

차례

머리말	1.1
소개	1.2
시작하기	1.3
다운로드 및 설치	1.3.1
기본적인 사용법	1.3.2
튜토리얼	1.4
기본 뉴럴 네트워크	1.4.1
MNIST 초급	1.4.1.1
MNIST 고급	1.4.1.2
텐서플로우 구조	1.4.1.3
MNIST 데이터 다운로드	1.4.1.4
tf.contrib.learn을 사용한 간편한 머신러닝	1.4.2
tf.contrib.learn 시작하기	1.4.2.1
tf.contrib.learn 선형모델 소개	1.4.2.2
선형모델 튜토리얼	1.4.2.3
와이드앤 딥 러닝 튜토리얼	1.4.2.4
텐서플로우 서빙	1.4.3
텐서플로우 서빙	1.4.3.1
이미지 프로세싱	1.4.4
콘볼루션 뉴럴 네트워크	1.4.4.1
이미지 인식	1.4.4.2
언어와 시퀀스 프로세싱	1.4.5
word2vec 모델	1.4.5.1
리커런트 뉴럴 네트워크	1.4.5.2
seq2seq 모델	1.4.5.3
SyntaxNet	1.4.5.4
비머신러닝 어플리케이션	1.4.6
만델브로트	1.4.6.1
편미분 방정식	1.4.6.2
하우트	1.5

변수	1.5.1
텐서플로우 구조	1.5.2
텐서보드	1.5.3
그래프 시각화	1.5.4
데이터 로딩	1.5.5
쓰레드와 큐	1.5.6
분산처리	1.5.7
커스텀 연산자	1.5.8
문서화	1.5.9
커스텀 데이터 포맷	1.5.10
GPU	1.5.11
공유 변수	1.5.12
모델 파일	1.5.13
부분 학습	1.5.14
메타 그래프	1.5.15
정량화	1.5.16
API	1.6
개요	1.6.1
Python API	1.6.2
그래프 생성	1.6.2.1
상수, 시퀀스, 난수 생성	1.6.2.2
변수	1.6.2.3
텐서 변환	1.6.2.4
수학 함수	1.6.2.5
제어 연산자	1.6.2.6
이미지 처리	1.6.2.7
희소 텐서	1.6.2.8
입력 처리	1.6.2.9
데이터 IO	1.6.2.10
뉴럴 네트워크	1.6.2.11
그래프 실행	1.6.2.12
학습	1.6.2.13
함수 연산자	1.6.2.14
테스트	1.6.2.15

레이어	1.6.2.16
유ти리티	1.6.2.17
C++ API	1.6.3
class tensorflow::Env	1.6.3.1
class tensorflow::RandomAccessFile	1.6.3.2
class tensorflow::WritableFile	1.6.3.3
class tensorflow::EnvWrapper	1.6.3.4
class tensorflow::Session	1.6.3.5
class tensorflow::SessionOptions	1.6.3.6
class tensorflow::Status	1.6.3.7
class tensorflow::State	1.6.3.8
class tensorflow::Tensor	1.6.3.9
class tensorflow::TensorShape	1.6.3.10
class tensorflow::TensorShapeDim	1.6.3.11
class tensorflow::TensorShapeUtils	1.6.3.12
class tensorflow::PartialTensorShape	1.6.3.13
class tensorflow::PartialTensorShapeUtils	1.6.3.14
class tensorflow::Thread	1.6.3.15
class tensorflow::ThreadOptions	1.6.3.16
참고자료	1.7
텐서플로우 논문	1.7.1
사용하는 곳	1.7.2
자주하는 질문	1.7.3
용어	1.7.4
랭크, 크기, 타입	1.7.5
텐서플로우 버전	1.7.6
로드맵	1.7.7

TensorFlow 한글 문서

tensorflow.org 문서를 한글로 번역하여 깃북으로 제공하기 위한 레파지토리입니다.

작업하려는 부분을 깃허브의 이슈에 등록하고 자유롭게 진행하려고 합니다. 여러 사람이 모여 진행을 하다보면 자연스럽게 어느정도 규칙적인 리듬을 가지게 될 수 있을거라 생각합니다. 이 프로젝트는 페이스북 [텐서플로우 그룹](#)과 함께 진행하고 있습니다. 많은 분들이 참여해 주셨으면 좋겠습니다.

깃허브의 주소는 <https://github.com/tensorflowkorea/tensorflow-kr> 입니다.

깃북의 주소는 <https://tensorflowkorea.gitbooks.io/tensorflow-kr> 입니다.

깃허브의 작업 방법에 대해서는 [블로그 포스팅](#)을 참고해 주세요.

감사합니다.

번역 및 마크다운 참고사항

번역 범위

현재 텐서플로우는 버전이 빠르게 변하고 있어서 API 문서가 자주 변경됩니다. 따라서 가능하시면 api_docs 폴더 이외의 다른 문서를 먼저 번역하는 것이 합리적으로 생각됩니다. 각 폴더별 번역 진행 상황은 [@hunkim](#) 님께서 작업해 주신 [progress.md](#) 파일을 참고해 주세요. api_docs 문서를 제외하고 번역이 완료되면 몇명이서 전체 번역을 다듬으려고 합니다. 이 때 필요한 자원봉사자를 페이스북 텐서플로우 그룹을 통해 신청받으려고 합니다.

버전 관리 안내

텐서플로우 버전이 업데이트 되면서 번역된 문서 간의 버전 차이가 있을 수 있습니다. master 브랜치에는 v0.9의 영문 문서가 들어가 있습니다. 그리고 master 브랜치외에 r0.9, r0.10 두개의 브랜치가 추가 되었습니다. 각각 v0.9, v0.10의 영문 문서가 담겨 있습니다. 두개의 브랜치를 비교하면서 기 번역된 문서를 업데이트하거나 새로운 번역 문서를 추가할 수 있습니다.

master 브랜치의 번역 문서는 아래와 같이 번역된 버전을 제목 아래에 적어 주십시오. 만약 버전이 적혀있지 않을 경우에는 v0.9로 간주합니다.

```
# 순환 신경망(Recurrent Neural Networks)
(v0.9)
```

아래 pip-installation의 경우처럼 페이지내 이동을 위해서 네임드 앵커 태그가 없이 사용만 되는 경우가 있습니다.

- * [Pip 설치](#pip-installation): 이 방식으로 텐서플로우를 설치하거나 업그레이드할 때는 이 전에 작성했던 파이썬 프로그램에 영향을 미칠 수 있습니다.

이럴 때에는 pip-installation 이 가리키는 위치에 네임드 앵커 태그를 넣어주시면 됩니다.

```
...
<a id="pip-installation"></a>
## Pip Installation
...
```

TensorFlow

소개

텐서플로우(TensorFlow™)는 데이터 플로우 그래프(Data flow graph)를 사용하여 수치 연산을 하는 오픈소스 소프트웨어 라이브러리입니다. 그래프의 노드(Node)는 수치 연산을 나타내고 엣지(edge)는 노드 사이를 이동하는 다차원 데이터 배열(텐서,tensor)를 나타냅니다. 유연한 아키텍처로 구성되어 있어 코드 수정없이 데스크탑, 서버 혹은 모바일 디바이스에서 CPU나 GPU를 사용하여 연산을 구동시킬 수 있습니다. 텐서플로우는 원래 머신러닝과 딥 뉴럴 네트워크 연구를 목적으로 구글의 인공지능 연구 조직인 구글 브레인 팀의 연구자와 엔지니어들에 의해 개발되었습니다. 하지만 이 시스템은 여러 다른 분야에도 충분히 적용될 수 있습니다. 다음 문서에서 어떻게 텐서플로우를 설치하고 사용하는지에 대해 설명하겠습니다.

Table of Contents

시작하기

텐서플로우를 실제로 작동시켜 봅시다!

시작하기 전에 앞으로 무엇을 배울지 힌트를 얻기위해 파이썬 API로 된 텐서플로우 코드를 잠깐 보겠습니다.

이 코드는 2차원 샘플 데이터를 사용하여 분포에 맞는 직선을 찾는(역주: 회귀분석) 간단한 파이썬 프로그램입니다.

```
import tensorflow as tf
import numpy as np

# Create 100 phony x, y data points in NumPy, y = x * 0.1 + 0.3
x_data = np.random.rand(100).astype(np.float32)
y_data = x_data * 0.1 + 0.3

# Try to find values for W and b that compute y_data = W * x_data + b
# (We know that W should be 0.1 and b 0.3, but Tensorflow will
# figure that out for us.)
W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b

# Minimize the mean squared errors.
loss = tf.reduce_mean(tf.square(y - y_data))
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)

# Before starting, initialize the variables. We will 'run' this first.
init = tf.global_variables_initializer()

# Launch the graph.
sess = tf.Session()
sess.run(init)

# Fit the line.
for step in range(201):
    sess.run(train)
    if step % 20 == 0:
        print(step, sess.run(W), sess.run(b))

# Learns best fit is W: [0.1], b: [0.3]

# Close the Session when we're done.
sess.close()
```

코드의 앞 부분은 데이터 플로우 그래프를 만들고 있습니다. 텐서플로우는 세션이 만들어져서 `run` 함수가 호출되기 전까지 어떤 것도 실제로 실행하지 않습니다.

좀 더 흥미를 돋우기 위해 전형적인 머신러닝 모델이 텐서플로우에서 어떻게 구현되는지 살펴보시면 좋습니다. 뉴럴 네트워크 분야에서 가장 전형적인 문제는 MNIST 손글씨 숫자를 분류하는 것입니다. 우리는 여기서 두 가지 버전의 설명 즉 하나는 머신러닝 초보자를 위한 것과 하나는 전문가를 위한 버전을 제공합니다. 만약 다른 소프트웨어 패키지로 MNIST 모델을 여러번 훈련시킨 적이 있다면 붉은 알약을 선택하세요. 만약 MNIST에 대해 들어본 적이 없다면 푸른 알약을 선택하면 됩니다. 초보자와 전문가 사이의 어디라면 푸른색 말고 붉은 알약을 선택하시면 됩니다.



Images licensed CC BY-SA 4.0; original by W. Carter

바로 텐서플로우를 설치하고 배우고 싶다면 이 내용은 넘어가고 다음으로 진행해도 됩니다. 텐서플로우 기능을 설명하는 기술적인 튜토리얼에서 MNIST 예제를 또 사용하므로 다시 볼 수 있습니다.

Recommended Next Steps

- 다운로드 및 설치
- 기본적인 사용법
- 텐서플로우 구조
- 텐서플로우 플레이그라운드

다운로드와 셋업

텐서플로우는 바이너리 패키지나 깃허브 소스를 이용해 설치할 수 있습니다.

필요사항

텐서플로우 파이썬 API는 파이썬 2.7과 파이썬 3.3+을 지원합니다.

GPU 버전(아직 리눅스용만 있습니다)은 Cuda Toolkit 7.5 과 cuDNN v4 와 가장 잘 작동합니다. 소스를 이용해 설치하면 다른 버전(Cuda toolkit >= 7.0 과 cuDNN 6.5(v2), 7.0(v3), v5)도 사용할 수 있습니다. 자세한 내용은 [Cuda 설치](#) 부분을 참고해 주세요.

개요

여러가지 설치 방법을 지원하고 있습니다:

- [Pip 설치](#): 이 방식으로 텐서플로우를 설치하거나 업그레이드할 때는 이 전에 작성했던 파이썬 프로그램에 영향을 미칠 수 있습니다.
- [Virtualenv 설치](#): 텐서플로우를 각각의 디렉토리 안에 설치하므로 다른 프로그램에 영향을 미치지 않습니다.
- [아나콘다\(Anaconda\) 설치](#): 텐서플로우를 각 아나콘다 환경에 설치하므로 다른 프로그램에 영향을 미치지 않습니다.
- [도커\(Docker\) 설치](#): 텐서플로우를 도커 컨테이너에서 실행하므로 컴퓨터의 다른 프로그램과 분리되어 운영됩니다.
- [소스에서 설치](#): 텐서플로우를 pip wheel을 이용하여 빌드하고 설치합니다.

만약 Pip, Virtualenv, 아나콘다(Anaconda) 나 도커(Docker)를 잘 알고 있다면 필요에 맞게 설치 과정을 응용해도 좋습니다. pip 패키지 이름이나 도커 이미지 이름은 각 설치 섹션에 기재되어 있습니다.

설치시 에러가 발생하면 [자주 발생하는 문제](#)를 참고하세요.

Pip 설치

[Pip](#))는 파이썬 패키지를 설치하고 관리하는 패키지 매니저 프로그램입니다.

설치되는 동안 추가되거나 업그레이드 될 파이썬 패키지 목록은 [setup.py 파일의 REQUIRED_PACKAGES 섹션](#)에 있습니다.

pip가 설치되어 있지 않다면 먼저 pip를 설치해야 합니다(python 3일 경우는 pip3):

```
# Ubuntu/Linux 64-bit  
$ sudo apt-get install python-pip python-dev  
  
# Mac OS X  
$ sudo easy_install pip  
$ sudo easy_install --upgrade six
```

적절한 텐서플로우 바이너리를 선택합니다:

```
# Ubuntu/Linux 64-bit, CPU 전용, Python 2.7  
$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.9.0-cp27-none-linux_x86_64.whl  
  
# Ubuntu/Linux 64-bit, GPU 버전, Python 2.7  
# CUDA toolkit 7.5 와 CuDNN v4 필수. 다른 버전을 사용하려면 아래 "소스에서 설치" 섹션을 참고하세요.  
$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow-0.9.0-cp27-none-linux_x86_64.whl  
  
# Mac OS X, CPU 전용, Python 2.7  
$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/mac/tensorflow-0.9.0-py2-none-any.whl  
  
# Ubuntu/Linux 64-bit, CPU 전용, Python 3.4  
$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.9.0-cp34-cp34m-linux_x86_64.whl  
  
# Ubuntu/Linux 64-bit, GPU 버전, Python 3.4  
# CUDA toolkit 7.5 와 CuDNN v4 필수. 다른 버전을 사용하려면 아래 "소스에서 설치" 섹션을 참고하세요.  
$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow-0.9.0-cp34-cp34m-linux_x86_64.whl  
  
# Ubuntu/Linux 64-bit, CPU 전용, Python 3.5  
$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.9.0-cp35-cp35m-linux_x86_64.whl  
  
# Ubuntu/Linux 64-bit, GPU 버전, Python 3.5  
# CUDA toolkit 7.5 와 CuDNN v4 필수. 다른 버전을 사용하려면 아래 "소스에서 설치" 섹션을 참고하세요.  
$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow-0.9.0-cp35-cp35m-linux_x86_64.whl  
  
# Mac OS X, CPU 전용, Python 3.4 or 3.5:  
$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/mac/tensorflow-0.9.0-py3-none-any.whl
```

텐서플로우를 설치합니다:

```
# Python 2  
$ sudo pip install --upgrade $TF_BINARY_URL  
  
# Python 3  
$ sudo pip3 install --upgrade $TF_BINARY_URL
```

NOTE: 만약 텐서플로우 0.7.1 버전 이하에서 업그레이드하는 경우라면 protobuf 업데이트를 반영하기 위해 반드시 `pip uninstall` 을 사용하여 텐서플로우 이전 버전과 protobuf 를 언인스톨한 후 진행해야 합니다.

[설치 후 테스트](#)를 해 보세요.

Virtualenv 설치

[Virtualenv](#) 은 각기 다른 파이썬 프로젝트에서 필요한 패키지들의 버전이 충돌되지 않도록 다른 공간에서 운영되도록 하는 툴입니다. 텐서플로우를 Virtualenv 로 설치하면 기존 파이썬 패키지들을 덮어쓰지 않게됩니다.

[Virtualenv](#) 설치 과정은 다음과 같습니다:

- pip 와 Virtualenv 를 설치합니다.
- Virtualenv 환경을 만듭니다.
- Virtualenv 환경을 활성화 하고 그 안에서 텐서플로우를 설치합니다.
- 설치 후에는 텐서플로우를 사용하고 싶을 때마다 Virtualenv 환경을 활성화하면 됩니다.

pip 와 Virtualenv 를 설치합니다:

```
# Ubuntu/Linux 64-bit  
$ sudo apt-get install python-pip python-dev python-virtualenv  
  
# Mac OS X  
$ sudo easy_install pip  
$ sudo pip install --upgrade virtualenv
```

디렉토리 `~/tensorflow` 에 Virtualenv 환경을 만듭니다:

```
$ virtualenv --system-site-packages ~/tensorflow
```

환경을 활성화 합니다:

```
$ source ~/tensorflow/bin/activate # bash를 사용할 경우  
$ source ~/tensorflow/bin/activate.csh # csh를 사용할 경우  
(tensorflow)$ # 프롬프트가 바뀌게 됩니다
```

이제 pip 설치 방식과 동일하게 텐서플로우를 설치합니다. 먼저 적절한 바이너리를 선택합니다:

```
# Ubuntu/Linux 64-bit, CPU 전용, Python 2.7
(tensorflow)$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/cpu
/tensorflow-0.9.0-cp27-none-linux_x86_64.whl

# Ubuntu/Linux 64-bit, GPU 버전, Python 2.7
# CUDA toolkit 7.5 와 CuDNN v4 필수. 다른 버전을 사용하려면 아래 "소스에서 설치" 섹션을 참고하세요.
(tensorflow)$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/gpu
/tensorflow-0.9.0-cp27-none-linux_x86_64.whl

# Mac OS X, CPU 전용, Python 2.7
(tensorflow)$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/mac/tenso
rflow-0.9.0-py2-none-any.whl

# Ubuntu/Linux 64-bit, CPU 전용, Python 3.4
(tensorflow)$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/cpu
/tensorflow-0.9.0-cp34-cp34m-linux_x86_64.whl

# Ubuntu/Linux 64-bit, GPU 버전, Python 3.4
# CUDA toolkit 7.5 와 CuDNN v4 필수. 다른 버전을 사용하려면 아래 "소스에서 설치" 섹션을 참고하세요.
(tensorflow)$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/gpu
/tensorflow-0.9.0-cp34-cp34m-linux_x86_64.whl

# Ubuntu/Linux 64-bit, CPU 전용, Python 3.5
(tensorflow)$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/cpu
/tensorflow-0.9.0-cp35-cp35m-linux_x86_64.whl

# Ubuntu/Linux 64-bit, GPU 버전, Python 3.5
# CUDA toolkit 7.5 와 CuDNN v4 필수. 다른 버전을 사용하려면 아래 "소스에서 설치" 섹션을 참고하세요.
(tensorflow)$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/gpu
/tensorflow-0.9.0-cp35-cp35m-linux_x86_64.whl

# Mac OS X, CPU 전용, Python 3.4 or 3.5:
(tensorflow)$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/mac/tenso
rflow-0.9.0-py3-none-any.whl
```

마지막으로 텐서플로우를 설치합니다:

```
# Python 2
(tensorflow)$ pip install --upgrade $TF_BINARY_URL

# Python 3
(tensorflow)$ pip3 install --upgrade $TF_BINARY_URL
```

Virtualenv 활성화하고 [설치 테스트](#)를 할 수 있습니다.

텐서플로우 작업을 마쳤을 때에는 환경을 비활성화 합니다.

```
(tensorflow)$ deactivate  
$ # 프롬프트가 원래대로 되돌아 옵니다
```

나중에 텐서플로우를 다시 사용하려면 Virtualenv 환경을 다시 활성화해야 합니다:

```
$ source ~/tensorflow/bin/activate # bash를 사용할 경우  
$ source ~/tensorflow/bin/activate.csh # csh를 사용할 경우  
(tensorflow)$ # 프롬프트가 변경되었습니다  
# 텐서플로우를 사용한 프로그램을 실행시킵니다  
...  
# 텐서플로우 사용을 마쳤을 때에는 환경을 비활성화 합니다  
(tensorflow)$ deactivate
```

Anaconda 설치

Anaconda 는 여러 수학, 과학 패키지를 기본적으로 포함하고 있는 파이썬 배포판입니다. Anaconda 는 "conda" 로 불리는 패키지 매니저를 사용하여 Virtualenv 와 유사한 [환경 시스템](#)을 제공합니다. (역주: 텐서플로우 뿐만이 아니라 일반적인 데이터 사이언스를 위해서도 아나콘다를 추천합니다)

Virtualenv 처럼 conda 환경은 각기 다른 파이썬 프로젝트에서 필요한 패키지들의 버전이 충돌되지 않도록 다른 공간에서 운영합니다. 텐서플로우를 Anaconda 환경으로 설치하면 기존 파이썬 패키지들을 덮어쓰지 않게됩니다.

- Anaconda를 설치합니다.
- conda 환경을 만듭니다.
- conda 환경을 활성화 하고 그 안에 텐서플로우를 설치합니다.
- 설치 후에는 텐서플로우를 사용하고 싶을 때마다 conda 환경을 활성화하면 됩니다.

Anaconda를 설치합니다:

[Anaconda 다운로드 사이트](#)의 안내를 따릅니다.

tensorflow 이름을 갖는 conda 환경을 만듭니다:

```
# Python 2.7  
$ conda create -n tensorflow python=2.7  
  
# Python 3.4  
$ conda create -n tensorflow python=3.4  
  
# Python 3.5  
$ conda create -n tensorflow python=3.5
```

환경을 활성화시키고 그 안에서 pip를 이용하여 텐서플로우를 설치합니다. `easy_install` 관련한 에러를 방지하려면 `--ignore-installed` 플래그를 사용합니다.

```
$ source activate tensorflow  
(tensorflow)$ # 프롬프트가 바뀝니다
```

이제 pip 설치 방식과 동일하게 텐서플로우를 설치합니다. 먼저 적절한 바이너리를 선택합니다:

```
# Ubuntu/Linux 64-bit, CPU 전용, Python 2.7  
(tensorflow)$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/cpu  
/tensorflow-0.9.0-cp27-none-linux_x86_64.whl  
  
# Ubuntu/Linux 64-bit, GPU 버전, Python 2.7  
# CUDA toolkit 7.5 와 CuDNN v4 필수. 다른 버전을 사용하려면 아래 "소스에서 설치" 섹션을 참고하세요.  
(tensorflow)$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/gpu  
/tensorflow-0.9.0-cp27-none-linux_x86_64.whl  
  
# Mac OS X, CPU 전용, Python 2.7  
(tensorflow)$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/mac/tensorflow-0.9.0-py2-none-any.whl  
  
# Ubuntu/Linux 64-bit, CPU 전용, Python 3.4  
(tensorflow)$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/cpu  
/tensorflow-0.9.0-cp34-cp34m-linux_x86_64.whl  
  
# Ubuntu/Linux 64-bit, GPU 버전, Python 3.4  
# CUDA toolkit 7.5 와 CuDNN v4 필수. 다른 버전을 사용하려면 아래 "소스에서 설치" 섹션을 참고하세요.  
(tensorflow)$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/gpu  
/tensorflow-0.9.0-cp34-cp34m-linux_x86_64.whl  
  
# Ubuntu/Linux 64-bit, CPU 전용, Python 3.5  
(tensorflow)$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/cpu  
/tensorflow-0.9.0-cp35-cp35m-linux_x86_64.whl  
  
# Ubuntu/Linux 64-bit, GPU 버전, Python 3.5  
# CUDA toolkit 7.5 와 CuDNN v4 필수. 다른 버전을 사용하려면 아래 "소스에서 설치" 섹션을 참고하세요.  
(tensorflow)$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/gpu  
/tensorflow-0.9.0-cp35-cp35m-linux_x86_64.whl  
  
# Mac OS X, CPU 전용, Python 3.4 or 3.5:  
(tensorflow)$ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/mac/tensorflow-0.9.0-py3-none-any.whl
```

마지막으로 텐서플로우를 설치합니다:

```
# Python 2  
(tensorflow)$ pip install --upgrade $TF_BINARY_URL  
  
# Python 3  
(tensorflow)$ pip3 install --upgrade $TF_BINARY_URL
```

conda 활성화하고 [설치 테스트](#)를 할 수 있습니다.

텐서플로우 작업을 마쳤을 때에는 환경을 비활성화 합니다.

```
(tensorflow)$ source deactivate  
  
$ # Your prompt should change back
```

나중에 텐서플로우를 다시 사용하려면 conda 환경을 다시 활성화해야 합니다::

```
$ source activate tensorflow  
(tensorflow)$ # 프롬프트가 바뀌었습니다  
# 텐서플로우를 사용한 프로그램을 실행시킵니다  
...  
# 텐서플로우 사용을 마쳤을 때에는 환경을 비활성화 합니다  
(tensorflow)$ source deactivate
```

도커(Docker) 설치

Docker는 로컬 컴퓨터에서 컨테이너로 리눅스 운영체제를 운영할 수 있는 시스템입니다. 도커를 사용하여 텐서플로우를 설치하고 사용한다면 이는 로컬 컴퓨터의 패키지와 완전히 분리된 것입니다.

네개의 도커 이미지가 제공됩니다:

- gcr.io/tensorflow/tensorflow : TensorFlow CPU 바이너리 이미지.
- gcr.io/tensorflow/tensorflow:latest-devel : CPU 바이너리 이미지와 소스 코드.
- gcr.io/tensorflow/tensorflow:latest-gpu : TensorFlow GPU 바이너리 이미지.
- gcr.io/tensorflow/tensorflow:latest-devel-gpu : GPU 바이너리 이미지와 소스 코드.

최근 릴리즈는 버전대신 latest 태그를 표시합니다(예, 0.9.0-gpu).

도커를 이용한 설치는 아래와 같습니다:

- 로컬 컴퓨터에 도커를 설치합니다.
- sudo 없이 컨테이너를 시작할 수 있도록 [도커 그룹](#) 을 만듭니다.
- 텐서플로우 이미지로 도커 컨테이너를 시작합니다. 처음 시작할 때 자동으로 이미지를 다운로드합니다.

로컬 컴퓨터에 도커를 설치하는 설명은 [도커 설치](#) 를 참고하세요.

도커가 설치되면 텐서플로우 바이너리 이미지로 아래와 같이 도커 컨테이너를 실행합니다.

```
$ docker run -it -p 8888:8888 gcr.io/tensorflow/tensorflow
```

옵션 `-p 8888:8888` 은 로컬(호스트) 컴퓨터가 도커 컨테이너로 접속할 수 있는 포트를 지정합니다. 여기서는 쥬피터(Jupyter) 노트북 연결을 위한 포트입니다.

포트를 매핑하는 형식은 `호스트포트:컨테이너포트` 입니다. 컨테이너 포트 `8888` 에 대한 호스트 포트는 임의의 포트를 지정할 수 있습니다.

NVidia GPU를 위해서는 최신 NVidia 드라이버와 [nvidia-docker](#) 를 설치하고 아래와 같이 실행합니다.

```
$ nvidia-docker run -it -p 8888:8888 gcr.io/tensorflow/tensorflow:latest-gpu
```

더 자세한 것은 [텐서플로우 도커](#) 문서를 참고하세요.

도커 컨테이너 안에서 [설치 테스트](#)를 할 수 있습니다.

텐서플로우 설치 테스트

(선택사항, Linux) GPU 활성화

텐서플로우 GPU 버전을 설치했다면 반드시 Cuda Toolkit 7.5 and cuDNN v4 도 설치해야 합니다. [Cuda 설치](#)을 참고하세요.

`LD_LIBRARY_PATH` 와 `CUDA_HOME` 환경 변수를 지정해야 합니다. 아래 명령을 `~/.bash_profile` 파일에 추가하는 것이 좋습니다. 이 명령은 `/usr/local/cuda` 에 CUDA 가 설치되어있다고 가정한 것입니다:

```
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/local/cuda/lib64"  
export CUDA_HOME=/usr/local/cuda
```

커맨드 라인에서 텐서플로우 실행하기

에러가 발생하면 [자주 발생하는 문제](#) 섹션을 참고하세요.

터미널을 열고 아래 명령을 실행합니다:

```
$ python
...
>>> import tensorflow as tf
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()
>>> print(sess.run(hello))
Hello, TensorFlow!
>>> a = tf.constant(10)
>>> b = tf.constant(32)
>>> print(sess.run(a + b))
42
>>>
```

텐서플로우 데모 모델 실행

데모 모델을 포함해 텐서플로우의 모든 패키지는 파이썬 라이브러리로 설치되어 있습니다. 파이썬 라이브러리의 정확한 경로는 설치된 시스템마다 다릅니다. 하지만 보통 아래 중에 하나일 것입니다:

```
/usr/local/lib/python2.7/dist-packages/tensorflow
/usr/local/lib/python2.7/site-packages/tensorflow
```

아래 명령으로 정확한 디렉토리를 찾을 수 있습니다(텐서플로우를 설치한 파이썬을 사용해야 합니다. 예를 들면 파이썬 3에서 텐서플로우를 설치했다면 `python` 대신 `python3` 를 사용해야 합니다):

```
$ python -c 'import os; import inspect; import tensorflow; print(os.path.dirname(inspect.getfile(tensorflow)))'
```

MNIST 데이터셋을 이용한 손글씨 숫자를 분류하는 간단한 데모 모델은 `models/image/mnist/convolutional.py` 에 있습니다. 커맨드라인에서 다음과 같이 실행시킬 수 있습니다(텐서플로우를 설치한 파이썬인지 확인하세요):

```
# 파일 검색 범위에서 프로그램을 찾기 위해서 'python -m' 명령을 이용합니다:  
$ python -m tensorflow.models.image.mnist.convolutional  
Extracting data/train-images-idx3-ubyte.gz  
Extracting data/train-labels-idx1-ubyte.gz  
Extracting data/t10k-images-idx3-ubyte.gz  
Extracting data/t10k-labels-idx1-ubyte.gz  
...etc...  
  
# 파일 인터프리터에 모델 프로그램의 파일 경로를 전달할 수 있습니다.  
# (텐서플로우가 설치된 파일 버전을 사용해야 합니다).  
# 예를 들어, 파일 3의 경우는 .../python3.X/... 가 됩니다.).  
$ python /usr/local/lib/python2.7/dist-packages/tensorflow/models/image/mnist/convolut  
ional.py  
...
```

소스에서 설치

소스에서 설치하려면 pip를 사용해서 진행할 수 있도록 pip 훈(wheel)을 만듭니다. pip를 설치하려면 [Pip 설치](#) 섹션을 참고하세요.

텐서플로우 레파지토리 클론(Clone)하기

```
$ git clone https://github.com/tensorflow/tensorflow
```

아래 방법은 최신 마스터 브랜치의 텐서플로우를 설치하는 것입니다. 만약 특정 브랜치(릴리즈 브랜치 같은)를 설치하고 싶다면 `git clone` 명령에 `-b <branchname>` 옵션을 추가하고 r0.8 과 그 이전 버전에서는 protobuf 라이브러리를 추가하기 위해 `--recurse-submodules` 옵션을 추가합니다.

리눅스 설치

Bazel 설치

Bazel에 필요한 소프트웨어를 [여기](#)를 따라 설치합니다. [자신의 컴퓨터에 맞는 인스톨러](#)를 사용하여 최신 안정버전의 `bazel`을 다운로드 하여 아래와 같이 실행합니다:

```
$ chmod +x PATH_TO_INSTALL.SH  
$ ./PATH_TO_INSTALL.SH --user
```

`PATH_TO_INSTALL.SH` 부분을 다운로드 받은 인스톨러의 경로를 바꾸어 줍니다.

마지막으로 실행 경로에 `bazel` 을 추가하기 위해 화면의 설명을 따릅니다.

다른 의존성 라이브러리 설치

```
# Python 2.7:  
$ sudo apt-get install python-numpy swig python-dev python-wheel  
# Python 3.x:  
$ sudo apt-get install python3-numpy swig python3-dev python3-wheel
```

설치환경 설정

루트 디렉토리에 있는 `configure` 스크립트를 실행합니다. 환경설정 스크립트는 파이썬 인터프리터의 경로를 요청하고 (선택사항으로)CUDA 라이브러리를 설정합니다 ([아래](#)를 참고하세요).

이 단계에서는 파이썬과 넘파이(numpy) 헤더파일을 찾습니다.

```
$ ./configure  
Please specify the location of python. [Default is /usr/bin/python]:
```

선택사항: CUDA 설치 (리눅스 GPU)

GPU 버전의 텐서플로우를 설치하고 실행하기 위해서는 엔비디아(NVIDIA)의 쿠다 툴킷(Cuda Toolkit) (>= 7.0)과 cuDNN(>= v2)을 설치해야 합니다.

텐서플로우 GPU 버전은 엔비디아(NVidia)의 Compute Capability >= 3.0 이상을 지원하는 GPU 카드를 필요로 합니다. 지원되는 카드는 아래 목록을 포함하고 있습니다:

- NVidia Titan
- NVidia Titan X
- NVidia K20
- NVidia K40

GPU 카드의 NVIDIA Compute Capability 체크

<https://developer.nvidia.com/cuda-gpus>

Cuda Toolkit 다운로드 및 설치

<https://developer.nvidia.com/cuda-downloads>

텐서플로우의 바이너리 릴리즈를 사용하려면 버전 7.5를 설치하세요.

`/usr/local/cuda` 등에 툴킷을 설치합니다.

cuDNN 다운로드 및 설치

<https://developer.nvidia.com/cudnn>

cuDNN v4를 다운로드합니다(v5는 현재 릴리즈 후보 상태로 텐서플로우를 소스에서 설치할 때만 사용할 수 있습니다).

압축을 풀어 툴킷 디렉토리에 cuDNN 파일을 복사합니다. `/usr/local/cuda` 에 툴킷이 설치되어 있다고 가정하고 아래 명령을 실행합니다(다운로드 받은 cuDNN의 적절한 버전을 반영해 주세요):

```
tar xvzf cudnn-7.5-linux-x64-v4.tgz  
sudo cp cudnn-7.5-linux-x64-v4/cudnn.h /usr/local/cuda/include  
sudo cp cudnn-7.5-linux-x64-v4/libcudnn* /usr/local/cuda/lib64  
sudo chmod a+r /usr/local/cuda/include/cudnn.h /usr/local/cuda/lib64/libcudnn*
```

텐서플로우에서 Cuda 라이브러리 선택

소스 디렉토리의 맨 위에서 `configure` 스크립트를 실행하고 텐서플로우를 GPU 지원하도록 빌드 할지 물어볼 때 `y` 를 선택하세요. 만약 여러가지 버전의 Cuda와 cuDNN이 설치되어 있다면 디폴트 대신 구체적으로 어떤 버전을 사용할지 지정해야 합니다. 아래와 같은 질문들을 보게됩니다:

```
$ ./configure
Please specify the location of python. [Default is /usr/bin/python]:
Do you wish to build TensorFlow with GPU support? [y/N] y
GPU support will be enabled for TensorFlow

Please specify which gcc nvcc should use as the host compiler. [Default is
/usr/bin/gcc]: /usr/bin/gcc-4.9

Please specify the Cuda SDK version you want to use, e.g. 7.0. [Leave
empty to use system default]: 7.5

Please specify the location where CUDA 7.5 toolkit is installed. Refer to
README.md for more details. [default is: /usr/local/cuda]: /usr/local/cuda

Please specify the Cudnn version you want to use. [Leave empty to use system
default]: 4.0.4

Please specify the location where the cuDNN 4.0.4 library is installed. Refer to
README.md for more details. [default is: /usr/local/cuda]: /usr/local/cudnn-r4-rc/

Please specify a list of comma-separated Cuda compute capabilities you want to
build with. You can find the compute capability of your device at:
https://developer.nvidia.com/cuda-gpus.
Please note that each additional compute capability significantly increases your
build time and binary size. [Default is: "3.5,5.2"]: 3.5

Setting up Cuda include
Setting up Cuda lib64
Setting up Cuda bin
Setting up Cuda nvvm
Setting up CUPTI include
Setting up CUPTI lib64
Configuration finished
```

시스템에 있는 Cuda 라이브러리를 가리키는 기본으로 사용할 심볼릭 링크들을 만듭니다. `bazel` 빌드 명령을 실행하기 전에 Cuda 라이브러리 경로를 바꾸게 되면 이 단계를 다시 거쳐야 합니다. Cudnn 라이브러리 R2는 '6.5'를 R3는 '7.0'을 R4-RC는 '4.0.4'를 선택합니다.

GPU를 지원하도록 빌드하기

소스 트리 맨 위에서 실행:

```
$ bazel build -c opt --config=cuda //tensorflow/cc:tutorials_example_trainer

$ bazel-bin/tensorflow/cc/tutorials_example_trainer --use_gpu
# 많은 출력이 나옵니다. 이 튜토리얼은 GPU에서 2x2 행렬의 고유값을 반복해서 계산합니다.
# 마지막 몇 줄은 아래와 같습니다.

000009/000005 lambda = 2.000000 x = [0.894427 -0.447214] y = [1.788854 -0.894427]
000006/000001 lambda = 2.000000 x = [0.894427 -0.447214] y = [1.788854 -0.894427]
000009/000009 lambda = 2.000000 x = [0.894427 -0.447214] y = [1.788854 -0.894427]
```

GPU 지원을 활성화하기 위해서는 "--config=cuda" 옵션이 필요합니다.

알려진 이슈

- 하나의 소스 트리에서 Cuda 와 non-Cuda 두가지 설정으로 모두 빌드가 가능하지만 설정을 바꾸려면 `bazel clean` 을 실행해 주세요.
- `bazel` 빌드를 하기 전에 환경 설정을 해야 합니다. 그렇지 않으면 빌드가 실패합니다. 향후에는 빌드 프로세스 안에 환경 설정 단계를 포함시켜 좀 더 편리하게 만드려고 생각하고 있습니다.

Mac OS X 설치

`bazel`과 SWIG 설치를 위해서는 [homebrew](#)를 사용하고 `easy_install`이나 `pip`을 사용하여 파이썬 라이브러리를 설치하길 권장합니다.

물론 `homebrew`을 사용하지 않고 소스에서 `Swig`를 설치할 수도 있습니다. 그런 경우에 의존성 라이브러리인 [PCRE](#)를 설치해야 합니다. PCRE2가 아닙니다.

의존성 라이브러리

`bazel`의 의존성 라이브러리를 설치하려면 [이곳](#)의 안내를 따르세요. `bazel`과 SWIG 설치를 위해 `homebrew`을 사용할 수 있습니다:

```
$ brew install bazel swig
```

`easy_install`이나 `pip`을 사용하여 파이썬 의존성을 설치할 수 있습니다. `easy_install`을 사용할 경우 아래를 실행합니다

```
$ sudo easy_install -U six
$ sudo easy_install -U numpy
$ sudo easy_install wheel
```

기능이 강화된 파이썬 웰인 [ipython](#)을 권장합니다. 다음과 같이 설치합니다:

```
$ sudo easy_install ipython
```

GPU 지원이 되도록 빌드하려면 [homebrew](#)를 사용해 GNU coreutils가 설치되어 있어야 합니다:

```
$ brew install coreutils
```

다음은 [NVIDIA](#) 사이트에서 OSX 버전에 맞는 패키지를 다운로드 하거나 [Homebrew Cask](#) 확장 을 사용하여 최신의 [CUDA Toolkit](#)을 설치해야 합니다:

```
$ brew tap caskroom/cask  
$ brew cask install cuda
```

CUDA 툴킷을 설치하면 필요한 환경 변수를 `~/.bash_profile` 파일에 아래와 같이 셋팅해야 합니다:

```
export CUDA_HOME=/usr/local/cuda  
export DYLD_LIBRARY_PATH="$DYLD_LIBRARY_PATH:$CUDA_HOME/lib"  
export PATH="$CUDA_HOME/bin:$PATH"
```

마지막으로 [Accelerated Computing Developer Program](#) 계정이 필요한 [CUDA Deep Neural Network](#) (cuDNN)를 설치할 수도 있습니다. 로컬 컴퓨터에 다운로드 받고 난 후 압축을 풀고 헤더 파일과 라이브러리를 CUDA 툴킷 폴더에 옮깁니다:

```
$ sudo mv include/cudnn.h /Developer/NVIDIA/CUDA-7.5/include/  
$ sudo mv lib/libcudnn* /Developer/NVIDIA/CUDA-7.5/lib  
$ sudo ln -s /Developer/NVIDIA/CUDA-7.5/lib/libcudnn* /usr/local/cuda/lib/
```

설치환경 설정

소스 트리 맨 위에서 `configure` 명령을 실행합니다. 이 스크립트는 파이썬 인터프리터의 경로를 묻습니다.

이 단계에서 CUDA와 툴킷이 설치되어 있을 때 GPU 지원을 활성화 하는 것은 물론 파이썬과 numpy 헤더 파일들의 위치를 찾습니다. 예를 들면:

```
$ ./configure
Please specify the location of python. [Default is /usr/bin/python]:
Do you wish to build TensorFlow with Google Cloud Platform support? [y/N] N
No Google Cloud Platform support will be enabled for TensorFlow
Do you wish to build TensorFlow with GPU support? [y/N] y
GPU support will be enabled for TensorFlow
Please specify which gcc nvcc should use as the host compiler. [Default is /usr/bin/gcc]:
Please specify the Cuda SDK version you want to use, e.g. 7.0. [Leave empty to use system default]: 7.5
Please specify the location where CUDA 7.5 toolkit is installed. Refer to README.md for more details. [Default is /usr/local/cuda]:
Please specify the Cudnn version you want to use. [Leave empty to use system default]: 5
Please specify the location where cuDNN 5 library is installed. Refer to README.md for more details. [Default is /usr/local/cuda]:
Please specify a list of comma-separated Cuda compute capabilities you want to build with.
You can find the compute capability of your device at: https://developer.nvidia.com/cuda-gpus.
Please note that each additional compute capability significantly increases your build time and binary size.
[Default is: "3.5,5.2"]: 3.0
Setting up Cuda include
Setting up Cuda lib
Setting up Cuda bin
Setting up Cuda nvvm
Setting up CUPTI include
Setting up CUPTI lib64
Configuration finished
```

pip 패키지 생성 및 설치

소스에서 설치할 때 pip 패키지를 만들고 설치해야 합니다.

```
$ bazel build -c opt //tensorflow/tools/pip_package:build_pip_package

# To build with GPU support:
$ bazel build -c opt --config=cuda //tensorflow/tools/pip_package:build_pip_package

$ bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tensorflow_pkg

# The name of the .whl file will depend on your platform.
$ sudo pip install /tmp/tensorflow_pkg/tensorflow-0.9.0-py2-none-any.whl
```

텐서플로우 개발자 셋팅

텐서플로우 자체를 수정할 때 텐서플로우를 재 설치하지 않고 파이썬 대화식 쉘에서 변경 내용을 테스트할 수 있다면 매우 유용할 것입니다.

모든 파일이 시스템 디렉토리로 부터 링크(복사가 아닌)되도록 텐서플로우를 셋팅하기 위해서는 텐서플로우 루트 디렉토리에서 다음 명령을 실행합니다:

```
bazel build -c opt //tensorflow/tools/pip_package:build_pip_package

# To build with GPU support:
bazel build -c opt --config=cuda //tensorflow/tools/pip_package:build_pip_package

mkdir _python_build
cd _python_build
ln -s ../bazel-bin/tensorflow/tools/pip_package/build_pip_package.runfiles/org_tensorflow/* .
ln -s ../tensorflow/tools/pip_package/* .
python setup.py develop
```

C++ 파일을 변경하거나 어떤 파이썬 파일이든지 추가, 삭제, 이동될 때 혹은 bazel 빌드 룰을 바꿀 때마다 `//tensorflow/tools/pip_package:build_pip_package` 을 다시 빌드해야 합니다.

텐서플로우로 첫번째 뉴럴 네트워크 모델을 학습 시키기

소스 트리 루트에서 실행합니다:

```
$ cd tensorflow/models/image/mnist
$ python convolutional.py
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting data/train-images-idx3-ubyte.gz
Extracting data/train-labels-idx1-ubyte.gz
Extracting data/t10k-images-idx3-ubyte.gz
Extracting data/t10k-labels-idx1-ubyte.gz
Initialized!
Epoch 0.00
Minibatch loss: 12.054, learning rate: 0.010000
Minibatch error: 90.6%
Validation error: 84.6%
Epoch 0.12
Minibatch loss: 3.285, learning rate: 0.010000
Minibatch error: 6.2%
Validation error: 7.0%
...
...
```

자주 발생하는 문제

GPU 관련 이슈들

텐서플로우 프로그램을 실행할 때 다음과 같은 에러를 만날 경우:

```
ImportError: libcudart.so.7.0: cannot open shared object file: No such file or directory
```

GPU 설치 [가이드](#)를 따랐는지 확인하세요. 소스에서 설치할 때 Cuda나 cuDNN 버전은 비워둔 채 진행했다면 명시적으로 지정하여 다시 시도해 보세요.

Protobuf 라이브러리 관련 이슈들

텐서플로우 pip 패키지는 protobuf pip 패키지 버전 3.0.0b2를 필요로 합니다. [PyPI](#)에서 다운받을 수 있는(`pip install protobuf`를 사용해서) Protobuf의 pip 패키지는 파이썬 만으로 개발된 라이브러리로 C++ 구현보다 직렬화/역직렬화시 10~50배 느립니다. Protobuf는 빠른 프로토콜 파싱을 위한 C++ 바이너리 확장을 지원합니다. 이 확장은 표준 파이썬 PIP 패키지에는 포함되어 있지 않습니다. 우리는 이 바이너리 확장을 포함한 protobuf pip 패키지를 자체적으로 만들었습니다. 다음 명령을 사용해 자체적으로 만든 protobuf pip 패키지를 설치할 수 있습니다:

```
# Ubuntu/Linux 64-bit:  
$ pip install --upgrade https://storage.googleapis.com/tensorflow/linux/cpu/protobuf-3  
.0.0b2.post2-cp27-none-linux_x86_64.whl  
  
# Mac OS X:  
$ pip install --upgrade https://storage.googleapis.com/tensorflow/mac/protobuf-3.0.0b2  
.post2-cp27-none-any.whl
```

Python 3 에서는 :

```
# Ubuntu/Linux 64-bit:  
$ pip3 install --upgrade https://storage.googleapis.com/tensorflow/linux/cpu/protobuf-  
3.0.0b2.post2-cp34-none-linux_x86_64.whl  
  
# Mac OS X:  
$ pip3 install --upgrade https://storage.googleapis.com/tensorflow/mac/protobuf-3.0.0b2  
.post2-cp35-none-any.whl
```

`pip install tensorflow` 명령은 파이썬으로된 기본 pip 패키지를 설치하므로 위 패키지를 설치하려면 반드시 텐서플로우를 설치하고 난 후에 합니다. 위 pip 패키지는 이미 설치된 protobuf 패키지를 덮어 씁니다. 바이너리 pip 패키지는 64M 넘는 메세지에 대한 지원을 이미 하고 있어 아래와 같은 에러가 이미 해결 되었습니다:

```
[libprotobuf ERROR google/protobuf/src/google/protobuf/io/coded_stream.cc:207] A  
protocol message was rejected because it was too big (more than 67108864 bytes).  
To increase the limit (or to disable these warnings), see  
CodedInputStream::SetTotalBytesLimit() in google/protobuf/io/coded_stream.h.
```

Pip 설치 이슈들

Cannot import name 'descriptor'

```
ImportError: Traceback (most recent call last):  
  File "/usr/local/lib/python3.4/dist-packages/tensorflow/core/framework/graph_pb2.py"  
, line 6, in <module>  
    from google.protobuf import descriptor as _descriptor  
ImportError: cannot import name 'descriptor'
```

최신 버전의 텐서플로우로 업그레이드할 때 위와 같은 에러가 발생하면 텐서플로우와 protobuf를 모두 언인스톨하고 텐서플로우를 다시 재설치합니다(올바른 protobuf 의존성을 찾기 위해서).

Can't find setup.py

`pip install` 하는 동안 아래와 같은 에러를 만나면:

```
...
IOError: [Errno 2] No such file or directory: '/tmp/pip-o6Tpui-build/setup.py'
```

해결책: pip를 업그레이드 합니다:

```
pip install --upgrade pip
```

pip 설치가 어떻게 되어 있느냐에 따라 `sudo` 를 필요로 할지 모릅니다.

SSLError: SSL_VERIFY_FAILED

URL로 부터 pip 인스톨을 하는 동안 아래와 같은 에러를 만나면:

```
...
SSLError: [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed
```

해결책: curl이나 wget으로 수동으로 wheel을 다운로드 받아 로컬에서 `pip install` 합니다.

Operation not permitted

`sudo` 를 사용함에도 아래와 같은 에러를 만나면:

```
...
Installing collected packages: setuptools, protobuf, wheel, numpy, tensorflow
Found existing installation: setuptools 1.1.6
Uninstalling setuptools-1.1.6:
Exception:
...
[Errno 1] Operation not permitted: '/tmp/pip-a1DXRT-uninstall/System/Library/Frameworks/Python.framework/Versions/2.7/Extras/lib/python/_markerlib'
```

해결책: pip 명령에 `--ignore-installed` 플래그를 추가합니다.

Linux 이슈들

이런 에러가 나오면:

```
...
"__add__", "__radd__",
^
SyntaxError: invalid syntax
```

해결책: 파이썬 2.7을 사용하고 있는지 확인합니다.

Mac OS X: ImportError: No module named copyreg

Mac OS X에서 텐서플로우를 임포트할 때 아래와 같은 에러가 나올 수 있습니다.

```
>>> import tensorflow as tf
...
ImportError: No module named copyreg
```

해결책: 텐서플로우는 `six-1.10.0` 파이썬 패키지를 필요로하는 protobuf에 의존성이 있습니다.
애플의 기본 파이썬 설치에는 `six-1.4.1`이 제공됩니다.

다음과 같은 방법으로 이를 해결 할 수 있습니다:

- 최신 버전의 `six`로 업그레이드 합니다:

```
$ sudo easy_install -U six
```

- 별도의 파이썬 환경에서 텐서플로우를 설치합니다:
 - [Virtualenv](#) 사용.
 - [Docker](#) 사용.
- [Homebrew](#)나 [MacPorts](#)를 사용하여 별도의 파이썬 버전을 설치한 후 텐서플로우를 그 파이썬에서 재 설치합니다.

Mac OS X: OSError: [Errno 1] Operation not permitted:

엘 캐피탄에서 "six"는 수정할 수 없는 스페셜 패키지라서 이 에러는 "pip install" 명령으로 이 패키지를 수정하려고 할 때 나타납니다. 이 문제를 해결하기 위해서는 "ignore-installed" 플래그를 사용합니다.

```
sudo pip install --ignore-installed six https://storage.googleapis.com/....
```

Mac OS X: TypeError: __init__() got an unexpected keyword argument 'syntax'

Mac OS X에서 텐서플로우를 임포트할 때 이런 에러가 나타날 수 있습니다.

```
>>> import tensorflow as tf
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.7/site-packages/tensorflow/__init__.py", line 4, in <module>
    from tensorflow.python import *
  File "/usr/local/lib/python2.7/site-packages/tensorflow/python/__init__.py", line 13, in <module>
    from tensorflow.core.framework.graph_pb2 import *
...
  File "/usr/local/lib/python2.7/site-packages/tensorflow/core/framework/tensor_shape_pb2.py", line 22, in <module>
    serialized_pb=_b('\n tensorflow/core/framework/tensor_shape.proto\x12\ntensorflow\x1d\n\x10TensorShapeProto\x12-\n\x03\x64im\x18\x02 \x03(\x0b\x32 .tensorflow.TensorShapeProto.Dim\x1a!\n\x03\x44im\x12\x0c\x04size\x18\x01 \x01(\x03\x12\x0c\x04name\x18\x02 \x01(\tb\x06proto3')
TypeError: __init__() got an unexpected keyword argument 'syntax'
```

이 에러는 protobuf 버전간의 충돌 때문입니다(protobuf 3.0.0이 필요합니다). 가장 좋은 해결책은 예전 버전의 protobuf를 업그레이드 하는 것입니다:

```
$ pip install --upgrade protobuf
```

기본 사용법(Basic Usage)

(v1.0)

TensorFlow를 사용하기 위해 먼저 TensorFlow가 어떻게 동작하는지를 이해해 봅시다.

- 연산은 graph로 표현합니다.(역자 주: graph는 점과 선, 업계 용어로는 노드와 엣지로 이뤄진 수학적인 구조를 의미합니다.)
- graph는 session 내에서 실행됩니다.
- 데이터는 tensor로 표현합니다.
- 변수(variable) 는 (역자 주: 여러 graph들이 작동할 때도) 그 상태를 유지합니다.
- 작업(operation 혹은 op)에서 데이터를 입출력 할 때 feed와 fetch를 사용할 수 있습니다.

개요(Overview)

TensorFlow는 graph로 연산(역자 주: 앞으로도 'computation'을 연산으로 번역합니다)을 나타내는 프로그래밍 시스템입니다. graph에 있는 노드는 작업(op)(작업(operation)의 약자)라고 부릅니다. 작업(op)은 0개 혹은 그 이상의 tensor를 가질 수 있고 연산도 수행하며 0개 혹은 그 이상의 Tensor를 만들어 내기도 합니다. Tensorflow에서 Tensor는 정형화된 다차원 배열(a typed multi-dimensional array)입니다. 예를 들어, 이미지는 부동소수점 수(floating point number)를 이용한 4차원 배열([batch, height(가로), width(세로), channels(역자 주: 예를 들어 RGB)])로 나타낼 수 있습니다.

TensorFlow에서 graph는 연산을 표현해놓은 것이라서 연산을 하려면 graph가 session 상에 실행되어야 합니다. session은 graph의 작업(op)(역자 주: operation. graph를 구성하는 노드)을 CPU나 GPU같은 Device에 배정하고 실행을 위한 메서드들을 제공합니다. 이런 메서드들은 작업(op)을 실행해서 tensor를 만들어 냅니다. tensor는 파이썬에서 numpy ndarray 형식으로 나오고 C와 C++에서는 TensorFlow::Tensor 형식으로 나옵니다.

연산 graph(The computation graph)

TensorFlow 프로그램은 보통 graph를 조립하는 '구성 단계(construction phase)'와 session을 이용해 graph의 op을 실행시키는 '실행 단계(execution phase)'로 구성됩니다.

예를 들어 뉴럴 네트워크를 표현하고 학습시키기 위해 구성 단계에는 graph를 만들고 실행 단계에는 graph의 훈련용 작업들(set of training ops)을 반복해서 실행합니다.

TensorFlow는 C, C++, 파이썬을 이용해서 쓸 수 있습니다. 지금은 graph를 만들기 위해 파이썬 라이브러리(역자 주: '파이썬 라이브러리'는 파이썬 라이브러리로 나온 TensorFlow를 의미합니다.)를 사용하는 것이 훨씬 쉽습니다. C, C++에서는 제공하지 않는 많은 헬퍼 함수들을 쓸 수 있기 때문이죠.

(역자 주: TensorFlow의) session 라이브러리들은 3개 언어에 동일한 기능을 제공합니다.

graph 만들기(Building the graph)

graph를 만드는 것은 상수(constant) 같이 아무 입력값이 필요없는 작업(op)을 정의하는 것에서부터 시작합니다. 이 op을 연산이 필요한 다른 op들에게 입력값으로 제공하는 것입니다.

파이썬 라이브러리의 작업 생성 함수(op constructor)는 만들어진 작업(op)들의 결과값을 반환합니다. 반환된 작업들의 결과값은 다른 작업(op)을 생성할 때 함수의 입력값으로 이용할 수 있습니다.

파이썬 라이브러리에는 작업 생성 함수로 노드를 추가할 수 있는 *default graph*라는 것이 있습니다. *default graph*는 다양하게 이용하기 좋습니다. [Graph class](#) 문서에서 여러 그래프를 명시적으로 관리하는 방법을 알 수 있습니다.

```
import tensorflow as tf

# 1x2 행렬을 만드는 constant op을 만들어 봅시다.
# 이 op는 default graph에 노드로 들어갈 것입니다.
# Create a constant op that produces a 1x2 matrix. The op is
# added as a node to the default graph.
#
# 생성함수에서 나온 값은 constant op의 결과값입니다.
# The value returned by the constructor represents the output
# of the constant op.
matrix1 = tf.constant([[3., 3.]])

# 2x1 행렬을 만드는 constant op을 만들어봅시다.
# Create another Constant that produces a 2x1 matrix.
matrix2 = tf.constant([[2.],[2.]])  

# 'matrix1'과 'matrix2'를 입력값으로 하는 Matmul op(역자 주: 행렬곱 op)을
# 만들어 봅시다.
# 이 op의 결과값인 'product'는 행렬곱의 결과를 의미합니다.
# Create a Matmul op that takes 'matrix1' and 'matrix2' as inputs.
# The returned value, 'product', represents the result of the matrix
# multiplication.
product = tf.matmul(matrix1, matrix2)
```

*default graph*에는 이제 3개의 노드가 있습니다. 2개는 상수(constant) 작업(op)이고 하나는 행렬곱(matmul) 작업(op)이죠. 그런데 사실 행렬을 곱해서 결과값을 얻으려면 Session 에다 graph를 실행해야 합니다.

session에서 graph 실행하기(Launching the graph in a session)

graph를 구성하고 나면 `Session` 오브젝트를 만들어서 graph를 실행할 수 있습니다. `op` 생성함수에서 다른 graph를 지정해줄 때까지는 default graph가 `Session`에서 실행됩니다. 관련 내용은 [Session class](#)에서 확인할 수 있습니다.

```
# default graph를 실행시켜 봅시다.
# Launch the default graph.
sess = tf.Session()

# 행렬곱 작업(op)을 실행하기 위해 session의 'run()' 메서드를 호출해서 행렬곱
# 작업의 결과값인 'product' 값을 넘겨줍시다. 그 결과값을 원한다는 뜻입니다.
# To run the matmul op we call the session 'run()' method, passing 'product'
# which represents the output of the matmul op. This indicates to the call
# that we want to get the output of the matmul op back.

#
# 작업에 필요한 모든 입력값들은 자동적으로 session에서 실행되며 보통은 병렬로
# 처리됩니다.
# All inputs needed by the op are run automatically by the session. They
# typically are run in parallel.
#
# 'run(product)'가 호출되면 op 3개가 실행됩니다. 2개는 상수고 1개는 행렬곱이죠.
# The call 'run(product)' thus causes the execution of three ops in the
# graph: the two constants and matmul.
#
# 작업의 결과물은 numpy `ndarray` 오브젝트인 'result' 값으로 나옵니다.
# The output of the op is returned in 'result' as a numpy `ndarray` object.
result = sess.run(product)
print(result)
# ==> [[ 12.]]

# 실행을 마치면 Session을 닫읍시다.
# Close the Session when we're done.
sess.close()
```

연산에 쓰인 시스템 자원을 돌려보내려면 `session`을 닫아야 합니다. 시스템 자원을 더 쉽게 관리하려면 `with` 구문을 쓰면 됩니다. 각 `Session`에 컨텍스트 매니저(역자 주: 파이썬의 요소 중 하나로 주로 'with' 구문에서 쓰임)가 있어서 'with' 구문 블락의 끝에서 자동으로 '`close()`'가 호출됩니다.

```
with tf.Session() as sess:
    result = sess.run([product])
    print(result)
```

TensorFlow의 구현 코드(TensorFlow implementation)를 통해 graph에 정의된 내용이 실행 가능한 작업들(operation)로 변환되고 CPU나 GPU같이 이용 가능한 연산 자원들에 뿌려집니다. 코드로 어느 CPU 혹은 GPU를 사용할지 명시적으로 지정할 필요는 없습니다. 작업을 가능한 한 많이

처리하기 위해 TensorFlow는 (컴퓨터가 GPU를 가지고 있다면) 첫 번째 GPU를 이용하니까요.

만약 컴퓨터에 복수의 GPU가 있어서 이를 사용하려면, op을 어느 하드웨어에 할당할지 명시적으로 밝혀야 합니다. 작업에 사용할 CPU 혹은 GPU를 지정하려면 `with...Device` 구문을 사용하면 됩니다.

```
with tf.Session() as sess:
    with tf.device("/gpu:1"):
        matrix1 = tf.constant([[3., 3.]])
        matrix2 = tf.constant([[2.], [2.]])
        product = tf.matmul(matrix1, matrix2)
    ...
```

이용할 CPU 혹은 GPU는 문자열로 지정할 수 있습니다. 현재 지원되는 것은 아래와 같습니다.

- `"/cpu:0"` : 컴퓨터의 CPU.
- `"/gpu:0"` : 컴퓨터의 1번째 GPU.
- `"/gpu:1"` : 컴퓨터의 2번째 GPU.

GPU와 TensorFlow에 대한 더 자세한 정보는 [Using GPUs](#)를 참조하시기 바랍니다.

분산된 session에서 graph 실행하기(Launching the graph in a distributed session)

TensorFlow 클러스터를 만들기 위해 클러스터에 포함된 각 머신에 TensorFlow 서버를 실행시켜 봅시다. Session을 클라이언트에 인스턴스화할 때는 Session을 클러스터 머신의 네트워크 위치로 넘겨야 합니다.:

```
with tf.Session("grpc://example.org:2222") as sess:
    # sess.run(...)을 호출하면 클러스터에서 실행될 것입니다.
    # Calls to sess.run(...) will be executed on the cluster.
    ...
```

세션을 받은 머신은 해당 Session의 마스터가 됩니다. 머신 내에서는 Tensorflow의 구현 코드(implementation)가 머신 내 연산 자원에게 작업을 나눠주지만, 클러스터에서는 마스터가 클러스터 내의 다른 머신들에게 graph를 분배하는 것입니다.

"`with tf.device():`" 구문을 이용해서 특정 머신에게 직접 graph의 특정 부분을 지정해 줄 수도 있습니다.

```
with tf.device("/job:ps/task:0"):
    weights = tf.Variable(...)
    biases = tf.Variable(...)
```

Session 분산과 클러스터에 대한 더 자세한 정보는 [Distributed TensorFlow How To](#)를 참조하시기 바랍니다.

인터렉티브한 이용법(Interactive Usage)

이 문서에 있는 파이썬 예제들은 `Session` 을 실행시키고 `Session.run()` 메서드를 이용해서 graph의 작업들을 처리한다.

IPython 같은 인터렉티브 파이썬 환경에서의 이용편의성을 위해 `InteractiveSession` 클래스와 `Tensor.eval()`, `Operation.run()` 메서드를 대신 이용할 수도 있다. `session` 내에서 변수를 계속 유지할 필요가 없기 때문이다.

```
# 인터렉티브 TensorFlow Session을 시작해봅시다.
# Enter an interactive TensorFlow Session.
import tensorflow as tf
sess = tf.InteractiveSession()

x = tf.Variable([1.0, 2.0])
a = tf.constant([3.0, 3.0])

# 초기화 op의 run() 메서드를 이용해서 'x'를 초기화합니다.
# Initialize 'x' using the run() method of its initializer op.
x.initializer.run()

# 'x'에서 'a'를 빼는 작업을 추가하고 실행시켜서 결과를 봅시다.
# Add an op to subtract 'a' from 'x'. Run it and print the result
sub = tf.subtract(x, a)
print(sub.eval())
# ==> [-2. -1.]

# 실행을 마치면 Session을 닫읍시다.
# Close the Session when we're done.
sess.close()
```

Tensors

TensorFlow 프로그램은 모든 데이터를 tensor 데이터 구조로 나타냅니다. 연산 graph에 있는 작업들(op) 간에는 tensor만 주고받을 수 있기 때문입니다. TensorFlow의 tensor를 n 차원의 배열이나 리스트라고 봐도 좋습니다. tensor는 정적인 타입(static type), 차원(rank 역자 주: 예를 들어 1차원, 2차원하는 차원), 형태(shape, 역자 주: 예를 들어 2차원이면 $m \times n$) 값을 가집니다. TensorFlow가 어떻게 이 개념들을 다루는지 알고 싶으시면 [Rank, Shape, and Type](#)을 참조하시기 바랍니다.

Variables

그래프를 실행하더라도 변수(variable)의 상태는 유지됩니다. 아래에서 간단한 카운터 예제를 통해 변수에 대해서 알 수 있습니다. 더 자세한 내용은 [Variables](#)를 참조하시기 바랍니다.

```
# 값이 0인 스칼라로 초기화된 변수를 만듭니다.
# Create a Variable, that will be initialized to the scalar value 0.
state = tf.Variable(0, name="counter")

# 'state'에 1을 더하는 작업(op)을 만듭니다.
# Create an Op to add one to `state`.

one = tf.constant(1)
new_value = tf.add(state, one)
update = tf.assign(state, new_value)

# 그래프를 한 번 작동시킨 후에는 'init' 작업(op)을 실행해서 변수를 초기화해야
# 합니다. 먼저 'init' 작업(op)을 추가해 봅시다.
# Variables must be initialized by running an `init` Op after having
# launched the graph. We first have to add the `init` Op to the graph.
init_op = tf.global_variables_initializer()

# graph와 작업(op)들을 실행시킵니다.
# Launch the graph and run the ops.
with tf.Session() as sess:
    # 'init' 작업(op)을 실행합니다.
    # Run the 'init' op
    sess.run(init_op)
    # 'state'의 시작값을 출력합니다.
    # Print the initial value of 'state'
    print(sess.run(state))
    # 'state' 값을 업데이트하고 출력하는 작업(op)을 실행합니다.
    # Run the op that updates 'state' and print 'state'.
    for _ in range(3):
        sess.run(update)
        print(sess.run(state))

# output:
# 0
# 1
# 2
# 3
```

이 코드에서 `assign()` 작업은 `add()` 작업처럼 `graph`의 한 부분입니다. 그래서 `run()`이 `graph`를 실행시킬 때까지 실제로 작동하지 않습니다.

우리는 보통 통계 모델의 파라미터를 변수로 표현합니다. 예를 들어 뉴럴 네트워크의 비중값을 변수인 `tensor`로 표현할 수 있습니다. 학습을 진행할 때 훈련용 `graph`를 반복해서 실행시키고 이 `tensor` 값을 업데이트하는 것입니다.

Fetches

작업의 결과를 가져오기 위해 `Session` 오브젝트에서 `run()` 을 호출해서 `graph`를 실행하고 `tensor`로 결과값을 끌어냅니다. 앞의 예제에서는 'state' 하나의 노드만 가져왔지만 복수의 `tensor`를 받아올 수도 있습니다.:

```
input1 = tf.constant([3.0])
input2 = tf.constant([2.0])
input3 = tf.constant([5.0])
intermed = tf.add(input2, input3)
mul = tf.multiply(input1, intermed)

with tf.Session() as sess:
    result = sess.run([mul, intermed])
    print(result)

# output:
# [array([ 21.], dtype=float32), array([ 7.], dtype=float32)]
```

여러 `tensor`들의 값을 계산해내기 위해 수행되는 작업(`op`)들은 각 `tensor` 별로 각각 수행 되는 것 이 아니라 전체적으로 한 번만 수행됩니다.

Feeds

위의 예제에서 살펴본 `graph`에서 `tensor`들은 상수(Constant) 와 변수(Variable) 로 저장되었습니다. TensorFlow에서는 `graph`의 연산에게 직접 `tensor` 값을 줄 수 있는 'feed 메커니즘'도 제공합니다.

`feed` 값은 일시적으로 연산의 출력값을 입력한 `tensor` 값으로 대체합니다. `feed` 데이터는 `run()` 으로 전달되어서 `run()` 의 변수로만 사용됩니다. 가장 일반적인 사용방법은 `tf.placeholder()`를 사용해서 특정 작업(`op`)을 "feed" 작업으로 지정해 주는 것입니다.

```
input1 = tf.placeholder(tf.float32)
input2 = tf.placeholder(tf.float32)
output = input1 * input2

with tf.Session() as sess:
    print(sess.run([output], feed_dict={input1:[7.], input2:[2.]}))

# output:
# [array([ 14.], dtype=float32)]
```

만약 `feed` 를 제대로 제공하지 않으면 `placeholder()` 연산은 에러를 출력할 것입니다. `feed`를 이용하는 다른 예시는 [MNIST fully-connected feed tutorial\(source code\)](#)를 참조하시기 바랍니다.

튜토리얼

기본 뉴럴 네트워크

MNIST 초급

만약 기계학습을 처음 배우신다면, 여기서부터 출발하기를 권장드립니다. MNIST라는 손으로 쓰여진 숫자를 분류하는 전통적인 문제에 대해 배워, 다중 분류에 관한 가벼운 입문을 할 수 있습니다.

[튜토리얼 보기](#)

MNIST 고급

만약 이미 다른 딥러닝 소프트웨어 패키지와 MNIST에 익숙하시다면, 이 튜토리얼이 텐서플로우에 관한 매우 간단한 시작점이 될 것입니다.

[튜토리얼 보기](#)

텐서플로우 구조

규모 있는 모델을 학습시키기 위해 텐서플로우의 인프라를 조금 더 자세하게 알아보고자 할 때의 기술적인 튜토리얼입니다. MNIST를 예시로 이용합니다.

[튜토리얼 보기](#)

MNIST 데이터 다운로드

MNIST 숫자 데이터셋을 다운로드하는 것에 대한 자세한 사항입니다. 꽤 흥미로운 자료입니다.

[튜토리얼 보기](#)

tf.contrib.learn을 사용한 간편한 머신러닝

tf.contrib.learn 시작하기

텐서플로우의 고수준 API인 tf.contrib.learn의 간략한 입문입니다. 단 몇 줄의 코드로 신경망을 만들고, 훈련하고, 평가합니다.

[튜토리얼 보기](#)

tf.contrib.learn 선형모델 소개

텐서플로우에서 선형 모델을 작업하기 위해서, tf.contrib.learn의 풍부한 도구에 대해 소개합니다.

[튜토리얼 보기](#)

선형모델 튜토리얼

이 튜토리얼은 tf.contrib.learn을 이용해 선형 모델을 만드는 코드를 살펴볼 수 있습니다.

[튜토리얼 보기](#)

와이드앤 딥러닝 튜토리얼

이 튜토리얼은 각 모델의 장점을 결합하기 위해 선형 모델과 딥 신경망을 tf.contrib.learn을 이용해 동시에 학습시키는 방법을 보여줍니다.

[튜토리얼 보기](#)

텐서플로우 서빙

텐서플로우 서빙

생산적인 환경 마련을 위한, 기계학습 모델을 제공하는 유연하고 우수한 시스템인 텐서플로우 서빙을 소개합니다.

[튜토리얼 보기](#)

이미지 프로세싱

콘볼루션 뉴럴 네트워크

CIFAR-10 데이터셋을 이용한 콘볼루션 신경망에 관한 소개입니다. 시각적인 자료에 관해 더 함축적이고 효과적인 표현을 산출하기 위해 변환 불변성을 이용하기 때문에, 콘볼루션 신경망은 특별히 이미지 처리에 맞게 설계되어 있습니다.

[튜토리얼 보기](#)

이미지 인식

ImageNet Challenge 데이터와 라벨 데이터셋으로 훈련된 콘볼루션 신경망을 이용해 물체를 인식하는 법을 배웁니다.

[튜토리얼 보기](#)

Deep Dream 시각 환상

인셉션 인식 모델을 구축한, Deep Dream 신경망 시각 환상 소프트웨어의 텐서플로우 버전을 배포합니다.

[튜토리얼 보기](#)

언어와 시퀀스 프로세싱

word2vec 모델

이 튜토리얼은 단어를 벡터로 표현하는 방법(워드 임베딩)을 배우는 게 어째서 유용한지에 관해 여러분의 흥미를 자극할 것입니다. 임베딩을 학습하기 위한 효과적인 모델로 word2vec 모델을 소개합니다. 또한, 노이즈-대조 학습 방법을 지지하는 높은 수준의 디테일도 다룰 것입니다.

[튜토리얼 보기](#)

리커런트 뉴럴 네트워크

영어 문장에서 다음 단어를 예측하기 위해 LSTM 네트워크를 학습하는 RNN에 관한 소개입니다. (언어 모델링이라고도 부르는 작업입니다.)

[튜토리얼 보기](#)

seq2seq 모델

RNN 튜토리얼에 이어, 기계 번역을 위해 시퀀스-시퀀스 모델을 결합합니다. 전체적으로 머신이 학습해, 자신 만의 영어-프랑스어 번역기를 만드는 것을 배웁니다.

[튜토리얼 보기](#)

SyntaxNet

텐서플로우를 위한 자연언어처리 프레임워크인 SyntaxNet을 소개합니다.

[튜토리얼 보기](#)

비머신러닝 애플리케이션

만델브로트

텐서플로우는 기계학습과 관련이 없는 연산 작업에도 활용될 수 있습니다. 여기서는 만델브로트 데이터셋을 시각화하는 나이브한 구현을 보여줍니다.

[튜토리얼 보기](#)

편미분 방정식

기계학습과 관련이 없는 또 다른 예시로, 연못에 떨어지는 빗방울에 관한 나이브한 PDF 시뮬레이션의 예시를 제공합니다.

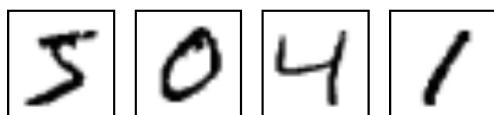
[튜토리얼 보기](#)

MNIST 초급

이 튜토리얼은 머신러닝과 텐서플로우를 처음 접해본 독자들을 위해서 구성되어 있습니다. 만약 *MNIST*가 무엇인지 알고 소프트맥스(다변량 로지스틱) 회귀가 무엇인지 이미 알고 계신다면, [더 깊이 있는 튜토리얼](#)을 선호하실 것이라고 생각합니다. 튜토리얼을 진행하시기 전에, 자신의 머신에 텐서플로우가 설치되어 있는지 확인해주세요.

프로그래밍을 어떻게 하는지 배울 때, 언제나 "Hello World."를 가장 먼저 출력해본다는 전통이 있습니다. 그것처럼, 머신러닝에선 MNIST를 가장 먼저 다룹니다.

MNIST는 간단한 컴퓨터 비전 데이터셋입니다. 이는 다음과 같이 손으로 쓰여진 이미지로 구성되어 있습니다.



또한, 이것은 그 숫자가 어떤 숫자인지 알려주는 각 이미지에 관한 라벨을 포함하고 있습니다. 예를 들어서, 위의 이미지의 경우에 라벨은 5, 0, 4, 그리고 1입니다.

이 튜토리얼에서는 모델이 이미지를 보고 어떤 숫자인지 예측하는 모델을 훈련시킬 것입니다. 우리의 목적은 가장 최신의 성능을 발휘하는 정교한 시스템을 만드는 것이 아닙니다.(물론 뒤에 가서 하게될 것입니다!) 대신에, 여기서는 텐서플로우에 발을 살짝만 담궈봅시다. 우리는 소프트맥스 회귀라고 불리는 아주 간단한 모델로 시작할 것입니다.

이 튜토리얼에 쓰이는 실제 코드는 매우 짧고, 흥미로운 일들은 단지 세 줄 안에 모두 일어납니다. 그러나 우리에게는 그 세 줄 안에 담겨있는 텐서플로우의 동작 원리와 머신러닝의 핵심 개념을 아는 것이 중요합니다. 때문에, 우리는 코드를 하나하나 주의 깊게 다룰 것입니다.

MNIST 데이터셋

MNIST 데이터셋은 [Yann LeCun의 웹사이트](#)에 호스팅되어 있습니다. 좀 편하게 하기 위해서, 우리는 이 데이터를 다운로드받고 설치하는 파이썬 코드를 넣어놨습니다. [여기](#)에서 코드를 다운받은 후 아래와 같이 코드를 불러올 수도 있습니다. 아니면 그냥 복사하고 붙여넣기를 하십시오.

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

다운로드된 데이터는 55,000개의 학습 데이터(`mnist.train`), 10,000개의 테스트 데이터 (`mnist.test`), 그리고 5,000개의 검증 데이터(`mnist.validation`) 이렇게 세 부분으로 나뉩니다. 데이터가 이렇게 나뉜다는 것은 매우 중요합니다. 왜냐하면 우리가 학습시키지 않는 데이터를 통

해, 우리가 학습한 것이 정말로 일반화되었다고 확신할 수 있기 때문입니다!

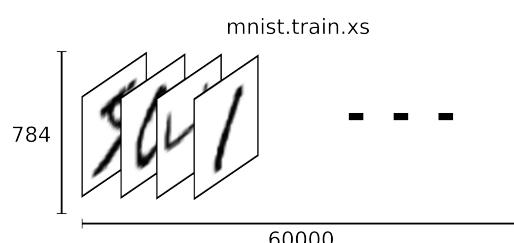
앞서 언급했듯이, 각 MNIST 데이터셋은 두 부분으로 나뉩니다. 손으로 쓴 숫자와 그에 따른 라벨입니다. 우리는 이미지를 "xs"라고 부르고 라벨을 "ys"라고 부를 것입니다. 학습 데이터셋과 테스트 데이터셋은 둘 다 xs와 ys를 가집니다. 예를 들어, 학습 이미지는 `mnist.train.images`이며, 학습 라벨은 `mnist.train.labels`입니다.

각 이미지는 28x28 픽셀입니다. 우리는 이를 숫자의 큰 배열로 해석할 수 있습니다.

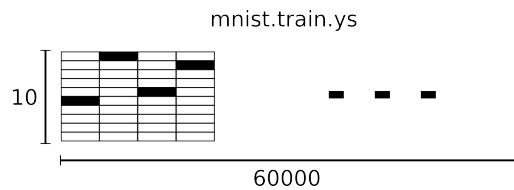
우리는 이 배열을 펼쳐서 $28 \times 28 = 784$ 개의 벡터로 만들 수 있습니다. 이미지들 간에 일관적으로 처리하기만 한다면, 배열을 어떻게 펼치든지 상관없습니다. 이러한 관점에서, MNIST 이미지는 [매우 호화스러운 구조](#)(주의 : 연산을 많이 요하는 시각화입니다)를 가진, 단지 784차원 벡터 공간에 있는 여러 개의 데이터일 뿐입니다.

데이터를 펼치면 이미지의 2D 구조에 대한 정보가 완벽하게 사라집니다. 그게 나빠보이지 않나요? 음, 가장 좋은 컴퓨터 비전의 방법론은 2D 구조를 쓰고, 나중의 튜토리얼에서 다룹니다. 하지만 여기서 사용하는, 간단한 소프트맥스 회귀는 그렇지 않습니다.

데이터를 펼친 결과로 `mnist.train.images` 는 [55000, 784] 의 형태를 가진 텐서(n차원 배열)가 됩니다. 첫 번째 차원은 이미지를 가리키며, 두 번째 차원은 각 이미지의 픽셀을 가르킵니다. 텐서의 모든 성분은 특정 이미지의 특정 픽셀을 특정하는 0과 1사이의 픽셀값입니다.



MNIST에서 각각에 대응하는 라벨은 0과 9사이의 숫자이며, 각 이미지가 어떤 숫자인지를 말해줍니다. 이 튜토리얼의 목적을 위해서 우리는 라벨을 "원-핫 벡터"로 바꾸길 원합니다. 원-핫 벡터는 단 하나의 차원에서만 1이고, 나머지 차원에서는 0인 벡터입니다. 이 경우, n번째 숫자는 n번째 차원이 1인 벡터로 표현될 것입니다. 예를 들어서, 3은 `[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]` 입니다. 결과적으로, `mnist.train.labels` 는 `[55000, 10]` 의 모양을 같은 실수 배열이 됩니다.(역자 주 : 정수 배열이 아니라, 실수 배열로 취급하는 데에는 이후 소프트맥스 회귀의 결과가 정수형이 아닌 실수형으로 산출되기 때문입니다.)



그럼 이제 실제로 우리의 모델을 만들 준비가 되었습니다.

소프트맥스 회귀 (softmax regression)

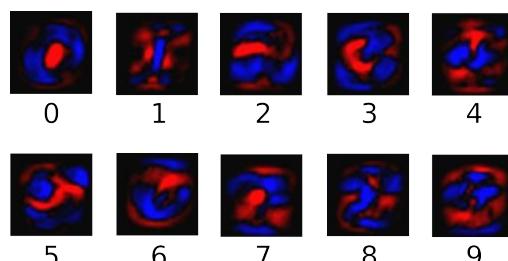
우리는 MNIST의 각 이미지가 0부터 9 사이의 손으로 쓴 숫자라는 것을 알고 있습니다. 따라서 각 이미지는 10가지의 경우의 수 중 하나에 해당하겠지요. 우리는 이제 이미지를 보고 그 이미지가 각 숫자일 확률을 계산할 것입니다. 예를 들면, 우리가 만드는 모델은 9가 쓰여져 있는 이미지를 보고 이 이미지가 80%의 확률로 9라고 추측하지만, (윗쪽의 동그란 부분 때문에) 8일 확률도 5% 있다고 계산할 수도 있습니다. 또한 그 외의 다른 숫자일 확률도 조금씩 있을 수 있습니다. 확실하지 않으니까요.

이 상황은 소프트맥스 회귀를 사용하기에 아주 적절한 예입니다. 만약 당신이 어떤 것이 서로 다른 여러 항목 중 하나일 확률을 계산하고자 한다면, 소프트맥스가 딱 맞습니다. 소프트맥스는 각 값이 0과 1 사이의 값으로 이루어지고, 각 값을 모두 합하면 1이 되는 목록을 제공하기 때문입니다. 게다가 나중에 더 복잡한 모델을 트레이닝 할 때에도, 마지막 단계는 소프트맥스 레이어가 될 것입니다.

소프트맥스 회귀는 두 단계로 이루어집니다. 우선 입력한 데이터가 각 클래스에 속한다는 증거 (evidence)를 수치적으로 계산하고, 그 뒤엔 계산한 값을 확률로 변환하는 것입니다.

한 이미지가 특정 클래스에 속하는지 계산하기 위해서는 각 픽셀의 어두운 정도(intensity)를 가중치합(서로 다른 계수를 곱해 합하는 계산, weighted sum)을 합니다. 여기서 가중치는 해당 픽셀이 진하다는 것이 특정 클래스에 속한다는 것에 반하는 내용이라면 음(-)의 값을, 특정 클래스에 속한다는 것을 의미한다면 양(+)의 값을 가지게 됩니다.

아래 그림은 모델이 각 클래스에 대해 학습한 가중치를 나타내고 있습니다. 빨간 부분은 음의 가중치를, 파란 부분은 양의 가중치를 나타냅니다.



또한 여기서 바이어스(bias)라는 추가적인 항을 더하게 됩니다. 결과값의 일부는 입력된 데이터와는 독립적일 수 있다는 것을 고려하기 위함입니다. 이를 수식으로 표현하면, 입력값 $\langle x \rangle$ 가 주어졌을 때 클래스 $\langle i \rangle$ 에 대한 증거값은 다음과 같습니다:

$$\text{evidence}_i = \sum_j W_{i,j} x_j + b_i$$

여기서 (W_i) 는 가중치이며 (b_i) 는 클래스 i 에 대한 바이어스이고, (j) 는 입력 데이터로 사용한 이미지 (x) 의 픽셀 값을 합하기 위한 인덱스입니다. 이제 각 클래스에 대해 계산한 증거값들을 "소프트맥스" 함수를 활용해 예측 확률 (y) 로 변환합니다:

$$y = \text{softmax}(\text{evidence})$$

여기서 소프트맥스는 우리가 계산한 선형 함수를 우리가 원하는 형태 - 이 경우에는 10가지 경우에 대한 확률 분포 - 로 변환하는데 사용하는 "활성화" 또는 "링크" 함수의 역할을 합니다. 이번 예에서는 계산한 증거값들을 입력된 데이터 값이 각 클래스에 속할 확률로 변환하는 것이라고 생각하셔도 됩니다. 이 과정은 다음과 같이 정의됩니다:

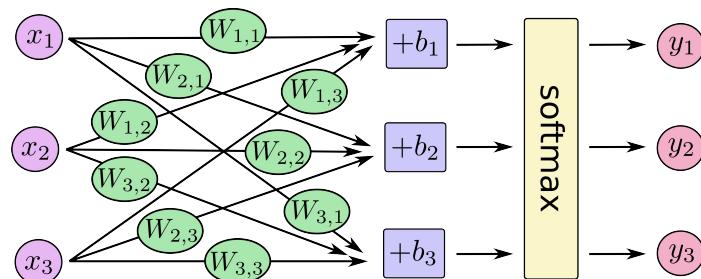
$$\text{softmax}(x) = \text{normalize}(\exp(x))$$

이 식을 전개하면 다음의 식을 얻습니다:

$$\text{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

많은 경우, 일단 소프트맥스를 입력값을 지수화한 뒤 정규화 하는 과정이라고 생각하는게 편합니다. 지수화란 증거값을 하나 더 추가하면 어떤 가설에 대해 주어진 가중치를 곱으로 증가시키는 것을 의미합니다. 또한 반대로, 증거값의 갯수가 하나 줄어든다는 것은 가설의 가중치가 기존 가중치의 분수비로 줄어들게 된다는 뜻입니다. 어떤 가설도 0 또는 음의 가중치를 가질 수 없습니다. 그런 뒤 소프트맥스는 가중치를 정규화한 후, 모두 합하면 1이 되는 확률 분포로 만듭니다. (소프트맥스 함수에 대해 더 알고싶다면, 상호작용이 가능한 시각화가 잘 되어 있는 마이클 닐슨의 책의 이 [챕터](#)를 참조하세요.)

소프트맥스 회귀는 다음 그림 같은 형태를 갖게 됩니다 (실제론 훨씬 (x) 가 훨씬 많지만요). 각각의 출력값에 대해, 가중치합을 계산하고 바이어스를 더한 뒤 소프트맥스를 적용하는 것입니다.



이를 수식으로 표현하면 다음과 같습니다:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax} \begin{pmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{pmatrix}$$

우리는 이 과정을 행렬곱과 벡터합으로 변경하여 "벡터화"할 수 있습니다. 벡터화는 계산의 효율화에 도움이 됩니다 (또한 머리로 생각하기에도 좋은 방법입니다).

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax} \left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

더 간략하게는 다음과 같이 표현할 수 있습니다:

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

회귀 구현하기

파이썬에서 효율적인 수치 연산을 하기 위해, 우리는 다른 언어로 구현된 보다 효율이 높은 코드를 사용하여 행렬곱 같은 무거운 연산을 수행하는 NumPy등의 라이브러리를 자주 사용합니다. 그러나 아쉽게도, 매 연산마다 파이썬으로 다시 돌아오는 과정에서 여전히 많은 오버헤드가 발생할 수 있습니다. 이러한 오버헤드는 GPU에서 연산을 하거나 분산 처리 환경같은, 데이터 전송에 큰 비용이 발생할 수 있는 상황에서 특히 문제가 될 수 있습니다.

텐서플로우 역시 파이썬 외부에서 무거운 작업들을 수행하지만, 텐서플로우는 이런 오버헤드를 피하기 위해 한 단계 더 나아간 방식을 활용합니다. 파이썬에서 하나의 무거운 작업을 독립적으로 실행하는 대신, 텐서플로우는 서로 상호작용하는 연산간의 그래프를 유저가 기술하도록 하고, 그 연산 모두가 파이썬 밖에서 동작합니다 (이러한 접근 방법은 다른 몇몇 머신러닝 라이브러리에서 볼 수 있습니다).

텐서플로우를 사용하기 위해서는 이를 임포트해야 합니다.

```
import tensorflow as tf
```

우리는 이 상호작용하는 연산들을 심볼릭 변수를 활용해 기술하게 됩니다. 하나 만들어 보죠:

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

`x`에 특정한 값이 주어진 것은 아닙니다. 이는 'placeholder'로, 우리가 텐서플로우에서 연산을 실행할 때 값을 입력할 자리입니다. 여기서는 784차원의 벡터로 변형된 MNIST 이미지의 데이터를 넣으려고 합니다. 우린 이걸 `[None, 784]`의 형태를 갖고 부동소수점으로 이루어진 2차원 텐서로 표현합니다. (여기서 `None`은 해당 차원의 길이가 어떤 길이든지 될 수 있음을 의미합니다)

또한 우리의 모델에는 가중치와 바이어스 역시 필요합니다. 우리는 이를 부가적인 입력처럼 다루는 방법을 생각할 수도 있지만, 텐서플로우는 `Variable`이라고 불리는 보다 나은 방법을 갖고 있습니다. `Variable`은 서로 상호작용하는 연산으로 이루어진 텐서플로우 그래프 안에 존재하는, 수정 가능한 텐서입니다. `Variable`은 연산에 사용되기도 하고, 연산을 통해 수정되기도 합니다. 머신러닝에 이를 사용할 때에는 주로 모델의 변수를 `Variable` 들로 사용하게 됩니다.

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

우리는 `tf.variable`에 `variable`의 초기값을 넘겨줌으로써 이 `variable`들을 생성합니다: 여기서는 `w`와 `b` 둘 다 0으로 이루어진 텐서로 초기화를 합니다. 이제부터 `w`와 `b`를 학습해 나갈 것이므로, 각각의 초기값은 크게 중요하지 않습니다.

`w` 가 [784, 10]의 형태를 갖는 것에 주목해주시기 바랍니다. 이러한 형태로 만든 이유는 `w`에 784차원의 이미지 벡터를 곱해서 각 클래스에 대한 증거값을 나타내는 10차원 벡터를 얻고자 하기 때문입니다. `b` 는 그 10차원 벡터에 더하기 위해 [10]의 형태를 갖는 것입니다.

이제 우리는 모델을 구현할 수 있습니다. 단 한줄로요!

```
y = tf.nn.softmax(tf.matmul(x, w) + b)
```

우선, `tf.matmul(x, w)` 로 `x` 와 `w` 를 곱합니다. 이 표현은 위에서 본 수식에서 곱했던 순서인 (Wx) 와 반대인데 (행렬이므로 순서가 중요하죠), `x` 가 여러 입력값을 갖는 2차원 텐서인 경우에도 대응하기 위한 작은 트릭입니다. 그 다음엔 `b` 를 더하고, 마지막으로 `tf.nn.softmax` 을 적용합니다.

됐습니다. 우리 모델을 세팅하기 위해 단 한줄의 코드만을 사용했습니다. 간단한 몇 줄 짜리 준비 작업을 한 뒤에요. 이렇게 간단하게 할 수 있는 건 텐서플로우에서 소프트맥스 회귀가 특히 구현하기 쉽기 때문이 아닙니다. 텐서플로우는 머신러닝 모델에서부터 물리학 시뮬레이션까지 다양한 종류의 수치 연산을 표현할 수 있는 매우 유연한 방법이기 때문입니다. 게다가 한 번 작성한 모델은 여러 기기에서 실행할 수 있습니다: 당신의 컴퓨터에 있는 CPU, GPU, 심지어 휴대폰에서까요!

학습

우리의 모델을 학습시키기 위해서는 우선 모델이 좋다는 것은 어떤 것인지를 정의해야 합니다. 사실 머신러닝에서는 모델이 안좋다는 것이 어떤 의미인지를 주로 정의합니다. 우리는 이를 주로 비용(cost) 또는 손실(loss)이라고 부르며, 이것들은 우리의 모델이 우리가 원하는 결과에서 얼마나 떨어져있는지를 보여주는 값입니다. 우리는 그 격차를 줄이기 위해 노력하며, 그 격차가 적으면 적을수록 우리의 모델은 좋다고 말합니다.

모델의 손실을 정의하기 위해 자주 사용되는 좋은 함수 중 하나로 "크로스 엔트로피"가 있습니다. 원래 크로스 엔트로피는 정보 이론 분야에서 정보를 압축하는 방법으로써 고안된 것이지만, 현재는 도박에서 머신러닝에 이르기까지 여러 분야에서 중요한 아이디어로 사용되고 있습니다. 크로스 엔트로피는 다음과 같이 정의됩니다:

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

\hat{y} 는 우리가 예측한 확률 분포이며, y 는 실제 분포(우리가 입력하는 원-핫 벡터)입니다. 대략적으로 설명하자면, 크로스 엔트로피는 우리의 예측이 실제 값을 설명하기에 얼마나 비효율적인지를 측정하는 것입니다. 크로스 엔트로피에 대해서 더 자세하게 다루는 것은 이 튜토리얼의 범위를 벗어나는 내용입니다만, [알아둘 가치는 있습니다.](#)

크로스 엔트로피를 구현하기 위해서는 올바른 답을 넣기 위한 새로운 placeholder를 추가하는 것부터 시작해야 합니다.

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

이제 우리는 크로스 엔트로피 $-\sum y \log(\hat{y})$ 를 구현할 수 있습니다:

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

우선, `tf.log` 는 `y`의 각 원소의 로그 값을 계산합니다. 그 다음, `y_`의 각 원소를 `tf.log(y)`의 해당하는 원소들과 곱합니다. 그리고 `tf.reduce_sum` 으로 `y`의 2번째 차원(`reduction_indices=[1]`이라는 파라미터가 주어졌으므로)의 원소들을 합합니다. 마지막으로, `tf.reduce_mean` 으로 배치(batch)의 모든 예시에 대한 평균을 계산합니다.

(수학적으로 불안정한 계산이기 때문에, 소스 코드에서는 이 연산을 사용하지 않고 있는 것에 주의하시기 바랍니다. 대신, 정규화 되지 않은 로짓(logit)에 대해

`tf.nn.softmax_cross_entropy_with_logits` 을 적용합니다(즉, `tf.matmul(x, w) + b` 에 `softmax_cross_entropy_with_logits` 을 사용합니다). 이렇게 하는 이유는 이 수학적으로 보다 안정적인 함수가 내부적으로 소프트맥스 활성을 계산하기 때문입니다. 당신의 코드에서도 `tf.nn.(sparse_)softmax_cross_entropy_with_logits`를 사용하는 것을 고려해보시기 바랍니다.)

우리의 모델이 할 일을 우리가 알고있다면, 이를 텐서플로우를 통해 학습시키는 것은 매우 간단합니다. 텐서플로우는 당신이 하고자 하는 연산의 전체 그래프를 알고 있으므로, 손실(당신이 최소화하고 싶어하는 것이죠)에 당신이 설정한 변수들이 어떻게 영향을 주는지를 [역전파\(backpropagation\)](#) 알고리즘을 자동으로 사용하여 매우 효율적으로 정의할 수 있기 때문입니다. 그리고나서 텐서플로우는 당신이 선택한 최적화 알고리즘을 적용하여 변수를 수정하고 손실을 줄일 수 있습니다.

```
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

여기서는 텐서플로우에게 학습 비율 0.5로 경사 하강법(gradient descent algorithm)을 적용하여 크로스 엔트로피를 최소화하도록 지시합니다. 경사하강법이란 텐서플로우가 각각의 변수를 비용을 줄이는 방향으로 조금씩 이동시키는 매우 단순한 방법입니다. 그러나 텐서플로우는 [다른 여러 최적화 알고리즘](#)을 제공합니다: 그 중 하나를 적용하는 것은 코드 한 줄만 수정하면 될 정도로 간단합니다.

여기서 텐서플로우가 실제로 뒤에서 하는 일은, 역전파와 경사하강이라는 새로운 작업을 당신의 그래프에 추가하는 것입니다. 이제 텐서플로우가 실행되면 비용을 감소시키기 위해 변수들을 살짝 수정하는 경사 하강 학습 작업 한 번을 돌려줄 것입니다.

이제 우리 모델은 학습할 준비가 되었습니다. 학습을 실행시키기 전에 마지막으로, 우리가 작성한 변수들을 초기화하는 작업을 추가해야 합니다:

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

이제 `Session`에서 모델을 실행시키고, 변수들을 초기화하는 작업을 실행시킬 수 있습니다:

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

학습을 시킵시다 -- 여기선 학습을 1000번 시킬 겁니다!

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

반복되는 루프의 각 단계마다, 우리는 학습 데이터셋에서 무작위로 선택된 100개의 데이터로 구성된 "배치(batch)"를 가져옵니다. 그 다음엔 `placeholder`의 자리에 데이터를 넣을 수 있도록 `train_step`을 실행하여 배치 데이터를 넘깁니다.

무작위 데이터의 작은 배치를 사용하는 방법을 확률적 학습(stochastic training)이라고 부릅니다 -- 여기서는 확률적 경사 하강법입니다. 이상적으로는 학습의 매 단계마다 전체 데이터를 사용하고 싶지만(그렇게 하는게 우리가 지금 어떻게 하는게 좋을지에 대해 더 잘 알려줄 것이므로), 그렇게 하면 작업이 무거워집니다. 따라서 그 대신에 매번 서로 다른 부분집합을 사용하는 것입니다. 이렇게 하면 작업 내용은 가벼워지지만 전체 데이터를 쓸 때의 이점은 거의 다 얻을 수 있기 때문입니다.

모델 평가하기

우리가 작성한 모델은 성능이 어느 정도일까요?

흐음, 우선 모델이 라벨을 올바르게 예측했는지 확인해봅시다. `tf.argmax` 는 텐서 안에서 특정 축을 따라 가장 큰 값의 인덱스를 찾기에 매우 유용한 함수입니다. 예를 들면, `tf.argmax(y, 1)` 는 우리의 모델이 생각하기에 각 데이터에 가장 적합하다고 판단한(가장 증거값이 큰) 라벨이며, `tf.argmax(y_, 1)` 는 실제 라벨입니다. 우리는 `tf.equal` 을 사용하여 우리의 예측이 맞았는지 확인할 수 있습니다.

```
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
```

이렇게 하면 부울 값으로 이루어진 리스트를 얻게 됩니다. 얼마나 많이 맞았는지 판단하려면, 이 값을 부동소수점 값으로 변환한 후 평균을 계산하면 됩니다. 예를 들면, [True, False, True, True] 는 [1, 0, 1, 1] 로 환산할 수 있고, 이 값의 평균을 계산하면 0.75 가 됩니다.

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

마지막으로, 우리의 테스트 데이터를 대상으로 정확도를 계산해 봅시다.

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

결과는 약 92% 정도가 나올 것입니다.

좋은 결과일까요? 글쎄요, 딱히 그렇진 않습니다. 사실, 매우 안 좋은 결과입니다. 왜냐하면 우리가 매우 단순한 모델을 사용했기 때문입니다. 약간만 바꾸면, 97%의 정확도를 얻을 수 있습니다. 가장 좋은 모델은 정확도가 99.7%도 넘을 수 있지요! (더 알고 싶으시다면 다음의 [결과 목록](#)을 확인해보세요)

여기서 중요한 것은 우리가 이 모델을 통해 배운 것입니다. 혹시 아직도 이 결과가 조금 실망스러우시면 [다음 튜토리얼](#)을 읽어보시기 바랍니다. 거기선 우리가 훨씬 더 좋은 결과값도 얻고, 텐서플로 우로 더 복잡한 모델을 작성하는 방법도 배우게 된답니다!

MNIST 고급

TensorFlow는 큰 규모의 수치 계산에 적합한 강력한 라이브러리입니다. TensorFlow가 강력한 힘을 발휘하는 작업 중 하나는, 심층 신경망을 구성하고 학습시키는 것입니다. 이 튜토리얼에서는 MNIST 데이터를 분류하는 심층 합성곱(convolutional) 신경망을 구성하면서, TensorFlow에서 신경망 모델을 구성하는 기본 블록에 대해 알아볼 것입니다.

이 튜토리얼은 인공 신경망과 *MNIST* 데이터셋에 익숙한 독자를 위해 구성되어 있습니다. 만약 이들에 익숙하지 않다면, [MNIST 초급](#) 튜토리얼이 도움이 될 것입니다. 진행하기 전, [Tensorflow가 설치되어 있는지 확인해 주세요](#).

설정

모델을 생성하기 전, 먼저 MNIST 데이터셋을 불러오고, TensorFlow 세션을 시작할 것입니다.

MNIST 데이터셋 불러오기

편의를 위해서, 자동으로 MNIST 데이터셋을 다운받은 뒤 불러오는 [스크립트](#)가 준비되어 있습니다. 아래와 같이 해당 스크립트를 import 하여 실행하면, 현재 디렉토리 하위에 '`MNIST_data`' 폴더를 생성하여 자동으로 데이터 파일을 저장할 것입니다.

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

위에서 `mnist` 는 훈련(training), 테스트(testing) 그리고 검증(validation) 데이터를 NumPy 배열로 저장하는 클래스입니다. 아래에서 사용될 미니배치(minibatch)를 추출하는 함수 또한 이 클래스 안에 포함되어 있습니다.

TensorFlow InteractiveSession 시작하기

TensorFlow는 계산을 위해 고효율의 C++ 백엔드(backend)를 사용합니다. 이 백엔드와의 연결을 위해 TensorFlow는 세션(session)을 사용합니다. 일반적으로 TensorFlow 프로그램은 먼저 그래프를 구성하고, 그 이후 그래프를 세션을 통해 실행하는 방식을 따릅니다.

여기서는 대신 TensorFlow 코드를 보다 유연하게 작성할 수 있게 해 주는 `InteractiveSession` 클래스를 사용할 것입니다. 이 클래스는 [계산 그래프](#)(computation graph)를 구성하는 작업과 그 그래프를 실행하는 작업을 분리시켜 줍니다. 즉, `InteractiveSession` 을 쓰지 않는다면, 세션을 시작하여 [그래프를 실행](#)하기 전에 이미 전체 계산 그래프가 구성되어 있어야 하는 것입니다.

```
import tensorflow as tf
sess = tf.InteractiveSession()
```

계산 그래프 (Computational Graph)

Python에서 효율적인 수치 계산을 하기 위해서, 주로 NumPy와 같이 Python 외부에서 다른 언어로 된 고효율의 코드를 통해 행렬 곱셈과 같은 고비용의(expensive) 연산을 수행하는 라이브러리를 이용합니다. 불행히도, 이렇게 하면 연산 결과를 일일이 Python으로 다시 불러들이는 데 많은 오버헤드가 발생합니다. 특히 계산 과정을 여러 GPU에 분산시키는 경우, 데이터를 이동시키는 데 드는 비용이 매우 커지게 됩니다.

TensorFlow도 마찬가지로 고비용의 연산은 Python 외부에서 실행합니다. 하지만, 위와 같은 오버헤드 문제를 피하기 위해 현명한 방법을 활용합니다. 각각의 고비용 연산을 Python에서 독립적으로 실행하는 대신, TensorFlow는 상호작용하는 연산을 그래프로 묶어 그 전체를 Python 바깥에서 실행시키는 방법을 사용합니다. Theano나 Torch와 같은 라이브러리에서 활용되는 방법과 비슷합니다.

따라서 Python에서 작성하는 코드의 역할은, 이러한 외부의 계산 그래프를 구성하고, 이 계산 그래프의 어떤 부분이 실행되어야 하는지 지시하는 것입니다. 자세한 내용은 [계산 그래프 및 기본 사용법](#)을 참고하세요.

소프트맥스 회귀 모델 구성

이 절에서는 단일 계층의 소프트맥스 회귀 모델(softmax regression model)을 구성할 것입니다. 그리고 다음 절에서 이를 확장시켜, 다중 계층의 합성곱 신경망(convolutional network)을 구성할 것입니다.

플레이스홀더 (Placeholder)

계산 그래프를 구성하기 위해, 먼저 입력될 이미지와 각각의 출력 클래스에 해당하는 노드를 생성할 것입니다.

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

위 코드에서 `x` 와 `y_` 에 특정한 값이 부여된 것은 아닙니다. 그들은 나중에 TensorFlow가 계산을 실행할 때 값을 넣어 줄 자리인 `placeholder`입니다.

입력될 이미지를 `x` 는 부동 소수점 실수(float) 값들의 2D 텐서입니다. 위 코드에서 `shape`에 `[None, 784]` 를 넣어 주었는데, 여기서 `784` 는 28×28 의 크기를 가지는 MNIST 이미지를 한 줄로 펼친 크기에 해당합니다. 배치(batch)의 크기에 해당하는 첫 번째 차원 크기의 `None` 은 크기를 여

기서 정하지 않는다(어떤 배치 크기라도 가능하다)는 것을 의미합니다. 출력 클래스인 `y_` 또한 2D 텐서입니다. 각 열은 해당하는 MNIST 이미지의 숫자 클래스를 10차원 one-hot 벡터로 나타냅니다.

`tf.placeholder`에 `shape` 매개변수가 필수는 아닙니다. 하지만, 이를 명시해 줌으로써 TensorFlow가 잘못된 텐서 구조(shape)에 따른 오류를 자동으로 잡아낼 수 있게 됩니다.

변수 (Variable)

이제 모델에 사용할 가중치(weight) `w` 와 편향(bias) `b` 를 정의합니다. 이들을 추가적인 입력으로 대할 수도 있겠지만, TensorFlow는 이러한 변수들을 다루기 위해 `variable` 을 제공합니다.

`variable` 이란 TensorFlow의 계산 그래프 안에 있는 값입니다. 이들은 계산에 사용될 수 있을 뿐만 아니라, 계산에 의해 변경될 수도 있습니다. 따라서 머신 러닝에 활용되는 모델 매개변수는 주로 `variable` 들로 구성됩니다.

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

`tf.Variable` 을 사용할 때에는 변수의 초기 값을 지정해 주어야 합니다. 위의 경우, `w` 와 `b` 모두 0으로만 구성된 텐서로 초기화됩니다. `w` 는 784x10 행렬(입력 이미지 벡터의 크기가 784, 출력 숫자 클래스가 10개)이며, `b` 는 10차원 벡터입니다.

`variable` 들은 세션이 시작되기 전에 초기화되어야 합니다. 아래 코드는 모든 `variable` 들 각각에 대해 미리 지정된 초기 값을(위에서 지정된 0으로만 구성된 텐서)를 넣어 주는 역할을 합니다.

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

클래스 예측 및 비용 함수(Cost Function)

이제 회귀 모델을 도입할 수 있습니다. 한 줄만으로요! 벡터화된 입력 이미지인 `x` 를 가중치 행렬인 `w` 와 곱하고, 여기에 편향 `b` 를 더한 뒤, 각각의 클래스에 대한 소프트맥스 함수의 결과를 계산하면 됩니다.

```
y = tf.nn.softmax(tf.matmul(x, w) + b)
```

모델 훈련 과정에서 최소화될 비용 함수(cost function) 또한 간단하게 도입할 수 있습니다. 여기서 사용될 비용 함수는 실제 클래스와 모델의 예측 결과 간 크로스 엔트로피(cross-entropy) 함수입니다.

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

여기서 `tf.reduce_sum` 은 모든 클래스에 대해 결과를 합하는 함수, `tf.reduce_mean` 은 사용된 이미지를 각각에서 계산된 합의 평균을 구하는 함수입니다.

모델 훈련시키기

이제 모델과 훈련의 비용 함수가 정의되었으니, TensorFlow로 모델을 훈련시키는 일만 남았습니다. TensorFlow에 전체 계산 그래프의 정보가 입력되어 있으므로, 라이브러리가 자동으로 미분을 통해 각각의 변수에 대한 비용 함수의 기울기(gradient)를 계산합니다. TensorFlow는 다양한 [내장된 최적화 알고리즘](#)을 가지고 있습니다. 여기서는 아래 코드와 같이 학습 속도 0.5의 경사 하강법(steepest gradient descent) 알고리즘을 사용하여 크로스 엔트로피를 최소화할 것입니다.

```
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

위의 코드 한 줄에서 TensorFlow가 실제로 하는 것은 계산 그래프에 기울기를 계산하고, 얼마나 매개변수를 변경해야 할지 계산하고, 매개변수를 변경하는 새로운 계산들을 추가하는 것입니다.

반환된 `train_step` 은 실행되었을 때 경사 하강법을 통해 각각의 매개변수를 변화시키게 됩니다. 따라서, 모델을 훈련시키려면 이 `train_step` 을 반복해서 실행하면 됩니다.

```
for i in range(1000):
    batch = mnist.train.next_batch(50)
    train_step.run(feed_dict={x: batch[0], y_: batch[1]})
```

각각의 훈련 단계(iteration)에서, 50개의 훈련 샘플이 추출됩니다. 그리고 `train_step` 을 실행하며 `feed_dict` 를 통해 `placeholder` 텐서인 `x` 와 `y_` 에 훈련 샘플을 넣어줍니다. 참고로, `feed_dict` 는 `placeholder` 외에도 계산 그래프 안의 어떤 텐서라도 변경할 수 있습니다.

모델 평가하기

이렇게 훈련된 모델은 얼마나 정확할까요?

먼저, 모델이 정확한 레이블을 예측했는지 확인해 볼 것입니다. `tf.argmax` 함수는 텐서의 한 차원을 따라 가장 큰 값의 인덱스를 반환합니다. 예로, `tf.argmax(y, 1)` 은 모델이 입력을 받고 가장 그럴듯하다고 생각한 레이블이고, `tf.argmax(y_, 1)` 은 실제 레이블입니다. 이제 `tf.equal` 함수를 사용해 두 레이블이 일치하는지 다음과 같이 확인할 수 있습니다.

```
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
```

위의 코드는 불리언으로 이루어진 리스트를 반환합니다. 전체에서 얼마나 맞았는지를 확인하려면, 불리언을 부동 소수점 실수로 형변환하여 리스트의 평균을 구하면 됩니다. 예로, 결과가 `[True, False, True, True]` 였다면 이는 형변환을 통해 `[1, 0, 1, 1]` 이 되고, 평균인 `0.75` 가 예측 결과

의 정확도가 됩니다.

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

이제 아래와 같이 `feed_dict`로 `mnist.test`를 전달하여 테스트 데이터셋에 대한 예측 정확도를 확인할 수 있습니다. 대략 92% 정도의 정확도가 얻어질 것입니다.

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

다중 계층 합성곱 신경망

MNIST 데이터에서 91% 정확도를 얻는 것은 그다지 좋은 결과라고 할 수 없습니다. 그래서 이번 장에서는, 정확도를 높이기 위해 합성곱 신경망(convolutional neural network)이라는 약간 복잡한 모형을 사용할 것입니다. 이를 통해 99.2% 정도의 정확도를 얻을 수 있습니다. 최신 결과에는 미치지 못하지만, 어느 정도 그럴듯한 결과입니다.

가중치 초기화

합성곱 신경망 모델을 구성하기 위해서는 많은 수의 가중치와 편향을 사용하게 됩니다. 대칭성을 깨뜨리고 기울기(gradient)가 0이 되는 것을 방지하기 위해, 가중치에 약간의 잡음을 주어 초기화 합니다. 또한, 모델에 ReLU 뉴런이 포함되므로, "죽은 뉴런"을 방지하기 위해 편향을 작은 양수 (0.1)로 초기화합니다. 매번 모델을 만들 때마다 반복하는 대신, 아래 코드와 같이 이러한 일을 해주는 함수 두 개를 생성합니다.

```
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

합성곱(Convolution)과 풀링(Pooling)

TensorFlow는 합성곱과 풀링 계층(layer)을 유연하게 다룰 수 있도록 해 줍니다. 경계의 패딩(padding)과 스트라이드(stride)에 대해 다양한 선택을 할 수 있습니다. 이번 예시에서는 스트라이드를 1로, 출력 크기가 입력과 같게 되도록 0으로 패딩하도록 설정합니다. 풀링은 2x2 크기의 맥스 풀링을 적용합니다. 마찬가지로 코드를 간단히 하기 위해 합성곱과 풀링을 위한 함수를 아래 코드와 같이 생성합니다.

```

def conv2d(x, w):
    return tf.nn.conv2d(x, w, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                          strides=[1, 2, 2, 1], padding='SAME')

```

첫 번째 합성곱 계층

이제 첫 번째 계층을 만들 것입니다. 이는 합성곱 계층과 맥스 풀링 계층으로 구성됩니다. 합성곱 계층에서는 5×5 의 윈도우(patch라고도 함) 크기를 가지는 32개의 필터를 사용하며, 따라서 구조(shape)가 `[5, 5, 1, 32]`인 가중치 텐서를 정의해야 합니다. 처음 두 개의 차원은 윈도우의 크기, 세 번째는 입력 채널의 수, 마지막은 출력 채널의 수(즉, 얼마나 많은 특징을 사용할 것인가)를 나타냅니다. 또한, 각각의 출력 채널에 대한 편향을 정의해야 합니다. 이 과정에서 앞에서 만든 함수를 사용합니다.

```

w_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])

```

이 계층에 이미지를 입력하려면 먼저 `x` 를 4D 텐서로 `reshape` 해야 합니다. 두 번째와 세 번째 차원은 이미지의 가로와 세로 길이, 그리고 마지막 차원은 컬러 채널의 수를 나타냅니다.

```

x_image = tf.reshape(x, [-1, 28, 28, 1])

```

이제 `x_image` 와 가중치 텐서에 합성곱을 적용하고, 편향을 더한 뒤 ReLU 함수를 적용합니다. 출력 값을 구하기 위해 마지막으로 맥스 풀링을 적용합니다.

```

h_conv1 = tf.nn.relu(conv2d(x_image, w_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

```

두 번째 합성곱 계층

심층 신경망을 구성하기 위해서, 앞에서 만든 것과 비슷한 계층을 쌓아올릴 수 있습니다. 여기서는 두 번째 합성곱 계층이 5×5 윈도우에 64개의 필터를 가집니다.

```

w_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, w_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)

```

완전 연결 계층 (Fully-Connected Layer)

두 번째 계층을 거친 뒤 이미지 크기는 7×7 로 줄어들었습니다. 이제 여기에 1024개의 뉴런으로 연결되는 완전 연결 계층을 구성합니다. 이를 위해서 7×7 이미지의 배열을 `reshape` 해야 하며, 완전 연결 계층에 맞는 가중치 행렬과 편향 행렬을 구성합니다. 최종적으로 완전 연결 계층의 끝에 `ReLU` 함수를 적용합니다.

```
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

드롭아웃 (Dropout)

오버피팅(overfitting) 되는 것을 방지하기 위해, [드롭아웃](#)을 적용할 것입니다. 뉴런이 드롭아웃되지 않을 확률을 저장하는 `placeholder`를 만듭니다. 이렇게 하면 나중에 드롭아웃이 훈련 과정에는 적용되고, 테스트 과정에서는 적용되지 않도록 설정할 수 있습니다. TensorFlow의 `tf.nn.dropout` 함수는 뉴런의 출력을 자동으로 스케일링(scaling)하므로, 추가로 스케일링 할 필요 없이 그냥 드롭아웃을 적용할 수 있습니다.¹

```
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

최종 소프트맥스 계층

마지막으로, 위에서 단일 계층 소프트맥스 회귀 모델을 구성할 때와 비슷하게 아래 코드와 같이 소프트맥스 계층을 추가합니다.

```
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_conv = tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
```

모델의 훈련 및 평가

이렇게 훈련된 모델은 얼마나 정확할까요?

훈련 및 평가 또한 위의 단일 계층 모델과 거의 같습니다. 차이가 있다면, 이번에는 경사 하강법 알고리즘 대신 더 복잡한 ADAM 최적화 알고리즘을 사용합니다. 또한, 드롭아웃 확률을 설정하는 추가 변수인 `keep_prob` 을 `feed_dict` 인수를 통해 전달합니다. 아래의 코드는 훈련 과정에서 100회 반복 시마다 로그를 작성합니다.

```

cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y_conv), reduction_indices=[1]))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
sess.run(tf.global_variables_initializer())
for i in range(20000):
    batch = mnist.train.next_batch(50)
    if i%100 == 0:
        train_accuracy = accuracy.eval(feed_dict={
            x:batch[0], y_: batch[1], keep_prob: 1.0})
        print("step %d, training accuracy %g"%(i, train_accuracy))
    train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

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

```

코드를 실행시켜서 얻은 최종 정확도는 약 99.2%가 될 것입니다.

이렇게 하여 TensorFlow를 이용해 쉽고 빠르게 '어느 정도 복잡한 딥 러닝 모델'을 구성하고, 훈련시키고, 평가하는 과정을 배워 보았습니다.

1: 드롭아웃은 오버피팅을 줄이는 데 매우 효과적이지만, 이번에 다룬 작은 합성곱 신경망에 대해서는, 성능이 드롭아웃을 적용한 경우와 하지 않은 경우가 비슷합니다. 드롭아웃은 큰 신경망을 훈련시킬 때 유용합니다. ↪

TensorFlow 메커니즘 기초

코드: [tensorflow/examples/tutorials/mnist/](#)

이 튜토리얼의 목표는 TensorFlow를 사용해 어떻게 트레이닝 하는지 그리고 전형적인 MNIST 데이터 셋을 사용해 손으로 쓴 숫자를 구별하는 간단한 feed-forward neural network를 평가하는지 보여주는 것이다. 이 튜토리얼 대상 독자는 TensorFlow 사용에 관심이 있는 머신러닝 유경험자다.

이 튜토리얼은 일반적인 머신러닝 교육에 적합하지 않다.

반드시 [TensorFlow 설치](#) 지시를 따랐는지 확인하라.

튜토리얼 파일

이 튜토리얼은 아래와 같은 파일들을 참조한다:

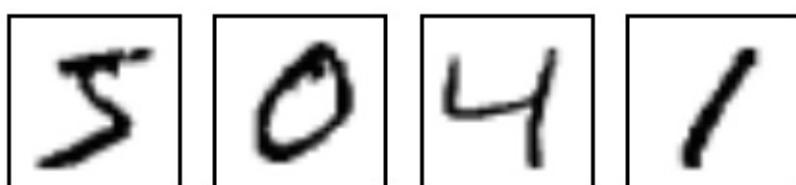
파일	목적
mnist.py	이 코드는 완전히 연결된 MNIST 모델을 구축한다.
fully_connected_feed.py	메인 코드는 feed dictionary를 사용해 다운로드 한 데이터 셋에 대해 구축된 MNIST모델을 트레이닝 한다.

트레이닝을 시작하기 위해 직접 `fully_connected_feed.py` 파일을 간단히 실행해 보라:

```
python fully_connected_feed.py
```

데이터 준비

MNIST는 머신 러닝에서 고전적인 문제다. 이 문제는 그레이 스케일(greyscale)인 손으로 쓴 숫자 28x28 픽셀 이미지를 보고 그 이미지가 표현하는 숫자가 0부터 9 까지 숫자 중 어떤 것인지 판단하는 것이다.



더 많은 정보는 [Yann LeCun's MNIST page](#) 또는 [Chris Olah's visualizations of MNIST](#) 참고하라.

다운로드

`run_training()` 메소드의 맨 위에는, `input_data.read_data_sets()` 함수가 당신의 트레이닝 폴더에 올바른 데이터가 다운되었는지 확인하고, `DataSet` 인스턴스의 딕셔너리에 반환하기 위해 그 데이터의 압축을 해제한다.

```
data_sets = input_data.read_data_sets(FLAGS.train_dir, FLAGS.fake_data)
```

주의: `fake_data` flag는 유닛 테스트의 목적으로 쓰이며 무시해도 이상이 없다.

데이터 �셋	목적
<code>data_sets.train</code>	기본 트레이닝을 위한 55000개의 이미지와 레이블.
<code>data_sets.validation</code>	트레이닝 정확도를 반복해서 검증하기 위한 5000개의 이미지와 레이블.
<code>data_sets.test</code>	트레이닝된 정확도를 마지막으로 테스트하기 위한 10000개의 이미지와 레이블.

데이터에 대한 더 많은 정보는 [Download tutorial](#)을 읽어 보세요.

입력과 플레이스 홀더(Placeholders)

`placeholder_inputs()` 함수는 두개의 `tf.placeholder` ops를 생성한다. 이 ops는 `batch_size`를 포함해, 남은 그래프를 위한 입력 형태와 실제 트레이닝 example의 입력 형태를 정의한다.

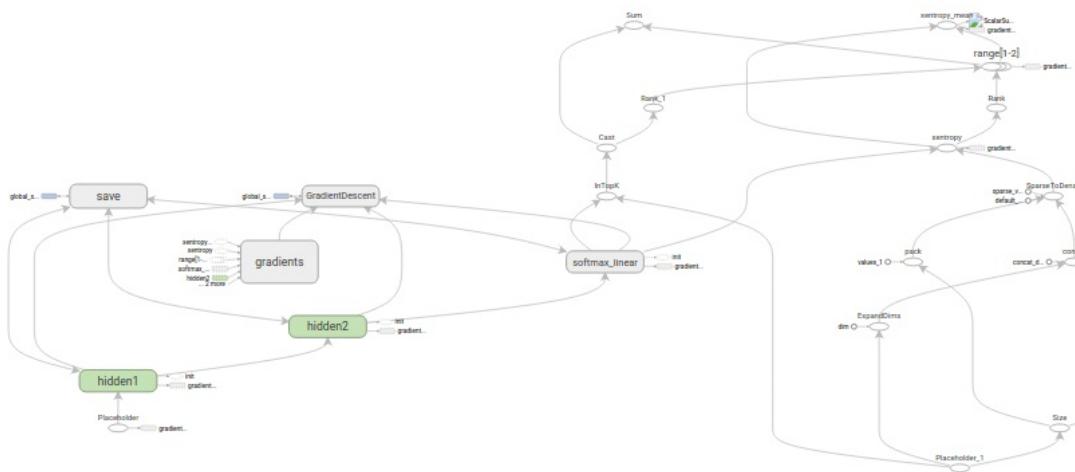
```
images_placeholder = tf.placeholder(tf.float32, shape=(batch_size,
                                                       mnist.IMAGE_PIXELS))
labels_placeholder = tf.placeholder(tf.int32, shape=(batch_size))
```

트레이닝 반복 루프 더 아래 부분에서, 전체 이미지와 레이블 데이터셋이 각 순서에서 `batch_size`에 맞게 나누어지고 이 플레이스 홀더 ops들과 매치된다. 그리고 나서 `feed_dict` 변수를 사용해 `sess.run()` 함수에 전달된다.

Build the Graph

데이터를 위한 플레이스 홀더를 생성한 후에, 3-스테이지 패턴(3-stage pattern): `inference()`, `loss()`, `training()` 을 따라서 `mnist.py` 파일로부터 그래프가 생성됩니다.

1. `inference()` - 예측을 위해 network forward 실행에 필요한 수준의 그래프를 작성한다.
2. `loss()` - `inference` 그래프에 `loss`를 생성하기 위해 필요한 ops를 더한다.
3. `training()` - `loss` 그래프에 계산과 그라디언트(gradients)를 적용하기 위한 op를 더한다.



Inference

`inference()` 함수는 그래프를 작성하는데, 이 그래프는 예측한 출력을 가지는 `tensor`를 반환하는데 필요한 정도까지 작성된다.

이것은 이미지 플레이스 홀더를 입력으로 취하고 그 위에 출력 `logits`를 지정한 10 노드 선형 층(ten node linear layer)을 동반하는 `ReLU` activation을 가진 한 쌍의 완전 연결 층(fully connected layer)을 만든다.

각 층은 고유한 `tf.name_scope` 아래에서 생성된다. 이것은 해당 범위(scope) 안에서 생성된 것에게 접두어와 같은 기능을 한다.

```
with tf.name_scope('hidden1'):
```

정의된 범위 내, `weights`와 `biases`의 층을 요구되는 형태로 `tf.Variable` 인스턴스 안에서 생성해 사용한다:

```
weights = tf.Variable(
    tf.truncated_normal([IMAGE_PIXELS, hidden1_units],
                        stddev=1.0 / math.sqrt(float(IMAGE_PIXELS))),
    name='weights')
biases = tf.Variable(tf.zeros([hidden1_units]),
                     name='biases')
```

예를 들어, 이것들이 `hidden1` 범위 내에서 생성될 때는 `weights` 변수에 부여된 고유한 이름은 "`hidden1/weights`"다.

각 변수에게 `initializer ops`가 생성자(construction)의 일부로서 주어져 있다.

보통의 경우에, `weights`는 `tf.truncated_normal`로 초기화 되고 2-D `tensor`의 형태가 된다. 첫 번째 `dim`(차원 dimension)은 `weights`가 연결해 나온 층의 유닛(units) 갯수이고 두 번째 `dim`은 `weights`가 연결한 층의 유닛 갯수이다. `hidden1`이라고 이름붙여진 첫 번째 레이어의 차원은

[IMAGE_PIXELS, hidden1_units] 다. 왜냐하면 weights가 이미지 입력과 hidden1 layer를 연결하고 있기 때문이다. `tf.truncated_normal_initializer`는 주어진 평균과 표준 편차를 가지고 임의의 분포를 생성한다.

그 후에 biases가 모두 0 값을 가지고 시작하도록 biases를 `tf.zeros`로 초기화한다. 그리고 그것의 형태는 단순히 연결된 층의 유닛 수가 된다.

그래프의 세가지 기본적인 ops -- 숨겨진 층(hidden layer) `tf.matmul` 을 감싸는 두개의 `tf.nn.relu` ops와 logits를 위한 추가 `tf.matmul` 하나 -- 가 분리된 `tf.Variable` 인스턴스와 함께 각각 차례대로 생성된다. 이 인스턴스는 각각의 입력 플레이스 홀더 또는 이전 레이어의 출력 tensor와 연결되어 있다.

```
hidden1 = tf.nn.relu(tf.matmul(images, weights) + biases)
```

```
hidden2 = tf.nn.relu(tf.matmul(hidden1, weights) + biases)
```

```
logits = tf.matmul(hidden2, weights) + biases
```

마지막으로, 출력을 가질 `logits` tensor가 반환된다.

Loss

`loss()` 함수는 필요한 loss ops를 더해 그래프를 더 발전시킨다.

첫 번째로, `labels_placeholder`에서 나온 값이 64비트 정수로 변환된다. 그 다음, `tf.nn.sparse_softmax_cross_entropy_with_logits` 가 `labels_placeholder`에서 1-hot label을 자동으로 생성하고 `inference()` 함수의 1-hot labels 출력 `logits`를 비교하기 위해 추가된다.

```
labels = tf.to_int64(labels)
cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
    logits, labels, name='xentropy')
```

그 후에 batch dimension(첫 번째 dimension)에 걸친 cross entropy 값을 총 손실(loss)로 구하기 위해 `tf.reduce_mean` 를 사용한다.

```
loss = tf.reduce_mean(cross_entropy, name='xentropy_mean')
```

그리고 loss 값을 가질 tensor가 반환된다.

주의: Cross-entropy는 무엇이 정말 참인지를 고려해 볼 때, neural network의 예측을 믿는 것이 얼마나 나쁜지를 설명하게 해 준 정보 이론에서 가져온 아이디어다. 더 많은 정보는 Visual Information Theory 블로그 포스트를 읽어 보라 (<http://colah.github.io/posts/2015-09-Visual-Information/>)

Training

`training()` 함수는 Gradient Descent를 통해 손실을 최소화하기 위해 필요한 작업을 추가한다.

첫째로, `loss()` 함수로부터 loss tensor를 가지고 `tf.scalar_summary`에 넘겨준다.

`tf.scalar_summary`는 `SummaryWriter`와 쓰일 때 이벤트 파일에 요약 값(summary values)을 생성하는 op다. 이 경우에, 이것은 요약이 기록될 때마다 손실 값의 스냅샷(snapshot)을 내보낸다.

```
tf.scalar_summary(loss.op.name, loss)
```

다음으로, 요청된 학습률에 gradients를 적용하는 `tf.train.GradientDescentOptimizer` 인스턴스를 생성한다.

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
```

그런 다음, 글로벌 트레이닝 단계(global training step)를 위한 카운터를 가진 변수 하나를 생성한다. `minimize()` op는 시스템 내에서 트레이닝 가능한 weights와 글로벌 단계의 진행을 업데이트 한다. 관례상, 이것은 `train_op`로 알려져 있다. 그리고 이것은 트레이닝의 전체적인 단계를 진행하기 위해 반드시 TensorFlow session에서 실행되어야 한다.(아래 확인)

```
global_step = tf.Variable(0, name='global_step', trainable=False)
train_op = optimizer.minimize(loss, global_step=global_step)
```

트레이닝 op의 출력을 가진 tensor가 반환된다.

Train the Model

일단 그래프가 작성되면, 반복해서 트레이닝 할 수 있고 반복 루프(loop)에서 실행할 수 있습니다. 반복 루프(loop)는 `fully_connected_feed.py`에 있는 유저 코드에 의해 컨트롤됩니다.

그래프

`run_training()` 함수의 상단에 python 명령어 `with`이 있다. 이 명령어는 만들어진 모든 ops가 default global `tf.Graph` 인스턴스와 관련이 있음을 나타낸다.

```
with tf.Graph().as_default():
```

`tf.Graph` 는 그룹으로 함께 실행되는 ops의 모임이다. 대부분의 TensorFlow 사용은 오직 하나의 기본 그래프에 의존해야 한다.

다수의 그래프로 더 복잡한 사용이 가능하지만 이 간단한 튜토리얼의 범위에 벗어난다.

세션(Session)

만들 준비가 모두 완료되고 필요한 모든 ops가 생성되었다면, 그래프를 실행하기 위해 `tf.Session` 을 만든다.

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

다른 방법으로, 범위 지정을 위한 `with` 블록에서 `Session` 을 생성할 수 있다:

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

세션에 빈 파라미터는 이 코드가 기본 로컬 세션에 연결될 것임(아직 로컬 세션이 생성되지 않았다면 생성할 것임)을 나타냅니다.

세션을 생성한 직후 `tf.Variable` 의 초기화 op에서 `sess.run()` 를 호출해 모든 `tf.Variable` 인스턴스가 초기화됩니다.

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

`sess.run()` 메소드는 파라미터로 전달된 op(s)에 대응하는 그래프의 완벽한 부분집합을 실행합니다. 첫 번째 경우에, `init` op는 변수들의 initializer만을 가지고 있는 `tf.group` 입니다. 그래프의 남은 부분 중 어떤 것도 여기서는 실행되지 않습니다. 그것은 아래의 트레이닝 반복 루프에서 일어납니다.

Train Loop

세션으로 변수들을 초기화 한 후, 트레이닝이 시작되었습니다.

사용자의 코드는 단계별로 트레이닝을 제어합니다. 쓸만한 트레이닝을 할 수 있는 간단한 루프:

```
for step in xrange(FLAGS.max_steps):
    sess.run(train_op)
```

그러나 이 튜토리얼은 이전에 만든 플레이스 홀더에 맞게 각 단계에서 입력 데이터를 다듬어야 하기 때문에 약간 복잡하다.

Feed the Graph

각 단계에서, 코드는 이 단계를 위한 트레이닝 예시 셋을 포함하고 플레이스 홀더 ops가 키값이 되는 feed 딕셔너리를 생성한다.

`fill_feed_dict()` 함수에서, 주어진 `dataSet` 은 다음 이미지와 레이블의 `batch_size` 셋을 검색한다. 그리고 다음 이미지와 레이블을 포함해 플레이스 홀더와 매칭된 `tensors`가 채워진다.

```
images_feed, labels_feed = data_set.next_batch(FLAGS.batch_size,
                                              FLAGS.fake_data)
```

그러면 플레이스 홀더를 키 값으로하고 feed tensors를 value 값으로 해 python 딕셔너리 객체가 생성된다.

```
feed_dict = {
    images_placeholder: images_feed,
    labels_placeholder: labels_feed,
}
```

이것은 이 단계의 트레이닝에 대한 입력 예시를 제공하기 위해 `sess.run()` 함수의 `feed_dict` 파라미터로 전달된다.

Check the Status

이 코드는 실행 호출에서 가져오기 위해 두 값을 지정한다: `[train_op, loss]`.

```
for step in xrange(FLAGS.max_steps):
    feed_dict = fill_feed_dict(data_sets.train,
                               images_placeholder,
                               labels_placeholder)
    _, loss_value = sess.run([train_op, loss],
                           feed_dict=feed_dict)
```

가져올 값이 두개기 때문에 `sess.run()` 는 두개의 아이템을 가진 튜플을 반환한다. 가져올 값의 리스트에 있는 각 `Tensor` 는 반환된 튜플에 있는 `numpy` 배열과 대응한다. 그리고 이 트레이닝 단계 동안 그 `tensor`의 값으로 채워진다. `train_op` 는 출력값이 없는 작업 이기 때문에 반환된 튜플에서 대응하는 요소는 `None` 이다. 그래서 버린다. 그러나 `loss tensor`의 값은 트레이닝 중에 모델이 나누어지면 `NaN`이 된다. 로그 기록을 위해 이 값을 캡쳐해 둔다.

트레이닝이 NaNs 없이 잘 실행되었다고 가정하면, 사용자가 트레이닝의 상태를 알게 하기 위해 트레이닝 루프가 매 100번째 단계마다 간단한 상태를 출력한다.

```
if step % 100 == 0:  
    print 'Step %d: loss = %.2f (%.3f sec)' % (step, loss_value, duration)
```

상태 시작화

TensorBoard에서 사용된 이벤트 파일을 내보내기 위해서, 그래프 작성 단계에서 모든 요약자료를 (이 경우에는 하나) 하나의 op에 모아야 한다.

```
summary_op = tf.merge_all_summaries()
```

세션이 만들어진 후에, 그래프와 요약 값을 포함한 이벤트 파일을 작성하기 위해

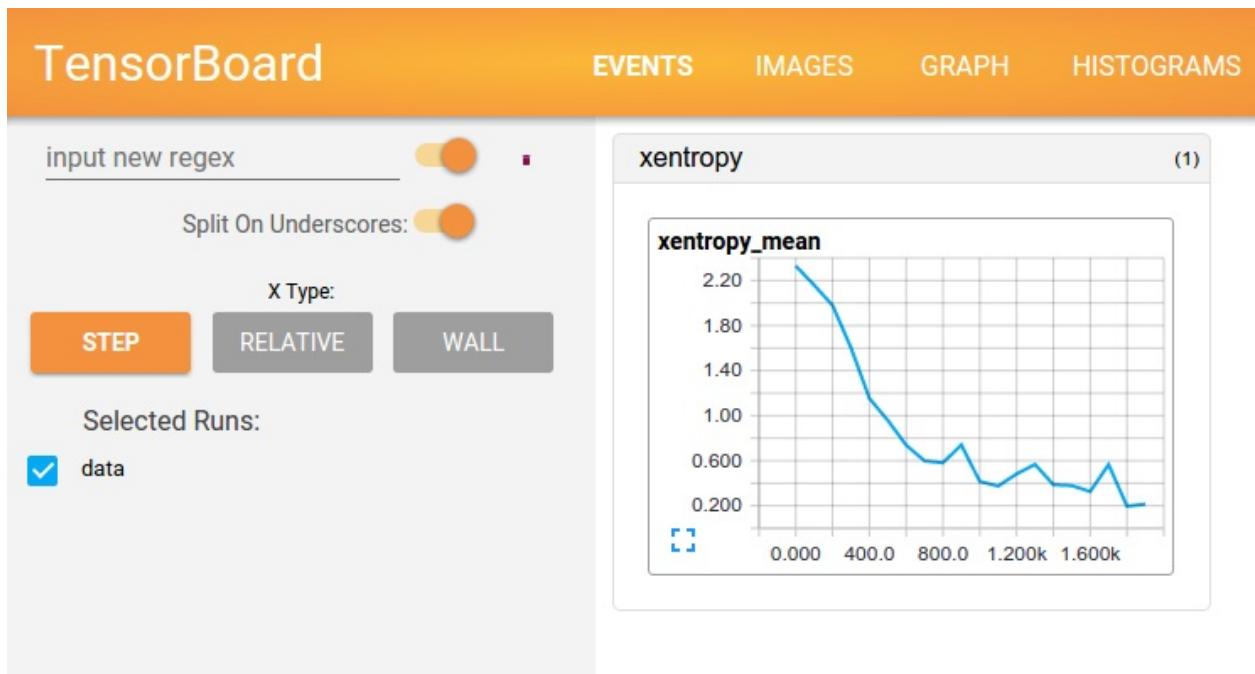
`tf.train.SummaryWriter` 인스턴스가 생성되었을 것이다.

```
summary_writer = tf.train.SummaryWriter(FLAGS.train_dir, sess.graph)
```

마지막으로, 이벤트 파일은 `summary_op` 가 실행되고 작성자의 `add_summary()` 함수에 출력이 전달될 때마다 새로운 요약 값으로 업데이트 된다.

```
summary_str = sess.run(summary_op, feed_dict=feed_dict)  
summary_writer.add_summary(summary_str, step)
```

이벤트 파일이 쓰여지면, 요약 값을 보여주기 위해 TensorBoard가 트레이닝 폴더에 대해 실행될 것이다.



주의: 어떻게 Tensorboard를 만들고 실행하는지에 대한 더 많은 정보는, 동봉된 튜토리얼을 보시기 바랍니다. [Tensorboard: 학습을 시각화하기](#).

Save a Checkpoint

나중에 추가적인 트레이닝이나 평가를 위해 모델을 복구하는데 쓰일 수 있는 checkpoint 파일을 내보내기 위해서, `tf.train.Saver` 인스턴트를 생성합니다.

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

트레이닝 루프에서, 모든 트레이닝 할 수 있는 변수들의 현재 값을 트레이닝 디렉토리에 있는 checkpoint 파일에 쓰기 위해 `saver.save()` 메소드를 정기적으로 호출한다.

```
saver.save(sess, FLAGS.train_dir, global_step=step)
```

미래에 나중에 생성된 몇개의 포인터에서, 모델 파라미터를 불러오기 위해 `saver.restore()` 메소드를 사용해 트레이닝을 재개할 수도 있다.

```
saver.restore(sess, FLAGS.train_dir)
```

Evaluate the Model

매 1000번째 단계마다, 코드는 트레이닝과 테스트 데이터셋에 대해 모델 평가를 시도한다. Training, validation, test dataset 을 위해 `do_eval()` 함수를 세 번 호출한다.

```

print 'Training Data Eval:'
do_eval(sess,
        eval_correct,
        images_placeholder,
        labels_placeholder,
        data_sets.train)
print 'Validation Data Eval:'
do_eval(sess,
        eval_correct,
        images_placeholder,
        labels_placeholder,
        data_sets.validation)
print 'Test Data Eval:'
do_eval(sess,
        eval_correct,
        images_placeholder,
        labels_placeholder,
        data_sets.test)

```

더 복잡한 데이터를 다룰 때는 일반적으로 매우 많은 양의 hyperparameter를 조절한 후, `data_sets.test` 만 체크한다. 그러나 간단한 MNIST 문제에 대해서는 모든 데이터에 대해 확인한다.

Build the Eval Graph

트레이닝 루프에 들어가기 전에, `loss()` 함수와 같은 logits/labels 파라미터로 `mnist.py`에서 `evaluation()` 함수를 호출해 Eval op를 생성했어야 한다.

```
eval_correct = mnist.evaluation(logits, labels_placeholder)
```

`evaluation()` 함수는 단순히 `tf.nn.in_top_k` op를 생성한다. 이 op는 자동적으로 참인 레이블이 K most-likely 예측에서 발견되면, 각 모델의 출력을 올바르다고 채점한다. 이 경우에 참인 레이블에 대해 예측이 옳았을 경우만 K의 값을 1로 설정합니다.

```
eval_correct = tf.nn.in_top_k(logits, labels, 1)
```

Eval Output

`feed_dict` 를 채우기 위해 루프를 만들수 있고 `eval_correct` op에 대해 `sess.run()` 를 호출해서 주어진 데이터셋의 모델을 평가할 수 있습니다.

```
for step in xrange(steps_per_epoch):
    feed_dict = fill_feed_dict(data_set,
                               images_placeholder,
                               labels_placeholder)
    true_count += sess.run(eval_correct, feed_dict=feed_dict)
```

`true_count` 변수는 간단히 `in_top_k` op가 옳다고 판단한 모든 예측들을 측정합니다.
그것을 간단히 예시의 총 갯수로 나누어 정확도를 계산합니다.

```
precision = true_count / num_examples
print(' Num examples: %d  Num correct: %d  Precision @ 1: %0.04f' %
      (num_examples, true_count, precision))
```

MNIST Data Download

코드: [tensorflow/examples/tutorials/mnist/](https://github.com/tensorflow/tutorials/blob/master/mnist/input_data.py)

이 튜토리얼의 목적은 (고전적인) MNIST 데이터를 활용한 필기 숫자의 분류(classification)를 위해 데이터를 어떻게 다운로드 받아야 하는지를 알려주는 것입니다.

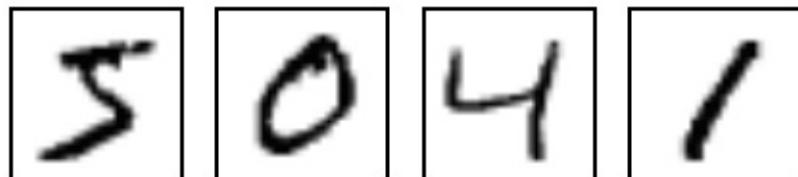
튜토리얼 파일

이 튜토리얼은 다음 파일을 참조합니다.

파일	목적
input_data.py	학습과 추정을 위한 MNIST 데이터셋을 다운로드하는 코드

데이터 준비

MNIST는 머신러닝의 고전적인 문제입니다. 이 문제는 필기 숫자들의 그레이스케일 28x28 픽셀 이미지를 보고, 0부터 9까지의 모든 숫자들에 대해 이미지가 어떤 숫자를 나타내는지 판별하는 것입니다.



좀 더 많은 정보를 원하시면, [Yann LeCun's MNIST page](http://yann.lecun.com/exdb/mnist/) 또는 [Chris Olah's visualizations of MNIST](http://chrisolah.github.io/2015/07/25/mnist-visualizations/)를 참고하면 됩니다.

Download

[Yann LeCun's MNIST page](http://yann.lecun.com/exdb/mnist/) 또한 다운로드를 위한 학습과 테스트 데이터를 호스팅하고 있습니다

파일	목적
<code>train-images-idx3-ubyte.gz</code>	학습 셋 이미지 - 55000개의 트레이닝 이미지, 5000개의 검증 이미지
<code>train-labels-idx1-ubyte.gz</code>	이미지와 매칭되는 학습 셋 레이블
<code>t10k-images-idx3-ubyte.gz</code>	테스트 셋 이미지 - 10000개의 이미지
<code>t10k-labels-idx1-ubyte.gz</code>	이미지와 매칭되는 테스트 셋 레이블

`input_data.py` 파일에서 `maybe_download()` 함수는 학습을 위한 파일들을 로컬 데이터 폴더에 넣을 수 있는지를 확인해줍니다.

폴더명은 `fully_connected_feed.py` 파일의 맨 위에 있는 플래그 변수에 의해 정해지며 원한다면 바꿀 수 있습니다.

풀기(Unpack, 언팩)와 변형(Reshape)

파일들 자체는 표준 이미지 포맷이 아니며 직접 `input_data.py` 에 있는 `extract_images()` 와 `extract_labels()` 함수를 사용하여 언패킹할 수 있습니다.

이미지 데이터는 `[image index, pixel index]` 형태의 이차원 텐서(여기선 2차원 배열을 의미함)로 추출될 수 있습니다. 각 엔트리는 특정 이미지에서 특정 픽셀의 휘도값이며, `[0, 255]`에서 `[0, 1]` 까지 재조정됩니다. "image index"는 데이터셋에 있는 이미지를 가리키며, 0부터 데이터셋의 크기까지 카운팅됩니다. 그리고 "pixel index"는 어떤 이미지에서의 특정 픽셀을 가리키며, 0부터 이미지에 존재하는 픽셀의 갯수까지 존재합니다.

`train-*` 파일들에 있는 60000개의 예시들은 학습을 위한 55000개의 예시들과 검증을 위한 5000개의 예시들로 나뉘어집니다. 데이터셋에 있는 모든 28x28 픽셀의 그레이스케일 이미지의 크기는 784이고 따라서 학습 셋 이미지를 위한 출력값 텐서는 `[55000, 784]` 의 형태가 됩니다.

레이블 데이터는 각 예시를 위한 클래스 식별자를 값으로써 가지며 `[image index]` 형태의 일차원 텐서로 추출될 수 있습니다. 학습 셋 레이블은 `[55000]` 의 형태가 될 것입니다.

데이터셋 객체

이 기본 코드는 다운로드와 압축풀기 그리고 다음의 데이터셋들을 위해 이미지와 레이블을 변형할 것입니다.

데이터셋	목적
data_sets.train	초기 학습을 위한 55000개의 이미지들과 레이블들
data_sets.validation	학습 정확도의 반복적 검증을 위한 5000개의 이미지와 레이블들
data_sets.test	학습 정확도의 마지막 테스팅을 위한 10000개의 이미지와 레이블들

`read_data_sets()` 함수는 각 세가지 데이터 셋을 위한 `DataSet` 인스턴스를 가진 딕셔너리를 리턴합니다. `DataSet.next_batch()` 메서드는 `batch_size` 개의 이미지 리스트와 레이블들로 이루어진 튜플을 실행중인 TensorFlow 세션에 넣기위해 사용될 수 있습니다.

```
images_feed, labels_feed = data_set.next_batch(FLAGS.batch_size)
```

tf.contrib.learn 시작하기

(v1.0)

텐서플로우의 고수준 머신러닝 API(tf.contrib.learn)는 다양한 머신러닝 모델을 쉽게 설정하고, 훈련하고, 평가할 수 있도록 해줍니다. 이 튜토리얼에서는 tf.contrib.learn 을 사용하여 [신경망](#) 분류기를 만들고, [Iris 데이터셋](#)에 있는 꽃받침과 꽃잎의 정보를 이용하여 꽃의 종류를 예측할 수 있도록 분류기를 훈련시킬 것입니다. 코드는 다음의 다섯 단계로 수행됩니다:

1. Iris 훈련/테스트 데이터를 담은 CSV 파일을 텐서플로우 `Dataset` 으로 불러옵니다
2. [신경망](#) 분류기를 만듭니다
3. 훈련 데이터를 이용하여 모델을 훈련 시킵니다
4. 모델의 정확도를 평가합니다
5. 새로운 표본을 분류합니다

참고: 이 튜토리얼을 시작하기 전에 [텐서플로우를 설치해야 합니다](#).

시작하기

다음은 이 신경망 분류기의 전체 코드입니다:

```

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import tensorflow as tf
import numpy as np

# 데이터셋
IRIS_TRAINING = "iris_training.csv"
IRIS_TEST = "iris_test.csv"

# 데이터셋을 불러옵니다.
training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
    filename=IRIS_TRAINING,
    target_dtype=np.int,
    features_dtype=np.float32)
test_set = tf.contrib.learn.datasets.base.load_csv_with_header(
    filename=IRIS_TEST,
    target_dtype=np.int,
    features_dtype=np.float32)

# 모든 특성이 실수값을 가지고 있다고 지정합니다
feature_columns = [tf.contrib.layers.real_valued_column("", dimension=4)]

# 10, 20, 10개의 유닛을 가진 3층 DNN를 만듭니다
classifier = tf.contrib.learn.DNNClassifier(feature_columns=feature_columns,
                                             hidden_units=[10, 20, 10],
                                             n_classes=3,
                                             model_dir="/tmp/iris_model")

# 모델을 학습시킵니다.
classifier.fit(x=training_set.data,
                y=training_set.target,
                steps=2000)

# 정확도를 평가합니다.
accuracy_score = classifier.evaluate(x=test_set.data,
                                       y=test_set.target)["accuracy"]
print('정확도: {:.f}'.format(accuracy_score))

# 새로운 두 개의 꽃 표본을 분류합니다.
new_samples = np.array(
    [[6.4, 3.2, 4.5, 1.5], [5.8, 3.1, 5.0, 1.7]], dtype=float)
y = list(classifier.predict(new_samples, as_iterable=True))
print ('예측: {}'.format(str(y)))

```

다음에서 이 코드를 자세하게 살펴 보겠습니다.

Iris CSV 데이터를 텐서플로우로 불러오기

Iris 데이터셋은 Iris 품종인 *Iris setosa*, *Iris virginica*, *Iris versicolor*가 각각 50개씩의 표본으로 구성된 150개의 행을 가진 데이터입니다.



왼쪽에서 오른쪽으로, *Iris setosa* ([Radomil](#), CC BY-SA 3.0), *Iris versicolor* ([Dlanglois](#), CC BY-SA 3.0), and *Iris virginica* ([Frank Mayfield](#), CC BY-SA 2.0).

각각의 행은 각 꽃의 표본에 대한 다음의 정보를 담고 있습니다 : 꽃받침 길이, 꽃받침 너비, 꽃잎 길이, 꽃잎 너비, 그리고 꽃의 종류. 꽃의 종류는 정수로 표현되어 있으며, 0은 *Iris setosa*, 1은 *Iris versicolor*, 그리고 2는 *Iris virginica*를 나타냅니다.

꽃받침 길이	꽃받침 너비	꽃잎 길이	꽃잎 너비	종
5.1	3.5	1.4	0.2	0
4.9	3.0	1.4	0.2	0
4.7	3.2	1.3	0.2	0
...
7.0	3.2	4.7	1.4	1
6.4	3.2	4.5	1.5	1
6.9	3.1	4.9	1.5	1
...
6.5	3.0	5.2	2.0	2
6.2	3.4	5.4	2.3	2
5.9	3.0	5.1	1.8	2

이 튜토리얼을 위해서 Iris data를 랜덤하게 섞은 후에, 두 개의 CSV 파일로 나누어 놓았습니다:

- 120개의 표본을 갖는 훈련 데이터([iris_training.csv](#))
- 30개의 표본을 갖는 테스트 데이터([iris_test.csv](#))

이 두 CSV 파일을 파이썬 코드와 같은 디렉토리에 놓습니다.

시작하려면, 먼저 텐서플로우와 numpy를 임포트합니다:

```

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import tensorflow as tf
import numpy as np

```

그 다음, `learn.datasets.base` 에 있는 `load_csv_with_header()` 함수를 이용하여 훈련 셋과 테스트 셋을 `Dataset` 으로 불러옵니다. `load_csv_with_header()` 함수는 세 개의 인자를 요구합니다:

- `filename` , CSV 파일의 경로
- `target_dtype` , 데이터셋에 있는 타깃 값의 `numpy` 데이터형
- `features_dtype` , 데이터셋에 있는 특성 값의 `numpy` 데이터형

여기에서 타깃 값(모델을 훈련시켜 예측하려고 하는 값)은 0–2의 정수로 구성된 꽃의 종입니다. 따라서, 적절한 `numpy` 데이터형은 `np.int` 입니다:

```

# 데이터셋
IRIS_TRAINING = "iris_training.csv"
IRIS_TEST = "iris_test.csv"

# 데이터셋을 불러옵니다.
training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
    filename=IRIS_TRAINING,
    target_dtype=np.int,
    features_dtype=np.float32)
test_set = tf.contrib.learn.datasets.base.load_csv_with_header(
    filename=IRIS_TEST,
    target_dtype=np.int,
    features_dtype=np.float32)

```

`tf.contrib.learn`의 `Dataset` 은 파이썬의 [네임드 튜플](#)이며, `data` 와 `target` 필드를(역주 : [namedtuple](#)의 `field_name`을 말합니다) 이용해 특성 데이터와 타깃 값에 접근할 수 있습니다. `training_set.data` 와 `training_set.target` 은 각각 훈련 셋의 특성 데이터와 타깃 값을 가지고 있고, `test_set.data` 와 `test_set.target` 은 테스트 셋의 특성 데이터와 타깃 값을 가지고 있습니다.

나중에 "[Iris 훈련 데이터로 DNNClassifier 훈련시키기](#)"에서, `training_set.data` 와 `training_set.target` 을 이용하여 모델을 훈련시키고, "[모델 정확도 평가하기](#)"에서는 `test_set.data` 와 `test_set.target` 를 이용할 것입니다. 그전에 먼저, 다음 섹션에서 모델을 구성해 보겠습니다.

딥 신경망 분류기 만들기

tf.contrib.learn은 데이터를 가지고 훈련과 평가를 위해 곧장 사용할 수 있는 `Estimator` 라 불리는 여러 가지의 미리 정의된 모델을 제공합니다. 여기에서는 Iris data를 학습시키기 위해 딥 신경망 모델을 구성하겠습니다. tf.contrib.learn을 이용하면, `DNNClassifier` 의 객체를 몇 줄의 코드로 간단히 만들 수 있습니다:

```
# 모든 특성이 실수값을 가지고 있다고 지정합니다
feature_columns = [tf.contrib.layers.real_valued_column("", dimension=4)]

# 10, 20, 10개의 유닛을 가진 3층 DNN를 만듭니다
classifier = tf.contrib.learn.DNNClassifier(feature_columns=feature_columns,
                                             hidden_units=[10, 20, 10],
                                             n_classes=3,
                                             model_dir="/tmp/iris_model")
```

위 코드는 먼저 데이터 셋에 있는 특성의 데이터 타입을 지정하기 위해 모델의 특성 열 (`feature_columns`)을 정의합니다. 모든 특성이 연속적인 값이므로

`tf.contrib.layers.real_valued_column` 가 특성 열을 구성하는 데 사용하기 적합한 함수입니다. 이 데이터 셋에는 네 개의 특성(꽃받침 너비, 꽃받침 길이, 꽃잎 너비, 꽃잎 길이)이 있으므로 모두 포함시키기 위해 `dimensions` 을 4 로 지정합니다.

그런 다음 아래 인자를 사용하여 `DNNClassifier` 모델을 만듭니다:

- `feature_columns=feature_columns` . 위에서 만든 특성 열을 지정합니다.
- `hidden_units=[10, 20, 10]` . 각각 10, 20, 10 개의 뉴런을 가진 세 개의 히든 레이어.
- `n_classes=3` . 세 가지 붓꽃의 품종을 나타내는 타깃 클래스.
- `model_dir=/tmp/iris_model` . 모델이 학습하는 동안 체크 포인트를 저장할 디렉토리. 텐서플로우의 로깅과 모니터링에 대해서는 [Logging and Monitoring Basics with tf.contrib.learn](#)을 참고하세요.

Iris 훈련 데이터로 DNNClassifier 훈련 시키기

이제 DNN `classifier` 모델을 설정했으니, `fit` 메소드를 이용하여 Iris 훈련 데이터에 학습시킬 수 있습니다. 특성 데이터(`training_set.data`)와 타깃 값(`training_set.target`), 그리고 훈련할 반복 횟수(여기서는 2000)를 인자로 넘겨줍니다:

```
# 모델 학습
classifier.fit(x=training_set.data, y=training_set.target, steps=2000)
```

`classifier` 는 모델의 상태를 저장하고 있습니다. 따라서 원한다면 모델을 반복하여 학습시킬 수 있습니다. 예를 들어서, 위의 한 줄은 다음의 두 줄과 완벽하게 같습니다:

```
classifier.fit(x=training_set.data, y=training_set.target, steps=1000)
classifier.fit(x=training_set.data, y=training_set.target, steps=1000)
```

하지만 만약 학습되는 동안에 모델을 추적하고 싶다면, 대신 텐서플로우의 `monitor` 를 사용하여 로그를 남기는 게 낫습니다. 이에 대한 내용은 “[Logging and Monitoring Basics with tf.contrib.learn](#)”를 참고하세요.

모델 정확도 평가하기

Iris 테스트 데이터를 사용해 `DNNClassifier` 모델을 학습시켰습니다. 이제, `evaluate` 메소드를 이용하여 Iris 테스트 데이터로 모델의 정확도를 확인해 보겠습니다. `evaluate` 는 `fit` 과 같이 특성 데이터와 타깃 값을 인자로 받고, 평가 결과를 하나의 딕셔너리로 반환합니다. 다음의 코드는 Iris 테스트 데이터— `test_set.data` 와 `test_set.target` —를 전달하고, 결과 값에서 `accuracy` 를 출력합니다:

```
accuracy_score = classifier.evaluate(x=test_set.data, y=test_set.target)["accuracy"]
print('Accuracy: {:.2f}'.format(accuracy_score))
```

전체 스크립트를 실행하고 정확도 결과를 확인합니다:

```
Accuracy: 0.966667
```

결과 값이 조금 다를 수 있지만 90%보다는 높게 나올 것입니다. 비교적 적은 데이터셋 치고는 나쁘지 않습니다!

새로운 표본 분류하기

새로운 표본을 분류하기 위해 모델의 `predict()` 메소드를 이용합니다. 예를 들어, 다음의 새로운 꽃의 표본이 두 개가 있습니다:

꽃받침 길이	꽃받침 너비	꽃잎 길이	꽃잎 너비
6.4	3.2	4.5	1.5
5.8	3.1	5.0	1.7

다음 코드에서 이 데이터의 종을 예측할 수 있습니다:

```
# 새로운 두 꽃의 표본을 분류합니다.  
new_samples = np.array(  
    [[6.4, 3.2, 4.5, 1.5], [5.8, 3.1, 5.0, 1.7]], dtype=float)  
y = list(classifier.predict(new_samples, as_iterable=True))  
print('Predictions: {}'.format(str(y)))
```

`predict()` 메소드는 각 표본의 예측 결과를 표본마다 하나씩 할당하여 배열로 반환합니다:

```
Prediction: [1 2]
```

이 모델은 첫 번째 표본을 *Iris versicolor*, 두 번째 표본을 *Iris virginica*로 예측하였습니다.

추가적인 자료

- `tf.contrib.learn`에 대해 추가적인 참고 자료를 원한다면, 공식적인 [API docs](#)를 살펴보세요.
- `tf.contrib.learn`을 이용해 선형 모델을 생성하는 방법을 배우려면 [Large-scale Linear Models with TensorFlow](#)를 살펴보세요.
- `tf.contrib.learn` API를 이용하여 자신만의 Estimator 클래스를 만들고 싶다면 [Building Machine Learning Estimator in TensorFlow](#)를 살펴보세요.
- 브라우저에서의 신경망 모델링과 시각화를 체험해 보려면 [Deep Playground](#)를 살펴보세요.
- 신경망에 대한 좀 더 심화된 튜토리얼을 원한다면 [Convolutional Neural Networks](#)와 [Recurrent Neural Networks](#)를 살펴보세요.

TensorFlow를 활용한 대형 스케일 선형 모델

tf.learn API는 TensorFlow에서 선형 모델을 사용하기 위한 (다른 것들중에서도) 풍부한 도구들을 제공합니다. 이 문서는 이러한 툴들에 대한 개요를 제공합니다. 이 문서에선 다음의 것들을 설명합니다.

- 선형 모델이 무엇인지.
- 왜 선형 모델을 사용하는지.
- tf.learn이 TensorFlow에서 선형 모델을 얼마나 쉽게 구축할 수 있는지.
- 양 쪽의 장점을 얻기 위해 tf.learn을 사용하여 어떻게 선형 모델과 딥러닝을 결합할 수 있는지.

tf.learn 선형 모델 도구가 당신에게 유용한지 아닌지를 결정하려면 이 문서를 읽어보십시오. 그리고 [Linear Models tutorial](#)를 시도해보십시오. 이 개요는 튜토리얼의 코드 샘플을 사용하지만, 튜토리얼은 코드를 더욱 상세하게 파고듭니다.

이 개요를 이해하기 위해선 머신 러닝의 기초 개념에 친숙해지는것과 [tf.learn](#)를 보는것이 도움이 될 것입니다.

[TOC]

선형 모델이란?

선형 모델은 예측을 만들기위해 피쳐들의 단일 가중치 합을 사용합니다. 예를 들어, 인구의 나이, 교육 기간, 그리고 주별 근무시간에 대한 [data](#)를 가지고 있다면 가중치 합이 개인의 급여를 추정하도록 숫자 각각에 대한 가중치를 학습할 수 있습니다.

몇 가지 선형 모델은 가중치 합을 더욱 편리한 형태로 변환할 수 있습니다. 예를 들면, 로지스틱 회귀는 출력값을 0과 1사이의 값으로 바꾸기위해 가중치 합을 로지스틱 함수로 연결합니다. 그러나 여전히 입력 피쳐들에 대한 단 하나의 가중치만 가지고 있습니다.

왜 선형 모델을 사용하는가?

최근 연구에서 다중 레이어를 가진 매우 복잡한 신경망의 강력함이 입증되고 있는 시점에 왜 이러한 매우 단순한 모델을 사용할까요?

선형 모델은:

- 깊은 신경망에 비해 학습이 빠릅니다.
- 매우 큰 피쳐 집합에서도 잘 동작합니다.
- 학습 속도와 귀찮은 작업등이 많이 필요하지 않는 알고리즘으로 훈련이 가능합니다.

- 신경망에 비해 인터프리팅과 디버깅을 매우 쉽게 할 수 있습니다. 예측에 가장 큰 영향을 주는 피쳐가 무엇인지 찾기 위해 각 피쳐들에게 할당된 가중치들을 검사할 수 있습니다.
- 머신러닝을 배우는데에 훌륭한 시작 지점을 제공합니다. (처음 머신러닝을 배우는데 적합합니다.)
- 산업에서 널리 사용되고 있습니다.

tf.learn이 어떻게 선형 모델을 구축하는데 도움을 주는가?

당신은 특별한 API의 도움 없이도 TensorFlow의 스크래치에서 선형 모델을 구축할 수 있습니다. 하지만 tf.learn은 효율적인 대형 스케일의 선형 모델을 쉽게 구축할 수 있는 몇 가지 도구를 제공합니다.

피쳐 컬럼과 변환(transformations)

선형 모델을 설계하는 대다수의 작업은 로우(raw) 데이터를 적당한 입력 피쳐들로 변환하는 작업들로 이루어집니다. tf.learn은 이러한 변환들을 가능하게하기 위해 `FeatureColumn` 추상화를 사용합니다.

`FeatureColumn` 은 데이터에서 하나의 피쳐를 나타냅니다. `FeatureColumn` 은 'height'와 같은 양적 수치를 나타낼 수도 있고, {'blue', 'brown', 'green'}와 같은 이산 확률 집합에서 뽑혀진 값인 'eye_color'와 같은 카테고리를 나타낼 수도 있습니다.

'height'와 같은 연속적인 피쳐와 'eye_color'와 같은 카테고리성 피쳐에서 데이터의 단일값은 모델로 입력되기 전에 숫자들의 시퀀스로 변환이 될 것입니다. `FeatureColumn` 추상화는 이러한 사실에도 불구하고 피쳐를 하나의 의미있는 단위로써 조작하도록 합니다. 당신은 변환을 지정할 수 있으며 당신이 모델에 넣을 텐서의 특정 인덱스를 처리하지 않고 포함시킬 피쳐들을 선택할 수 있습니다.

희소 컬럼

선형 모델에서 카테고리성 피쳐들은 일반적으로 각각의 가능한 값이 인덱스나 아이디를 가지고 있는 희소 벡터로 변환됩니다. 예를 들면, 만약 `eye_color` 를 길이가 3인 벡터로써 표현할 수 있는 딱 3가지의 가능한 `eye_color`가 있다고 해봅시다. 그러면 'brown'은 [1, 0, 0], 'blue'는 [0, 1, 0], 'green'은 [0, 0, 1]로 표현될 수 있습니다. 이 벡터들은 가능한 값들이 매우 많아질 경우(가령 모든 영어단어 라던지), 많은 제로값을 가지면서 매우 길어질 수 있기 때문에 "희소(sparse)"라고 부릅니다.

tf.learn 선형 모델을 사용하기 위해서 희소 컬럼을 사용할 필요는 없지만, 선형 모델의 강점 중 하나는 매우 큰 희소 벡터를 처리하는 능력입니다. 희소 피쳐들은 tf.learn 선형 모델 도구의 가장 기본적인 사용 사례입니다.

희소 컬럼 코드화

`FeatureColumn` 는 카테고리성 값들의 벡터로의 변환을 자동으로 처리합니다. 다음의 코드를 보세요:

```
eye_color = tf.contrib.layers.sparse_column_with_keys(
    column_name="eye_color", keys=["blue", "brown", "green"])
```

`eye_color` 는 원데이터의 컬럼명입니다.

카테고리성 피쳐들의 모든 가능한 값들을 알 수 없는 경우에도 `FeatureColumn` 을 생성할 수 있습니다. 이 경우엔 피쳐값들에게 인덱스를 할당하기위해 해쉬 함수를 사용하는 `sparse_column_with_hash_bucket()` 을 사용할 수 있습니다.

```
education = tf.contrib.layers.sparse_column_with_hash_bucket(\n    "education", hash_bucket_size=1000)
```

피쳐 교차

선형 모델은 피쳐들을 나누기위해 독립적인 가중치를 할당하기 때문에, 특정한 피쳐들의 조합의 상대적인 중요성은 학습할 수 없습니다. 만약 'favorite_sport'피쳐와 'home_city'피쳐를 가지고 있으며 어떤 사람이 빨간 옷을 좋아하는지 아닌지를 예측하려고 할 때, 당신의 선형 모델은 특히 빨간 옷을 좋아하는 St. Louis의 야구 팬들은 학습할 수 없을 것입니다.

당신은 'favorite_sport_x_home_city'라는 새로운 피쳐를 생성함으로써 이러한 한계를 극복할 수 있습니다. 주어진 사람에 대한 이 피쳐의 값은 단순히 두 개의 원래 피쳐값들을 하나로 잇는 것입니다. 이러한 종류의 피쳐 조합을 피쳐 교차(*feature cross*)라고 부릅니다.

`crossed_column()` 메서드는 피쳐 교차 생성을 쉽게 만들어줍니다.

```
sport = tf.contrib.layers.sparse_column_with_hash_bucket(\n    "sport", hash_bucket_size=1000)\n\ncity = tf.contrib.layers.sparse_column_with_hash_bucket(\n    "city", hash_bucket_size=1000)\nsport_x_city = tf.contrib.layers.crossed_column(\n    [sport, city], hash_bucket_size=int(1e4))
```

연속적인 컬럼

연속적인 피쳐는 다음과 같이 지정할 수 있습니다:

```
age = tf.contrib.layers.real_valued_column("age")
```

비록 하나의 실수이기는 하지만, 연속적인 피쳐는 종종 모델로 바로 입력될 수 있으며, 뿐만 아니라 tf.learn은 이러한 종류의 컬럼을 위해 유용한 변환을 제공합니다.

버킷화 (Bucketization)

버킷화(Bucketization)는 연속적인 컬럼을 카테고리성 컬럼으로 변환합니다. 이 변환은 연속적인 피쳐를 피쳐 교차에서 사용하도록 하거나 특별한 중요성을 가진 특정한 값 범위의 케이스를 학습하도록 합니다.

버킷화는 가능한 값들의 범위를 버킷이라 불리우는 부분 범위로 나눕니다.

```
age_buckets = tf.contrib.layers.bucketized_column(  
    age, boundaries=[18, 25, 30, 35, 40, 45, 50, 55, 60, 65])
```

버킷은 그 값이 해당 값을 위한 카테고리 라벨이 됩니다.

입력 함수

`FeatureColumn` 은 모델의 입력 데이터를 위해, 어떻게 데이터를 표현하고 변환하는지를 나타내는 명세를 제공합니다. 그러나 데이터 그 자체를 제공하지는 않습니다. 데이터는 입력 함수를 통해 제공해줘야 합니다.

입력 함수는 반드시 텐서들의 딕셔너리를 반환해야합니다. 각 키는 `FeatureColumn` 의 이름을 가리킵니다. 각 키의 값은 모든 데이터 인스턴스에 대한 피쳐의 값들을 포함하는 텐서입니다. 입력 함수의 예시는 [linear models tutorial code](#)에서 `input_fn` 을 보십시오.

`fit()` 과 `evaluate()` 로 전달되는 입력 함수는 다음 섹션에서 보게될 훈련과 테스팅의 시작을 호출합니다.

선형 추정량

tf.learn의 추정량 클래스는 통합된 훈련과 회귀와 분류 모델을 위한 평가 하네스를 제공합니다. 이는 훈련과 평가 루프들의 상세한것들을 다루며 사용자는 모델 입력과 아키텍쳐에만 집중할 수 있도록 해줍니다.

선형 추정량을 구축하기 위해선 각각 분류와 회귀를 위해 `tf.contrib.learn.LinearClassifier` 추정량 또는 `tf.contrib.learn.LinearRegressor` 추정량을 사용할 수 있습니다.

모든 tf.learn의 추정량에 대해, 실행은 단지 다음과 같이 하면됩니다.

1. 추정량 클래스의 인스턴스를 만듭니다. 두 개의 선형 추정량 클래스를 위해선, `FeatureColumn` 의 리스트를 생성자에 전달하면 됩니다.
2. 훈련을 위해 추정량의 `fit()` 메서드를 호출합니다.
3. 어떻게 수행되는지 보기 위해 추정량의 `evaluate()` 메서드를 호출합니다.

예시:

```
e = tf.contrib.learn.LinearClassifier(feature_columns=[  
    native_country, education, occupation, workclass, marital_status,  
    race, age_buckets, education_x_occupation, age_buckets_x_race_x_occupation],  
    model_dir=YOUR_MODEL_DIRECTORY)  
e.fit(input_fn=input_fn_train, steps=200)  
# 한 단계 평가하기 (테스트 데이터를 통해 전달)  
results = e.evaluate(input_fn=input_fn_test, steps=1)  
  
# 평가에 대한 통계를 출력합니다.  
for key in sorted(results):  
    print "%s: %s" % (key, results[key])
```

넓고 깊은 학습

tf.learn API는 또한 선형 모델과 깊은 신경망을 함께 훈련시킬 수 있도록하는 추정량 클래스를 제공합니다. 이 새로운 접근법은 신경망의 일반화 능력과 선형 모델의 키 피쳐들을 기억할 수 있는 능력을 결합합니다. 이러한 "넓고 깊은" 종류의 모델을 생성하려면

`tf.contrib.learn.DNNLinearCombinedClassifier` 사용하세요.

```
e = tf.contrib.learn.DNNLinearCombinedClassifier(  
    model_dir=YOUR_MODEL_DIR,  
    linear_feature_columns=wide_columns,  
    dnn_feature_columns=deep_columns,  
    dnn_hidden_units=[100, 50])
```

더 상세한 내용은 [Wide and Deep Learning tutorial](#)를 보십시오.

텐서플로우 선형 모델 튜토리얼

(v1.0)

이번 강의에서 우리는 이진 분류 문제를 사람에 나이, 성별, 교육, 그리고 직업(특성들)에 관한 인구 조사 데이터를 가지고 한 사람의 연봉이 50,000불이 넘는지를 TensorFlow에 TF.Learn API를 사용해서 풀어 볼 것이다(목표 레이블). 우리는 **로지스틱 회귀 모델**을 주어진 개인들에 정보를 가지고 교육 할 것이고 모델은 개인의 연봉이 50,000달러 이상일 가능성으로 해석 될 수 있는 0과1 사이의 숫자를 출력한다.

설치

이번 튜토리얼 코드를 실행해보기 위해서:

1. 텐서플로우를 설치하지 않았다면 [텐서플로 설치](#)
2. [튜토리얼 코드](#) 다운로드.
3. **pandas** 데이터 분석 라이브러리 설치.

`tf.learn`에 **pandas**가 필수적인 요소는 아니지만 `tf.learn`이 판다를 지원하고 또한 이번 튜토리얼에서 판다를 하기 때문. **pandas**를 설치하기 위해서:

- i. `pip` 설치:

```
# Ubuntu/Linux 64-bit
$ sudo apt-get install python-pip python-dev

# Mac OS X
$ sudo easy_install pip
$ sudo easy_install --upgrade six
```

- ii. `pip`로 **pandas** 설치하기:

```
$ sudo pip install pandas
```

만약에 판다 설치에 어려움을 느낀다면, 판다 사이트에서 [설명] (<http://pandas.pydata.org/pandas-docs/stable/install.html>) 을 참고하시오.

4. 이 튜토리얼에 설명된 선형모델을 훈련하기 위해 튜토리얼 코드를 아래의 명령어로 실행하시오:

```
$ python wide_n_deep_tutorial.py --model_type=wide
```

코드가 어떻게 선형모델을 구축하는지 계속 읽어 보자.

인구조사 데이터 읽어보기

우리가 사용할 데이터 세트는 [소득 인구조사 데이터세트] (<https://archive.ics.uci.edu/ml/datasets/Census+Income>). [훈련 데이터] (<https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data>) 그리고 테스트 데이터를 수동 또는 코드를 이용해서 내려받을 수 있습니다.

```
# 텐서플로 선형 모델 튜토리얼
```

이번 강의에서 우리는 이진 분류 문제를 사람에 나이, 성별, 교육, 그리고 직업(특성들)에 관한 인구조사 데이터를 가지고 한 사람의 연봉이 50,000불이 넘는지를 TensorFlow에 TF.Learn API를 사용해서 풀어 볼 것이다(목표 레이블). 우리는 **로지스틱 회귀** 모델을 주어진 개인들에 정보를 가지고 교육할 것이고 모델은 개인의 연봉이 50,000달러 이상일 가능성으로 해석 될 수 있는 0과 1 사이의 숫자를 출력한다.

```
## 설치
```

이번 튜토리얼 코드를 실행해보기 위해서:

1. 텐서플로우를 설치하지 않았다면 [텐서플로 설치](../../get_started/os_setup.md)
2. [튜토리얼 코드] (https://www.tensorflow.org/code/tensorflow/examples/learn/wide_n_deep_tutorial.py) 다운로드.
3. pandas 데이터 분석 라이브러리 설치.

`tf.learn`에 `pandas`가 필수적인 요소는 아니지만 `tf.learn`이 판다를 지원하고 또한 이번 튜토리얼에서 판다를 하기 때문. `pandas`를 설치하기 위해서:

1. `pip` 설치:

```
```shell
Ubuntu/Linux 64-bit
$ sudo apt-get install python-pip python-dev

Mac OS X
$ sudo easy_install pip
$ sudo easy_install --upgrade six
```

2. `pip`로 pandas 설치하기:

```
```shell
$ sudo pip install pandas
```

```

만약에 판다 설치에 어려움을 느낀다면, 판다 사이트에서 [설명]

(<http://pandas.pydata.org/pandas-docs/stable/install.html>) 을 참고하시오.

1. 이 튜토리얼에 설명된 선형모델을 훈련하기 위해 튜토리얼 코드를 아래의 명령어로 실행하시오:

```
$ python wide_n_deep_tutorial.py --model_type=wide
```

코드가 어떻게 선형모델을 구축하는지 계속 읽어 보자.

## 인구조사 데이터 읽어보기

우리가 사용할 데이터 세트는 [소득 인구조사 데이터세트]

(<https://archive.ics.uci.edu/ml/datasets/Census+Income>). [훈련 데이터]

(<https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data>) 그리고 테스트 데이터를 수동 또는 코드를 이용해서 내려받을 수 있습니다.

```
import tempfile
import urllib
train_file = tempfile.NamedTemporaryFile()
test_file = tempfile.NamedTemporaryFile()
urllib.urlretrieve("http://mlr.cs.umass.edu/ml/machine-learning-databases/adult/adult.
data", train_file.name)
urllib.urlretrieve("http://mlr.cs.umass.edu/ml/machine-learning-databases/adult/adult.
test", test_file.name)
```

CSV 파일들에 다운로드가 완료됐다면, [Pandas] (<http://pandas.pydata.org/>) 데이터프레임에 입력시켜 보자.

```
import pandas as pd
COLUMNS = ["age", "workclass", "fnlwgt", "education", "education_num",
 "marital_status", "occupation", "relationship", "race", "gender",
 "capital_gain", "capital_loss", "hours_per_week", "native_country",
 "income_bracket"]
df_train = pd.read_csv(train_file, names=COLUMNS, skipinitialspace=True)
df_test = pd.read_csv(test_file, names=COLUMNS, skipinitialspace=True, skiprows=1)
```

이번 과제가 이진 분류 문제이기 때문에 수입이 50,000달러가 넘는다면 1을 그렇지 않다면 0에 값 을 가지는 열의 이름이 "label"인 표를 만들 것이다.

```
LABEL_COLUMN = "label"
df_train[LABEL_COLUMN] = (df_train["income_bracket"].apply(lambda x: ">50K" in x)).astype(int)
df_test[LABEL_COLUMN] = (df_test["income_bracket"].apply(lambda x: ">50K" in x)).astype(int)
```

다음으로, 데이터프레임에서 어떤 열들이 목표 label을 예측하는 데 사용 될 수 있는지 살펴보자. 열들은 categorical 또는 continuous 두 타입으로 구분 될 수 있다.

- 만약에 값이 오직 유한집합 범주 안에 있을 때 **categorical** 열이라 불린다. 예를 들어 사람에 국적(미국, 인도, 일본 등)이나 교육 수준(고등학교, 대학 등)이 categorical 열들이다.
- 만약에 값이 어떤 수치로 나올 수 있다면 **continuous** 열이라 불린다. 예를 들어, 한 사람에 소득이 continuous열이다.

```
CATEGORICAL_COLUMNS = ["workclass", "education", "marital_status", "occupation",
 "relationship", "race", "gender", "native_country"]
CONTINUOUS_COLUMNS = ["age", "education_num", "capital_gain", "capital_loss", "hours_per_week"]
```

수입 인구조사 데이터 세트에 나오는 열 리스트:

| 열 이름      | 타입          | 설명     | {.sortable} |
|-----------|-------------|--------|-------------|
| age       | Continuous  | 나이     |             |
| workclass | Categorical | 고용주 타입 |             |

:::(정부, 군대, 기업 등) : | fnlwgt | Continuous | The number of people the census :::  
takers believe that observation :::: represents (sample weight). This :::: variable will not  
be used. : | education | Categorical | 최고 학력 || education\_num | Continuous | 숫자 형식의  
최고 학력 || marital\_status | Categorical | 혼인 여부 || occupation | Categorical | 직업 ||  
relationship | Categorical | Wife, Own-child, Husband, | :::: Not-in-family, Other-relative, :::::  
Unmarried. : | race | Categorical | 인종 ( 백인, Asian-Pac-Islander, | :::: Amer-Indian-Eskimo,  
Other, 흑인. : | gender | Categorical | 성별 (남, 여) || capital\_gain | Continuous | 기록된 양도  
소득. || capital\_loss | Continuous | 기록된 자본 손실. || hours\_per\_week | Continuous | 주당  
근무시간. || native\_country | Categorical | 출생지 || income | Categorical | ">50K" 또는 "  
<=50K", 개인의 일년 | :::: 수입이 5만불 이상인지 아닌지 뜻함 :

## 데이터를 텐서들로 바꾸기

TF.Learn 모델을 구축 할 때, 입력 데이터는 Input Builder 함수에 의해서 명시된다. 이 builder 함수는 TF.Learn에 `fit`이나 `evaluate`와 같은 메소드들에게 넘겨 질때 까지 호출되지 않는다. 이 함수의 목적은 입력 데이터를 `Tensors`나 `SparseTensors` 형태로 구성하기 위함에 있다. 더 구체적으로, Input Builder 함수는 다음과 같은 한 쌍을 반환한다:

1. `feature_cols` : A dict from feature column names to `Tensors` or `SparseTensors`.
2. `label` : 라벨 열을 포함 하고 있는 `Tensor`.

`feature_cols` 에 키들은 다음 부분에서 열을 구성하는 데 사용 될 것이다.

우리는 `fit`과 `evaluate` 메소드들을 서로 다른 데이터로 호출 하고 싶으므로, 서로 같지만 단지 다른 데이터를 `input_fn`에 전달하는 input builder 함수인 `train_input_fn` 그리고 `test_input_fn` 를 정의 했습니다.

여기서 눈여겨볼 것은 `input_fn` 가 그래프 실행 중이 아니라 텐서플로 그래프를 생성하는 도중에 호출된다는 것입니다. Input Builder가 반환하는 것은 입력 데이터를 대표하는 텐서플로 연산 기본 단위인 `Tensor` (또는 `SparseTensor`)입니다.

우리 모델은 입력 데이터의 정수값을 대표하는 `constant` 텐서로 나타낸다, 이 경우에는 `df_train`나 `df_test`에 열에 값을 대표한다. 이 방법이 텐서플로에 데이터를 전달하는 가장 간단한 방법이다. 다른 심화한 방법으로는 파일이나 다른 데이터 소스를 대표하는 Input Reader를 만들어서 파일을 텐서플로가 그래프를 실행하는 동안에 읽어 나가는 것이다.

훈련 또는 테스트 데이터 프레임에 있는 각각의 `continuous` 열들은 밀집 데이터를 대표하기 좋은 `Tensor`로 변환될 것입니다.

우리는 `categorical` 데이터를 반드시 `sparseTensor`로 나타내야 합니다. Categorical 데이터 포맷은 희소 데이터를 대표하기 좋습니다.

```

import tensorflow as tf

def input_fn(df):
 # Creates a dictionary mapping from each continuous feature column name (k) to
 # the values of that column stored in a constant Tensor.
 continuous_cols = {k: tf.constant(df[k].values)
 for k in CONTINUOUS_COLUMNS}

 # Creates a dictionary mapping from each categorical feature column name (k)
 # to the values of that column stored in a tf.SparseTensor.
 categorical_cols = {k: tf.SparseTensor(
 indices=[[i, 0] for i in range(df[k].size)],
 values=df[k].values,
 shape=[df[k].size, 1])
 for k in CATEGORICAL_COLUMNS}

 # Merges the two dictionaries into one.
 feature_cols = dict(continuous_cols.items() + categorical_cols.items())
 # Converts the label column into a constant Tensor.
 label = tf.constant(df[LABEL_COLUMN].values)
 # Returns the feature columns and the label.
 return feature_cols, label

def train_input_fn():
 return input_fn(df_train)

def eval_input_fn():
 return input_fn(df_test)

```

## 모델을 위한 선택과 공학 특징

올바른 특성 열 세트를 선택하고 만드는 것이 효과적인 모델 학습에 핵심입니다.

**feature column**은 기존 데이터 프레임(기본 특성 열)에 가공되지 않은 열 중 이거나, 하나 또는 여러 개의 기본 열(파생 특성 열)에 변화를 기본으로 새롭게 생성된 열이다.

근본적으로, "feature column"은 목표 레이블을 예상하는 데 사용 가능한 추상적인 개념의 비가공 또는 파생 변수이다.

### 기본 Categorical 특성 열

categorical 특성을 위한 특성열을 정의하기 위해서 우리는 TF.Learn API로 `SparseColumn` 생성할 수 있습니다.

만약에 모든 열에 특성값 세트를 알고 있고 또한 몇 개 안된다면, `sparse_column_with_keys`를 사용할 수 있습니다. 리스트에 안에 각각의 키들은 0부터 시작해서 자동으로 증가하는 아이디가 할당된다. 예를 들어 우리는 아래와 같이 성별 열에 특성 문자열 "female"에게 숫자 아이디 1을 그리고 "male"에게는 1을 할당 할 수 있다

```
gender = tf.contrib.layers.sparse_column_with_keys(
 column_name="gender", keys=["female", "male"])
```

만약에 가능한 값의 세트를 미리 알 수 없다면 어떻게 해야 하나? 문제없다. 우리는 `sparse_column_with_hash_bucket` 을 대신 사용할 수 있다.

```
education = tf.contrib.layers.sparse_column_with_hash_bucket("education", hash_bucket_size=1000)
```

`education` 에 각 특성열에 가능한 값들은 훈련중에 정수 아이디로 해시 되어질 것이다. 아래의 실례를 보자:

| ID  | Feature |
|-----|---------|
| ... |         |
| 9   | "학사"    |
| ... |         |
| 103 | "박사"    |
| ... |         |
| 375 | "석사"    |
| ... |         |

어떤 방법으로 `SparseColumn` 를 정의 하던지 특성 문자열들은 정해진 맵핑 또는 해쉬를 정수 ID를 찾을 것이다. 여기서 해쉬 충돌이 일어날 수 있지만, 모델의 질에 큰 영향을 끼치지는 않을 것이다. 내부적으로 `LinearModel` 클래스가 모델의 각 특성 ID의 모델 매개변수(모델 무게로 알려진)를 저장하는데 사용되는 `tf.Variable` 의 생성과 측정을 책임지고 있다. 모델의 매개변수들은 우리가 나중에 배우게 될 모델 훈련 과정에서 알게 될 것입니다.

우리는 다른 `categorical` 특성들을 정의하기 위해 비슷한 기술을 사용할 것이다.

```
race = tf.contrib.layers.sparse_column_with_keys(column_name="race", keys=[
 "Amer-Indian-Eskimo", "Asian-Pac-Islander", "Black", "Other", "White"])
marital_status = tf.contrib.layers.sparse_column_with_hash_bucket("marital_status", hash_bucket_size=100)
relationship = tf.contrib.layers.sparse_column_with_hash_bucket("relationship", hash_bucket_size=100)
workclass = tf.contrib.layers.sparse_column_with_hash_bucket("workclass", hash_bucket_size=100)
occupation = tf.contrib.layers.sparse_column_with_hash_bucket("occupation", hash_bucket_size=1000)
native_country = tf.contrib.layers.sparse_column_with_hash_bucket("native_country", hash_bucket_size=1000)
```

## 기초 Continuous 특성 열

비슷하게, 모델에서 사용하고 싶은 각 continuous 특성열들에게 `RealValuedColumn` 을 정의 할 수 있습니다.

```
age = tf.contrib.layers.real_valued_column("age")
education_num = tf.contrib.layers.real_valued_column("education_num")
capital_gain = tf.contrib.layers.real_valued_column("capital_gain")
capital_loss = tf.contrib.layers.real_valued_column("capital_loss")
hours_per_week = tf.contrib.layers.real_valued_column("hours_per_week")
```

## 버킷화를 통해 Continuous 특성들을 범주화 하기

때때로 continuous 특성과 라벨의 관계는 비선형이다. 가상의 한 예와 같이, 한 사람의 수입이 사회 생활 초기에는 나이와 함께 증가할 것이고, 어느 때가 되면 수입의 증가는 더뎌지고, 그리고 마침내 은퇴 후에는 수입이 줄어들 것이다. 모델이 아래의 3가지 경우 중 하나만 습득할 수 있으므로 이 시나리오에서 비가공 `age` 를 실수특성 열로 사용하는 것은 좋은 선택이 아닐 것이다.

1. 수입이 나이의 증가에 따라 항상 같은 비율로 증가 (양의 상관관계)
2. 수입이 나이의 증가에 따라 항상 같은 비율로 감소 (음의 상관관계), 또는
3. 수입이 나이에 상관 없이 항상 같음(관계없음)

만약 우리가 수입과 각 나이 그룹과의 세밀한 상관관계를 학습하고 싶다면 **버킷화** 를 활용할 수 있다. 버킷화는 continuous 특성 전체를 연속적인 빈/버킷들의 세트로 나누고, 이후에 버킷의 값에 따라 원래의 수적 특성을 버킷 아이디(categorical feature)로 변환하는 과정이다. 그래서 우리는 `age` 에 대해 `bucketized_column` 을 정의 할 수 있다:

```
age_buckets = tf.contrib.layers.bucketized_column(age, boundaries=[18, 25, 30, 35, 40,
45, 50, 55, 60, 65])
```

`boundaries` 는 버킷 경계의 목록이다.

이 경우에 10개의 경계가 있고, 결과적으로 11개의 버킷 그룹이 생성된다 ( 0세부터 17까지, 18-24, 25-29.... 65세 이상).

## 다수의 열을 CrossedColumn으로 교차하기

각 기본 feature 열을 나눠 사용하는 것만으로는 데이터를 설명하는데 충분하지 않을 것이다. 예를 들어 교육과 레이블(수입 > 50,000)에 상관관계는 아마 직업에 따라 다를 것이다. 그렇기에 우리가 오직 한가지 모델 `education="학사"` and `education="석사"` 의 무게를 학습한다면, 우리는 모든 경

우의 교육-직업 조합 `education="학사" AND occupation="경영자" and education="학사" AND occupation="수리공"` )의 차이를 알아 낼 수 없다. 다른 특성 조합들의 차이를 알기 위해서 우리는 **crossed feature columns** 을 모델에 부여할 수 있습니다.

```
education_x_occupation = tf.contrib.layers.crossed_column([education, occupation], hash_bucket_size=int(1e4))
```

우리는 또한 `CrossedColumn` 을 두 가지 이상의 열에 생성할 수 있습니다. 각 `constituent` 열은 기본 특성 열 `categorical( SparseColumn )`이나 버킷화된 실수 특성 열(`BucketizedColumn`), 심지어 다른 `CrossColumn` 이 될 수 있습니다. 하나의 예:

```
age_buckets_x_race_x_occupation = tf.contrib.layers.crossed_column(
 [age_buckets, race, occupation], hash_bucket_size=int(1e6))
```

## 로지스틱 회귀 모델 정의하기

입력 데이터를 가공하고 모든 특성열들을 정의 한 다음, 이제 모든 것을 한자리에 모아 로지스틱 회귀 모델을 구축할 준비가 되었다. 이전 부분에서 우리는 아래의 특성열들을 포함한 여러 가지의 기본 그리고 파생 특성 열을 보았다.

- `SparseColumn`
- `RealValuedColumn`
- `BucketizedColumn`
- `CrossedColumn`

위의 모든 특성열들은 추상 클래스 `FeatureColumn` 에 하위 클래스 들이며, 모델의 `feature_columns` 필드에 추가될 수 있습니다.

```
model_dir = tempfile.mkdtemp()
m = tf.contrib.learn.LinearClassifier(feature_columns=[
 gender, native_country, education, occupation, workclass, marital_status, race,
 age_buckets, education_x_occupation, age_buckets_x_race_x_occupation],
 model_dir=model_dir)
```

모델은 자동으로 `feature`들을 보지 않고도 예측을 제어 할 수 있는 편향 용어를 자동으로 학습할 것입니다(더 많은 설명을 위해서는 "로지스틱 회귀가 어떻게 작동하나" 보시오). 학습된 모델 파일은 `model_dir`에 저장될 것입니다.

## 모델을 훈련, 평가하기

모델에 모든 특성을 추가한 다음 어떻게 실제로 모델을 훈련 시키는지 알아보자. 모델을 훈련하는 것은 TF.Learn API를 사용하면 한 줄이면 된다.

```
m.fit(input_fn=train_input_fn, steps=200)
```

모델을 훈련 시킨 뒤 모델이 얼마나 홀드 아웃 데이터의 라벨을 잘 예측하는지 평가해 볼 수 있다

```
results = m.evaluate(input_fn=eval_input_fn, steps=1)
for key in sorted(results):
 print "%s: %s" % (key, results[key])
```

첫 번째 줄의 결과는 정확도 83.6%를 뜻하는 `accuracy: 0.83557522` 와 같이 나올 것이다. 당신이 더 좋은 결과를 낼 수 있는지 자유롭게 더 많은 기능을 시도해보고 변경해봐라

만약에 처음부터 끝까지 작동하는 예제를 보고 싶다면 우리의 [예제코드를](#) 내려받고, `model_type` 플래그를 `wide` 로 설정하세요.

## 과적화를 피하기위해 정규화 추가하기

정규화는 과적화를 피하려고 사용되는 기술이다. 과적화는 모델이 훈련에 사용했던 데이터로는 잘 작동하지만, 처음 보는 데이터에 대해서는 안 좋은 결과를 보일 때 일어난다. 과적화는 모델이 너무 과도하게 복잡할 때 일어난다. 예를 들어 관찰한 자료에 비해 매개변수가 너무 많을 경우이다. 정규화는 모델의 복잡성을 제어 할 수 있게 해주고, 처음 접하는 데이터에 대해 좀 더 일반화가 가능하게 해준다.

선형모델 라이브러리에 당신은 L1과 L2 정규화 둘을 다음과 같이 모델에 추가할 수 있습니다:

```
m = tf.contrib.learn.LinearClassifier(feature_columns=[gender, native_country, education, occupation, workclass, marital_status, race, age_buckets, education_x_occupation, age_buckets_x_race_x_occupation], optimizer=tf.train.FtrlOptimizer(learning_rate=0.1, l1_regularization_strength=1.0, l2_regularization_strength=1.0), model_dir=model_dir)
```

L1과 L2 정규화의 한 가지 중요한 차이점은 L1 정규화는 모델의 무게를 0에 있게 하려고 하고 `sparser` 모델을 만들지만, L2 정규화는 모델에 무게를 0에 가깝게 만들려고 노력한다 하지만 꼭 0 일 필요는 없습니다. 그렇기에 당신이 만약 L1 정규화를 강화한다면 많은 모델에 무게가 0일 것이라 작은 모델을 가지게 될 것입니다. 정규화는 feature 크기가 크지만 흩어져 있는 경우와 자원이 제한적이라 아주 큰 모델을 실행하기 힘든 경우에 종종 바람직한 방법입니다.

실제로는 과적화와 모델의 크기를 조절 하기에 최고의 매개변수를 찾기 위해 L1과 L2의 강도를 다양하게 시도해봐야 합니다.

## 로지스틱 회귀가 어떻게 작동하는가

만약 로지스틱 회귀 모델에 익숙하지 않다면 여기서 잠시 멈춰 로지스틱 회귀모델이 실제로 어떻게 생겼는지 보도록 하자. 우리는 라벨을  $Y$ 로, 관찰된 feature들의 세트를 feature 벡터  $\mathbf{x} = [x_1, x_2, \dots, x_d]$ 로 표현할 것이다. 우리는 수입이 50,000달러가 넘는다면  $Y = 1$  그렇지 않다면  $Y = 0$  정의 할 것입니다. 로지스틱 회귀에서 주어진 features  $\mathbf{x}$ 에 대해 라벨이 양수( $Y = 1$ )가 될 가능성을 나타내는 것은:

$$P(Y = 1|\mathbf{x}) = \frac{1}{1 + \exp(-(\mathbf{w}^T \mathbf{x} + b))}$$

where  $\mathbf{w} = [w_1, w_2, \dots, w_d]$  are the model weights for the features

$$\mathbf{x} = [x_1, x_2, \dots, x_d]$$

$\mathbf{w} = [w_1, w_2, \dots, w_d]$ 는 features  $\mathbf{x} = [x_1, x_2, \dots, x_d]$ 의 모델 무게이다.  $b$ 는 정수이며 종종 모델의 편향이라고 불린다. 공식은 두 parts-A 선형모델과 로지스틱 함수로 구성 되어 있다.

- **선형 모델:** 먼저, 우리는 결과가 입력 features  $\mathbf{x}$  함수인 선형 모델  $\mathbf{w}^T \mathbf{x} + b = b + w_1 x_1 + \dots + w_d x_d$ 로 볼 수 있다. 편향  $b$ 는 아무런 feature들을 관찰하지 않고 한 예측이다. 모델의 무게  $w_i$ 는 feature와 어떻게 양의 label과 상관관계를 가지는지 나타낸다. 만약에  $x_i$ 가 양수 label과 양의 상관관계를 가지면 무게  $w_i$ 는 증가할 것이고  $P(Y = 1|\mathbf{x})$ 는 1에 가까울 것이다. 반면에,  $x_i$ 가 양수 label과 음의 상관관계를 가지면 무게  $w_i$ 는 감소 할 것이고  $P(Y = 1|\mathbf{x})$ 는 0에 가까울 것이다.
- **로지스틱 함수:** 두 번째로, 우리는 로지스틱 함수(시그모이드 함수로 알려진)  $S(t) = 1 / (1 + \exp(-t))$ 가 선형 모델에 적용되는 것을 볼 수 있다. 확률로 해석되어질 수 있는 어떠한 0과 1 사이의 실수로에서 나온 선형모델  $\mathbf{w}^T \mathbf{x} + b$ 의 출력 값을 변환하는데 사용된다.

모델 훈련은 최적화에 대한 문제이다. 다시말해, 데이터를 학습하는 동안 정의된 손실 함수를 최소화 할수 있는 모델 weights 세트(즉, 모델 매개변수)를 찾는 것이다. 손실 함수는 라벨의 실측 라벨과 모델의 예측 차이점을 측정한다. 만약에 예측이 실수 라벨과 아주 가깝다면 손실 값은 낮을 것이고 반대로 예측값이 라벨과 많은 차이가 날 때 손실 값은 높을 것이다.

## 심화학습

만약 더 많은 것을 배우고 싶다면 어떻게 선형 모델과 심층 신경망에 강점을 TF.Learn API을 이용하여 훈련 시키는지 알아 볼 수 있는 [Wide & Deep Learning Tutorial](#)을 확인해 보시오.

# TensorFlow Wide & Deep Learning 튜토리얼

(v1.0)

이전 튜토리얼 [선형모델\(Linear Model\)](#) 튜토리얼에서, [Census Income Dataset](#)을 이용하여 연소득이 \$50,000 이상인 사람을 예측하기 위한 로지스틱 회귀모델을 학습했다. TensorFlow는 deep neural networks(신경망) 학습에 탁월함은 물론, 어느 것을 잘 선택해야 하는가에 대한 것을 생각할 수 있다. 이들을 동시에 안될까? 하나의 모델에 둘의 strength를 조합하는 것은 가능할까?

이 튜토리얼에서, TF를 사용하는 법에 대해 소개할 것이다. wide 선형 모델과 deep feed-forward 신경망을 함께 학습하기 위한 API를 배우자. 이 접근법은 저장법과 일반화의 강점을 조합한 것이다. 이 방법은 sparse 입력 feature들(즉, 많은 수의 가능한 특징값을 가진 분류적 특징)의 일반적으로 큰 규모의 회귀법과 분류법 문제들에 유용하다. Wide & Deep 학습의 학습 동작에 대해 관심이 있다면, 우리 논문[research paper](#)을 참고하기 바란다.

![Wide & Deep Spectrum of Models] (../../images/wide\_n\_deep.svg "Wide & Deep")

위 그림은 wide 모델(sparse feature와 변환법의 로지스틱 회귀분석), deep 모델(embedding layer(층)과 여러 hidden layer(층)들의 feed-forward 신경망)을 비교하여 보여준다. 가장 높은 수준에서는, TF를 이용한 wide, deep, wide & deep 모델의 설정 3 단계만 존재한다. API를 배워보자:

1. wide 부분에 대한 features 선택 : 사용하길 원하는 sparse base column과 crossed column들을 선택한다
2. deep 부분에 대한 features 선택 : 연속된 열, 각 분류 열의 embedding dimension, 그리고 hidden layer 크기를 선택한다
3. 이들을 Wide & Deep 모델에 적용한다(`DNNLinearCombinedClassifier`)

다 됐다! 간단한 예제를 통해 알아보자.

## 설정

이 튜토리얼을 위한 코드를 실행하기 위해서,

1. 아직 설치되지 않았다면, TensorFlow를 설치한다 [Install TensorFlow](#)
2. 튜토리얼 코드를 다운로드 한다 [the tutorial code](#)
3. pandas 데이터 분석 라이브러리를 설치한다. tf.learn은 pandas를 필요로 하지 않지만 이를 지원한다. 그리고 이 튜토리얼은 pandas를 사용한다. `pandas`를 설치하기 위해서 :

## i. Get pip :

```
Ubuntu/Linux 64-bit
$ sudo apt-get install python-pip python-dev

Mac OS X
$ sudo easy_install pip
$ sudo easy_install --upgrade six
```

```
```shell
# Ubuntu/Linux 64-bit
$ sudo apt-get install python-pip python-dev
```

```
# Mac OS X
$ sudo easy_install pip
$ sudo easy_install --upgrade six
```
```

```
```shell
# Ubuntu/Linux 64-bit
$ sudo apt-get install python-pip python-dev
```

```
# Mac OS X
$ sudo easy_install pip
$ sudo easy_install --upgrade six
```
```

## 2. Use `pip` to install pandas:

2. `pip` 를 이용하여 pandas 설치:

```
```shell
$ sudo pip install pandas
```
```

pandas 설치에 문제가 있다면, pandas 사이트의 [instructions](<http://pandas.pydata.org/pandas-docs/stable/install.html>) 를 찾아보아라.

## 1. 이 튜토리얼에 소개된 선형 모델을 학습시키기 위해 이어지는 명령어로 튜토리얼 코드를 실행 하자:

```
$ python wide_n_deep_tutorial.py --model_type=wide_n_deep
```

이 코드가 어떻게 이 선형 모델을 만들어가는 알아보기 위해 계속 읽어보자.

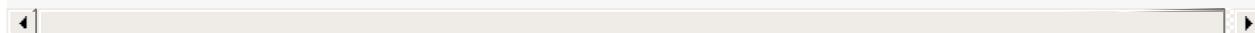
## Base Feature Columns 정의

우선, 사용할 base categorical feature column 과 continuous feature column 들을 정의하자. 이들 base column 들은 모델의 wide 부분과 deep 부분에 모두 사용될 building block 들이 된다.

```
import tensorflow as tf

Categorical base columns.
gender = tf.contrib.layers.sparse_column_with_keys(column_name="gender", keys=["female", "male"])
race = tf.contrib.layers.sparse_column_with_keys(column_name="race", keys=["Amer-Indian-Eskimo", "Asian-Pac-Islander", "Black", "Other", "White"])
education = tf.contrib.layers.sparse_column_with_hash_bucket("education", hash_bucket_size=1000)
marital_status = tf.contrib.layers.sparse_column_with_hash_bucket("marital_status", hash_bucket_size=100)
relationship = tf.contrib.layers.sparse_column_with_hash_bucket("relationship", hash_bucket_size=100)
workclass = tf.contrib.layers.sparse_column_with_hash_bucket("workclass", hash_bucket_size=100)
occupation = tf.contrib.layers.sparse_column_with_hash_bucket("occupation", hash_bucket_size=1000)
native_country = tf.contrib.layers.sparse_column_with_hash_bucket("native_country", hash_bucket_size=1000)

Continuous base columns.
age = tf.contrib.layers.real_valued_column("age")
age_buckets = tf.contrib.layers.bucketized_column(age, boundaries=[18, 25, 30, 35, 40, 45, 50, 55, 60, 65])
education_num = tf.contrib.layers.real_valued_column("education_num")
capital_gain = tf.contrib.layers.real_valued_column("capital_gain")
capital_loss = tf.contrib.layers.real_valued_column("capital_loss")
hours_per_week = tf.contrib.layers.real_valued_column("hours_per_week")
```



## The Wide Model: Linear Model with Crossed Feature Columns

wide 모델은 sparse 하고 crosseed feature column 들의 다양한 집합의 선형 모델이다:

```
wide_columns = [
 gender, native_country, education, occupation, workclass, marital_status, relationship,
 age_buckets,
 tf.contrib.layers.crossed_column([education, occupation], hash_bucket_size=int(1e4)),
 tf.contrib.layers.crossed_column([native_country, occupation], hash_bucket_size=int(1e4)),
 tf.contrib.layers.crossed_column([age_buckets, race, occupation], hash_bucket_size=int(1e6))]
```

crossed feature column 들의 Wide 모델은 feature들 간의 sparse interactions(상호작용들)을 효율적으로 저장할 수 있다. 그럼에도, crossed feature columns 의 한가지 제한되는 점은 학습된 데 이터에 나타나지 않는 feature 조합들을 생성해내지 못한다는 것이다. 이를 수정하기 위해 embeddings 를 포함한 deep 모델을 추가해보자.

## The Deep Model: Neural Network with Embeddings

이전 그림에서 보이는 것과 같이 deep 모델은 feed-forward 신경망이다. 각각의 sparse, high-dimensional(고차원의) categorical feature 들은 종종 embedding 벡터라 불리는 low-dimensional(저차원의), dense real-valued 벡터로 변환된다. 이 low-dimensional(저차원의) dense embedding 벡터들은 continuouse feature 들로 연계되어 변환되고, 계속해서 다음 신경망의 hidden layers 로 연결된다. embedding value 들은 무작위로 초기화되고, 학습 loss(손실)을 최소화하는 다른 모든 모델 파라미터들과 함께 학습된다. embeddings 에 대한 학습에 흥미가 있다면 TensorFlow 튜토리얼 [Vector Representations of Words](#) 이나 위키피디아 [Word Embedding](#) 를 확인해보자.

`embedding_column` 을 이용하여 categorical column 들을 위한 embeddings 를 설정할 것이다. 그리고 이들을 continuous columns 과 연관지울 것이다.

```
deep_columns = [
 tf.contrib.layers.embedding_column(workclass, dimension=8),
 tf.contrib.layers.embedding_column(education, dimension=8),
 tf.contrib.layers.embedding_column(marital_status, dimension=8),
 tf.contrib.layers.embedding_column(gender, dimension=8),
 tf.contrib.layers.embedding_column(relationship, dimension=8),
 tf.contrib.layers.embedding_column(race, dimension=8),
 tf.contrib.layers.embedding_column(native_country, dimension=8),
 tf.contrib.layers.embedding_column(occupation, dimension=8),
 age, education_num, capital_gain, capital_loss, hours_per_week]
```

embedding 의 더 높은 `dimension` 은 더 높은 자유도로 features 의 대표들을 학습할 수 있다. 간단하게, 여기 모든 feature columns 의 dimension(차원)을 8 로 정했다. 경험적으로, a more informed decision 차원의 수는  $k \log_2(n)$  나  $k\sqrt[4]{n}$  의 차수 값에서 시작한다. 이때  $n$  은 feature column 에서 유니크한 feature 의 수이고  $k$  는 작은 상수값(일반적으로 10보다 작은 값)이다.

dense embeddings 를 통해, deep 모델은 학습된 데이터에서 이전에 보여지지 못한 feature 쌍들에 대한 생성을 더 잘 할 수 있고 예측할 수 있다. 그러나 두 feature column 간의 기본 interaction matrix(상호작용 메트릭스)가 sparse 하고 high-rank 일 때, feature column 들에 대한 low-dimensional representation 들의 효율적인 학습은 어렵다. 이러한 경우, 대부분의 feature 쌍들 간 interaction(상호작용)은 몇몇을 제외하는 0(zero) 이 되어야 한다. 하지만 dense embeddings

는 모든 feature 쌍들에 대해 none-zero(0 이 아닌) 예측을 이끌어 냄에 따라, 지나치게 일반화할 수 있다. 반면, crossed features 선형모델은 이러한 "exception rules"(예외들)을 더 적은 모델 파라미터들을 이용해 효율적으로 저장할 수 있다.

이제 어떻게 wide 모델과 deep 모델을 함께 학습시키고 그들 각각의 강점과 약점을 보완해 가는지 살펴보자.

## Wide and Deep 모델을 하나로 조합하기

wide 모델과 deep 모델들은 예측으로써 그들의 마지막 출력 로그 나머지(final output log odd)의 합에 의해 조합된다. 그 다음 로지스틱 loss 함수의 예측에 전달된다. 모든 그래프 정의와 변수 할당량들은 이미 내부에서 다뤄졌기 때문에, `DNNLinearCombinedClassifier` 를 만들어 볼 필요가 있다.:

```
import tempfile
model_dir = tempfile.mkdtemp()
m = tf.contrib.learn.DNNLinearCombinedClassifier(
 model_dir=model_dir,
 linear_feature_columns=wide_columns,
 dnn_feature_columns=deep_columns,
 dnn_hidden_units=[100, 50])
```

## 모델 학습하기와 평가하기

모델 학습시키기 전에, 우리가 [TensorFlow Linear Model tutorial](#) 에서 했던 인구조사 데이터 세트를 입력하자. 입력 데이터 처리를 위한 코드는 편의를 위해 다시 제공한다:

```
import pandas as pd
import urllib

Define the column names for the data sets.
COLUMNS = ["age", "workclass", "fnlwgt", "education", "education_num",
 "marital_status", "occupation", "relationship", "race", "gender",
 "capital_gain", "capital_loss", "hours_per_week", "native_country", "income_bracket"]
]
LABEL_COLUMN = 'label'
CATEGORICAL_COLUMNS = ["workclass", "education", "marital_status", "occupation",
 "relationship", "race", "gender", "native_country"]
CONTINUOUS_COLUMNS = ["age", "education_num", "capital_gain", "capital_loss",
 "hours_per_week"]

Download the training and test data to temporary files.
Alternatively, you can download them yourself and change train_file and
test_file to your own paths.
train_file = tempfile.NamedTemporaryFile()
```

```

test_file = tempfile.NamedTemporaryFile()
urllib.urlretrieve("http://mlr.cs.umass.edu/ml/machine-learning-databases/adult/adult.
data", train_file.name)
urllib.urlretrieve("http://mlr.cs.umass.edu/ml/machine-learning-databases/adult/adult.
test", test_file.name)

Read the training and test data sets into Pandas dataframe.
df_train = pd.read_csv(train_file, names=COLUMNS, skipinitialspace=True)
df_test = pd.read_csv(test_file, names=COLUMNS, skipinitialspace=True, skiprows=1)
df_train[LABEL_COLUMN] = (df_train['income_bracket'].apply(lambda x: '>50K' in x)).astype(int)
df_test[LABEL_COLUMN] = (df_test['income_bracket'].apply(lambda x: '>50K' in x)).astype(int)

def input_fn(df):
 # Creates a dictionary mapping from each continuous feature column name (k) to
 # the values of that column stored in a constant Tensor.
 continuous_cols = {k: tf.constant(df[k].values)
 for k in CONTINUOUS_COLUMNS}
 # Creates a dictionary mapping from each categorical feature column name (k)
 # to the values of that column stored in a tf.SparseTensor.
 categorical_cols = {k: tf.SparseTensor(
 indices=[[i, 0] for i in range(df[k].size)],
 values=df[k].values,
 shape=[df[k].size, 1])
 for k in CATEGORICAL_COLUMNS}
 # Merges the two dictionaries into one.
 feature_cols = dict(continuous_cols.items() + categorical_cols.items())
 # Converts the label column into a constant Tensor.
 label = tf.constant(df[LABEL_COLUMN].values)
 # Returns the feature columns and the label.
 return feature_cols, label

def train_input_fn():
 return input_fn(df_train)

def eval_input_fn():
 return input_fn(df_test)

```

데이터를 읽고 난 후, 모델을 학습하고 평가할 수 있다:

```

m.fit(input_fn=train_input_fn, steps=200)
results = m.evaluate(input_fn=eval_input_fn, steps=1)
for key in sorted(results):
 print "%s: %s" % (key, results[key])

```

결과의 첫줄에는 정확도: 0.84429705 와 같이 나타나야 한다. wide 만 사용한 선형모델에서 83.6% 인 정확도가 Wide & Deep 모델을 사용한 것에서 84.4% 로 향상된 것을 볼 수 있다. 완전한 예제를 경험하길 원한다면, [example code](#) 를 다운로드 할 수 있다.

이 튜토리얼은 API 에 친숙하게 하기 위한 작은 데이터세트(dataset)의 간단한 예제임을 알아두자. Wide & Deep 러닝은 가능한 많은 수의 feature value 값들을 가지는 많은 sparse feature column 들의 데이터세트(dataset)로 시도할 경우 더욱 강력할 것이다. 다시말해, Wide & Deep 러닝을 대규모의 실세계에 맞춰진 러닝 문제들을 어떻게 적용하는지에 대한 많은 아이디어를 위해 우리의 [research paper](#) 을 자유롭게 참고하라.

# TensorFlow Serving

## Introduction

텐서플로우 서빙은 제품 환경을 위해 디자인 되었으며, 머신러닝 모델을 위해 유연하며 고성능의 서빙 시스템을 제공합니다. 텐서플로우 서빙을 통해 같은 서버 아키텍쳐와 API를 유지하는 동안, 새로운 알고리즘과 실험을 쉽게 배포할 수 있습니다.

## Basic Serving Tutorial

텐서플로우 서빙 페이지의 [basic tutorial](#)을 통해 학습된 텐서플로우 모델을 어떻게 내보낼지 그리고 서버를 통해 모델을 어떻게 올릴 수 있는지에 대해 학습하실 수 있습니다.

## Advanced Serving Tutorial

텐서플로우 서빙 페이지의 [advanced tutorial](#)을 통해 동적으로 서버를 구축하는 방법을 배우고 최신 버전의 학습된 텐서플로우 모델을 경험하실 수 있습니다.

## Serving Inception Model Tutorial

텐서플로우 서빙 페이지의 [serving inception tutorial](#)을 통해 초기 모델을 텐서플로우 서빙과 Kubernetes으로 올리는 방법을 학습하실 수 있습니다.

# 컨볼루션 뉴럴 네트워크

**NOTE:** 이 튜토리얼은 텐서플로우에 능숙한 사용자를 대상으로 하며, 기계학습에 대한 전문 지식과 경험을 갖고 있다는 전제로 쓰였습니다.

## 개요

CIFAR-10 분류는 기계학습에서 흔히 사용되는 벤치마크 문제입니다. 이 분류 문제는 RGB 32x32 픽셀 이미지를 다음의 10개 카테고리로 분류하는 것이 목표입니다 : 비행기, 자동차, 새, 고양이, 사슴, 개, 개구리, 말, 배, 트럭.

더 자세한 설명을 원하신다면 [CIFAR-10 페이지](#)와 Alex Krizhevsky의 [기술 보고서](#)를 참조하세요.

## 목표

이 튜토리얼의 목표는 이미지를 인식하는 상대적으로 작은 [컨볼루션 뉴럴 네트워크]를 만드는 것입니다. 이 과정에서, 튜토리얼에서는

1. 네트워크 구조와 학습 및 평가의 표준적인 구성에 주목하고,
2. 더 크고 복잡한 모델에 대한 예제를 제공합니다.

CIFAR-10 분류가 선택된 이유는 더 큰 모델을 다루는 데에 필요한 텐서플로우의 많은 기능들을 연습하기에 충분히 복잡하기 때문입니다. 그와 동시에, 충분히 작은 모델이기 때문에 학습이 빨라 새로운 아이디어를 적용해보거나 새로운 테크닉을 실험해보기에 적합하기 때문입니다.

## 튜토리얼의 주안점

CIFAR-10 튜토리얼은 텐서플로우로 더 크고 복잡한 모델을 디자인하기 위한 몇몇의 주요 구성들을 설명합니다.

- 주요 수학적 요소 : [Convolution \(wiki\)](#), [rectified linear activations \(wiki\)](#)), [Max Pooling \(wiki\)](#) and [local response normalization](#) (Chapter 3.3 in [AlexNet paper](#)).
- 활성화(Activations)와 경사(gradients)의 손실(loss) 및 분포와 입력된 이미지를 포함하는 학습 중인 네트워크의 활동 [시각화](#)
- 학습된 변수의 [이동 평균\(moving average\)](#)을 계산하는 방법과 평가를 할 때 예측 성능을 향상시키기 위하여 이 평균들을 이용하는 방법
- 체계적으로 시간에 따라 감소하는 [학습 비율\(learning rate\) 스케줄](#)의 구현

- 디스크 자연과 비싼 이미지 전처리를 모델로부터 분리하기 위한 입력 데이터 큐의 선인출(prefetching)

또한 저희는 모델의 [다중-GPU 버전](#)을 제공합니다. 이 모델은 다음과 같은 사항들을 설명합니다:

- 다수의 GPU 카드에서 병렬로 훈련할 모델을 구성하기
- 다수의 GPU 간에 변수들을 공유하고 업데이트하기

우리는 이 튜토리얼이 TensorFlow로 영상(vision) 작업에 필요한 큰 CNN을 구축하기 위한 시발점이 되었으면 합니다.

## 모델 구조

CIFAR-10 튜토리얼의 모델은 컨볼루션과 비선형이 교차되어있는 다중 레이어 구조로 구성되어 있습니다. 이 레이어들 뒤로는 Softmax 분류기로 이어지는 Fully connected layer가 있습니다. 상위 몇몇 레이어를 제외하고, 이 모델은 [Alex Krizhevsky](#)가 만든 모델을 따르고 있습니다.

이 모델은 GPU에서 몇시간의 학습을 거친 후 최대 86%의 정확도를 달성하였습니다. 좀더 자세한 사항은 [아래](#)와 코드를 참조하세요. 이 모델은 1,068,298개의 학습 가능한 매개변수로 구성되어 있으며, 단일 이미지를 추론하는 데에 19.5M의 곱셈-덧셈 연산이 필요합니다.

## 코드 구성

이 튜토리얼의 코드는 [tensorflow/models/image/cifar10/](#)에 있습니다.

| 파일                                         | 목적                              |
|--------------------------------------------|---------------------------------|
| <a href="#">cifar10_input.py</a>           | CIFAR-10 바이너리 파일 포맷을 읽어들입니다.    |
| <a href="#">cifar10.py</a>                 | CIFAR-10 모델을 만듭니다.              |
| <a href="#">cifar10_train.py</a>           | CIFAR-10 모델을 CPU 혹은 GPU로 학습합니다. |
| <a href="#">cifar10_multi_gpu_train.py</a> | CIFAR-10 모델을 다중 GPU로 학습합니다.     |
| <a href="#">cifar10_eval.py</a>            | CIFAR-10 모델의 예측 성능을 평가합니다.      |

## CIFAR-10 모델

CIFAR-10 네트워크는 주로 '[cifar10.py](#)'에 들어있습니다. 전체 훈련 그래프는 약 765개의 연산을 포함합니다. 우리는 아래의 모듈들로 그래프를 구성하는 것이 가장 재사용성이 높은 코드를 만드는 방법임을 알게 되었습니다:

- 모델 입력:** 'inputs()' 와 'distorted\_inputs()'는 각각 평가와 훈련을 위한 CIFAR 이미지를 읽고 전처리를 하는 연산들을 추가합니다.

2. **모델 예측:** 'inference()'는 추론을 수행하는 연산들을 추가합니다. 예) 제공된 이미지에 대한 분류
3. **모델 훈련:** 'loss()'와 'train()'은 손실(loss)과 경사(gradients), 변수 업데이트와 시각화 요약을 계산하는 연산들을 추가합니다.

## 모델 입력

모델의 입력 부분은 CIFAR-10 바이너리 데이터 파일로부터 이미지를 읽는 'inputs()'와 'distorted\_inputs()'로 구성되어 있습니다. 이 데이터 파일들은 고정 바이트 길이 레코드를 담고있어, 우리는 `tf.FixedLengthRecordReader`를 사용합니다. 'Reader' 클래스가 어떻게 작동하는지 더 알고 싶으시다면 [Reading Data](#)를 참조하세요.

이미지들은 아래의 과정을 통하여 처리됩니다.

- 이미지는  $24 \times 24$  픽셀로 잘라냅니다. 훈련을 위하여 [무작위로](#) 잘라내거나 혹은 평가를 위하여 중심만 잘라냅니다.
- 동적 범위 내에 모델이 둔감해지도록 [대략적인 화이트닝](#)을 합니다

훈련을 위하여 추가적으로 일련의 무작위 왜곡을 적용하여 인공적으로 데이터 셋의 크기를 키웁니다:

- 이미지를 좌에서 우로 [무작위로 뒤집기](#)
- [이미지 밝기](#)를 무작위로 왜곡하기
- [이미지 대비](#)를 무작위로 왜곡하기

가능한 왜곡의 목록은 [Images](#) 페이지를 참조하세요. 또한 `image_summary`를 이미지에 붙여 [TensorBoard](#)에서 시각화 할 수 있도록 하였습니다. 이는 입력이 제대로 만들어 졌는지 확인하기 위한 좋은 연습이 될 것 입니다.



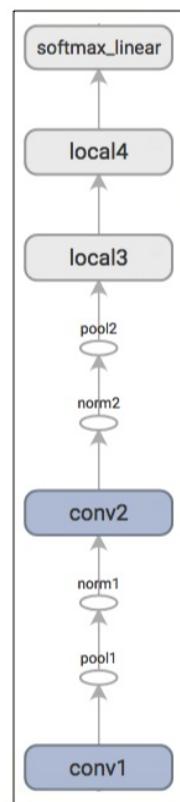
디스크에서 이미지를 읽고 왜곡을 하는 것은 적지 않은 양의 처리 시간이 필요할 수 있습니다. 이러한 작업이 훈련을 늦추는 것을 방지하기 위해, 우리는 이 작업을 16개의 독립된 스레드로 나누어 실행시킵니다. 이 스레드는 [TensorFlow 큐](#)를 계속해서 채웁니다.

## 모델 예측

모델의 예측 부분은 'inference()' 함수로 구성되어 있습니다. 이 함수는 예측의 로짓(logit)들을 계산하는 연산을 추가합니다. 모델의 해당 부분은 다음과 같이 구성되어 있습니다:

| 레이어 명          | 설명                                                                                 |
|----------------|------------------------------------------------------------------------------------|
| conv1          | 컨볼루션(convolution) 과 정류된 선형(rectified linear) 활성화 레이어.                              |
| pool1          | 최대 풀링(max pooling) 레이어.                                                            |
| norm1          | 지역 반응 정규화(local response normalization) 레이어.                                       |
| conv2          | 컨볼루션(convolution) 과 정류된 선형(rectified linear) 활성화 레이어.                              |
| norm2          | 지역 반응 정규화(local response normalization).                                           |
| pool2          | 최대 풀링(max pooling) 레이어.                                                            |
| local3         | 정류된 선형 활성화가 포함된 완전 연결 레이어(fully connected layer with rectified linear activation). |
| local4         | 정류된 선형 활성화가 포함된 완전 연결 레이어(fully connected layer with rectified linear activation). |
| softmax_linear | 로짓(logit)들을 생산하는 선형 변환(linear transformation)                                      |

아래의 그래프는 TensorBoard를 통해 생성된 추론(inference) 연산을 설명합니다.



연습: '추론(inference)'의 출력값은 정규화되지 않은 로짓(logit)입니다. `tf.nn.softmax()` 을 사용하여 정규화된 예측값을 리턴하도록 네트워크 구조를 수정해보세요.

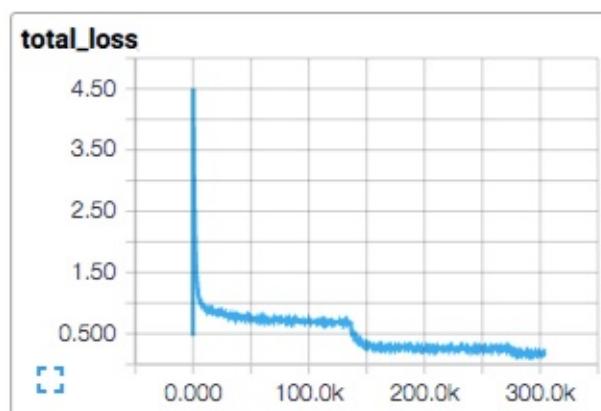
'inputs()'와 'inference()' 함수는 모델을 평가하는데 필요한 모든 컴포넌트들을 제공합니다. 이제 우리는 모델을 훈련하는 작업을 구축하는 것으로 초점을 옮겨봅시다.

연습: 'inference()'의 모델 구조는 [cuda-convnet](#)에서 명시하는 CIFAR-10 모델과 조금 다릅니다. 특히, Alex의 원본 모델의 최상위 레이어는 완전 연결(fully connected)이 아니라 국소 연결(locally connected) 되어있습니다. 최상위 레이어에서 국소 연결(locally connected) 구조를 정확하게 재현하도록 구조를 수정해보세요.

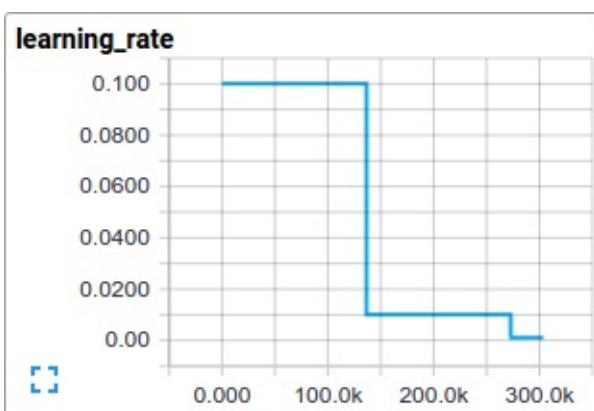
## 모델 훈련

N-way 분류를 수행하는 네트워크를 훈련시키는 일반적인 방법은 소프트맥스 회귀(*Softmax regression*)로 알려진 [다항 로지스틱 회귀\(multinomial logistic regression\)](#)입니다. 소프트맥스 회귀(Softmax regression)는 네트워크의 출력값에 softmax 비선형성을 적용하고, 정규화된 예측값과 [1-핫 인코딩\(1-hot encoding\)](#)된 라벨 사시의 크로스 엔트로피(cross-entropy)를 계산합니다. 규제(regularization)를 위하여, 우리는 모든 학습된 변수에 대하여 일반적인 [가중치 감소\(weight decay\)](#) 손실을 적용합니다. 모델의 목적함수는 크로스 엔트로피 손실의 합과 'loss()' 함수에 의해 리턴되는, 모든 가중치 감소(weight decay) 텁의 합입니다.

우리는 TensorBoard의 [scalar\\_summary](#) 를 사용하여 이를 시각화 하였습니다.



우리는 표준적인 [경사 강하\(gradients descent\)](#) 알고리즘 (다른 방법을 보려면 [Training](#)을 참조)을 사용하여 모델을 훈련합니다. 시간에 따라 [급격하게 감소\(exponentially decays\)](#)하는 학습 비율(learning rate)을 사용하였습니다.



'train()' 함수는 경사(gradients)를 계산하고 학습된 변수를 업데이트함으로써 목표를 최소화하는데에 필요한 기능을 추가합니다 ( 자세한 사항은 [GradientDescentOptimizer](#) 참조). 이 함수는 하나의 이미지 배치(batch)에 대하여 모델을 훈련하고 업데이트하는데 필요한 모든 연산을 실행하는

기능을 리턴해줍니다.

## 모델 실행 및 훈련 해보기

모델을 만들었으니, 이제 이 모델을 실행해보고 `cifar10_train.py` 스크립트를 사용하여 훈련 작업을 실행해봅시다.

```
python cifar10_train.py
```

**참고:** 여러분이 CIFAR-10 튜토리얼에서 처음 어떤 타겟을 실행하면, CIFAR-10 데이터셋이 자동으로 다운로드 됩니다. 데이터셋은 160MB 이하입니다. 아마 그동안 당신은 커피 한 잔이 떠오를지도 모릅니다.

출력을 보아야 합니다:

```
Filling queue with 20000 CIFAR images before starting to train. This will take a few minutes.
2015-11-04 11:45:45.927302: step 0, loss = 4.68 (2.0 examples/sec; 64.221 sec/batch)
2015-11-04 11:45:49.133065: step 10, loss = 4.66 (533.8 examples/sec; 0.240 sec/batch)
2015-11-04 11:45:51.397710: step 20, loss = 4.64 (597.4 examples/sec; 0.214 sec/batch)
2015-11-04 11:45:54.446850: step 30, loss = 4.62 (391.0 examples/sec; 0.327 sec/batch)
2015-11-04 11:45:57.152676: step 40, loss = 4.61 (430.2 examples/sec; 0.298 sec/batch)
2015-11-04 11:46:00.437717: step 50, loss = 4.59 (406.4 examples/sec; 0.315 sec/batch)
...
```

스크립트는 매 10단계마다 총 손실(total loss) 뿐만 아니라 데이터의 마지막 배치가 처리될 때의 처리속도도 보고합니다. 몇 가지 조언:

- 데이터의 첫 배치는 전처리 스레드가 20,000장의 처리된 CIFAR 이미지를 셔플링(shuffling) 큐에 채워넣는 만큼 지나치게 느릴 수 있습니다 (예를 들어, 수 분).
- 보고된 손실은 가장 최근 배치의 평균 손실입니다. 이 손실은 크로스 엔트로피의 합과 모든 가중치 감소(weight decay) 텀의 합임을 기억하세요.
- 배치 하나의 처리속도에 주목하세요. 위의 수치는 Tesla K40c로 얻은 값입니다. CPU에서 실행한다면, 좀더 느린 성능을 보일 것입니다.

**연습:** 실험할 때, 훈련의 첫 스텝이 오랜 시간이 소요되는 것이 때때로 짜증날 수 있습니다. 초기에 큐를 채우는 이미지의 수를 줄여보세요. `cifar10.py`에서 `NUM_EXAMPLES_PER_EPOCH_FOR_TRAIN`을 검색해보세요.

`cifar10_train.py`는 주기적으로 모든 모델 파라미터를 체크포인트 파일(checkpoint files)에 저장합니다. 하지만 모델 자체를 평가하지는 않습니다. 체크포인트 파일은 `cifar10_eval.py`에서 예측 성능을 측정하는데에 사용됩니다.(아래에 있는 [모델을 평가하기](#)를 보세요).

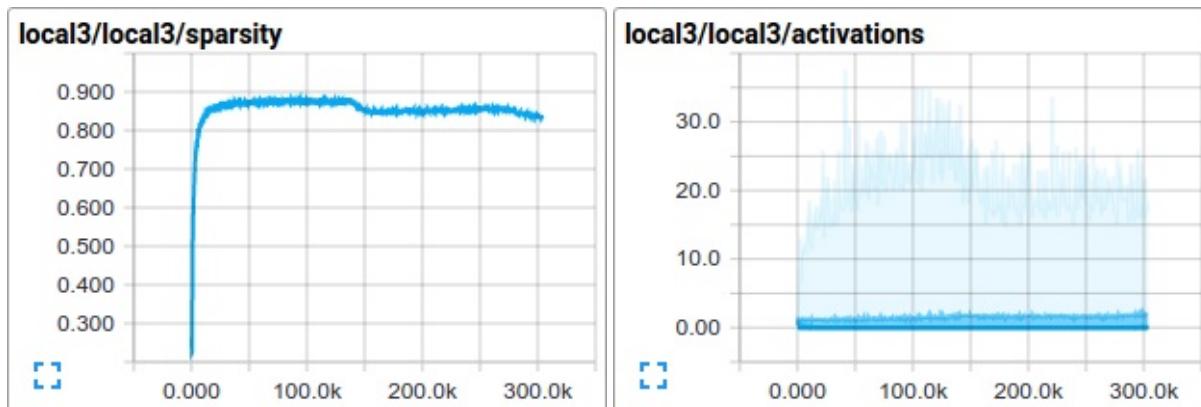
이전 단계들을 모두 따라왔다면, 당신은 CIFAR-10 모델의 훈련을 시작한 것입니다! [축하합니다!](#)

`cifar10_train.py`에서 리턴되는 terminal 텍스트는 모델을 어떻게 훈련할 것인지에 대한 최소한의 통찰(insight)을 제공합니다. 우리는 훈련하는 동안 모델에 대한 더욱 많은 통찰(insight)을 원합니다:

- 손실(loss)이 정말 감소하는지 혹은 단지 노이즈였는지?
- 모델이 적절한 이미지를 제공받는지?
- 경사(gradients), 활성화(activations), 그리고 가중치(weights)는 합당한지?
- 현재의 학습 비율(learning rate)은 무엇인지?

[TensorBoard](#)는 기능적으로, `cifar10_train.py`의 `SummaryWriter`를 통해 주기적으로 데이터를 추출하여 표시합니다.

예를 들어, 우리는 훈련하는 동안 활성화(activation)의 분포와, `local3` feature들의 희박함(sparsity)의 분포가 어떻게 진화(evolve) 하는지 볼 수 있습니다:



총 손실(total loss)뿐만 아니라, 개별적인 손실 함수(loss function)들은 특히 시간 경과에 따라 흥미롭습니다. 그러나, 손실(loss)은 훈련에 사용되는 작은 배치 사이즈에 따라 상당히 많은 양의 노이즈를 나타냅니다. 이 연습에서 우리는 원본 값에 더하여 그들의 이동 평균을 시각화하는데 매우 유용함을 발견하였습니다. 이러한 목적을 위하여 어떻게 스크립트가 [ExponentialMovingAverage](#)를 사용하는지 보세요.

## 모델 평가하기

이제 남아있는 데이터 셋에 대하여 학습된 모델이 얼마나 잘 작동하는지 평가해봅시다. 모델은 `cifar10_eval.py` 스크립트에 의해 평가됩니다. 이 스크립트는 `inference()` 함수로 모델을 구축하고 CIFAR-10 평가 데이터셋에 있는 10,000장의 이미지를 사용하여 1에서의 정밀도(*Precision at 1*): 가장 높은 예측값이 이미지의 실제 라벨과 얼마나 자주 일치 하는지를 계산합니다.

훈련하는 동안 모델이 어떻게 개선되는지 추적하기 위하여, `cifar10_train.py`가 생성하는 최근의 체크포인트 파일에서 평가 스크립트가 주기적으로 실행됩니다.

```
python cifar10_eval.py
```

같은 GPU에서 평가와 훈련 바이너리를 실행하지 않도록 주의하세요. 아마 메모리가 부족할 것입니다. 분리된 GPU에서 각각 실행하거나 같은 GPU에서 평가를 하는 동안에는 훈련을 잠시 중단하는 것을 고려하세요.

아래와 같은 결과를 보게 됩니다.

```
2015-11-06 08:30:44.391206: precision @ 1 = 0.860
```

```
...
```

스크립트는 주기적으로 오직 정밀도@1(precision@1)을 리턴합니다 -- 이 경우에는 86%의 정확도를 리턴하였습니다. 또한 `cifar10_eval.py` 는 TensorBoard에서 시각화를 해볼 수 있는 요약(summaries)을 내보냅니다. 이 요약들은 평가를 하는 동안 모델에 대한 추가적인 이해를 제공합니다.

훈련 스크립트는 모든 학습된 변수의 [이동평균\(moving average\)](#) 버전을 계산합니다. 평가 스크립트는 모든 학습된 모델 파라미터를 이동 평균 버전으로 치환합니다. 이 치환은 평가시 모델 성능을 향상시킵니다.

**연습:** 평균 파라미터를 사용하는 것은 예측 성능을 정밀도 @ 1(precision @ 1)에서 약 3%정도 향상시킬 수 있습니다. `cifar10_eval.py` 를 수정하여 평균 파라미터를 사용하지 않도록 해보고 예측 성능이 떨어지는 것을 확인해보세요.

## 다중 GPU 카드를 사용하여 모델 훈련하기

최신 워크스테이션은 과학적인 계산을 위하여 다수의 GPU를 갖추고 있습니다. TensorFlow는 이러한 환경을 이용하여 다수의 GPU 카드에서 동시에 훈련 연산을 실행할 수 있습니다.

병렬로 모델을 훈련하려면, 훈련 과정을 분산된 방식으로 조직할 필요가 있습니다. 다음에 나오는 모델 복제본이라는 용어는 데이터의 하위 집합으로 훈련한 모델의 복사본 중의 하나입니다.

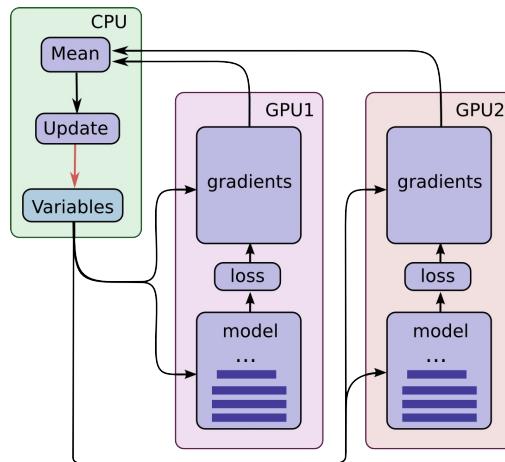
단순하게 비동기(asynchronous) 모델 파라미터 업데이트를 적용하면 최선의 값에 비해 조금 떨어지는 차선의 훈련 성능을 얻을 수 있습니다. 독립된 모델 복제본은 오래된 모델 파라미터의 복사본으로 학습되어 있기 때문입니다. 반대로, 완전한 동기(synchronous) 업데이트의 경우는 가장 느린 모델 복제본 만큼 느릴 것 입니다.

다수의 GPU 카드를 갖춘 워크스테이션에서, 각각의 GPU는 비슷한 속도와 CIFAR-10 모델 전체를 실행할 만한 충분한 메모리를 탑재하고 있을 것입니다. 그러므로, 우리는 우리의 훈련 시스템을 디자인 하는데에 아래와 같은 규칙을 따릅니다:

- 각각의 GPU에 개별의 모델 복제본을 올립니다.

- 모든 GPU가 한 배치의 데이터를 처리 완료 할때까지 기다려 모델 파라미터 업데이트를 동기적으로 수행합니다.

이 모델의 다이어그램은 다음과 같습니다.



각각의 GPU는 추론(inference)뿐만 아니라 경사(gradients)도 독자적인 데이터 배치를 사용하여 계산한다는 것에 주의하세요. 이러한 설정은 GPU간의 더 큰 데이터 배치를 효과적으로 분배 할 수 있도록 해줍니다.

이러한 설정은 모든 GPU들이 모델 파라미터를 공유해야 합니다. GPU간의 데이터 전송이 꽤 느린 작업이라는 사실은 잘 알려져 있습니다. 이러한 이유로, 우리는 모든 모델 파라미터를 CPU 위에 저장하고 업데이트 하기로 정하였습니다(위의 그림에서 녹색 박스). 모든 GPU에서 새로운 데이터 배치가 처리 되고 난 뒤 갓 생성된 모델 파라미터의 집합은 각각의 GPU로 전송됩니다.

GPU들은 연산에 동기화되어있습니다. 모든 경사(gradients)를 GPU들로부터 축적하여 평균을 구합니다(녹색 박스를 보세요). 모델 파라미터는 모든 모델 복제본들의 경사(gradients)의 평균을 사용하여 업데이트됩니다.

## 장치에 변수와 연산 배치시키기

장치에 변수와 연산을 배치시키는 것은 조금 특별한 추상화(abstraction)가 필요합니다.

우리에게 필요한 첫번째 추상화는 단일 모델 복제본에 대한 추론(inference)과 경사(gradients)를 계산하기 위한 함수입니다. 코드에서 우리는 이 추상화를 "타워"라는 용어로 부릅니다. 우리는 각각의 타워에 두가지 속성을 설정해야 합니다:

- 타워 안의 모든 연산들에 대한 유일한 이름. `tf.name_scope()` 는 scope를 붙여 이런 유일한 이름을 제공합니다. 예를 들면, 첫번째 타워의 모든 연산들은 `tower_0` 가 앞에 붙습니다. e.g. `tower_0/conv1/Conv2D`.
- 타워 안의 연산을 실행할 선호하는 하드웨어 장치. `tf.device()` 는 이를 특정해줍니다. 예를 들면, 첫번째 타워의 모든 연산들은 `device('/gpu:0')` 스코프 안에 존재하게 됩니다. 이는 해당 연산들을 첫번째 GPU에서 실행하라는 것을 나타냅니다.

모든 변수는 다중-GPU 버전에서 공유 되기 위하여 CPU에 고정되어있고, `tf.get_variable()`를 통하여 접근할 수 있습니다. 자세한 방법은 [변수 공유하기\(Sharing Variables\)](#)를 보세요.

## 다수의 GPU 카드에서 모델을 실행하고 훈련하기

만약 당신의 머신에 여러 대의 GPU 카드가 있다면 `cifar10_multi_gpu_train` 스크립트를 이용하여 모델을 좀더 빠르게 학습하는데 사용할 수 있습니다. 이 버전의 훈련 스크립트는 다수의 GPU 카드에 모델을 병렬화합니다.

```
python cifar10_multi_gpu_train.py --num_gpus=2
```

기본 GPU 카드 숫자는 1로 설정되어 있다는 것을 참고하세요. 추가적으로, 당신의 머신에 하나의 GPU만 사용가능하다면, 당신이 더욱 많은 GPU를 요청하더라도 모든 계산은 그 하나의 GPU에 위치하게 됩니다.

**연습:** `cifar10_train.py` 의 기본 설정은 128의 배치 크기로 실행 하는 것입니다. 64 배치 크기로 2대의 GPU를 사용하여 `cifar10_multi_gpu_train.py` 을 실행해보고 훈련 속도를 비교 해보세요.

## 다음 단계

[축하합니다!](#) 당신은 CIFAR-10 튜토리얼을 완료하였습니다.

만약 지금 당신만의 이미지 분류 시스템을 개발하고 훈련하는 데에 흥미가 있다면, 이 튜토리얼을 포크(fork)하고 당신의 이미지 분류 문제에 맞춰 구성요소를 교체하는 것을 추천합니다.

**연습:** [Street View House Numbers \(SVHN\)](#) 데이터셋을 다운로드 하세요. CIFAR-10 튜토리얼을 포크(fork)하고 SVHN을 입력 데이터로 교체하세요. 예측 성능을 향상시키기 위하여 네트워크 아키텍처를 조정해보세요.

# 이미지 인식

우리의 뇌를 생각하면 시각적으로 인식하는 일은 쉬워 보인다. 보통 사람이라면 사자와 재규어를 구별할 줄 알고 표지판을 읽을 수 있으며 다른 사람의 얼굴 또한 어렵지 않게 인식할 수 있다. 그러나 이는 이미지를 인식하는 뇌의 능력이 놀라울 정도로 뛰어나기 때문에 가능한 것이지 이와 같은 일을 컴퓨터를 통해 해결하는 것은 매우 어려운 문제다.

지난 수 년간 기계학습 분야는 이미지 인식에 대해 엄청난 진전을 이루어 냈다. 특히 딥 러닝 기법의 하나인 [convolutional neural network](#)를 통해 혁신적인 성과를 거두었는데, 일부 분야에서는 사람의 인식 능력에 버금가거나 더 나은 결과를 보여주기도 했다.

연구자들은 학계에서 시작된 컴퓨터 비전 프로젝트인 [ImageNet](#)에서 자신들의 작업을 검증해왔고, 그들의 연구는 [QuocNet](#), [AlexNet](#), [Inception \(GoogLeNet\)](#), [BN-Inception-v2](#)와 같은 최신식 모델을 만들어냈다. 구글 내부 연구자와 외부 연구자 모두 이러한 모델을 설명하는 자료를 발표해 왔지만 자료가 널리 배포되고 있지는 않다. 그래서 [TensorFlow](#)는 구글이 개발한 이미지 인식의 가장 최신 모델인 [Inception-v3](#)를 활용하는 코드를 공개한다.

[Inception-v3](#)는 [ImageNet](#)의 Large Visual Recognition Challenge에서 2012년 데이터를 사용하여 훈련된 모델이다. 모든 이미지를 "얼룩말", "달마시안", "식기세척기"와 같은 [1000 classes](#)로 분류하는 것이 컴퓨터 비전의 표준 작업이다. 다음의 예는 [AlexNet](#)이 몇 가지 사진을 분류한 결과이다:



모델의 성능을 비교할 때는 "top-5 error rate"를 측정한다. 이는 모델이 가장 높은 확률로 예측한 5 가지 예측이 정답이 아닌 빈도를 검토하는 것이다. 2012년 검증 데이터 세트에서 나타난 각 모델의 top-5 error rate는 [AlexNet](#)이 15.3%, [BN-Inception-v2](#)이 6.66%였고 [Inception-v3](#)는 3.46%를 달성했다.

ImageNet 챌린지에서 사람의 성과는 어떨까? Andrej Karpathy가 [blog post](#)에서 밝힌 바에 의하면 그의 top-5 error rate는 5.1%였다고 한다.

본 튜토리얼은 [Inception-v3](#)를 사용하는 방법을 알려줄 것이다. 먼저 Python이나 C++로 본 모델을 사용해서 이미지를 [1000 classes](#)로 분류하는 방법을 배운다. 그리고 이 모델을 통해 다른 이미지 인식 문제에서 다시 활용될 수 있는 고수준의 특징을 추출하는 방법 또한 논의할 것이다.

커뮤니티에서 이 모델을 어떤 모습으로 활용할지 기대되는 바이다.

## Python API로 사용하기

프로그램이 처음 실행될 때 `classify_image.py` 는 `tensorflow.org` 로 부터 훈련된 모델을 다운로드 받는다. 필요한 하드디스크의 여유 공간은 200MB이다.

PIP 패키지에서 TensorFlow를 설치하고 터미널을 TensorFlow의 root 디렉토리로 설정한 후에 다음 명령어를 실행한다.

```
cd tensorflow/models/image/imagenet
python classify_image.py
```

위 명령어는 다운로드 받았던 판다 곰의 사진을 분류한다.



만약 모델이 올바르게 작동한다면, 다음과 같은 내용이 출력된다:

```
giant panda, panda, panda bear, coon bear, Ailuropoda melanoleuca (score = 0.88493)
indri, indris, Indri indri, Indri brevicaudatus (score = 0.00878)
lesser panda, red panda, panda, bear cat, cat bear, Ailurus fulgens (score = 0.00317)
custard apple (score = 0.00149)
earthstar (score = 0.00127)
```

다른 JPEG 이미지를 추가하려면 `--image_file` 인자를 수정하면 된다.

모델 데이터를 다른 디렉토리에 다운로드 받았다면, `--model_dir` 를 다운로드 받은 디렉토리로 지정해야 한다.

## C++ API로 사용하기

C++의 프로덕션 환경에서도 [Inception-v3](#) 모델을 사용할 수 있다. 모델을 정의하는 GraphDef를 담고 있는 아카이브를 다운로드 받는 방법은 아래와 같다: (단, TensorFlow repository의 root 디렉토리에서 실행한다)

```
wget https://storage.googleapis.com/download.tensorflow.org/models/inception_dec_2015.zip
unzip tensorflow/examples/label_image/data/inception_dec_2015.zip -d tensorflow/examples/label_image/data/
```

다음으로 그래프를 불러오고 실행할 수 있는 코드를 담고 있는 C++ 바이너리를 컴파일 해야 한다. 만약 [the instructions to download the source installation of TensorFlow](#)에 나와 있는 지시사항을 자신의 플랫폼에 맞게 이행했다면, 터미널에서 다음과 같은 명령어를 실행하여 예제를 빌드할 수 있다:

```
bazel build tensorflow/examples/label_image/...
```

위 명령어가 입력되면 실행 가능한 바이너리 파일이 생성될 것이다. 파일을 실행하는 명령어는 다음과 같다:

```
bazel-bin/tensorflow/examples/label_image/label_image
```

위 명령어는 프레임워크가 함께 전달되는 기본(default) 예제 이미지를 사용하며 아래와 유사한 내용이 출력된다:

```
I tensorflow/examples/label_image/main.cc:200] military uniform (866): 0.647296
I tensorflow/examples/label_image/main.cc:200] suit (794): 0.0477196
I tensorflow/examples/label_image/main.cc:200] academic gown (896): 0.0232411
I tensorflow/examples/label_image/main.cc:200] bow tie (817): 0.0157356
I tensorflow/examples/label_image/main.cc:200] bolo tie (940): 0.0145024
```

기본 제공 이미지인 [Admiral Grace Hopper](#) 사진을 사용한 결과이다. 0.6이라는 높은 점수로 모델의 네트워크가 군복을 입고 있는 여성을 올바르게 인식하고 있음을 확인할 수 있다.



다음으로 --image= 인자를 추가하여 본인이 갖고 있는 이미지로 테스트해 볼 수 있다. 예를 들면 다음과 같다:

```
bazel-bin/tensorflow/examples/label_image/label_image --image=my_image.png
```

`tensorflow/examples/label_image/main.cc` 파일을 살펴보면 명령어가 어떻게 작동하는지 알 수 있다. 이 코드를 통해 사용자가 TensorFlow 라이브러리를 자신의 애플리케이션에 사용하는 데 도움이 되길 바라며 주요 함수들을 차례차례 살펴보자.

커맨드 라인 플래그(command line flags)는 파일을 불러온 위치와 입력된 이미지의 속성을 조정하는 기능을 한다. 모델은 정사각형 299x299 사이즈의 RGB 이미지를 취급하기 때문에 이를 `input_width` 와 `input_height` 플래그라고 한다. 그리고 픽셀(pixel) 값을 0과 255사이의 정수 값(integer)에서 그래프를 사용하기 위한 실수 값(float)으로 스케일링(scaling)해야 한다. 스케일링은 `input_mean` 과 `input_std` 플래그로 조절한다. 각 픽셀 값에서 `input_mean` 을 빼고 난 후 `input_std` 로 나눈다.

이러한 값들이 마술처럼 신기해 보일 수 있는데, 이는 모델을 만든 원작자가 훈련용 입력 이미지로서 사용하고 싶은 것으로서 정의했던 내용일 뿐이다. 만일 사용자가 스스로 훈련시킨 그래프가 있다면, 자신의 훈련 프로세스에 적합하도록 사용자가 원하는 값으로 조정하면 된다.

이러한 값들이 [ `ReadTensorFromFile()` ]

([https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/label\\_image/main.cc#L88](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/label_image/main.cc#L88))의 함수를 통해 이미지에 어떻게 적용되는지 확인해 보자.

```
// Given an image file name, read in the data, try to decode it as an image,
// resize it to the requested size, and then scale the values as desired.
Status ReadTensorFromFile(string file_name, const int input_height,
 const int input_width, const float input_mean,
 const float input_std,
 std::vector<Tensor>* out_tensors) {
 tensorflow::GraphDefBuilder b;
```

가장 먼저 실행하거나 불러올 모델을 설정하기 위해 `GraphDefBuilder` 를 만든다.

```
string input_name = "file_reader";
string output_name = "normalized";
tensorflow::Node* file_reader =
 tensorflow::ops::ReadFile(tensorflow::ops::Const(file_name, b.opts()),
 b.opts().WithName(input_name));
```

그 다음에 실행할 소형 모델(small model)의 노드를 생성한다. 이는 픽셀 값을 불러오고, 변경하고, 스케일링하는 데 사용되며 메인 모델(main model)의 입력값으로 쓰일 것이다. 첫 번째로 만든 노드는 `Const op`(그래프 위의 노드를 의미하는 operation의 줄임말)이며 이는 우리가 불러 올 이미지의 파일 이름이 담긴 `Tensor`를 갖고 있다. 이것은 `ReadFile op`에게 첫 번째 입력값으로 전달된다. `op`를 생성하는 모든 함수의 마지막 인자로 `b.opts()` 를 전달할 것임을 알 수 있을 것이다. 이 인자는 해당 노드가 `GraphDefBuilder` 가 담고 있는 모델 정의에 반드시 추가되도록 한다. 또한

`WithName()` 함수를 `b.opts()`에게 호출하여 `ReadFile` 연산자를 지정할 것이다. 이것은 노드의 이름을 지정하는 것인데, 사실 사용자가 따로 수행하지 않더라도 자동 이름이 생성되어 할당되므로 필수적인 과정은 아니다. 하지만 디버깅에 도움이 되는 과정이다.

```
// Now try to figure out what kind of file it is and decode it.
const int wanted_channels = 3;
tensorflow::Node* image_reader;
if (tensorflow::StringPiece(file_name).ends_with(".png")) {
 image_reader = tensorflow::ops::DecodePng(
 file_reader,
 b.opts().WithAttr("channels", wanted_channels).WithName("png_reader"));
} else {
 // Assume if it's not a PNG then it must be a JPEG.
 image_reader = tensorflow::ops::DecodeJpeg(
 file_reader,
 b.opts().WithAttr("channels", wanted_channels).WithName("jpeg_reader"));
}
// Now cast the image data to float so we can do normal math on it.
tensorflow::Node* float_caster = tensorflow::ops::Cast(
 image_reader, tensorflow::DT_FLOAT, b.opts().WithName("float_caster"));
// The convention for image ops in TensorFlow is that all images are expected
// to be in batches, so that they're four-dimensional arrays with indices of
// [batch, height, width, channel]. Because we only have a single image, we
// have to add a batch dimension of 1 to the start with ExpandDims().
tensorflow::Node* dims_expander = tensorflow::ops::ExpandDims(
 float_caster, tensorflow::ops::Const(0, b.opts()), b.opts());
// Bilinearly resize the image to fit the required dimensions.
tensorflow::Node* resized = tensorflow::ops::ResizeBilinear(
 dims_expander, tensorflow::ops::Const({input_height, input_width},
 b.opts().WithName("size")),
 b.opts());
// Subtract the mean and divide by the scale.
tensorflow::ops::Div(
 tensorflow::ops::Sub(
 resized, tensorflow::ops::Const({input_mean}, b.opts()), b.opts()),
 tensorflow::ops::Const({input_std}, b.opts()),
 b.opts().WithName(output_name));
```

그리고 나서 계속해서 노드를 추가하여, 파일 데이터를 이미지로 해독하고, 정수 값을 실수 값으로 바꾸고(scaling), 값의 크기를 변경하고(resizing), 마지막으로 픽셀값을 빼고 나누는 연산 과정을 거친다.

```
// This runs the GraphDef network definition that we've just constructed, and
// returns the results in the output tensor.
tensorflow::GraphDef graph;
TF_RETURN_IF_ERROR(b.ToGraphDef(&graph));
```

코드 마지막 줄에 변수 `b`에 담긴 모델 정의를 갖게 된다. `ToGraphDef()` 함수를 사용해서 변수 `b`를 완전한 그래프 정의로 변환한다.

```
std::unique_ptr<tensorflow::Session> session(
 tensorflow::NewSession(tensorflow::SessionOptions()));
TF_RETURN_IF_ERROR(session->Create(graph));
TF_RETURN_IF_ERROR(session->Run({}, {output_name}, {}, out_tensors));
return Status::OK();
```

그 다음으로 `Session` 오브젝트를 생성한다. 이는 그래프를 구현하는 인터페이스 오브젝트다. `Session` 오브젝트를 실행하여 어떤 노드로부터 출력을 얻을 것인지, 출력된 데이터를 어디에 둘지 설정한다.

이러한 작업은 `Tensor` 오브젝트에 대한 벡터 값을 준다 (벡터 값은 단 한 개의 오브젝트일 것이다). 여기서 `Tensor`의 의미는 다차원 배열로 생각하면 된다. `Tensor`는 높이 299pixel, 너비 299pixel 그리고 실수 값이 있는 3가지 채널 이미지를 속성으로 가진다. 만약 사용자가 만든 프로젝트에 스스로 제작한 이미지 프로세싱 프레임워크가 있다면 그것을 대신 사용해도 된다. 메인 그래프에 이미지를 넣기 전에 똑같은 변환을 적용할 수만 있다면 말이다.

지금까지 설명한 것은 C++에서 작은 TensorFlow 그래프를 그리는 간단한 예제이다. 그러나 미리 훈련된 Inception 모델에서는 파일에서 훨씬 더 큰 정의를 불러와야 하는데, 이때는 `LoadGraph()` 함수를 이용한다.

```
// Reads a model graph definition from disk, and creates a session object you
// can use to run it.
Status LoadGraph(string graph_file_name,
 std::unique_ptr<tensorflow::Session>* session) {
 tensorflow::GraphDef graph_def;
 Status load_graph_status =
 ReadBinaryProto(tensorflow::Env::Default(), graph_file_name, &graph_def);
 if (!load_graph_status.ok()) {
 return tensorflow::errors::NotFound("Failed to load compute graph at '",
 graph_file_name, "'");
 }
}
```

이미지를 불러오는 코드를 검토해봤다면 코드의 많은 부분이 익숙할 것이다. `GraphDef` 오브젝트를 생성하기 위해 `GraphDefBuilder`를 사용하기보다는 직접 `GraphDef`를 담고 있는 protobuf 파일을 불러온다.

```

session->reset(tensorflow::NewSession(tensorflow::SessionOptions()));
Status session_create_status = (*session)->Create(graph_def);
if (!session_create_status.ok()) {
 return session_create_status;
}
return Status::OK();
}

```

그리고 나서 `GraphDef`로부터 `Session` 오브젝트를 만들고 호출 함수(caller)에 전달하여 나중에 실행할 수 있도록 한다.

`GetTopLabels()` 함수는 이미지를 불러오는 다른 함수와 매우 유사하나 여기서는 메인 그래프를 그리고 그것을 가장 높은 점수의 레이블들로 분류한 리스트로 바꾼다는 점이 다르다. 이미지 불러오는 다른 함수처럼 `GetTopLabels()` 함수 또한 `GraphDefBuilder`를 만들고, `GraphDefBuilder`에 노드 여러 개를 더한 다음 한 쌍의 아웃풋 `tensors`를 얻기 위해 짧은 그래프를 그린다. 한 쌍의 아웃풋 `tensors`는 가장 높은 결과의 정렬된 점수와 인덱스 포지션을 나타낸다.

```

// Analyzes the output of the Inception graph to retrieve the highest scores and
// their positions in the tensor, which correspond to categories.
Status GetTopLabels(const std::vector<Tensor>& outputs, int how_many_labels,
 Tensor* indices, Tensor* scores) {
 tensorflow::GraphDefBuilder b;
 string output_name = "top_k";
 tensorflow::ops::TopK(tensorflow::ops::Const(outputs[0], b.opts()),
 how_many_labels, b.opts().WithName(output_name));
 // This runs the GraphDef network definition that we've just constructed, and
 // returns the results in the output tensors.
 tensorflow::GraphDef graph;
 TF_RETURN_IF_ERROR(b.ToGraphDef(&graph));
 std::unique_ptr<tensorflow::Session> session(
 tensorflow::NewSession(tensorflow::SessionOptions()));
 TF_RETURN_IF_ERROR(session->Create(graph));
 // The TopK node returns two outputs, the scores and their original indices,
 // so we have to append :0 and :1 to specify them both.
 std::vector<Tensor> out_tensors;
 TF_RETURN_IF_ERROR(session->Run({}, {output_name + ":0", output_name + ":1"},
 {}, &out_tensors));
 *scores = out_tensors[0];
 *indices = out_tensors[1];
 return Status::OK();
}

```

`PrintTopLabels()` 함수는 이러한 정렬된 결과값을 입력 받아서 그 값을 좀 더 친절한 방법으로 출력한다. `CheckTopLabel()` 함수도 이와 매우 비슷한데, 디버깅 목적이 포함되어 있다. 상단 레이블(top label)이 우리가 예상하는 값임을 점검한다.

마지막으로 `main()` 함수는 이러한 모든 함수의 호출을 하나로 묶는 함수이다.

```

int main(int argc, char* argv[]) {
 // We need to call this to set up global state for TensorFlow.
 tensorflow::port::InitMain(argv[0], &argc, &argv);
 Status s = tensorflow::ParseCommandLineFlags(&argc, argv);
 if (!s.ok()) {
 LOG(ERROR) << "Error parsing command line flags: " << s.ToString();
 return -1;
 }

 // First we load and initialize the model.
 std::unique_ptr<tensorflow::Session> session;
 string graph_path = tensorflow::io::JoinPath(FLAGS_root_dir, FLAGS_graph);
 Status load_graph_status = LoadGraph(graph_path, &session);
 if (!load_graph_status.ok()) {
 LOG(ERROR) << load_graph_status;
 return -1;
 }
}

```

메인 그래프를 불러온다.

```

// Get the image from disk as a float array of numbers, resized and normalized
// to the specifications the main graph expects.
std::vector<Tensor> resized_tensors;
string image_path = tensorflow::io::JoinPath(FLAGS_root_dir, FLAGS_image);
Status read_tensor_status = ReadTensorFromFile(
 image_path, FLAGS_input_height, FLAGS_input_width, FLAGS_input_mean,
 FLAGS_input_std, &resized_tensors);
if (!read_tensor_status.ok()) {
 LOG(ERROR) << read_tensor_status;
 return -1;
}
const Tensor& resized_tensor = resized_tensors[0];

```

입력된 이미지를 불러오고, 사이즈를 조정(resize)하고 처리한다.

```

// Actually run the image through the model.
std::vector<Tensor> outputs;
Status run_status = session->Run({{FLAGS_input_layer, resized_tensor}},
 {FLAGS_output_layer}, {}, &outputs);
if (!run_status.ok()) {
 LOG(ERROR) << "Running model failed: " << run_status;
 return -1;
}

```

입력된 이미지를 사용하여, 불러온 그래프를 구현한다(Here we run the loaded graph with the image as an input).

```
// This is for automated testing to make sure we get the expected result with
// the default settings. We know that label 866 (military uniform) should be
// the top label for the Admiral Hopper image.
if (FLAGS_self_test) {
 bool expected_matches;
 Status check_status = CheckTopLabel(outputs, 866, &expected_matches);
 if (!check_status.ok()) {
 LOG(ERROR) << "Running check failed: " << check_status;
 return -1;
 }
 if (!expected_matches) {
 LOG(ERROR) << "Self-test failed!";
 return -1;
 }
}
```

테스트 목적으로 우리가 예상하는 결과를 얻도록 점검할 수 있다.

```
// Do something interesting with the results we've generated.
Status print_status = PrintTopLabels(outputs, FLAGS_labels);
```

마지막으로 우리가 찾은 레이블을 출력한다.

```
if (!print_status.ok()) {
 LOG(ERROR) << "Running print failed: " << print_status;
 return -1;
}
```

여기서 오류 처리는 TensorFlow의 `Status` 오브젝트를 사용한다. 이 오브젝트는 매우 간편한데, `ok()` 함수를 사용해서 오류가 발생했는지 사용자에게 알려주고 오류 메시지를 출력해준다.

지금까지 대상을 인식하는 것에 대해 설명했지만, 다른 모든 영역에서도 사용자가 발견하거나 스스로 훈련시킨 다른 모델을 사용하여 이와 유사한 코드를 활용할 수 있다. 이 예제로 인해서 사용자들이 TensorFlow를 자신의 프로젝트에 사용하는 데 도움이 되길 바란다.

**연습문제:** 하나의 문제를 잘 풀 수 있다면, 그것을 활용해서 그 밖의 관련된 다른 문제를 풀 때 활용할 수 있다. 이를 전이학습(transfer learning)이라고 한다. 여기서 전이학습을 하는 한 가지 방법은 네트워크의 최종 분류 레이어를 제거하고 2048 차원의 벡터 값인 `next-to-last layer of the CNN`을 추출하는 것이다. 이에 대한 안내 문서는 [in the how-to section](#)을 참조하면 된다.

## 더 많은 학습을 위한 자료

신경망(neural network)에 대해 전반적으로 알고 싶다면, Michael Nielsen의 [free online book](#)이 훌륭한 자료가 될 것이다. convolutional neural network에 한해서는 Chris Olah의 [nice blog posts](#)와 Michael Nielsen의 [great chapter](#)을 참조하면 된다.

convolutional neural network을 사용하는 방법을 더 알고 싶다면 TensorFlow의 [deep convolutional networks tutorial](#)로 건너뛰거나 상대적으로 더 평이한 [ML beginner](#) 혹은 [ML expert](#) MNIST 입문자 튜토리얼로 시작해도 된다. 마지막으로 이 영역에 대한 연구에 박차를 가하고 싶다면, 이 튜토리얼이 참조한 모든 문서들의 최근 연구를 읽어보면 된다.

# 단어들의 벡터 표현

(v1.0)

이 튜토리얼에서 [Mikolov et al.](#) 의 word2vec 모델을 살펴본다. 이 모델은 "word embeddings" 라 불리는 단어들의 벡터 표현 학습에 사용된다.

## 강조점(Highlights)

이 튜토리얼은 TensorFlow에서 word2vec 모델을 만드는 흥미롭고 실질적인 부분들을 강조할 예정이다.

- 왜 단어들을 벡터들로 표현해야 하는지에 대한 동기부여를 주는 것에서 시작한다.
- 모델 넘어의 직관력과 이것이 어떻게 학습되어지는지(정확한 측정을 위한 수학 사용과 함께) 알아본다.
- 또한 TensorFlow에서 모델의 간단한 구현을 보인다.
- 마지막으로, 초기 버전 수준을 더 잘 만들수 있는 방법을 알아본다.

이후에 튜토리얼에서는 코드를 보여줄 것이나, 좀 더 자세히 알고 싶다면

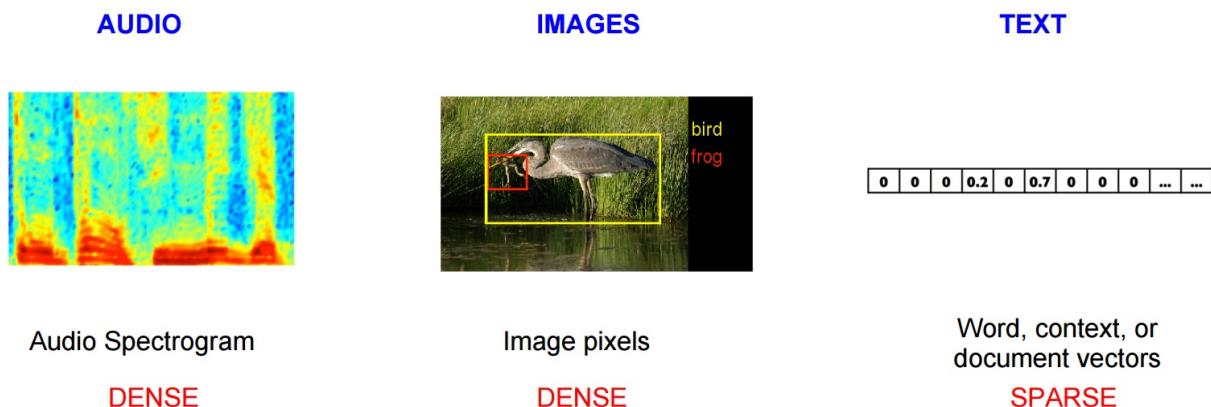
[tensorflow/examples/tutorials/word2vec/word2vec\\_basic.py](#)의 최소화된 구현을 참고하자. 이 기본 예제는 특정 데이터를 다운로드하기 위해 필요한 코드, 이것을 약간 학습하기 위한 코드, 그리고 결과를 시각화하기 위해 코드를 포함한다. 기본 버전을 읽고 실행하는데 익숙해지면, 쓰레드를 이용하여 어떻게 효율적으로 데이터를 텍스트 모델로 이동시키는지, 학습하는 동안 어떻게 체크하는지 등에 대한 좀 더 심화된 TensorFlow 원리들을 보여주는 심화 구현된 [tensorflow\\_models/tutorials/embedding/word2vec.py](#)을 시작할 수 있다.

하지만 우선, 전반부에 왜 word embeddings를 배워야 하는지 살펴보자. 자신이 Embedding을 잘 알고 자세한 설명들이 혼란스럽다고 생각하면 이 부분을 넘어가도 된다.

## 동기부여 : 왜 Word Embeddings를 배워야 하지?

이미지, 오디오 처리 시스템들은 이미지 데이터에 대하여 개별의 가공되지 않은 픽셀-강도값의 벡터들이나 오디오 데이터에 대한 파워스펙트럴밀도계수로 기록된 대량, 고차원 dataset들과 함께 한다. 객체나 연설 인식과 같은 문제에서 문제를 성공적으로 수행하기 위해 필요한 모든 정보는 데이터로 저장된다는 것을 알고 있다(사람은 이러한 문제를 가공되지 않은 데이터로부터 수행하기 때문이다). 그러나 자연어 처리 시스템들은 일반적으로 이산 원자 기호들(discrete atomic symbols)의 단어들로 다룬다. 따라서 'cat'은 'Id537'로, 'dog'은 'Id143'로 표현될 수 있다. 이 저장된 것들은 임의적이며, 개별 심볼들(symbols) 간에 존재하는 관계에 관해 시스템과는 의미없는 정보를 제공한다. 이것은 'dogs'라는 처리 데이터일 때 'cats'를 배우는 것이 영향력이 매우 적음을 의미한다.

(이들 모두 동물, 네 다리, 애완동물, 등) 유일하고 이산의(discrete) id 들로 단어들을 표현하는 것은 데이터를 더욱 드문드문(sparsity) 하게 만들고, 대체적으로 통계적 모델들을 성공적으로 학습하기 위해 더 많은 데이터가 필요함을 의미한다. 벡터 표현들을 사용하는 것은 다음과 같은 장애들을 해결할 수 있다.



벡터공간 모델(Vector space models) (VSMs)은 의미상 유사한 단어들은 가까운 지점으로 매핑되어지는(서로 가깝게 의미하는) 연속된 벡터 공간의 단어들로 표현(내포)한다. VSMs은 NLP에서 오래되고 깊은 역사를 가지고 있지만, 모든 방법들은 [Distributional Hypothesis](#)의 특정 방법이나 다른 방법에 따른다. 그리고 이 논문은 같은 맥락에서 나타나는 단어들은 시멘틱(semantic)의 미를 공유한다고 설명한다. 이 원리에 영향을 주는 다른 접근법들은 두 범주로 나눠진다: *count-based methods* (e.g. [Latent Semantic Analysis](#)) 와 *predictive methods* (e.g. [neural probabilistic language models](#)).

이 차이는 [Baroni et al.](#)에 의해 더 자세하게 설명되어 있다. 간략하게 말해: Count-based methods는 큰 텍스트 말뭉치에서 특정 단어가 그 주변 단어들과 함께 얼마나 자주 나타나는지에 대한 통계를 계산한다. 그리고 이 count-statistics를 각각의 단어에 대해 작고 dense 벡터로 상세히 묘사한다. 예측 모델(Predictive models)은 학습된 작고, dense embedding vectors (모델의 파라미터들로 고려된)에 관해 직접적으로 단어를 그 주변 단어들로부터 예측하려 한다.

Word2vec은 특히 가공하지 않은 텍스트로부터 학습한 단어 embeddings에 대해 계산적으로 효율적인 예측 모델이다. 이는 두 가지 종류로 나타난다, Continuous Bag-of-Word(CBOW) 모델과 Skip-Gram 모델([Mikolov et al.](#)의 Chapter 3.1과 3.2). 알고리즘적으로, 이들 모델들은 CBOW는 원본 컨텍스트 단어들('the cat sits on the')로부터 타겟 단어들(e.g. 'mat')을 예측하는 반면 skip-gram은 타겟 단어들로부터 원본 컨텍스트 단어들을 역으로 예측한다는 점을 제외하고 유사하다. 이 정반대는 임의적인 선택인 것처럼 보이지만, 통계적으로 CBOW는 많은 수의 분포상 정보를 바로잡는 효과를 가진다(전체 컨텍스트를 하나의 관찰로 처리함으로써). 대부분의 경우, 이러한 점은 작은 datasets 일 수록 유용한 것으로 밝혀졌다. 하지만 skip-gram은 각 컨텍스트-타겟 쌍을 새로운 발견으로 처리하고, 이것은 큰 규모의 datasets을 가질 때 더 잘 동작하는 경향이 있다. 이 튜토리얼의 나머지는 skip-gram 모델에 초점을 맞출 것이다.

# Noise-Contrastive 학습을 이용한 규모확장

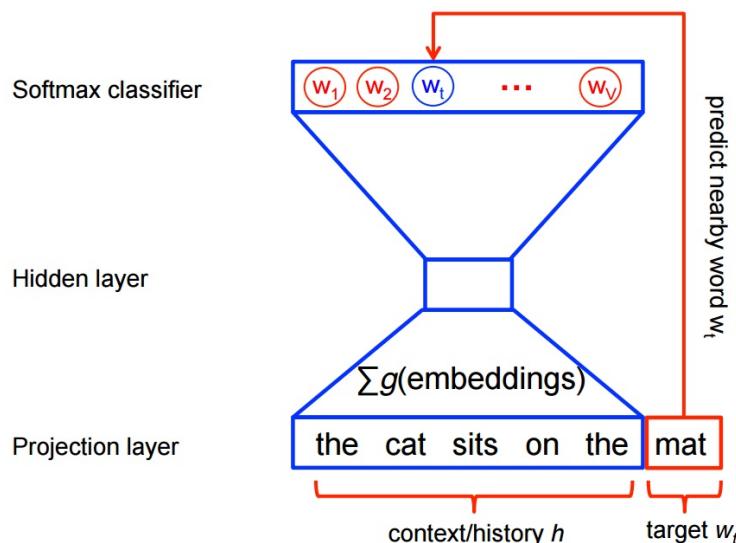
신경 확률 언어 모델들(Neural probabilistic language models)은 일반적으로 [softmax function](#)에 대해 이전에 주어진 단어들  $\{h\}$  (for "history")에서 다음 단어( $\{w_t\}$ ) (for "target")에 대한 확률을 최대화하는 [maximum likelihood](#) (ML) 원리를 이용하여 학습되어 진다.

$$\begin{aligned} P(w_t | h) &= \text{softmax(score}(w_t, h)) \\ &= \frac{\exp\{\text{score}(w_t, h)\}}{\sum_{\text{Word } w' \text{ in Vocab}} \exp\{\text{score}(w', h)\}} \end{aligned}$$

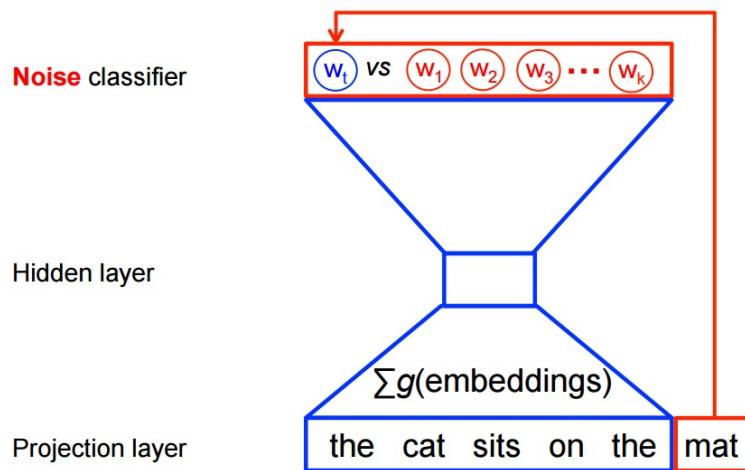
$\{\text{score}(w_t, h)\}$ 은 컨텍스트와 함께 하는 단어의 호환성을 계산한다(일반적으로 내적이 사용된다). 우리는 학습하는 set에서 이것의 [log-likelihood](#)를 최대화 함으로써 모델을 학습한다. 즉 최대화 하도록.

$$\begin{aligned} J_{\text{ML}} &= \log P(w_t | h) \\ &= \text{score}(w_t, h) - \log \left( \sum_{\text{Word } w' \text{ in Vocab}} \exp\{\text{score}(w', h)\} \right). \end{aligned}$$

이것은 언어 모델링에 대해서 적절하게 정규화된 확률 모델을 만들어 낸다. 하지만 이 방법은, 매 학습 스텝(at every training step), 현재 컨텍스트  $\{h\}$ 에서 다른 모든( $V$ ) words  $\{w'\}$ 에 대한 범위를 이용한 각 확률을 계산하고 정규화하는 것이 필요하기 때문에 그 비용이 매우 비싸다.



반면, word2vec의 feature 학습에 대하여 완전 확률 모델(a full probabilistic model)을 필요로 하지 않는다. 대신 CBOW와 skip-gram 모델은, 같은 컨텍스트 내에서, 실제 타겟 단어들  $\{\tilde{w}_t\}$ 을 가상(노이즈) 단어들  $\{\tilde{w}\}$ 로부터 구별해내기 위한 이진 분류 목적함수 ([logistic regression](#))를 이용하여 학습되어 진다. CBOW 모델에 대한 것은 아래에 도식화하였다. skip-gram에 대해 방향이 단순히 반대로 되어 있다.



수학적으로, 목적함수(각 예제에 대해)는 이를 최대화 한다.

$$J_{\text{NEG}} = \log Q_{\theta}(D = 1 | w_t, h) + k \mathbb{E}_{\tilde{w} \sim P_{\text{noise}}} [\log Q_{\theta}(D = 0 | \tilde{w}, h)]$$

$Q_{\theta}(D=1 | w, h)$ 은 학습된 embedding vectors  $\theta$ 에 대해 계산된 dataset  $D$  내 컨텍스트  $h$ 에서 보여지는 단어  $w$ 의 모델에 대해서 이진 로지스틱 회귀 확률인 조건이다. 실제로 노이즈 분포로부터 대조되는 단어는 찾아냄으로써 기대값을 가늠한다 (즉, Monte Carlo average 를 계산한다).

이 목적함수는 모델이 실제 단어들에 높은 확률을 할당하고 노이즈 단어들에 낮은 확률을 할당할 때 최대화된다. 기술적으로, 이를 Negative Sampling 이라 명하며, 이 손실(loss) 함수 사용에 대해 수학적으로 유리한 동기가 존재한다: 제시되는 업데이트들은 제한된 softmax 함수의 업데이트들을 근사값을 계산한다. 하지만 손실 함수의 계산을 우리가 선택한 noise words( $k$ )의 갯수, 어휘( $V$ ) 내 모든 단어(all words)가 아닌, 만으로 변경하여 계산한다는 점 때문에 계산적으로 특히 매력적이다. 이것은 학습을 더욱 빠르게 만든다. 우리는 noise-contrastive estimation (NCE) 손실 (loss) 와 매우 유사한 것, TensorFlow 가 가지고 있는 유용한 헬퍼 함수 `tf.nn.nce_loss()`, 를 활용할 것이다.

이제 실제로 어떻게 동작하는지에 직관적으로 이해해보자.

## Skip-gram 모델

예제로, dataset 을 생각해보자

```
the quick brown fox jumped over the lazy dog
```

우선 단어들과 컨텍스트들 자신들이 존재하는 dataset 을 만든다. 우리는 여러 타당한 방법으로 'context' 를 정의할 수 있고, 사실 사람들은 타겟의 좌측 단어, 타겟의 우측 단어 등을 통해 통사적 문맥(즉, 현재 타겟 단어의 통사적 의존성, 참고 Levy et al.) 을 살펴보게 된다. 이제부터, 평범한 정의와 연관지어보고 타겟 단어의 좌측과 우측의 단어들의 윈도우로 'context' 를 정의해보자. 크기 1 의 윈도우를 이용하면, (context, target) 쌍의 dataset 을 가지게 된다.

```
([the, brown], quick), ([quick, fox], brown), ([brown, jumped], fox), ...
```

skip-gram 은 컨텍스트들과 타겟들의 관계를 도치하고, 이들 타겟 단어로부터 각 컨텍스트 단어의 예측을 시도한다는 점을 상기하자. 그래서 문제는 'quick' 으로부터 'the' 와 'brown' 을, 'brown' 으로부터 'quick' 과 'fox' 등을 예측하는 것이 된다. 따라서 우리의 dataset 은 `(input, output)` 쌍의 dataset 이 된다.

```
(quick, the), (quick, brown), (brown, quick), (brown, fox), ...
```

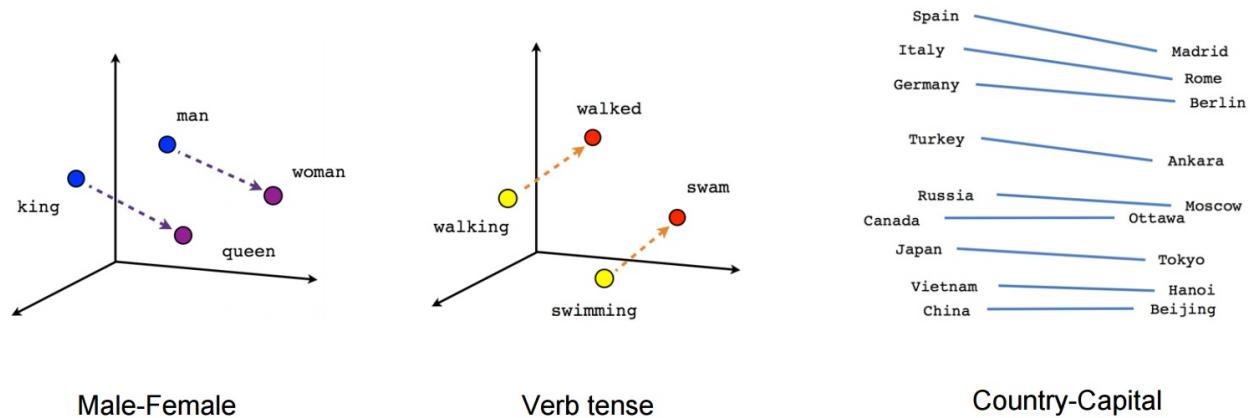
목적함수는 전체 dataset 에 대해 정의 되지만, 일반적으로 한번에 한 예를 이용한 [stochastic gradient descent](#)(SGD) 로 목적함수를 최적화 한다(또는 일반적으로 `16 <= batch_size <= 512` 인 `batch_size` 예제들의 'minibatch'). 그럼 이 과정의 한 단계를 살펴보자.

`quick` 에서 `the` 를 예측하는 것이 목표인 조건에서 우리가 관찰한 위 첫 학습 케이스의 학습 단계 $\backslash(t\backslash)$  를 상상해보자. 우리는 특정 노이즈 분포, 일반적으로 unigram 분포  $\backslash(P(w)\backslash)$ ,로부터 이끌어 낸 noisy (contrastive) 예제들의 `num_noise` 값을 선택했다. 간단하게, noisy 예제에서 `num_noise=1` 이라하고, `sheep` 을 선택한다. 이어서 이 관찰된 쌍과 noisy 예제들의 loss 를 계산 한다, 즉 time step  $\backslash(t\backslash)$  에서 목적함수는 아래와 같이 된다.

$$J_{\text{NEG}}^{(t)} = \log Q_\theta(D = 1 | \text{the}, \text{quick}) + \log(Q_\theta(D = 0 | \text{sheep}, \text{quick}))$$

목표는 이 목적 함수를 향상시키기 위한(여기서는 최대화) embedding parameters  $\backslash(\backslash theta\backslash)$  를 업데이트 시키는 것이다. 이는 embedding parameters  $\backslash(\backslash theta\backslash)$  에 대해서 loss 의 gradient 를 미분 함으로써 수행한다, 즉  $\backslash(\backslash frac{\backslash partial}{\backslash partial \backslash theta} J_{\text{NEG}}\backslash)$  (다행히 TensorFlow 는 이를 위해 쉬운 헬퍼 함수들을 제공한다!). 다음으로 gradient 방향으로 조금 진행하여 얻은 embeddings 의 업데이트를 수행한다. 전체 학습 set 에 걸쳐 이 과정을 반복할 때, 모델이 실제 단어들을 노이즈 단어들로부터 구별하는 것을 성공적으로 할 때까지 각 단어들에 대한 embedding 벡터들을 'moving' 하는 효과를 가진다.

예를 들어 [t-SNE dimensionality reduction technique](#) 와 같은 이용으로 학습된 벡터들을 2차원으로 투영하여 이들을 시각화 할 수 있다. 이들 시각화된 정보들을 살펴보면, 벡터들이 단어들과 그들과 다른 나머지들과의 관계에 대한 일반적이고, 사실 꽤 유용하고, 시멘틱(의미론적인, semantic) 정보를 담는다는 것을 분명히 할 수 있다. 유도된 벡터 공간에서의 특정 방향성은 특정 시멘틱 관계로 특징화되어 연결된다는 우리의 첫 발견은 매우 흥미로웠다, 즉 *male-femal, gender*, 그리고 심지어 *country-capital* 단어들 간의 관계, 아래 그림에 도식화하였다(예제 참고, [Mikolov et al., 2013](#)).



이것은 품사(part-of-speech) tagging이나 개체명 인식과 같은 여러 고전 NLP 예측 문제들에 대해 이들 벡터들이 왜 유용한 features인지 설명한다(원저작물인 [Collobert et al., 2011\(pdf\)](#) 예제를 참고하거나 후속 연구인 [Turian et al., 2010](#)을 참고하자).

이제 이들을 이용해서 멋진 그림들을 그려보자!

## Graph 구성하기

이것은 embeddings에 대한 모든 것이다. 그럼 우리의 embedding 매트릭스를 정의해보자. 이것은 시작하기 위한 랜덤 매트릭스일 뿐이다. 우리는 이 유닛 큐브의 값이 균일하도록 초기화 할 것이다.

```
embeddings = tf.Variable(
 tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
```

noise-contrastive estimation loss 는 로지스틱 회귀 모델에 관하여 정의 되어진다. 이를 위해, 어휘(vocabulary) 의 각 단어에 대한 가중치(weights)와 편향(biases) 을 정의할 필요가 있다( input embeddings 와 대조되어 output weights 라 불린다). 그럼 정의해보자.

```

nce_weights = tf.Variable(
 tf.truncated_normal([vocabulary_size, embedding_size],
 stddev=1.0 / math.sqrt(embedding_size)))
nce_biases = tf.Variable(tf.zeros([vocabulary_size]))

```

파라미터들이 준비되었고, 이제 우리의 skip-gram 모델 그래프를 정의할 수 있다. 간단하게, 우리의 텍스트 말뭉치(corpus)를 어휘(vocabulary)로 미리 정수화했다 가정하자. 그럼 각 단어는 정수로 표현되어 진다(자세한 사항은 다음을 참고하자).

[tensorflow/examples/tutorials/word2vec/word2vec\\_basic.py](https://github.com/tensorflow/examples/tree/master/tutorials/word2vec)). skip-gram 모델은 두 입력을 가진다. 하나는 원본 컨텍스트 단어들을 대표하는 정수들의 집합이고, 다른 하나는 타겟 단어들이다. 이를 입력들에 대한 placeholder 노드들을 만들어 보자. 그래야 나중에 데이터를 입력할 수 있다.

```
Placeholders for inputs
train_inputs = tf.placeholder(tf.int32, shape=[batch_size])
train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
```

이제 필요한 것은 집단(batch) 안의 각 원본 단어들에 대한 벡터를 살펴보는 것이다. TensorFlow는 이를 쉽게 해주는 편리한 헬퍼들을 가지고 있다.

```
embed = tf.nn.embedding_lookup(embeddings, train_inputs)
```

좋다, 이제 각 단어에 대한 embeddings을 만들었고, noise-contrastive 학습 목적함수를 이용한 타겟 단어 예측을 시도를 해보자.

```
매번 음수 라벨링 된 셈플을 이용한 NCE loss 계산
loss = tf.reduce_mean(
 tf.nn.nce_loss(nce_weights, nce_biases, embed, train_labels,
 num_sampled, vocabulary_size))
```

loss 노드를 만들었고, 이제 gradients를 계산하고 파라미터들을 업데이트 하는 등에 필요한 노드들을 추가할 필요가 있다. 이를 위해 stochastic gradient descent를 사용할 것이다. 그리고 TensorFlow는 이를 매우 쉽게 만들어주는 편리한 헬퍼들을 가지고 있다.

```
SGD optimizer를 사용
optimizer = tf.train.GradientDescentOptimizer(learning_rate=1.0).minimize(loss)
```

## 모델 학습하기

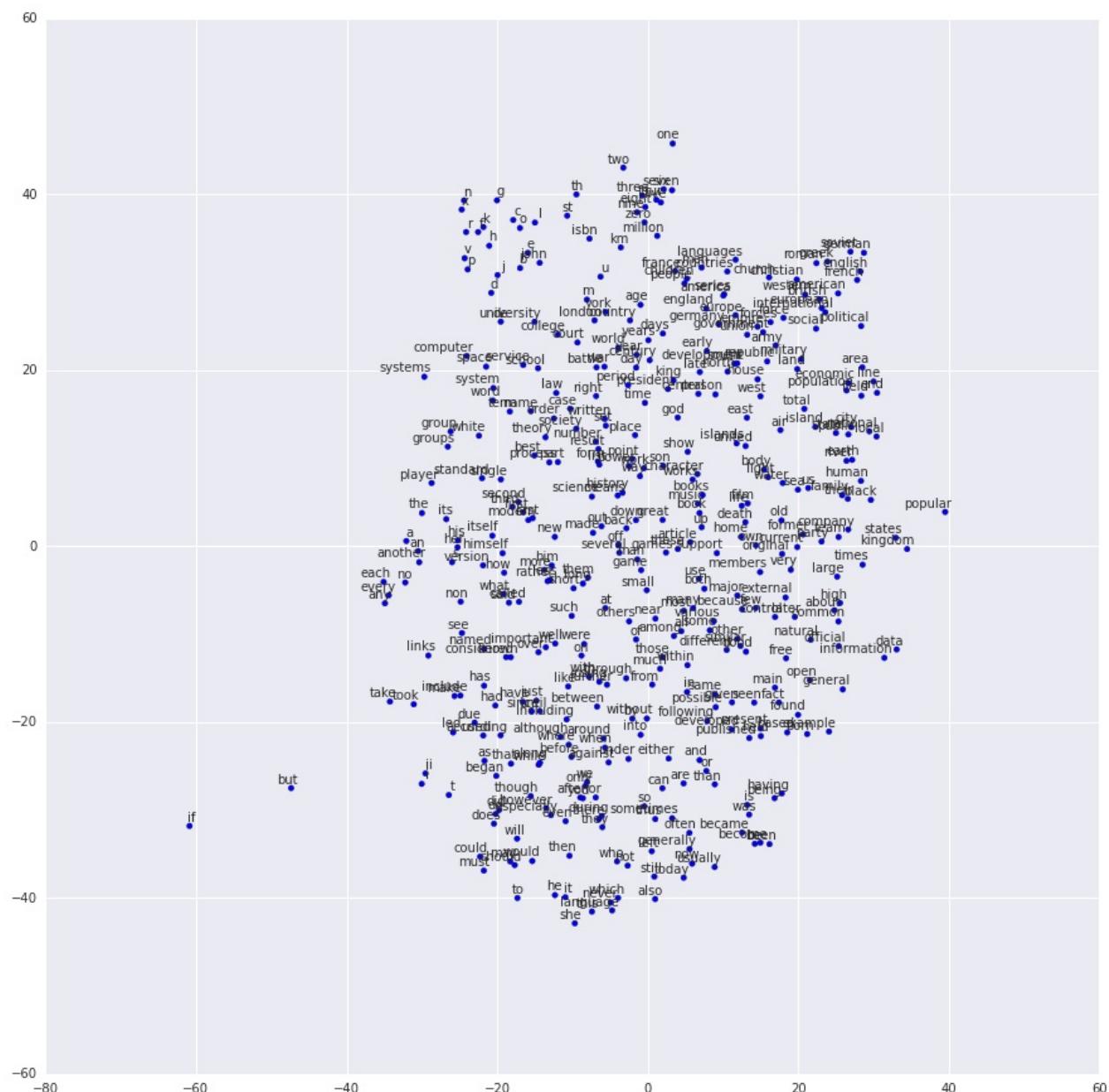
모델을 학습하는 것은 데이터를 placeholders에 넣기 위해 `feed_dict`를 사용하는 것 만큼 간단하며 루프 내에서 새로운 데이터와 함께 `session.run`를 불러 함수를 사용할 수 있다.

```
for inputs, labels in generate_batch(...):
 feed_dict = {training_inputs: inputs, training_labels: labels}
 _, cur_loss = session.run([optimizer, loss], feed_dict=feed_dict)
```

전체 예제 코드는 [tensorflow/examples/tutorials/word2vec/word2vec\\_basic.py](https://tensorflow/examples/tutorials/word2vec/word2vec_basic.py)을 살펴보자.

## 학습한 Embeddings 시각화하기

학습이 완료된 후 t-SNE를 사용하여 학습한 embeddings를 시각화 할 수 있다.



이제 다 됐어! 예상한 것처럼 비슷한 단어들은 결국 서로 가까운 집단화(clustering) 된다.

TensorFlow 의 더욱 진보된 features 를 보여주는 비중있는 word2vec 구현을 위해서,

[tensorflow\\_models/tutorials/embedding/word2vec.py](#) 을 참고하자.

## Embeddings 평가하기 : 유추(Analogical Reasoning)

Embeddings 는 NLP 의 다양한 예측 문제에 대해 유용하다. 완전 품사 모델이나 개체명 모델의 학습을 제외하고, embeddings 를 평가하는 한가지 간단한 방법은 이들을 직접 사용하여 `king is to queen as father is to ?` 와 같이 구문론적인 그리고 의미론적인 관계를 예측하는 것이다. 이 방법을 유추(*analogical reasoning*) 이라고 부르며 [Mikolov and colleagues](#) 에 의해 소개되었고, dataset 은 여기에서 다운로드 할 수 있다:

<https://word2vec.googlecode.com/svn/trunk/questions-words.txt>

어떻게 이 평가를 수행하는 알기 위해선, [tensorflow\\_models/tutorials/embedding/word2vec.py](#)의 `build_eval_graph()` 와 `eval()` 함수를 살펴봐라.

`hyperparameters` 의 설정은 이 문제의 정확도에 매우 큰 영향을 줄 수 있다. 이 문제에 대해 최고의 성과를 달성하기 위해선 매우 큰 `dataset` 을 학습하는 것, `hyperparameters` 에 대한 신중한 조절, 그리고 데이터의 이단추출과 같은 기법을 이용하는 것이 필요하다. 그리고 이 기법들은 이 튜토리얼의 범위를 넘어가는 것이다.

## 구현 최적화하기

우리의 평범한 구현은 TensorFlow 이 다루기 쉬움을의 보여준다. 예를들어 학습 목적함수의 변화는 `tf.nn.nce_loss()` 을 `tf.nn.sampled_softmax_loss()` 와 같은 대체 함수로 교체하는 것 만큼 간단하다. `loss` 함수에 대해 새로운 아이디어가 있다면, TensorFlow 내에서 새로운 목적함수에 대해 직접 고쳐 표현할 수 있으며 최적화 도구로 이것의 미분을 계산할 수 있다. 여러 다른 아이디어를 시도하거나 빠르게 반복할 경우, 이러한 용이성은 머신 러닝 모델 개발의 탐색 단계에서 매우 가치가 있다.

만족할 만한 모델 구조를 가지고 있다면, 당신의 구현을 더 효율적으로 실행하기 위해(그리고 적은 시간에 더 많은 데이터를 다룰수 있게 하기 위해) 최적화할 가치가 있을 수 있다. 예를 들어, 우리가 이 튜토리얼에서 사용한 간단한 코드는 데이터 아이템들을 읽고 대입하는데 --이들 각각은 TensorFlow back-end에서 매우 적게 고려된다-- Python 을 사용하기 때문에 절충된 속도로 수행된다. 만일 당신의 모델이 입력 데이터에 대해 심각한 병목현상을 겪는 것을 발견한다면, [New Data Formats](#) 에 설명된 것처럼, 수정된 데이터 리더(reader) 를 구현할 수 있을 것이다. Skip-gram 모델링의 경우, [tensorflow\\_models/tutorials/embedding/word2vec.py](#) 의 예제와 같이 이미 다루었다.

당신의 모델이 입출력 바운드 뿐만아니라 더 높은 성능을 원한다면, [Adding a New Op](#) 에 설명된 것처럼, 당신의 TensorFlow Ops 작성을 통해 자세한 설명을 할 수 있다. 이것에 대한 Skip-Gram 예제는 [tensorflow\\_models/tutorials/embedding/word2vec\\_optimized.py](#) 에 제시했다. 각 단계의 성능 향상 측정을 위해 각각에 대해 자유롭게 벤치마크 해보자.

## 결론

이 튜토리얼에서 word embeddings 학습에 대해 계산적으로 효율적인 모델인, word2vec 모델을 다뤘다. 우리는 왜 embeddings 가 유용한지 동기부여를 했고, 효율적인 학습 기술들에 대한 논의했으며, TensorFlow 에서 이 모든 것들을 어떻게 구현하는지 보였다. 종합하면, 이것들은 어떻게 TensorFlow 가 초기 실험에 필요한 용이성을 제공하고, 나중에 개인 맞춤의 최적화된 구현에 필요한 조작을 하는지 보여주기를 바란다.



# 순환 신경망(Recurrent Neural Networks)

(v0.9)

## 소개

순환 신경망과 LSTM에 관한 소개는 이 [블로그](#)를 참고하세요.

## 언어 모델링(Language Modeling)

이 튜토리얼에서 언어 모델링 문제에 대해 순환 신경망을 어떻게 학습시키는지 살펴 보겠습니다. 여기서 풀려고 하는 문제는 문장 구성에 대한 확률을 부여하는 모델을 최적화시키는 것입니다. 즉 이전에 나타난 단어의 기록을 보고 다음의 단어를 예측하는 것입니다. 우리가 사용할 데이터는 이런 종류의 모델의 성능을 평가하는 데 널리 사용되고 비교적 크기가 작아 학습하는 데 시간이 많이 걸리지 않는 [Penn Tree Bank \(PTB\)](#) 데이터셋입니다.

언어 모델링은 음성 인식, 기계 번역, 이미지 캡셔닝(captioning) 같이 인기있는 여러 분야의 핵심 요소입니다. 또한 매우 재미있습니다. [여기](#)를 한번 둘러보세요.

이 튜토리얼에서는 PTB 데이터셋을 사용하여 뛰어난 성과를 낸 [Zaremba 외., 2014 \(pdf\)](#)의 결과를 재현할 것 입니다.

## 튜토리얼 파일

이 튜토리얼에서 사용할 파일들은 `models/rnn/ptb` 에 있습니다.

| 파일                          | 설명                                  |
|-----------------------------|-------------------------------------|
| <code>ptb_word_lm.py</code> | 이 코드는 PTB 데이터셋을 사용하여 언어 모델을 학습시킵니다. |
| <code>reader.py</code>      | 이 코드는 데이터를 읽어 들이는데 사용됩니다.           |

## 데이터 다운로드하여 준비하기

이 튜토리얼에서 필요한 데이터는 Tomas Mikolov의 웹 페이지에서 다운 받은 파일의 `data` 디렉토리 안에 있습니다: <http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz>

이 데이터셋은 전처리된 총 만개의 단어를 가지고 있고 문장 끝을 나타내는 표시와 희귀 단어를 표시하는 심볼(<unk>)을 포함하고 있습니다. 이 데이터를 신경망에서 처리하기 적합하도록 `reader.py`에서 모든 단어를 고유한 정수 숫자로 바꿉니다.

## 학습 모델

### LSTM

학습 모델의 핵심 부분은 한번에 하나의 단어를 입력받아 문장에서 나타날 다음 단어의 확률을 계산하는 LSTM 셀(cell)로 이루어져 있습니다. 신경망의 메모리 상태는 0으로 초기화되고 단어를 읽으면서 업데이트 됩니다. 계산 효율을 높이기 위해 `batch_size` 크기의 미니배치(mini-batch)로 모델을 학습시킬 것입니다.

기본적인 의사코드는 아래와 같습니다:

```
lstm = rnn_cell.BasicLSTMCell(lstm_size)
LSTM 상태 메모리 초기화.
state = tf.zeros([batch_size, lstm.state_size])

loss = 0.0
for current_batch_of_words in words_in_dataset:
 # 상태 값은 배치를 처리한 후 업데이트 됩니다.
 output, state = lstm(current_batch_of_words, state)

 # LSTM의 출력 값을 사용하여 다음 단어를 예측합니다.
 logits = tf.matmul(output, softmax_w) + softmax_b
 probabilities = tf.nn.softmax(logits)
 loss += loss_function(probabilities, target_words)
```

## 부분 역전파(Truncated Backpropagation)

학습 과정을 지켜보기 위해서 일정 횟수(`num_steps`)만큼 학습을 진행한 후에 그 만큼의 그래디언트만 역전파 시키는 것이 보통입니다. 반복 루프안에서 한번에 `num_steps` 길이 만큼 입력 값을 주입하고 나서 역전파 시키면 됩니다.

부분 역전파를 위한 그래프를 만드는 코드의 간소화 버전은 아래와 같습니다:

```
한번 반복에서 처리할 입력 값을 위한 플레이스홀더
words = tf.placeholder(tf.int32, [batch_size, num_steps])

lstm = rnn_cell.BasicLSTMCell(lstm_size)
LSTM 상태 메모리 초기화.
initial_state = state = tf.zeros([batch_size, lstm.state_size])

for i in range(num_steps):
 # 상태 값은 배치를 처리한 후 업데이트 됩니다.
 output, state = lstm(words[:, i], state)

 # 이어진 코드
 # ...

final_state = state
```

그리고 아래는 전체 데이터셋에 대해 반복하는 부분을 구현한 것입니다:

```
배치를 처리한 후 LSTM의 상태를 저장하는 numpy 배열.
numpy_state = initial_state.eval()
total_loss = 0.0
for current_batch_of_words in words_in_dataset:
 numpy_state, current_loss = session.run([final_state, loss],
 # 이전 반복에서 얻은 결과를 사용해 LSTM 상태를 초기화.
 feed_dict={initial_state: numpy_state, words: current_batch_of_words})
 total_loss += current_loss
```

## 입력

단어의 아이디(ID)는 LSTM에 주입되기 전에 밀집 행렬(dense representation)([벡터 표현 튜토리얼](#)을 참고하세요)에 임베딩(embedding)될 것입니다. 이 방식은 특정 단어에 대한 정보를 효과적으로 표현할 수 있습니다. 아래와 같이 만들 수 있습니다:

```
embedding_matrix는 [단어수, 임베딩사이즈] 크기의 텐서입니다.
word_embeddings = tf.nn.embedding_lookup(embedding_matrix, word_ids)
```

임베딩 행렬은 랜덤하게 초기화되고 데이터를 처리하면서 단어의 의미를 구분하도록 학습됩니다.

## 손실 함수(Loss Function)

우리는 목적 단어의 로그 확률의 음수 평균을 최소화하려고 합니다:

$$\text{loss} = -\frac{1}{N} \sum_{i=1}^N \ln p_{\text{target}_i}$$

직접 구현하는 것도 어렵지 않으나 이미 `sequence_loss_by_example` 함수가 있어 이를 사용하겠습니다.

이 페이퍼에서 사용한 측정법은 평균 단어당 복잡도(perplexity)입니다(종종 그냥 복잡도라고 부릅니다). 아래 식과 같습니다.

$$e^{-\frac{1}{N} \sum_{i=1}^N \ln p_{\text{target}_i}} = e^{\text{loss}}$$

학습 과정 동안 이 값을 모니터링하도록 하겠습니다.

## LSTM 레이어 만들기

모델의 성능을 높이기 위해 데이터를 처리할 여러개의 LSTM 레이어를 만들 수 있습니다. 첫번째 레이어의 출력은 두번째 레이어의 입력이 되는 식입니다.

이런 작업을 위해 구현된 `MultiRNNCell` 클래스가 있습니다:

```
lstm = rnn_cell.BasicLSTMCell(lstm_size)
stacked_lstm = rnn_cell.MultiRNNCell([lstm] * number_of_layers)

initial_state = state = stacked_lstm.zero_state(batch_size, tf.float32)
for i in range(num_steps):
 # 상태 값은 배치를 처리한 후 업데이트 됩니다.
 output, state = stacked_lstm(words[:, i], state)

 # 이어진 코드
 #

final_state = state
```

## 코드 실행

독자가 이미 pip 패키지를 통해 텐서플로우를 설치했고 텐서플로우 깃 저장소(git repository)에서 클론하여 깃 트리의 최상위 디렉토리에 있다고 가정합니다. (만약 소스에서 직접 빌드 했다면 `bazel`을 이용해서 `tensorflow/models/rnn/ptb:ptb_word_lm` 타겟을 빌드하세요).

그 다음은:

```
cd tensorflow/models/rnn/ptb
python ptb_word_lm --data_path=/tmp/simple-examples/data/ --model small
```

이 튜토리얼에 포함된 코드에는 세가지의 모델 환경을 제공합니다: "small", "medium", "large" 입니다. 이들간의 차이는 LSTM 레이어의 크기와 학습에 사용될 하이퍼파라미터(hyperparameter) 설정입니다.

큰 모델일수록 더 좋은 결과가 나와야 합니다. 'small' 모델은 테스트 셋에 대한 복잡도가 120 아래에 도달하며 'large' 모델은 80 이하가 나오지만 학습에 여러시간이 소요될 수 있습니다.

## 그 다음엔?

여기서 언급하지 않았지만 모델의 성능을 더 좋게 만들기 위한 몇가지 기법이 있습니다:

- 학습 속도 감소를 스케줄링 하기,
- LSTM 레이어 간에 드롭아웃 적용.

코드를 연구하고 수정해서 모델의 성능을 개선해 보십시오.

# 시퀀스-투-시퀀스(Sequenc-to-Sequenc) 모델

러커런트 뉴럴 네트워크는 [RNN 튜토리얼](#)에서 이미 이야기 했던 것처럼 언어를 모델화하는 것을 학습할 수 있다.

(그것을 읽지 않았다면, 이 튜토리얼을 읽기 전에 그것을 먼저 읽기를 권한다). 흥미로운 질문이 있다: 몇몇의 입력으로 생성된 단어들을 조건화 할 수 있고 어떤 의미 있는 응답을 생성할 수 있을까? 예를 들어, 영어를 프랑스어로 번역하기 위한 뉴럴 네트워크를 학습할 수 있을까? 답은 예가 된다.

이 튜토리얼은 그런 앤드-투-앤드 (end-to-end) 시스템을 만들고 학습하는 방법 보여줄 것이다. pip를 통해 tensorflow를 이미 설치했고 tensorflow git 저장소를 클론(clone)했고, git tree의 root에 있다고 가정한다.

그리고 나서, 번역 프로그램을 실행함으로 시작할 수 있다:

```
cd tensorflow/models/rnn/translate
python translate.py --data_dir [your_data_directory]
```

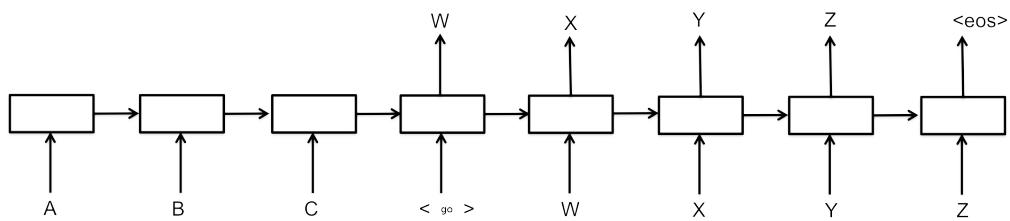
그 프로그램은 [WMT'15 웹사이트](#)에서 영어-프랑스어 번역 데이터를 다운받을 것이고 학습을 준비하고 학습할 것이다. 그 데이터는 20GB 공간을 차지할 것이고, 다운로드하고 준비하기 위해 좀 시간이 걸린다(상세히 알고 싶으면 [나중에](#) 보라). 그래서 이 튜토리얼을 읽는 동안 프로그램을 구동시키고 실행해 두자.

이 튜토리얼은 `models/rnn`에서 아래 파일을 참고한다.

| 파일                                      | 그 안에 무엇이 있나                     |
|-----------------------------------------|---------------------------------|
| <code>seq2seq.py</code>                 | 시퀀스-투-시퀀스 모델을 생성하기 위한 라이브러리.    |
| <code>translate/seq2seq_model.py</code> | 뉴럴 번역 시퀀스-투-시퀀스 모델.             |
| <code>translate/data_utils.py</code>    | 번역 데이터를 준비하기 위한 도움 함수들.         |
| <code>translate/translate.py</code>     | 번역 모델을 훈련하고 실행시키는 바이너리(binary). |

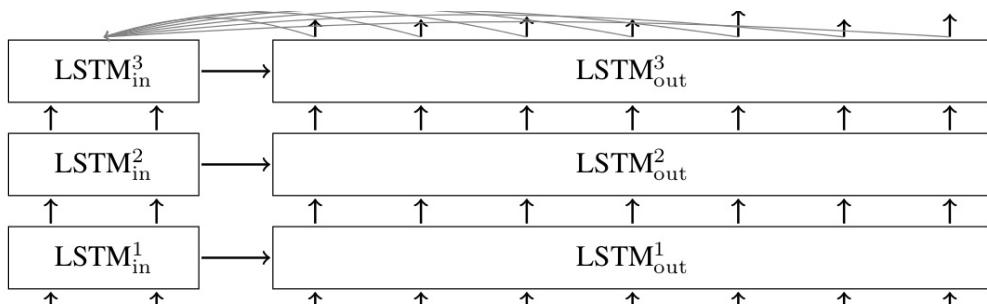
## 기본적인 시퀀스-투-시퀀스(Sequence-to-Sequence)

[Cho et al., 2014 \(pdf\)](#)에서 소개된 기본적인 시퀀스-투-시퀀스 모델은 두 개의 리커런트 뉴럴 네트워크(RNN)으로 구성된다: 입력을 처리하는 인코더와 결과를 생성하는 디코더. 이 기본 구조는 아래와 같이 묘사된다.



위 그림에 각 박스는 가장 일반적으로 GRU 셀이거나 LSTM 셀인 RNN 셀을 나타낸다([RNN Tutorial](#)를 참조하길 바란다). 인코더와 디코더는 가중치를 공유 할 수 있거나, 더 일반적으로는 다른 매개변수 집합을 사용한다. 다중층 셀들은 역시 시퀀스-투-시퀀스에서 성공적으로 사용되어 왔다. 예로 번역 [Sutskever et al., 2014 \(pdf\)](#)에서 알 수 있다.

위에서 설명된 기본 모델에서, 모든 입력은 디코더에 전달되는 유일한 것이기 때문에 고정된 크기를 가진 상태 벡터로 인코딩 되어야 한다. 디코더가 입력에 더 직접적인 접근을 가능하게 하기 위해, 주의(*attention*) 메카니즘이 [Bahdanau et al., 2014 \(pdf\)](#)에서 소개된다. 주의(*attention*) 메카니즘에 대해서 상세히 보지 않을 것이다(논문을 참고), 그것은 디코더가 모든 디코딩 단계에서 입력을 엿보게 해주는 것이라고 언급하는 것만으로도 충분하다. LSTM 셀을 가진 여러층의 시퀀스-투-시퀀스 네트워크와 디코더 안에 어텐션 메카니즘은 이처럼 보인다.



## 텐서플로우(TensorFlow) seq2seq 라이브러리

위에서 볼수 있듯이, 다른 많은 시퀀스-투-시퀀스 모델들이 있다. 각각 이러한 모델은 다른 RNN 셀들을 사용할 수 있다, 그러나 모두 인코더 입력과 디코디 입력을 받아야 한다. 이것은 텐스플로우 seq2seq 라이브러리( `models/rnn/seq2seq.py` )의 인터페이스에 동기를 준다.

기본적인 RNN 인코더-디코더 시퀀스-투-시퀀스 모델은 아래처럼 동작한다.

```
outputs, states = basic_rnn_seq2seq(encoder_inputs, decoder_inputs, cell)
```

위에 호출에서, `encoder_inputs` 는 인코더에 입력을 나타내는 텐서(tensor) 리스트이다. 예로, 위에 첫번째 그림에서 문자 *A*, *B*, *C*에 해당된다. 유사하게, `decoder_inputs` 는 디코더에 입력을 나타내는 텐서들이다. 첫번째 그림에 *GO*, *W*, *X*, *Y*, *Z* 이다.

`cell` 인수는 `models.rnn.rnn_cell.RNNCell` 클래스의 인스턴스이다, 그것은 어느 셀이 그 모델 안에서 사용할지를 결정한다. 예를 들어 `GRUCell` 또는 `LSTMCell`, 또는 자신의 것을 작성할 수 있다. 게다가, `rnn_cell` 은 여러층 셀을 만들고, 셀 입력과 결과에 드랍아웃(dropout)를 추가하거나

다른 변환을 하기 위한 랩퍼(wrapper)들을 제공한다. 예제들을 보기 위해서는 [RNN Tutorial](#)를 참고하기 바란다.

`basic_rnn_seq2seq` 호출은 두개의 인수를 리턴한다: `outputs` 와 `states`. 그것 둘다 `decoder_inputs` 와 동일한 크기의 탠서 리스트이다. 자연스럽게, `outputs` 는 각 시간 단계(time-step)에서 디코더 결과에 해당된다. 위 첫번째 그림에서 그것은  $W, X, Y, Z, EOS$ 가 된다. 반환된 `states` 는 모든 시간 단계에서 디코더의 내부 상태를 나타낸다.

시퀀스-투-시퀀스 모델의 많은 애플리케이션에서, 시간  $t$ 에 디코더 결과는 다시 시간  $t+1$ 에 디코더의 입력으로 전달된다. 테스트 시간에, 어떤 시퀀스를 디코딩할때, 이것은 그 시퀀스가 만들어지는 방법이다. 반면에, 학습동안 디코더가 전에 실수를 했어도 모든 시간 단계에 디코더에 올바른 입력을 제공하는 것이 일반적이다. `seq2seq.py` 에 함수들은 `feed_previous` 인수를 사용해서 두가지 모드를 지원한다. 예를 들어, 아래에 임베드 RNN모델 사용을 분석해 보자.

```
outputs, states = embedding_rnn_seq2seq(
 encoder_inputs, decoder_inputs, cell,
 num_encoder_symbols, num_decoder_symbols,
 output_projection=None, feed_previous=False)
```

`embedding_rnn_seq2seq` 모드에서, 모든 입력들은( `encoder_inputs` 과 `decoder_inputs` ) 이산 값(discrete value)을 나타내는 정수-텐서(integer-tensor)들이다. 그것들은 조밀한(dense) 표현으로 임베딩되어 질 것이다(임베딩에 대해 더 자세한 설명을 위해 [Vectors Representations Tutorial](#) 를 보길 권한다, 그러나 이러한 임베딩을 만들기 위해서 나타나는 이산 심볼의 최대 수를 지정할 필요가 있다: 인코더 쪽에 `num_encoder_symbols`, 그리고 디코더 쪽에 `num_decoder_symbols`.

위 호출에서, `feed_previous` 를 `False`로 설정했다. 이것은 디코더에 전달되는 `decoder_inputs` 를 사용할 것이다. `feed_previous` 가 `True`로 설정되면, 디코더는 `decoder_inputs` 의 단지 첫번째 원소만 사용할 것이다. 이 리스트에 모든 다른 텐서들은 무시 되어지고, 그대신 디코더의 이전 결과가 사용되어 질 것이다. 이것은 우리의 번역 모델에서 번역을 디코딩 하기 위해서 사용되어진다. [Bengio et al., 2015 \(pdf\)](#)와 유사하게 그 모델이 그 자신의 실수에 더 견고하기 하기 위해 또한 학습동안에도 사용되어 질수 있다.

위에서 사용된 더 중요한 한 인수는 `output_projection` 이다. 그것이 명시되지 않는다면, 임베딩 모델의 결과는 각 생성된 심볼에 대해 로짓(logit)들로 나타내기 때문에 배치사이즈  $\times$  `num_decoder_symbols` 의 형태를 가진 텐서가 될 것이다. 아주 큰 사전을 가지고 모델을 학습할때, 예를 들어, `num_decoder_symbols` 이 아주 클때, 아주 큰 이러한 텐서들을 저장하는것은 비실용적이다. 그대신, 좀더 작은 결과 텐서를 리턴하는 것이 좋다. 그 텐서는 나중에 `output_projection` 를 사용해서 큰 결과 텐서에 투여되어진다. 이것은 [Jean et. al., 2014 \(pdf\)](#)에서 설명하는 대로 샘플 소프트맥스 로스(sampled softmax loss)를 가진 seq2seq모델을 사용할수 있게 해준다.

`basic_run_seq2seq` 와 `embedding_rnn_seq2seq` 에 더해서, `seq2seq.py` 에 좀더 많은 시퀀스-투-시퀀스 모델들이 있고, 그것을 볼 것이다. 그것들은 유사한 인터페이스를 가진다, 그래서 상세히 설명하지는 않을 것이다. 아래에 우리의 번역 모델을 위해 `embedding_attention_seq2seq` 를 사용할

것이다.

## 뉴럴 번역 모델(Neural Translation Model)

시퀀스-투-시퀀스(sequence-to-sequence) 모델의 중요한 부분은 `models/rnn/seq2seq.py` 에 있는 함수에 의해 구성되어지지만, 여전히 `models/rnn/translate/seq2seq_model.py` 안에 우리 번역 모델에서 사용되어지고 언급할 가치가 있는 몇가지 기교가 있다.

### 샘플 소프트맥스와 결과 프로젝션(Sampled softmax and output projection)

하나는, 위에서 이미 언급한 대로, 우리는 아주 큰 사전을 다루기 위해서 샘플 소프트맥스(sampled softmax)를 사용하길 원한다. 그것으로부터 디코딩하기 위해 결과 프로젝션(output projection)을 기억할 필요가 있다. 샘플 소프트맥스 로스(loss)와 결과 프로젝트 둘다는 `seq2seq_model.py` 에 있는 다음 코드에 의해 구성되어진다.

```
if num_samples > 0 and num_samples < self.target_vocab_size:
 w = tf.get_variable("proj_w", [size, self.target_vocab_size])
 w_t = tf.transpose(w)
 b = tf.get_variable("proj_b", [self.target_vocab_size])
 output_projection = (w, b)

def sampled_loss(inputs, labels):
 labels = tf.reshape(labels, [-1, 1])
 return tf.nn.sampled_softmax_loss(w_t, b, inputs, labels, num_samples,
 self.target_vocab_size)
```

우선, 샘플 수(디폴트가 512)가 목적(target) 사전 크기보다 작다면, 단지 샘플 소프트맥스를 구성한다는 것을 기억하자. 512보다 작은 사전에 대해, 그냥 표준 소프트맥스 손실(softmax loss)를 사용하는 것이 더 나은 생각일지도 모른다.

```
if output_projection is not None:
 self.outputs[b] = [tf.matmul(output, output_projection[0]) +
 output_projection[1] for ...]
```

그리고 나서, 알다시피, 우리는 결과 프로젝트를 구성한다. 그것은 가중치 행렬과 바이어스(bias) 벡터로 구성된 순서쌍이 된다. 그것이 사용되어지면, rnn셀은 배치크기 x `target_vocab_size` 가 아니라 배치 크기x `size` 의 벡터들이 리턴될 것이다. 로짓(logit)를 복구하기 위해, `seq2seq_model.py` 에 124-126라인에서 했던 것처럼 가중치 행렬에 의해서 곱해지고, 바이어스가 더해질 필요가 있다.

### 버켓팅(Bucketing)과 패딩(padding)

샘플 소프트맥스에 더해서, 우리의 번역 모델은 또한 버켓팅(bucketing)를 사용한다, 그것은 다른 크기의 문장을 효과적으로 다루기 위한 방법이다. 우선 그 문제를 명확히 해 보자. 영어에서 프랑스어로 번역할 때, 입력으로 L1 크기의 영어문장과 결과로 L2 크기의 프랑스 문장이 있을 것이다. 영어 문장이 `encoder_inputs` 으로 전달되고 프랑스어 문장은 `decoder_inputs` (GO 심볼이 앞에 붙혀진다)로 나오기 때문에, 기본적으로 (L1, L2+1)의 모든 순서쌍에 대해 seq2seq 모델을 만들어야 한다. 이것은 아주 유사한 서브 그래프들로 구성되는 거대한 그래프를 만들게 될 것이다. 다른 한 편으로 특별한 PAD 심볼을 가지고 모든 문장에 간단히 메워 넣을 수 있다. 그러면, 우리는 메워진 크기에 대한 단지 하나의 seq2seq 모델만 만들 필요 있게 된다. 그러나 문장이 더 짧을 수록 쓸모없는 PAD 심볼에 대해 디코딩 인코딩 해야되기 때문에 우리의 모델은 비효율적이게 된다.

모든 길이의 순서쌍에 대해 그래프를 생성하는 것과 하나의 길이로 패딩하는 것 사이에 절충으로써 몇개의 버켓(bucket)를 사용하고 각 문장을 그 위에서 있는 버켓의 길이로 메워 넣는다. `translate.py` 에서 다음과 같은 기본적인 버켓들을 사용한다.

```
buckets = [(5, 10), (10, 15), (20, 25), (40, 50)]
```

이것은 3개의 토큰을 가진 영어 문장이 입력이고 6개의 토큰을 가진 프랑스어 문장이 그 입력에 대한 결과라고 한다면 그것들은 첫번째 버켓에 들어가게 되고 인코더 입력에 길이 5까지 메워넣고, 디코더 입력에 대해서는 길이 10까지 메워넣게 된다. 8개의 토큰을 가진 영어문장이고 그에 상응하는 프랑스어 문장이 18개의 토큰을 가진다면, 버켓 (10, 15)에 맞지 않고, 버켓 (20, 25)이 사용될 것이다. 예를 들어, 그 영어 문장은 길이 20까지 메워 질것이고, 프랑스어 문장은 길이 25까지 메워진다.

디코더 입력을 만들 때, 특별한 `go` 심볼을 입력 데이터 앞에 붙히는 것을 기억하자. 이것은 `seq2seq_model.py` 에 `get_batch()` 함수 안에서 이루어진다. 그것은 또한 영어 문장 입력을 반대로 변환한다. 입력을 반대로 변환하는 것은 [Sutskever et al., 2014 \(pdf\)](#)에서 뉴럴 번역 모델에 대한 결과를 향상시키는 것을 보여주었다. 이것 모든 것을 다 넣기 위해, "I go."라는 문장이 있다고 가정하고, 입력으로 `["I", "go", "."]`로 토큰화 되고 결과로 "Je vais"가 되고 `["Je", "vais", "."]`로 토큰화 된다. 그것은 `[PAD PAD "." "go" "I"]`로 인코더 입력이고, `[GO "Je" "vais" "." EOS PAD PAD PAD PAD PAD]` 는 디코더 입력으로 버켓 (5,10)에 들어가게 될 것이다.

## 그것을 실행해보자

위에 설명되어진 대로 모델을 훈련하기 위해, 우리는 아주 큰 영어-프랑스어 코퍼스(corpus)가 필요하다. [WMT'15 Website](#)에서 *10^9-French-English corpus*를 훈련을 위해 사용할 것이고, 개발셋(development set)으로 그 동일한 사이트에서 2013 news test를 사용할 것이다.

```
python translate.py
--data_dir [your_data_directory] --train_dir [checkpoints_directory]
--en_vocab_size=40000 --fr_vocab_size=40000
```

약 18GB 디스크 공간을 차지하고 훈련 코퍼스를 준비하기 위해 몇 시간이 걸린다. 그것이 풀려지면, 사전 파일들이 `data_dir`에 만들어 진다, 그리고 나서 그 코퍼스는 토근화 되어지고 정수 식별자(integer id)로 변환될 것이다. 사전 크기를 결정하는 매개변수들을 주시하자. 위 예에서, 가장 일반적인 단어 4만개에 있는 않는 모든 단어들은 모르는 단어를 나타내는 `UNK` 토근으로 변환 될 것이다. 사전 크기를 변경한다면 바이너리는 토근 식별자(token id)를 그 코퍼스에 다시 연결 할 것이다.

데이터가 준비된 뒤에, 훈련이 시작된다. `translate`에 있는 디폴트 매개변수는 꽤 큰 값으로 설정되어 있다. 아주 긴 시간 동안 학습된 큰 모델은 좋은 결과를 준다, 그러나 너무 긴 시간과 GPU에 너무 많은 메모리를 사용할지도 모른다. 다음 예처럼 더 작은 모델을 학습하는 것을 요청할 수 있다.

```
python translate.py
--data_dir [your_data_directory] --train_dir [checkpoints_directory]
--size=256 --num_layers=2 --steps_per_checkpoint=50
```

위 명령은 256 유닛(디폴트는 1024)을 가진 각 2개층(디폴트는 3)을 가진 모델을 학습할 것이고, 50 단계(디폴트는 200)마다 체크포인트(checkpoint)를 저장할 것이다. 여러분의 GPU의 메모리에 얼마나 큰 모델이 맞게 할 수 있는지 알기 위해 이러한 매개변수를 통해 할수 있다.

학습 동안, 모든 `steps_per_checkpoint` 단계에서 바이너리는 최근 단계로 부터 통계를 프린터 할 것이다. 디폴트 매개변수(크기 1024의 3개층)와 함께 첫번째 메시지는 이처럼 보인다.

```
global step 200 learning rate 0.5000 step-time 1.39 perplexity 1720.62
eval: bucket 0 perplexity 184.97
eval: bucket 1 perplexity 248.81
eval: bucket 2 perplexity 341.64
eval: bucket 3 perplexity 469.04
global step 400 learning rate 0.5000 step-time 1.38 perplexity 379.89
eval: bucket 0 perplexity 151.32
eval: bucket 1 perplexity 190.36
eval: bucket 2 perplexity 227.46
eval: bucket 3 perplexity 238.66
```

각 단계는 단지 1.4초보다 빠르게 걸린다는 것과 훈련 셋에 퍼플렉서티(perplexity)와 개발 셋에 퍼플렉서티들을 볼수 있다. 대략 3만 단계 뒤에, 짧은 문장(버켓이 0과 1)에 대한 퍼플렉서티가 한 자릿수로 된다. 훈련 코퍼스는 ~22M 문장을 포함하기 때문에, 각 epoch(훈련 데이터 전체 한번 통과)은 64의 배치크기로 대략 340K 단계가 걸린다. 이 시점에, 그 모델은 `--decode` 옵션을 사용해서 영어 문장에서 프랑스 문장으로 번역하기 위해 사용되어 질 수 있다.

```
python translate.py --decode
--data_dir [your_data_directory] --train_dir [checkpoints_directory]

Reading model parameters from /tmp/translate.ckpt-340000
> Who is the president of the United States?
Qui est le président des États-Unis ?
```

## 다음은?

위 예는 앤드-투-앤드(end-to-end)로 영어-에서-프랑스어(English-to-French) 번역기를 만드는 방법을 보여준다. 스스로 그것을 실행하고 그 모델이 어떻게 수행하는지 봐라. 그것이 꽤 괜찮은 성능을 보이지만, 디폴트 매개변수로 최고의 번역 모델이 주어지지 않는다. 여기 여러분이 그것을 향상 시킬수 있는 몇가지 방법이 있다.

무엇보다, `data_utils` 안에 `basic_tokenizer` 인 아주 기본적인 토큰생성자를 사용한다. 더 좋은 토큰 생성자는 [WMT'15 Website](#)에서 찾을수 있다. 저 토큰 생성자와 더 큰 사전을 사용하는 것은 번역을 향상 시킬 것이다.

또한, 그 번역 모델의 디폴트 매개변수는 최적화 되지 않았다. 학습 비율(learning rate)이나 디케이 (decay)를 변경하거나, 다른 방법으로 여러분의 모델의 가중치들을 초기하는 것을 시도 할수 있다. 또한 `seq2seq_model.py` 에 있는 디폴트인 `GradientDescentOptimizer` 를 더 혁신적인 `AdagradOptimizer` 로 변경할 수 있다. 이러한 것들을 시도하고 결과가 어떻게 향상 되는지 보자.

마지막으로, 위에서 나타난 모델은 번역뿐만 아니라 어떤 시퀀스-투-시퀀스 작업에 대해서도 사용되어 질수 있다. 하나의 시퀀스을 트리로 변환하기 원한다고 할지라도, 예를 들어 파싱 트리를 생성하기, 위처럼 동일한 모델은 [Vinyals & Kaiser et al., 2014 \(pdf\)](#)에서 보여진대로 최고의 결과를 준다. 그래서 여러분 자신의 번역기를 만들수 있을 뿐만 아니라 또한 파서, 챗봇, 또는 당신 머리에 떠 오르는 어떤 프로그램도 만들수 있다. 실험해보라!

# SyntaxNet

(v0.9)

## 소개

SyntaxNet는 텐서플로우를 위한 뉴럴 네트워크 방식의 자연어 처리 프레임워크입니다.

## SyntaxNet의 기본 튜토리얼

해당 [튜토리얼](#) 은 다음 항목들을 설명하고 있습니다:

- SyntaxNet를 설치하는 방법
- SyntaxNet 패키지에 포함된 Parsey McParseface 파서를 사용하는 방법
- 사용자의 품사 태거를 훈련하는 방법
- 사용자의 파서를 훈련하는 방법

# 만델브로트 집합

만델브로트 집합(Mandelbrot set)을 시각화하는 것은 머신 러닝과는 별 상관이 없지만, TensorFlow를 일반적인 수학에 사용하는 방법에 대한 재미있는 예시로 활용할 수 있습니다. 이 문서에서 소개되는 시각화 과정은 비교적 단순한(naive) 구현 방법을 사용했지만, 요점을 잘 보여줍니다. (더 아름다운 시각화를 위해, 더 정교한 구현 방법을 사용할 수도 있을 것입니다.)

참고: 이 튜토리얼은 IPython notebook에서의 구현을 바탕으로 작성되었습니다.

## 기본 설정

시작하기 전, 몇 개의 라이브러리를 import 해야 합니다.

```
시뮬레이션을 위한 라이브러리 import
import tensorflow as tf
import numpy as np

시각화를 위한 라이브러리 import
import PIL.Image
from io import BytesIO
from IPython.display import Image, display
```

이제 우리는 반복 횟수가 주어졌을 때 그림을 그리기 위한 함수를 정의합니다.

```
def DisplayFractal(a, fmt='jpeg'):
 """반복 횟수들의 배열을 다채로운 프랙탈 이미지로 나타냅니다."""
 a_cyclic = (6.28*a/20.0).reshape(list(a.shape)+[1])
 img = np.concatenate([10+20*np.cos(a_cyclic),
 30+50*np.sin(a_cyclic),
 155-80*np.cos(a_cyclic)], 2)
 img[a==a.max()] = 0
 a = img
 a = np.uint8(np.clip(a, 0, 255))
 f = BytesIO()
 PIL.Image.fromarray(a).save(f, fmt)
 display(Image(data=f.getvalue()))
```

## 세션 및 변수 초기화

이러한 작업을 위해서는 주로 인터랙티브 세션(interactive session)을 사용합니다. (일반적인 세션도 괜찮습니다.)

```
sess = tf.InteractiveSession()
```

NumPy와 TensorFlow를 자유롭게 연결해 쓸 수 있다는 것은 편리합니다.

```
Numpy를 이용하여 [-2,2]x[-2,2]의 복소수에 대한 2차원 배열 생성

Y, X = np.mgrid[-1.3:1.3:0.005, -2:1:0.005]
Z = X+1j*Y
```

이제 TensorFlow 텐서들을 정의하고 초기화합니다.

```
xs = tf.constant(Z.astype(np.complex64))
zs = tf.Variable(xs)
ns = tf.Variable(tf.zeros_like(xs, tf.float32))
```

TensorFlow의 변수는 사용하기 전에 명시적으로 초기화해야 합니다.

```
tf.initialize_all_variables().run()
```

## 정의 및 계산 실행

계산 과정에 대한 추가적인 부분을 더 작성하고...

```
새로운 z값의 계산: z^2 + x
zs_ = zs*zs + xs

새로운 z값은 발산(diverge)했을까?
not_diverged = tf.complex_abs(zs_) < 4

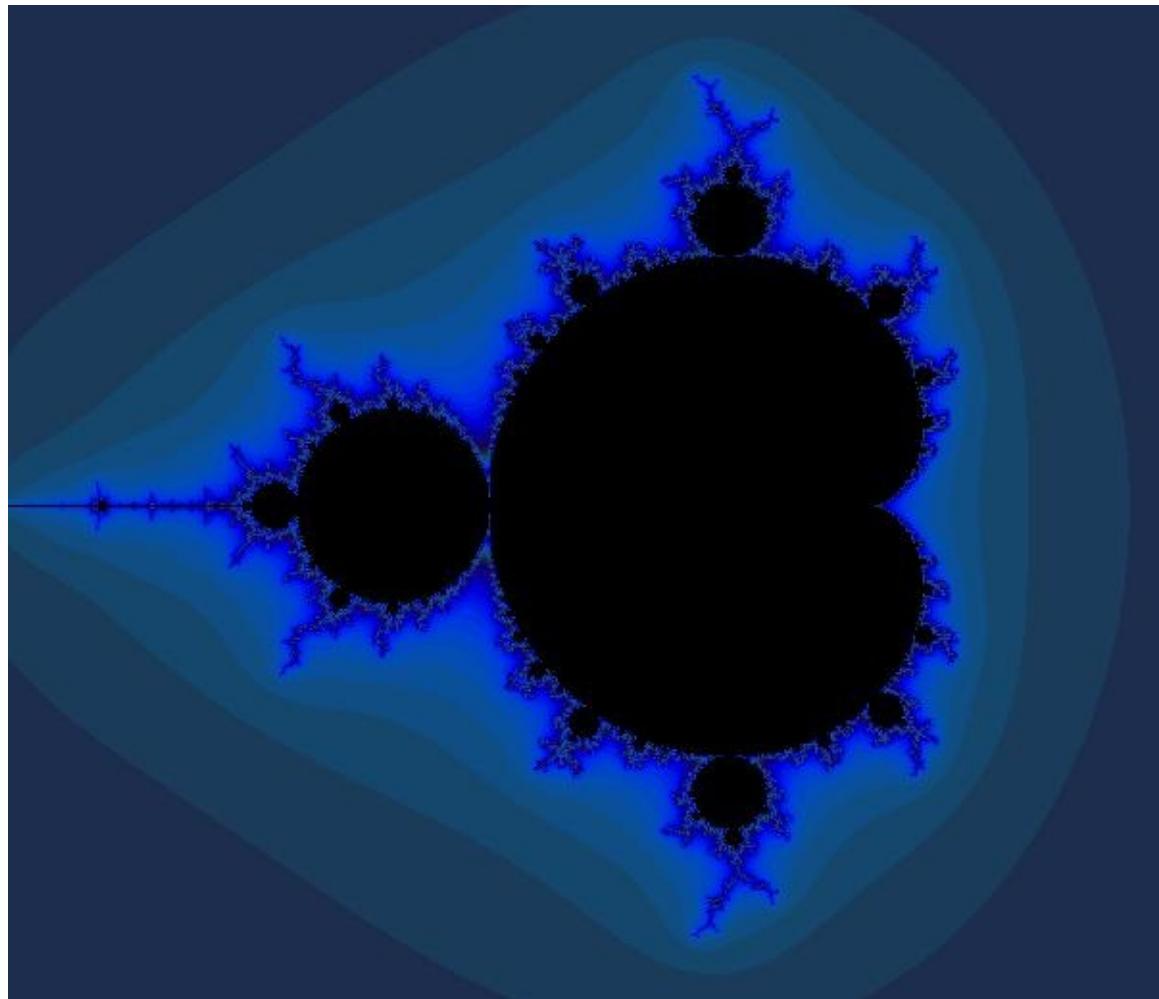
z값 및 반복 횟수 업데이트
#
참고: 이 코드는 zs가 발산한 뒤에도 계속 계산을 진행합니다.
이것은 낭비이며, 약간 더 복잡하더라도 더 좋은 방법이
있을 것입니다.
#
step = tf.group(
 zs.assign(zs_),
 ns.assign_add(tf.cast(not_diverged, tf.float32))
)
```

... 이것을 200번 정도 실행합니다.

```
for i in range(200): step.run()
```

결과를 보도록 하죠.

```
DisplayFractal(ns.eval())
```



나쁘지 않군요!

# 편미분 방정식

텐서플로우는 머신러닝만을 위한 것이 아닙니다. 편미분을 시뮬레이션 하기 위해 텐서플로우를 사용한 (약간 지루한) 예제가 있습니다. 빗방울이 떨어지는 사각형 연못의 표면을 시뮬레이션 해보겠습니다.

Note: 이 튜토리얼은 IPython notebook 으로 작성되었습니다.

## 기본 설정

몇 가지를 임포트 해야합니다.

```
#시뮬레이션을 위한 라이브러리 임포트
import tensorflow as tf
import numpy as np

#보여주기위한 임포트
import PIL.Image
from io import BytesIO
from IPython.display import clear_output, Image, display
```

연못 표면의 상태를 이미지로 보여주는 함수.

```
def DisplayArray(a, fmt='jpeg', rng=[0,1]):
 """Display an array as a picture."""
 a = (a - rng[0])/float(rng[1] - rng[0])*255
 a = np.uint8(np.clip(a, 0, 255))
 f = BytesIO()
 PIL.Image.fromarray(a).save(f, fmt)
 clear_output(wait = True)
 display(Image(data=f.getvalue()))
```

여기서 편의를 위해 인터랙티브 텐서플로우 세션을 시작합니다. 만약 .py 파일로 실행한다면 일반 세션도 잘 동작합니다.

```
sess = tf.InteractiveSession()
```

## 편의를 위한 함수

```

def make_kernel(a):
 """2차 배열을 콘볼루션 커널로 변환"""
 a = np.asarray(a)
 a = a.reshape(list(a.shape) + [1,1])
 return tf.constant(a, dtype=1)

def simple_conv(x, k):
 """간단한 2차 콘볼루션 연산"""
 x = tf.expand_dims(tf.expand_dims(x, 0), -1)
 y = tf.nn.depthwise_conv2d(x, k, [1, 1, 1, 1], padding='SAME')
 return y[0, :, :, 0]

def laplace(x):
 """2차 배열의 라플라시안 계산"""
 laplace_k = make_kernel([[0.5, 1.0, 0.5],
 [1.0, -6., 1.0],
 [0.5, 1.0, 0.5]])
 return simple_conv(x, laplace_k)

```

## PDE 정의

자연에서 발견되는 대부분의 연못과 같이 우리의 연못은  $500 \times 500$  의 정사각형입니다.

```
N = 500
```

연못을 만들고 빗방울을 떨어뜨려봅시다.

```

조건 초기화 -- 몇개의 빗방울이 연못에 떨어뜨립니다

전부 0으로 설정
u_init = np.zeros([N, N], dtype=np.float32)
ut_init = np.zeros([N, N], dtype=np.float32)

몇개의 빗방울이 연못의 임의 위치에 떨어뜨립니다
for n in range(40):
 a,b = np.random.randint(0, N, 2)
 u_init[a,b] = np.random.uniform()

DisplayArray(u_init, rng=[-0.1, 0.1])

```



이제 미분을 적용해봅시다.

```
매개변수:
eps -- 시간 해상도
damping -- 파고감쇠
eps = tf.placeholder(tf.float32, shape=())
damping = tf.placeholder(tf.float32, shape=())

시뮬레이션 상태를 위한 변수 생성
U = tf.Variable(u_init)
Ut = tf.Variable(ut_init)

이산화된 PDE 갱신 규칙
U_ = U + eps * Ut
Ut_ = Ut + eps * (laplace(U) - damping * Ut)

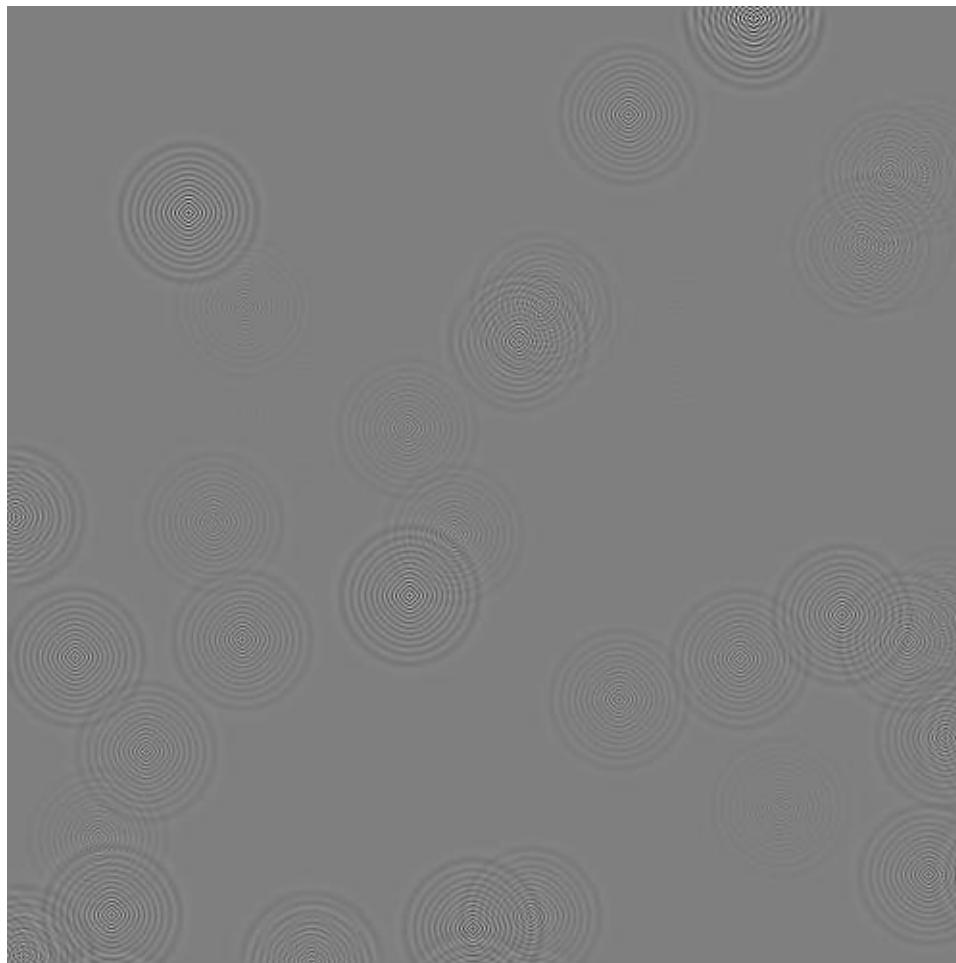
상태 갱신 명령
step = tf.group(
 U.assign(U_),
 Ut.assign(Ut_))
```

## 시뮬레이션 실행

이제 흥미로운 부분입니다 -- 간단한 for 반복문으로 실시간 진행합니다.

```
초기 조건에 대한 상태 초기화
tf.initialize_all_variables().run()

1000 번의 PDE 수행
for i in range(1000):
 # 단계 시뮬레이션
 step.run({eps: 0.03, damping: 0.04})
 DisplayArray(U.eval(), rng=[-0.1, 0.1])
```



보세요! 물결을!

## 하우투

### 변수: 생성, 초기화, 저장, 복구

텐서플로우 변수들은 텐서를 가지고 있는 메모리상의 버퍼입니다. 그것들을 이용해서 학습중에 어떻게 모델 파라미터들을 보관하고 업데이트 하는지를 설명합니다.

[튜토리얼 보기](#)

### 텐서플로우 구조 입문

텐서플로우 구조를 사용하여 어느정도 규모가 되는 모델을 학습하는 방법을 차근차근 상세하게 설명합니다. MNIST 필기 숫자 인식을 예제로 사용합니다.

[튜토리얼 보기](#)

### 텐서보드: 학습 시각화

텐서보드는 모델들의 학습 및 평가를 시각화 해주는 유용한 툴입니다. 이 튜토리얼을 통해 텐서보드를 어떻게 만들고 동작시키는지 알 수 있습니다. 그리고 Summary 연산자를 추가하여, 텐서보드가 시각화를 위해 사용하게 되는 이벤트 파일에 자동적으로 데이터를 출력하는 방법을 배웁니다.

[튜토리얼 보기](#)

### 텐서보드: 그래프 시각화

데이터 플로우를 그래프로 이해하고 수정을 가할 수 있도록, 텐서보드에서 그래프 시각화 툴을 사용하는 방법을 배웁니다.

[튜토리얼 보기](#)

## 데이터 로딩

텐서플로우 프로그램에 데이터를 로딩하는 세가지 방법을 설명합니다: Feeding, Reading, Preloading, 이렇게 세가지입니다.

[튜토리얼 보기](#)

## 분산처리

텐서플로우 서버 클러스터를 이용하여 텐서플로우 프로그램을 실행하는 방법을 설명합니다.

[튜토리얼 보기](#)

## 쓰레드와 큐

비동기 학습과 동시 학습을 구현하기 위한 다양한 텐서플로우 구조들을 설명합니다.

[튜토리얼 보기](#)

## 커스텀 연산자

텐서플로우는 이미 많은 수의 연산자들을 제공하고 있습니다. 스스로 커스텀 연산자를 만들 필요가 있는 경우, 이 튜토리얼을 참고하십시오.

[튜토리얼 보기](#)

## 텐서플로우 코드 작성 스타일

코드의 가독성을 높이고, 에러를 줄이며, 일관성을 장려하기 위해, 텐서플로우 개발자와 사용자들이 따라야 할 스타일 가이드입니다.

[스타일 가이드 보기](#)

## 문서화

텐서플로우의 도큐먼테이션들은 대부분 소스 코드들로부터 생성되었습니다. 이 튜토리얼을 통해, 문서의 포맷, 스타일 가이드, 그리고 소스로부터 업데이트된 도큐먼테이션을 생성하는 방법을 배울 수 있습니다.

[튜토리얼 보기](#)

## 커스텀 데이터 포맷

상당한 양의 커스텀 데이터를 가지고 있는 경우, 텐서플로우가 데이터 본래의 포맷으로 직접 읽어 들이게 하는 방법입니다.

[튜토리얼 보기](#)

## GPU 사용하기

GPU상에서 모델을 구축하고 실행하는 방법을 설명하는 튜토리얼입니다.

[튜토리얼 보기](#)

## 변수 공유

큰 모델을 하나 이상의 GPU에서 돌리거나, 복잡한 LSTM 또는 RNN을 전개하는 경우에, 모델을 만드는 코드상의 여러 곳으로부터 동일한 변수 객체에 접근할 필요가 자주 생깁니다.

그것을 실현하기 위한 "변수 범위(Variable Scope)" 방식이라는 것이 있습니다.

[튜토리얼 보기](#)

## 모델 파일

텐서플로우 모델이 저장되는 포맷을 이해하는 것은, 모델을 읽어들이고, 분석하고, 수정하는 데에 도움을 줍니다. 이 튜토리얼을 통해서 모델이 저장되는 포맷에 대해 자세하게 알 수 있습니다.

[튜토리얼 보기](#)

## 트랜스퍼 학습을 이용한 부분 학습

Inception과 같은 완성된 형태의 인식 모델을 학습시키는 데에는, 많은 수의 이미지와 오랜 시간이 필요합니다. 학습이 끝난 모델의 최종 레이어만 다시 학습함으로써, 다른 카테고리의 대상을 인식 할 수 있게 하는 트랜스퍼 학습 기법을 설명합니다. 모델을 처음부터 다시 학습하는 것보다 훨씬 빠르고 쉬운 방법입니다.

[튜토리얼 보기](#)

## 모델 Export 와 Import

모델을 사용하는 데에 필요한 일체를 Export하고, 나중에 Import하는 방법을 설명합니다.

[튜토리얼 보기](#)

## 텐서플로우로 뉴럴 네트워크를 정량화 하기

플로팅 포인트 모델을, 8비트 파라미터와 연산을 사용하도록 정량화(양자화)하는 방법을 배웁니다.  
프로그램의 이면에서 정량화가 어떻게 동작하는지도 설명합니다.

[튜토리얼 보기](#)

# 변수: 생성, 초기화, 저장, 복구

(v0.12)

모델을 학습 시킬 때, 매개 변수(parameter) 업데이트와 유지를 위해 [변수\(Variables\)](#)를 사용합니다. 변수는 텐서를 포함하는 인-메모리 버퍼입니다. 변수는 반드시 명시적으로 초기화해야 합니다. 그리고 학습 중 혹은 학습 후에 디스크에 저장할 수 있습니다. 저장된 값들을 모델 실행이나 분석을 위해 나중에 복원할 수도 있습니다.

이 문서는 다음 TensorFlow 클래스를 참조합니다. 링크를 따라가 API에 대한 자세한 설명이 있는 문서를 살펴보세요.

- [tf.Variable](#) 클래스.
- [tf.train.Saver](#) 클래스.

## 생성

[변수\(Variable\)](#)를 생성할 때 `Variable()` 생성자의 초기값으로 `Tensor`를 전달받게 됩니다. TensorFlow는 [상수\(constants\)](#) 또는 [임의\(random\)](#)의 값으로 초기화 하는 다양한 명령어(op)를 제공합니다.

이 모든 명령어는 `Tensor`들의 형태(shape)을 지정해줘야 합니다. 이 형태가 자동적으로 변수(Variable)의 형태가 됩니다. 변수는 대부분 고정된 형태를 가지지만 TensorFlow에서는 변수의 형태를 수정하기 위한(`reshape`) 고급 매커니즘도 제공합니다.

```
두 변수를 생성.
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35),
 name="weights")
biases = tf.Variable(tf.zeros([200]), name="biases")
```

`tf.Variable()` 호출은 그래프에 여러 오퍼레이션(op)을 추가합니다:

- `variable` 오퍼레이션은 그 변수의 값을 가지고 있습니다.
- `tf.assign` 오퍼레이션을 이용해서 변수의 초기값을 설정할 수 있습니다.
- 예제에서 편차(bias)를 설정할 때 쓰이는 `zeros` 오퍼레이션같이 초기값을 설정하는 오퍼레이션도 그래프에 추가됩니다.

`tf.Variable()`에서 반환되는 값은 Python 클래스의 인스턴스인 `tf.Variable`입니다.

## 디바이스에 변수 배치하기

변수를 생성할 때 `with tf.device(...):` 블록을 사용하여 특정 디바이스에 배치할 수 있습니다.

```
변수를 CPU에 배치
with tf.device("/cpu:0"):
 v = tf.Variable(...)

변수를 GPU에 배치.
with tf.device("/gpu:0"):
 v = tf.Variable(...)

변수를 특정 서버 테스크(task)에 배치.
with tf.device("/job:ps/task:7"):
 v = tf.Variable(...)
```

**주의** `v.assign()` 을 사용하거나 `tf.train.Optimizer`에서 매개변수를 변경하는 등 변수를 변경하는 작업을 할 때는 반드시 배치된 변수와 같은 디바이스에서 실행해야 합니다. 이 작업들을 생성할 때 다른 디바이스에 배치하라는 명령은 무시될 것입니다.

디바이스 배치는 설정을 복사해서 실행할 때 특히 문제가 될 수 있습니다. 설정을 복사한 모델에서 디바이스를 간단하게 설정하게 도와주는 디바이스 관련 함수는

`tf.train.replica_device_setter()`에서 자세한 내용을 확인할 수 있습니다.

## 초기화

변수 초기화는 모델의 다른 연산을 실행하기 전에 반드시 명시적으로 실행해야 합니다. 가장 쉬운 방법은 모든 변수를 초기화 하는 연산을 모델 사용 전에 실행하는 것입니다.

다른 방법으로는 체크포인트 파일에서 변수 값을 복원할 수 있습니다. 다음 챕터에서 다를 것입니다.

변수 초기화를 위한 작업(op)을 추가하기 위해 `tf.global_variables_initializer()`를 사용해봅시다. 모델을 모두 만들고 세션에 올린 후 이 작업을 실행할 수 있습니다.

```

두 변수를 생성
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35),
 name="weights")
biases = tf.Variable(tf.zeros([200]), name="biases")
...
변수 초기화 오퍼레이션을 초기화
init_op = tf.global_variables_initializer()

나중에 모델을 실행할때
with tf.Session() as sess:
 # 초기화 오퍼레이션을 실행
 sess.run(init_op)
 ...
 # 모델 사용
 ...

```

## 다른 변수값을 참조하여 초기화 하기

변수를 초기화할 때 다른 변수의 초기값을 참조해야 하는 경우가 있습니다.

`tf.global_variables_initializer()` 연산으로 모든 변수를 초기화 해야 하는 경우에는 조심해야 합니다.

다른 변수의 값을 이용해서 새로운 변수를 초기화할 때는 다른 변수의 `initialized_value()` 속성을 사용할 수 있습니다. 어떤 변수의 초기값을 바로 새로운 변수의 초기값으로 사용할 수도 있고 새로운 변수값을 위한 계산용으로 쓸 수도 있습니다.

```

랜덤 값으로 새로운 변수 초기화
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35),
 name="weights")
weights와 같은 값으로 다른 변수 초기화
w2 = tf.Variable(weights.initialized_value(), name="w2")
weights의 2배 값으로 다른 변수 초기화
w_twice = tf.Variable(weights.initialized_value() * 2.0, name="w_twice")

```

## 커스텀 초기화

`tf.global_variables_initializer()` 는 모델의 모든 변수를 초기화합니다. 하지만 사용자가 직접 초기화할 변수 리스트를 만들 수도 있습니다. [Variables Documentation](#)에서 변수 초기화 여부를 확인하는 기능을 포함한 옵션들을 볼 수 있습니다.

## 저장과 복구

모델을 저장하고 복구하기 위한 가장 쉬운 방법은 `tf.train.Saver` 오브젝트를 이용하는 것입니다. 이 오브젝트를 통해 그래프 전체 변수 또는 그래프의 지정된 리스트의 변수에 `save` 와 `restore` 오퍼레이션(op)을 추가할 수 있습니다. `Saver` 오브젝트는 이 오퍼레이션을 실행하는 메서드와 읽고 쓰는 체크포인트 파일의 지정된 경로를 제공합니다.

## 체크 포인트 파일

변수를 변수 이름과 텐서 값을 매핑해놓은 바이너리 파일에 저장할 수 있습니다.(역자 주: 이 바이너리 파일을 체크포인트 파일이라고 부릅니다. 확장자는 `.ckpt` 를 사용합니다.)

`Saver` 오브젝트를 생성할 때, 체크포인트 파일에 있는 변수명을 정할 수 있습니다. 변수명은 기본 적(default)으로 각 변수의 `variable.name` 을 사용합니다.

체크포인트 안에 어떤 변수가 있는지 알기 위해선, `inspect_checkpoint` 라이브러리를 사용할 수 있고 `print_tensors_in_checkpoint_file` 함수를 이용할 수도 있습니다.

## 변수 저장

`tf.train.Saver()` 로 `Saver` 를 생성하여 모든 변수를 관리할 수 있습니다.

```
몇개의 변수를 생성
v1 = tf.Variable(..., name="v1")
v2 = tf.Variable(..., name="v2")
...
변수 초기화를 위한 오퍼레이션 추가
init_op = tf.global_variables_initializer()

모든 변수의 저장과 복구를 위한 오퍼레이션 추가
saver = tf.train.Saver()

모델 실행, 변수 초기화, 몇 가지의 작업 실행, 디스크에 변수 저장
with tf.Session() as sess:
 sess.run(init_op)
 # 모델을 사용하여 작업
 ...
 # 디스크에 변수를 저장
 save_path = saver.save(sess, "/tmp/model.ckpt")
 print("Model saved in file: %s" % save_path)
```

## 변수 복구

똑같은 `Saver` 오브젝트를 변수를 복구하는데 사용할 수도 있습니다. 변수를 파일에서 복구할 때는 변수를 초기화 할 필요가 없는 점을 주의하세요.

```

몇개의 변수를 생성
v1 = tf.Variable(..., name="v1")
v2 = tf.Variable(..., name="v2")
...
변수 초기화를 위한 오퍼레이션 추가
saver = tf.train.Saver()

모델 실행, 변수 초기화, 몇 가지의 작업 실행, 디스크에 변수 저장
with tf.Session() as sess:
 # 디스크에서 변수를 복구
 saver.restore(sess, "/tmp/model.ckpt")
 print("Model restored.")
 # 모델을 사용하여 작업
 ...

```

## 저장 및 복구할 변수들의 선택

만약 `tf.train.Saver()` 에 아무런 인자도 전달하지 않는다면, `saver`는 그래프 안의 모든 변수를 다룹니다. 각 변수는 만들어질 때 부여된 이름으로 저장됩니다.

체크포인트 파일의 변수에 명시적으로 이름을 지정하는 것이 유용할 수 있습니다. 예를 들어, `"weights"` 라는 변수로 모델을 트레이닝 했지만 이 값을 `"params"` 라는 새로운 변수로 불러오고 싶을 수 있으니까요.

또한, 모델에 사용된 변수 중에 일부만 저장하거나 복원할 때도 유용합니다. 예를 들어, 5단 레이어를 가지는 신경망을 학습시켰고 나중에 6단 레이어를 가지는 새로운 모델을 학습시키고 싶을 때, 5단 레이어로 부터 훈련된 값을 새로운 모델(6단 레이어)의 첫 5단 레이어에 복원할 수 있습니다.

`tf.train.Saver()` 오브젝트에 파이썬 dictionary 타입의 데이터를 전달해서 저장할 이름과 변수를 쉽게 지정할 수 있습니다. 이때 dictionary의 키는 사용할 이름이며, 값(value)은 관리할 변수입니다.

주의:

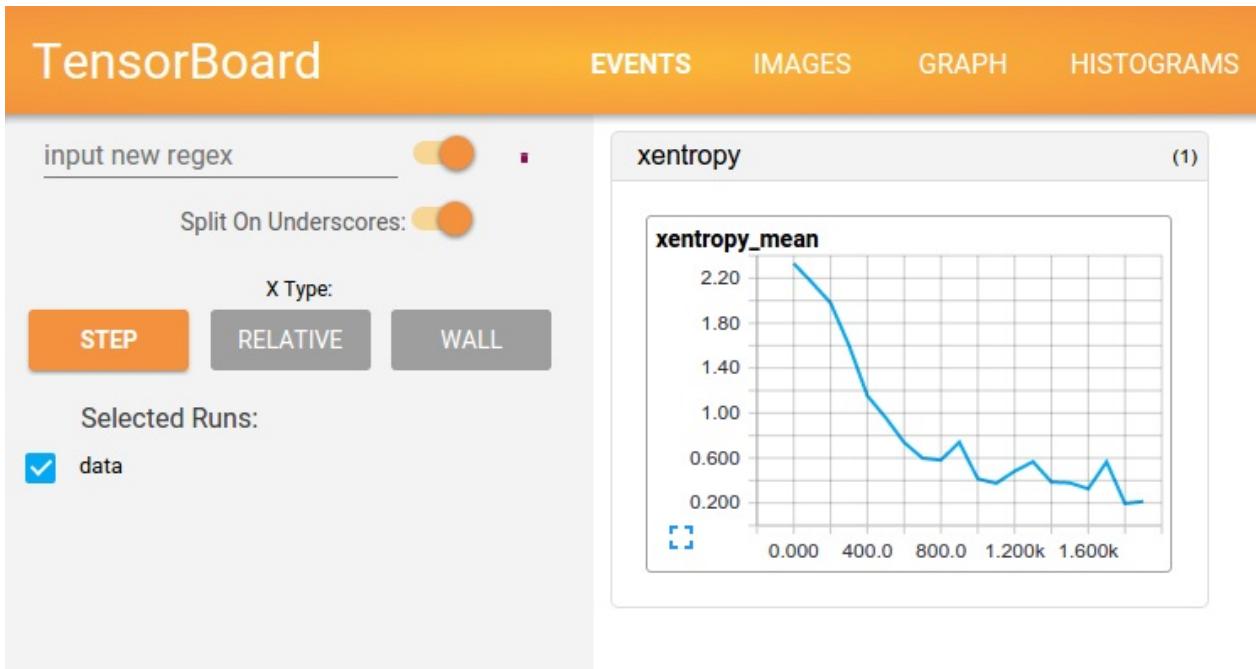
- 모델 변수의 일부분을 저장하고 복원해야 하는 경우 당신이 원하는 만큼 `saver` 오브젝트를 만들 수 있습니다. 같은 변수가 여러 개의 `saver` 오브젝트의 리스트에 있을 수 있습니다. 이 같은 `saver`의 `restore()` 메서드를 실행하여야만 변경 가능합니다.
- 만약 모델 변수의 일부분을 세션의 시작에서 복원 한다면 다른 변수들을 초기화하는 오퍼레이션을 실행하여야 합니다. `tf.initialize_variables()` 에서 자세한 사항을 확인하세요

```
몇개의 변수 생성
v1 = tf.Variable(..., name="v1")
v2 = tf.Variable(..., name="v2")
...
"my_v2"라는 이름을 이용하여 'v2'를 저장하고 복구 하는 오퍼레이션 추가
saver = tf.train.Saver({"my_v2": v2})
그 뒤 saver 오브젝트를 평상시 사용하던대로 사용
...
```

# TensorBoard: 학습 시각화

(v0.12)

TensorFlow를 쓸려는 연산은 거대한 심층 신경망 학습처럼 복잡하고 내용이 헷갈리는 것들입니다. 이해, 디버깅, 또 TensorFlow 최적화를 쉽게 만들기 위해서 시각화 도구 세트를 하나 넣어뒀습니다. 바로 TensorBoard입니다. 여러분은 TensorBoard를 이용해서 TensorFlow의 그래프를 시각화하고 그래프를 실행해서 얻은 행렬을 도표로 나타내고 이미지 파일같은 부가 데이터를 보여줄 수도 있습니다. TensorBoard가 완전히 셋팅되면 이렇게 보일 것입니다:



이 튜토리얼은 간단한 TensorBoard 사용법을 안내하는 것이 목적입니다. 참고할 만한 다른 문서들도 있어요! [TensorBoard 길라잡이](#)는 TensorBoard 사용법에 대한 팁이나 디버깅 방법 등 더 많은 정보를 담고 있습니다.

## 데이터 저장(serialize)

TensorBoard는 TensorFlow를 실행할 때 만들 수 있는 요약 데이터(summary data)가 들어간 TensorFlow 이벤트 파일을 이용합니다. TensorBoard에서 요약 데이터가 보여주는 라이프 사이클을 설명해드리겠습니다.

가장 먼저 요약 데이터를 얻고 싶은 TensorFlow 그래프를 만들어야겠죠. 그리고 [summary operations](#)을 이용해서 어느 노드를 기록할 지 결정합니다.

예를 들어 지금 곱집합 신경망을 학습시켜서 MNIST 숫자들(역자 주: 프로그래밍의 'hello world'처럼 이미지 인식에서 가장 기초적인 예제)을 인식하려고 있다고 해봅시다. 아마도 학습률이 어떻게 달라지는지, 목표함수가 어떻게 바뀌는지를 기록하고 싶을 것입니다. 학습률과 손실을 각각 만들어 내는 노드에 `scalar_summary` 작업(op)을 추가해서 데이터를 모을 수 있습니다. 그리고 각 `scalar_summary`에는 학습률이나 손실함수 같은 태그를 붙일 수도 있습니다.

특정 층에서 발생한 액티베이션, 그래디언트 혹은 가중치의 분포를 시각화하고 싶을 것 같기도 합니다. 이럴 때는 그래디언트 결과물이나 가중치 변수에 `histogram_summary` 작업(op)을 추가해서 데이터를 모을 수 있습니다.

가능한 모든 요약 작업(summary operation)들은 [summary operations](#) 문서를 확인하시기 바랍니다.

TensorFlow의 작업(op)들은 이용자가 그 작업이나 연관된 다른 작업을 실행시킬 때까지 아무 것도 하지 않습니다. 우리가 만든 요약 노드들은 그래프에서 지연적인 존재입니다. 아무 노드도 요약 노드들의 결과에 영향을 받지 않거든요. 그렇기 때문에 요약 데이터를 만들려면 반드시 모든 요약 노드들을 실행시켜야 합니다. 일일이 손으로 관리하는 것은 짜증나는 일이니까 `tf.merge_all_summaries` 를 사용해서 요약 노드들을 하나로 합쳐서 한 번에 모든 요약 데이터를 만들 수 있게 할 수 있습니다.

이제 통합된 요약 작업(summary op)을 실행시키면 모든 요약 데이터를 담은 `summary` 프로토버퍼 오브젝트를 만들 수 있습니다. 마지막으로 이 요약 데이터를 디스크에 저장하기 위해 프로토버퍼 오브젝트를 `tf.train.SummaryWriter` 로 넘겨야 합니다.

`SummaryWriter` 을 쓰기 위해서는 모든 요약 데이터를 저장할 디렉토리인 `logdir`을 정해줘야 합니다. `SummaryWriter` 는 때에 따라 그래프 도 이용할 수 있습니다. 만약 그래프 오브젝트를 이용하는 경우에는 TensorBoard가 텐서 형태(tensor shape) 정보에 덧붙여서 그래프도 보여줄 것입니다. 시각화된 그래프를 보면 그래프의 플로우에 대해서 더 잘 이해할 수 있겠죠. 자세한 내용은 [Tensor shape information](#)을 참조하세요.

드디어 그래프도 수정했고 `SummaryWriter` 도 얻었습니다. 네트워크를 실행할 준비가 끝났어요! 원한다면 통합된 요약 작업을 매 단계마다 실행시켜서 엄청난 학습 데이터를 기록할 수 있습니다. 그런데 매 단계마다 저장하면 필요한 것보다 훨씬 많을 거예요. `n` 단계마다 요약 작업을 시키는 것을 고려해봅시다.

아래의 코드 예시는 [초보자를 위한 MNIST](#)에 매 10단계마다 요약 작업을 하도록 조금 변형한 코드입니다. 이 코드를 실행시키고 `tensorboard --logdir=/tmp/mnist_logs` 를 실행하면 학습하는 동안 가중치나 정확도같은 통계값이 어떻게 변했는지 볼 수 있습니다. 아래 코드는 일부를 발췌한 것이니 코드 전체는 [여기](#)를 참조하세요.

```
def variable_summaries(var, name):
 """Attach a lot of summaries to a Tensor."""
 with tf.name_scope('summaries'):
 mean = tf.reduce_mean(var)
 tf.scalar_summary('mean/' + name, mean)
```

```

with tf.name_scope('stddev'):
 stddev = tf.sqrt(tf.reduce_sum(tf.square(var - mean)))
tf.scalar_summary('stddev/' + name, stddev)
tf.scalar_summary('max/' + name, tf.reduce_max(var))
tf.scalar_summary('min/' + name, tf.reduce_min(var))
tf.histogram_summary(name, var)

def nn_layer(input_tensor, input_dim, output_dim, layer_name, act=tf.nn.relu):
 """간단한 신경망 레이어를 만들기 위한 재사용가능 코드

 행렬곱을 하고 편차(bias)를 더하고 비선형화를 위한 액티베이션 함수로 ReLU
 (역자 주: Rectified Linear Unit. 자주 유용하게 쓰이는 액티베이션 함수.)를 사용.
 가독성을 위해 name scoping을 했고 다양한 요약 작업을 추가.

 """
 # name scope를 추가하여 그래프의 계층들을 논리적으로 분류
 with tf.name_scope(layer_name):
 # 레이어의 가중치를 저장할 변수
 with tf.name_scope('weights'):
 weights = weight_variable([input_dim, output_dim])
 variable_summaries(weights, layer_name + '/weights')
 with tf.name_scope('biases'):
 biases = bias_variable([output_dim])
 variable_summaries(biases, layer_name + '/biases')
 with tf.name_scope('Wx_plus_b'):
 preactivate = tf.matmul(input_tensor, weights) + biases
 tf.histogram_summary(layer_name + '/pre_activations', preactivate)
 activations = act(preactivate, 'activation')
 tf.histogram_summary(layer_name + '/activations', activations)
 return activations

hidden1 = nn_layer(x, 784, 500, 'layer1')

with tf.name_scope('dropout'):
 keep_prob = tf.placeholder(tf.float32)
 tf.scalar_summary('dropout_keep_probability', keep_prob)
 dropped = tf.nn.dropout(hidden1, keep_prob)

y = nn_layer(dropped, 500, 10, 'layer2', act=tf.nn.softmax)

with tf.name_scope('cross_entropy'):
 diff = y_ * tf.log(y)
 with tf.name_scope('total'):
 cross_entropy = -tf.reduce_mean(diff)
 tf.scalar_summary('cross entropy', cross_entropy)

with tf.name_scope('train'):
 train_step = tf.train.AdamOptimizer(FLAGS.learning_rate).minimize(
 cross_entropy)

with tf.name_scope('accuracy'):
 with tf.name_scope('correct_prediction'):
 correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
 with tf.name_scope('accuracy'):

```

```

accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
tf.scalar_summary('accuracy', accuracy)

모든 요약 내용을 합치고 /tmp/mnist_logs에 기록합니다.(기본 경로)
merged = tf.merge_all_summaries()
train_writer = tf.train.SummaryWriter(FLAGS.summaries_dir + '/train',
 sess.graph)
test_writer = tf.train.SummaryWriter(FLAGS.summaries_dir + '/test')
tf.initialize_all_variables().run()

```

SummaryWriters 를 초기화 시킨 후에는 모델을 학습시키거나 테스트할 때 요약 내용(summary)을 SummaryWriters 에 추가해야 합니다.

```

모델을 학습시키고 요약 내용을 기록해 봅시다.
매 10 단계마다 테스트 세트의 정확도를 측정하고 그 요약 내용을 기록합니다.
매 단계마다 train_step을 실행하고 학습 내용을 추가합니다.

def feed_dict(train):
 """TensorFlow feed_dict를 만들기: data를 Tensor 플레이스홀더에 매칭"""
 if train or FLAGS.fake_data:
 xs, ys = mnist.train.next_batch(100, fake_data=FLAGS.fake_data)
 k = FLAGS.dropout
 else:
 xs, ys = mnist.test.images, mnist.test.labels
 k = 1.0
 return {x: xs, y_: ys, keep_prob: k}

for i in range(FLAGS.max_steps):
 if i % 10 == 0: # 요약 내용과 테스트 세트 정확도를 기록합니다.
 summary, acc = sess.run([merged, accuracy], feed_dict=feed_dict(False))
 test_writer.add_summary(summary, i)
 print('Accuracy at step %s: %s' % (i, acc))
 else: # 학습한 세트에 대한 요약 내용을 기록하고 학습시킵니다.
 summary, _ = sess.run([merged, train_step], feed_dict=feed_dict(True))
 train_writer.add_summary(summary, i)

```

이제 TensorBoard를 이용해서 이 데이터를 시각화할 준비가 끝났습니다.

## TensorBoard 실행

TensorBoard를 실행해보기 위해서 아래 명령을 이용해 봅시다.

```
tensorboard --logdir=path/to/log-directory
```

`logdir` 은 데이터를 `SummaryWriter` 가 데이터를 저장(serialize)해놓은 디렉토리를 가리킵니다.  
만약 `logdir` 디렉토리에 다른 실행에 대한 데이터를 저장해놓은 하위 디렉토리가 있다면  
TensorBoard는 모두 다 시각화해서 보여줄 것입니다. 한 번 TensorBoard가 실행되면 웹브라우저  
주소창에 `localhost:6006` 을 입력해서 볼 수 있습니다.

TensorBoard 화면 오른쪽 상단에서 네비게이션 탭을 찾을 수 있습니다. 각 탭들은 저장된 데이터  
세트를 의미합니다.

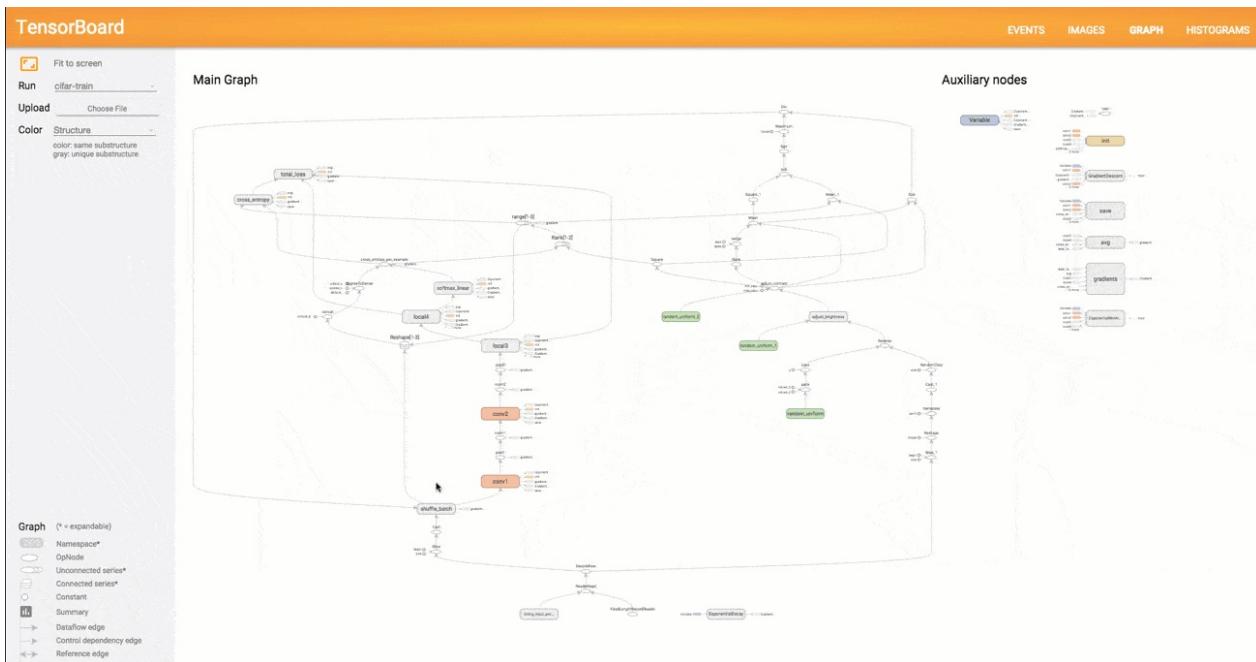
그래프를 시각화하는 그래프 탭에 대한 더 자세한 정보는 [TensorBoard: 그래프 시각화](#)를 참조하시  
기 바랍니다.

TensorBoard에 대한 전반적인 정보를 더 얻고 싶으시면 [TensorBoard 길라잡이](#)를 참조해 주세요.

# TensorBoard: 그래프 시각화

(v1.0)

TensorFlow 연산 그래프(computation graph)는 강력하지만 복잡합니다. 그래프를 시각화하면 이해와 디버그에 도움이 됩니다. 여기에 시각화 작동 예시가 있습니다.



*TensorFlow* 그래프의 시각화.

그래프를 보려면, TensorBoard를 실행할 때 로그 디렉토리를 입력한 후 상단 그래프 탭을 클릭하고 왼쪽 위 모서리에 있는 메뉴를 통해 적당한 작업을 선택하면 됩니다. TensorBoard를 어떻게 실행하는지와 필요한 정보를 기록하고 있는지 확인하는 방법에 대한 더 많은 정보를 보려면

[TensorBoard: 시각화 학습](#)을 참고하세요.

## 이름 범주화(Name scoping)와 노드

일반적으로 TensorFlow 그래프는 수천 개에 이르는 많은 노드를 가질 수 있습니다. 한 눈에 쉽게 보거나 보통의 그래프 도구를 이용해 그리기에는 너무 많죠. 그래서 변수의 이름을 그룹으로 묶어서(name scoping) 계층화하는 방법을 통해 간단하게 표현합니다. 처음에는 계층의 최상단에 있는 이름들만 보여지는 거죠. 여기 `tf.name_scope` 를 사용해 `hidden` name scope 아래에 세 가지 기능을 정의한 예가 있습니다:

```
import tensorflow as tf

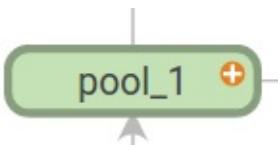
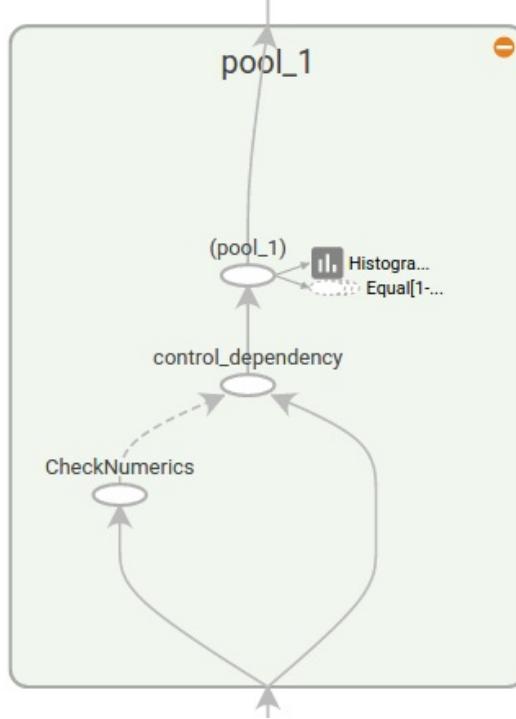
with tf.name_scope('hidden') as scope:
 a = tf.constant(5, name='alpha')
 W = tf.Variable(tf.random_uniform([1, 2], -1.0, 1.0), name='weights')
 b = tf.Variable(tf.zeros([1]), name='biases')
```

위의 예시 코드의 결과로 아래와 같은 세 가지 연산(op)의 이름이 나옵니다:

- 'hidden/alpha'
- 'hidden/weights'
- 'hidden/biases'

시각화를 거치면 이 세 가지 연산자(op)들은 `hidden` 라벨이 붙은 노드가 됩니다. 세부 내용은 그대로죠. 노드를 펼치려면 오른쪽 상단에 있는 주황색 + 표시를 클릭하거나 노드를 더블 클릭해 보세요. `alpha`, `weights`, `biases` 3개의 서브노드를 볼 수 있습니다.

이제 좀 더 복잡한 노드가 초기 상태와 펼쳐진 상태로 있는 현실적 예제를 살펴봅시다.

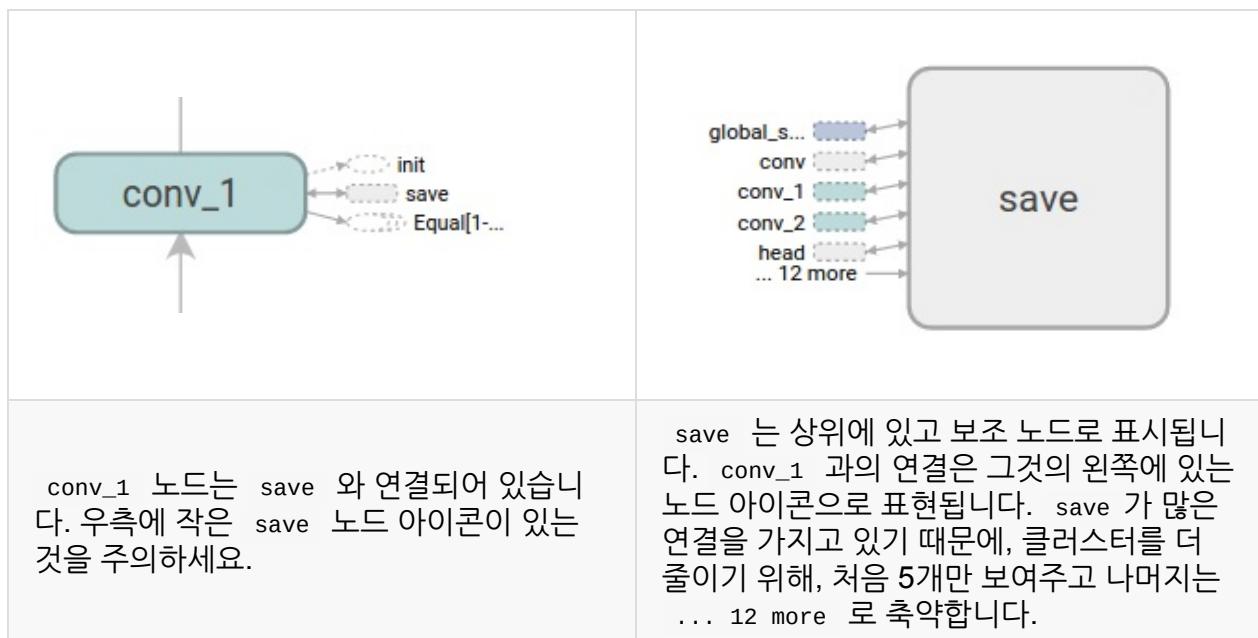
|                                                                                                       |                                                                                                                |
|-------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|
|                    |                             |
| <p>최상단 name scope <code>pool_1</code>의 초기 화면.<br/>우측 상단의 주황색 + 버튼을 클릭하거나<br/>노드를 더블 클릭하면 펼칠 수 있다.</p> | <p><code>pool_1</code> name scope가 펼쳐진 모습. 우측<br/>상단의 주황색 - 버튼을 클릭하거나 노드를<br/>더블 클릭하면 name scope를 접을 수 있다.</p> |

노드를 이름 범주(name scope)로 묶는 것은 읽기 쉬운 그래프를 만들 때 중요합니다. 모델을 만들 때 이름 범주화를 이용하면 시각화된 결과를 제어하기 좋습니다. 이름 범주(name scope)를 잘 쓰면 시각화가 잘 됩니다.

위 그림은 시각화의 두 번째 면을 보여줍니다. TensorFlow 그래프의 연결은 데이터 종속(data dependency)과 컨트롤 종속(control dependency) 두 가지 종류가 있습니다. 데이터 종속은 두 연산 사이 tensor의 흐름을 보여주는데 실선 화살표로 나타납니다. 반면 컨트롤 종속은 점선으로 나타납니다. (위 그림 오른쪽 부분) 펼쳐진 모습에서 `CheckNumerics` 와 `control_dependency` 를 연결하는 점선을 제외하고 모든 연결은 데이터 종속입니다.

레이아웃을 간단하게 하는 두 번째 트릭이 있습니다. 대부분의 TensorFlow 그래프는 다른 노드와 많이 연결된 몇 개의 노드로 이루어져 있습니다. 예를 들어, 많은 노드들이 초기화 단계에 컨트롤 종속을 가지고 있을 수 있습니다. 그래서 `init` 노드와 그 노드에 종속된 것들 사이의 모든 엣지(edge)를 그려보면 많은 실선들로 뭉쳐진 모습이 나타날 것입니다.

뭉쳐져 보이는 것을 줄이기 위해 시각화 프로그램은 모든 상위 노드를 우측에 있는 *auxiliary*(보조) 공간에 분리해 두고 엣지를 나타내는 선을 그리지 않습니다. 선 대신에 연결을 나타내기 위해 작은 노드 아이콘을 그립니다. 보조 노드를 떼어놓더라도 중요한 정보가 사라지는 일은 잘 없습니다. 왜냐하면 이 노드들은 보통 부기 기능(bookkeeping function)과 관련있기 때문입니다. 메인 그래프과 보조 영역 사이에서 노드를 어떻게 움직이는지 보려면 [Interaction](#)를 보세요.



마지막 구조 단순화는 *series collapsing* 입니다. 이름의 마지막 숫자만 다르고 같은 구조를 가진 노드인 순차적 모티프(sequential motif)는 아래에 나와 있는 것처럼 하나의 노드 스택으로 접을 수 있습니다. 긴 배열을 가진 네트워크의 경우, 이를 통해 모양이 굉장히 단순하게 됩니다. 노드의 계층을 나타낼 때와 마찬가지로, 더블 클릭해서 이 *series*를 펼칠 수 있습니다. 어떻게 특정 노드 셋이 접한 것을 비활성화/활성화 하는지는 [Interaction](#)를 보세요.



마지막으로, 가독성을 높이기 위해, 시각화는 상수와 요약 노드를 위해 특별한 아이콘을 사용합니다. 간단히, 노드 기호 표가 있습니다:

| 기호 | 의미                                                                       |
|----|--------------------------------------------------------------------------|
|    | <i>High-level</i> 노드는 name scope를 나타냅니다. high-level 노드를 펼치기 위해 더블 클릭하세요. |
|    | 서로 연결되지 않은 숫자가 매겨진 노드의 시퀀스.                                              |
|    | 서로 연결된 숫자가 매겨진 노드의 시퀀스.                                                  |
|    | 각각의 연산 노드.                                                               |
|    | 상수.                                                                      |
|    | 요약 노드.                                                                   |
|    | 간선은 연산 사이의 데이터 흐름을 보여줍니다.                                                |
|    | 간선은 연산 사이의 컨트롤 종속을 보여줍니다.                                                |
|    | 레퍼런스 간선은 나가는 연산 노드가 들어오는 tensor를 변형할 수 있다는 것을 보여줍니다.                     |

## Interaction

줌과 이동 기능을 이용해서 그래프를 탐색할 수 있습니다. 화면을 클릭하고 드래그하고 줌하기 위해 스크롤을 사용하세요. 연산 그룹을 보여주는 이름 범주(name scope)를 펼치기 위해서는 노드를 더블 클릭하거나 노드의 버튼을 누르세요. 줌과 이동 기능을 사용하더라도 지금의 위치를 손쉽게 파악할 수 있도록 우측 하단 모서리에 미니맵이 있습니다.

열린 노드를 닫기 위해 다시 한번 더블 클릭하거나 노드의 버튼을 누르세요. 노드를 선택하기 위해 한 번 누르셔도 됩니다. 노드는 어두운 색으로 변하고, 노드에 대한 세부 사항과 연결된 노드가 시각화의 우측 상단 모서리 정보 카드에 표시됩니다.

|                                                                                                                                                                                                                                                               |                                                                                                                                                                                                                                                       |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p><b>conv2</b><br/>Subgraph: 22 nodes</p> <p><b>Attributes (0)</b></p> <p><b>Inputs (1)</b><br/>norm1</p> <p><b>Outputs (6)</b><br/>norm2<br/>gradients<br/>GradientDescent<br/>ExponentialMovingAverage<br/>save</p> <p>^ Control dependencies<br/>init</p> | <p><b>DecodeRaw</b><br/>Operation: DecodeRaw</p> <p><b>Attributes (2)</b><br/>little_endian {"b":true}<br/>out_type {"type":"DT_UINT8"}<br/>Device /gpu:0</p> <p><b>Inputs (1)</b><br/>ReaderRead</p> <p><b>Outputs (2)</b><br/>Slice<br/>Slice_1</p> |
| <p>정보 카드가 conv2 name scope의 세부 정보를 보여줍니다. 입력과 출력은 name scope 내의 연산 노드의 입력과 출력에서 결합됩니다. name scope에 대한 속성은 보여지지 않습니다.</p>                                                                                                                                      | <p>정보 카드가 DecodeRaw 연산 노드의 세부 정보를 보여줍니다. 입력과 출력에 더해서, 카드는 현재 연산에 관련된 디바이스와 속성들을 보여줍니다.</p>                                                                                                                                                            |

TensorBoard는 그래프의 레이아웃을 바꿀 수 있는 몇 가지 방법을 제공합니다. 이것이 그래프의 연산 의미를 바꾸지는 않지만 네트워크 구조를 좀 더 명확하게 합니다. 노드를 우클릭하거나 정보 카드 하단에 있는 버튼을 눌러서 레이아웃에 아래와 같은 변화를 줄 수 있습니다:

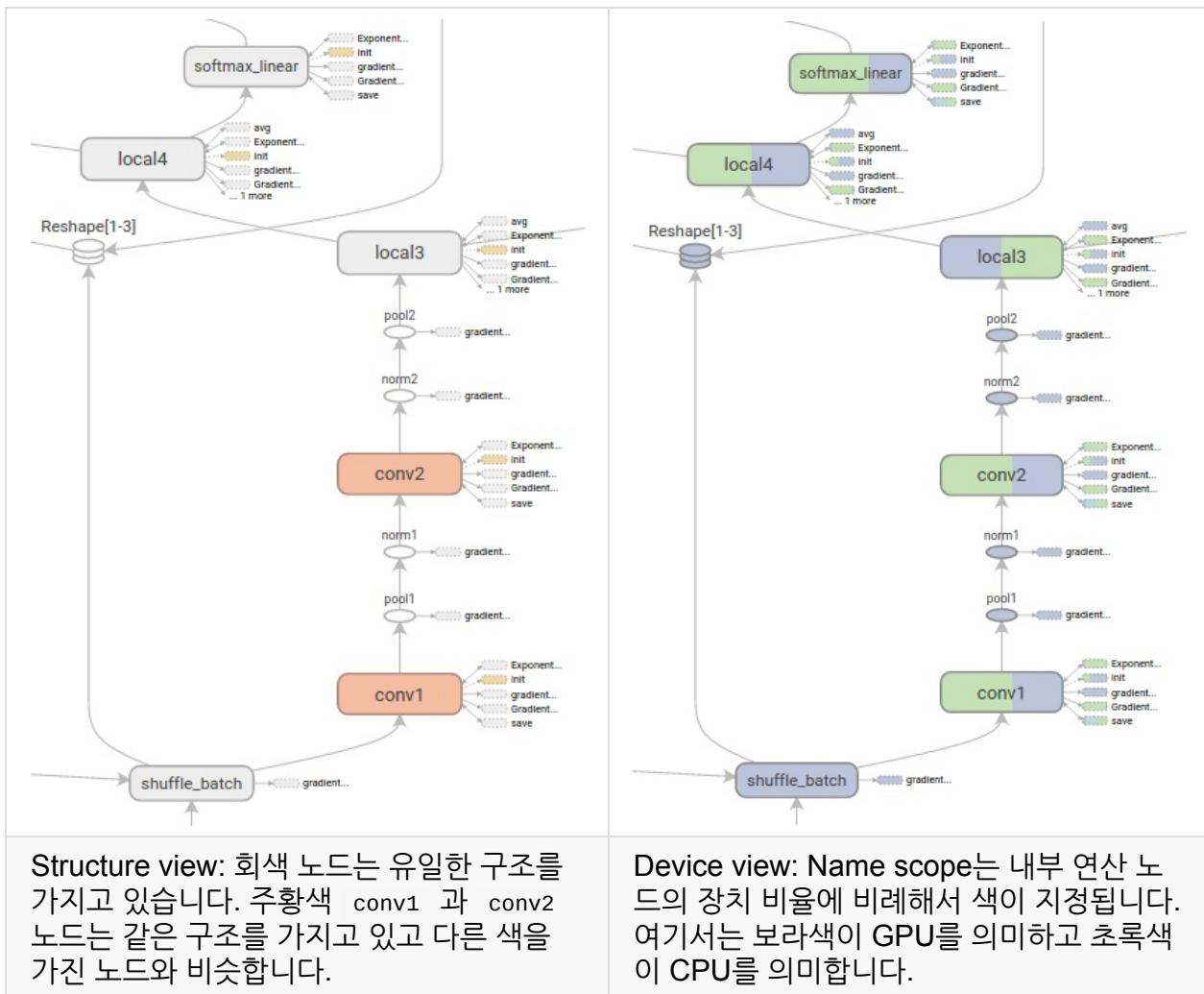
- 노드는 메인 그래프와 보조 영역 사이를 이동할 수 있습니다.
- 연속된(series) 노드를 그룹 해제해서 이 노드들이 그룹으로 함께 보여지지 않게 합니다. 그룹 해제된 series는 같은 방법으로 재그룹화 할 수 있습니다.

선택 기능은 상위 노드를 이해하는 것에도 도움이 됩니다. 어떤 상위 노드를 선택하면 다른 연결을 위해 대응하는 노드 아이콘도 또한 선택됩니다. 예를 들어, 이것은 어떤 노드가 저장되고 어떤 것이 안됐는지를 보기 쉽게 만들어 줍니다.

정보 카드에 있는 노드 이름을 클릭하면 해당 노드를 선택할 수 있습니다. 필요하다면, 노드를 볼 수 있도록 화면이 자동으로 이동합니다.

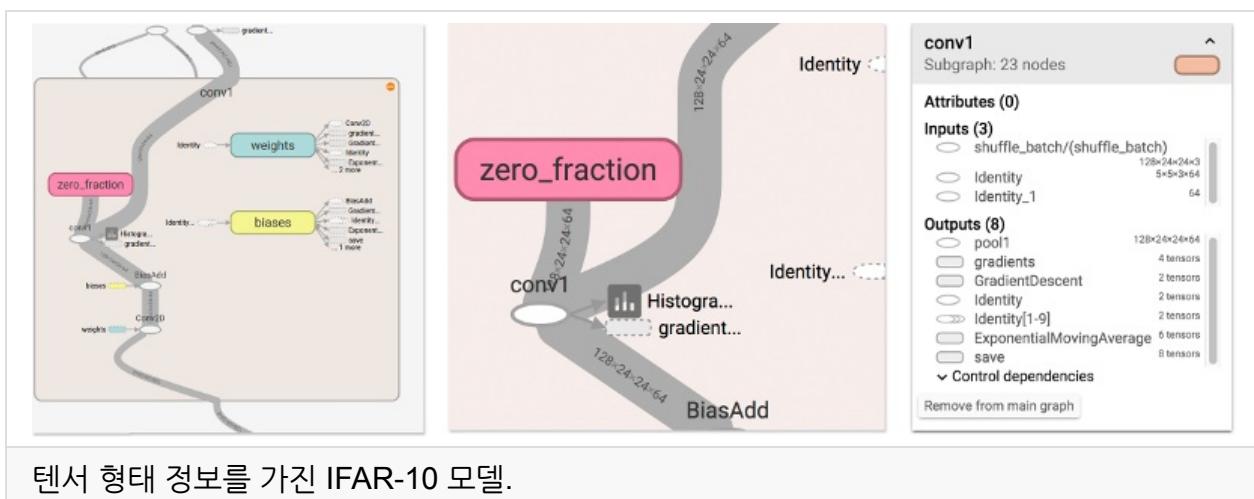
마지막으로, 상단 표에 있는 컬러 메뉴를 이용해 그래프에 두 가지 색을 선택할 수 있습니다. 기본 *Structure View*는 구조를 보여줍니다: 두 상위 레벨의 노드가 같은 구조이면 같은 색으로 표현됩니다. 유일한 구조를 가진 노드는 회색입니다. 어떤 디바이스가 다른 연산을 실행하고 있는지 보여주는 두 번째 화면도 있습니다. 이름 범주(name scope)는 내부 연산에 대한 장치의 비율에 비례해서 색이 지정됩니다.

아래의 이미지는 실제 그래프 중 일부분을 보여줍니다.



## 텐서의 형태 정보(Tensor shape information)

텐서의 형태가 연속된 GraphDef에 포함되면 그래프 시각화 프로그램이 텐서의 차원을 엣지에 표시하고 텐서의 크기는 엣지의 두께로 나타냅니다. GraphDef에 텐서 형태를 포함하기 위해서는 그 래프를 연속화할 때 실제 그래프 객체( sess.graph 와 같이)를 SummaryWriter에 전달합니다. 아래의 이미지는 텐서의 형태 정보를 가진 CIFAR-10 모델을 보여줍니다:



# Runtime statistics

실행할 때 총 메모리 사용량, 총 계산 시간, 노드의 tensor 형태와 같은 런타임 메타데이터를 수집하면 보통 도움이 됩니다. 아래의 코드 예제는 [simple MNIST tutorial](#)의 수정본 중 훈련과 테스트 부분에서 발췌한 내용으로 요약과 런타임 통계를 기록하는 부분입니다. 요약을 어떻게 기록하는지는 [Summaries Tutorial](#)을 보세요. 전체 소스는 [여기에 있습니다](#).

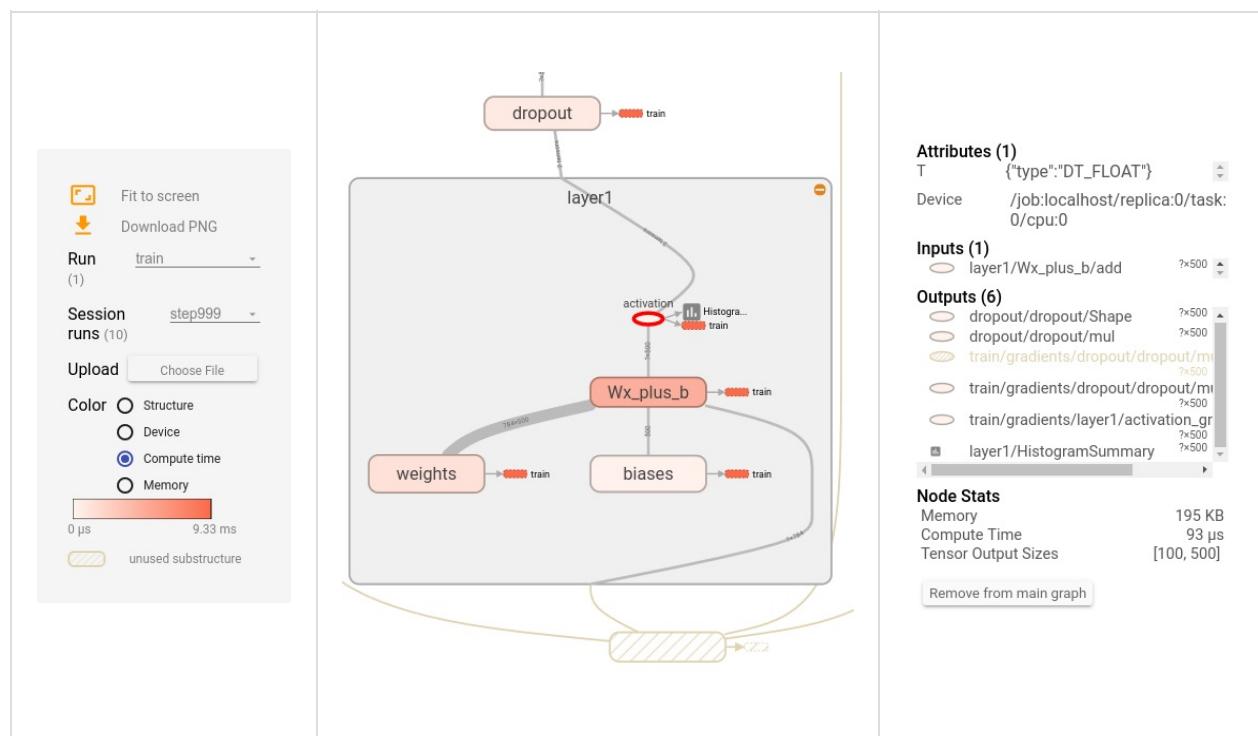
```
모델을 트레이닝하고 또한 요약을 작성합니다.
매 10번째 순서마다, 테스트 셋의 정확도를 측정하고 테스트 요약을 작성합니다.
다른 모든 순서에서 트레이닝 데이터로 트레이닝 단계를 실행하고 트레이닝 요약을 작성합니다.

def feed_dict(train):
 """Make a TensorFlow feed_dict that maps data onto Tensor placeholders."""
 if train or FLAGS.fake_data:
 xs, ys = mnist.train.next_batch(100, fake_data=FLAGS.fake_data)
 k = FLAGS.dropout
 else:
 xs, ys = mnist.test.images, mnist.test.labels
 k = 1.0
 return {x: xs, y_: ys, keep_prob: k}

for i in range(FLAGS.max_steps):
 if i % 10 == 0: # 요약과 테스트 셋 정확도를 기록한다
 summary, acc = sess.run([merged, accuracy], feed_dict=feed_dict(False))
 test_writer.add_summary(summary, i)
 print('Accuracy at step %s: %s' % (i, acc))
 else: # 트레이닝 셋 요약을 기록하고 트레이닝한다
 if i % 100 == 99: # 실행 통계를 기록한다
 run_options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
 run_metadata = tf.RunMetadata()
 summary, _ = sess.run([merged, train_step],
 feed_dict=feed_dict(True),
 options=run_options,
 run_metadata=run_metadata)
 train_writer.add_run_metadata(run_metadata, 'step%d' % i)
 train_writer.add_summary(summary, i)
 print('Adding run metadata for', i)
 else: # 요약을 기록한다
 summary, _ = sess.run([merged, train_step], feed_dict=feed_dict(True))
 train_writer.add_summary(summary, i)
```

이 코드는 99 번째부터 매 100 번째마다 런타임 통계를 내보냅니다.

tensorboard를 시작하고 그래프 탭으로 가면, 실행 메타데이터가 추가된 단계에 대응하는 "Sessopm runs" 아래 옵션들을 볼 수 있습니다. 이 중 하나를 선택하면 사용되지 않는 노드들은 가려줘서 해당 단계에서 네트워크의 상태를 볼 수 있습니다. 왼쪽의 컨트롤에서 총 메모리 또는 총 계산 시간으로 노드의 색을 지정할 수 있습니다. 추가적으로, 노드를 클릭하면 정확한 총 메모리, 계산 시간, 텐서 출력 크기를 보여줍니다.



# Reading data

(v1.0)

There are three main methods of getting data into a TensorFlow program:

- Feeding: Python code provides the data when running each step.
- Reading from files: an input pipeline reads the data from files at the beginning of a TensorFlow graph.
- Preloaded data: a constant or variable in the TensorFlow graph holds all the data (for small data sets).

[TOC]

## Feeding

TensorFlow's feed mechanism lets you inject data into any Tensor in a computation graph. A python computation can thus feed data directly into the graph.

Supply feed data through the `feed_dict` argument to a `run()` or `eval()` call that initiates computation.

```
with tf.Session():
 input = tf.placeholder(tf.float32)
 classifier = ...
 print(classifier.eval(feed_dict={input: my_python_preprocessing_fn()}))
```

While you can replace any Tensor with feed data, including variables and constants, the best practice is to use a `placeholder` op node. A `placeholder` exists solely to serve as the target of feeds. It is not initialized and contains no data. A placeholder generates an error if it is executed without a feed, so you won't forget to feed it.

An example using `placeholder` and feeding to train on MNIST data can be found in [tensorflow/examples/tutorials/mnist/fully\\_connected\\_feed.py](#), and is described in the [MNIST tutorial](#).

## Reading from files

A typical pipeline for reading records from files has the following stages:

1. The list of filenames
2. *Optional* filename shuffling
3. *Optional* epoch limit
4. Filename queue
5. A Reader for the file format
6. A decoder for a record read by the reader
7. *Optional* preprocessing
8. Example queue

## Filenames, shuffling, and epoch limits

For the list of filenames, use either a constant string Tensor (like `["file0", "file1"]` or `[("file%d" % i) for i in range(2)]`) or the `tf.train.match_filenames_once` function.

Pass the list of filenames to the `tf.train.string_input_producer` function.

`string_input_producer` creates a FIFO queue for holding the filenames until the reader needs them.

`string_input_producer` has options for shuffling and setting a maximum number of epochs. A queue runner adds the whole list of filenames to the queue once for each epoch, shuffling the filenames within an epoch if `shuffle=True`. This procedure provides a uniform sampling of files, so that examples are not under- or over-sampled relative to each other.

The queue runner works in a thread separate from the reader that pulls filenames from the queue, so the shuffling and enqueueing process does not block the reader.

## File formats

Select the reader that matches your input file format and pass the filename queue to the reader's read method. The read method outputs a key identifying the file and record (useful for debugging if you have some weird records), and a scalar string value. Use one (or more) of the decoder and conversion ops to decode this string into the tensors that make up an example.

## CSV files

To read text files in comma-separated value (CSV) format, use a `TextLineReader` with the `decode_csv` operation. For example:

```

filename_queue = tf.train.string_input_producer(["file0.csv", "file1.csv"])

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

Default values, in case of empty columns. Also specifies the type of the
decoded result.
record_defaults = [[1], [1], [1], [1], [1]]
col1, col2, col3, col4, col5 = tf.decode_csv(
 value, record_defaults=record_defaults)
features = tf.stack([col1, col2, col3, col4])

with tf.Session() as sess:
 # Start populating the filename queue.
 coord = tf.train.Coordinator()
 threads = tf.train.start_queue_runners(coord=coord)

 for i in range(1200):
 # Retrieve a single instance:
 example, label = sess.run([features, col5])

 coord.request_stop()
 coord.join(threads)

```

Each execution of `read` reads a single line from the file. The `decode_csv` op then parses the result into a list of tensors. The `record_defaults` argument determines the type of the resulting tensors and sets the default value to use if a value is missing in the input string.

You must call `tf.train.start_queue_runners` to populate the queue before you call `run` or `eval` to execute the `read`. Otherwise `read` will block while it waits for filenames from the queue.

## Fixed length records

To read binary files in which each record is a fixed number of bytes, use

`tf.FixedLengthRecordReader` with the `tf.decode_raw` operation. The `decode_raw` op converts from a string to a `uint8` tensor.

For example, [the CIFAR-10 dataset](#) uses a file format where each record is represented using a fixed number of bytes: 1 byte for the label followed by 3072 bytes of image data. Once you have a `uint8` tensor, standard operations can slice out each piece and reformat as needed. For CIFAR-10, you can see how to do the reading and decoding in [tensorflow\\_models/tutorials/image/cifar10/cifar10\\_input.py](#) and described in [this tutorial](#).

## Standard TensorFlow format

Another approach is to convert whatever data you have into a supported format. This approach makes it easier to mix and match data sets and network architectures. The recommended format for TensorFlow is a `TFRecords` file containing `tf.train.Example` `protocol buffers` (which contain `Features` as a field). You write a little program that gets your data, stuffs it in an `Example` protocol buffer, serializes the protocol buffer to a string, and then writes the string to a TFRecords file using the `tf.python_io.TFRecordWriter` class. For example, `tensorflow/examples/how_tos/reading_data/convert_to_records.py` converts MNIST data to this format.

To read a file of TFRecords, use `tf.TFRecordReader` with the `tf.parse_single_example` decoder. The `parse_single_example` op decodes the example protocol buffers into tensors. An MNIST example using the data produced by `convert_to_records` can be found in `tensorflow/examples/how_tos/reading_data/fully_connected_reader.py`, which you can compare with the `fully_connected_feed` version.

## Preprocessing

You can then do any preprocessing of these examples you want. This would be any processing that doesn't depend on trainable parameters. Examples include normalization of your data, picking a random slice, adding noise or distortions, etc. See `tensorflow_models/tutorials/image/cifar10/cifar10.py` for an example.

## Batching

At the end of the pipeline we use another queue to batch together examples for training, evaluation, or inference. For this we use a queue that randomizes the order of examples, using the `tf.train.shuffle_batch` function.

Example:

```

def read_my_file_format(filename_queue):
 reader = tf.SomeReader()
 key, record_string = reader.read(filename_queue)
 example, label = tf.some_decoder(record_string)
 processed_example = some_processing(example)
 return processed_example, label

def input_pipeline(filenames, batch_size, num_epochs=None):
 filename_queue = tf.train.string_input_producer(
 filenames, num_epochs=num_epochs, shuffle=True)
 example, label = read_my_file_format(filename_queue)
 # min_after_dequeue defines how big a buffer we will randomly sample
 # from -- bigger means better shuffling but slower start up and more
 # memory used.
 # capacity must be larger than min_after_dequeue and the amount larger
 # determines the maximum we will prefetch. Recommendation:
 # min_after_dequeue + (num_threads + a small safety margin) * batch_size
 min_after_dequeue = 10000
 capacity = min_after_dequeue + 3 * batch_size
 example_batch, label_batch = tf.train.shuffle_batch(
 [example, label], batch_size=batch_size, capacity=capacity,
 min_after_dequeue=min_after_dequeue)
 return example_batch, label_batch

```

If you need more parallelism or shuffling of examples between files, use multiple reader instances using the [tf.train.shuffle\\_batch\\_join function](#). For example:

```

def read_my_file_format(filename_queue):
 # Same as above

def input_pipeline(filenames, batch_size, read_threads, num_epochs=None):
 filename_queue = tf.train.string_input_producer(
 filenames, num_epochs=num_epochs, shuffle=True)
 example_list = [read_my_file_format(filename_queue)
 for _ in range(read_threads)]
 min_after_dequeue = 10000
 capacity = min_after_dequeue + 3 * batch_size
 example_batch, label_batch = tf.train.shuffle_batch_join(
 example_list, batch_size=batch_size, capacity=capacity,
 min_after_dequeue=min_after_dequeue)
 return example_batch, label_batch

```

You still only use a single filename queue that is shared by all the readers. That way we ensure that the different readers use different files from the same epoch until all the files from the epoch have been started. (It is also usually sufficient to have a single thread filling the filename queue.)

An alternative is to use a single reader via the `tf.train.shuffle_batch` function with `num_threads` bigger than 1. This will make it read from a single file at the same time (but faster than with 1 thread), instead of N files at once. This can be important:

- If you have more reading threads than input files, to avoid the risk that you will have two threads reading the same example from the same file near each other.
- Or if reading N files in parallel causes too many disk seeks.

How many threads do you need? the `tf.train.shuffle_batch*` functions add a summary to the graph that indicates how full the example queue is. If you have enough reading threads, that summary will stay above zero. You can [view your summaries as training progresses using TensorBoard](#).

## Creating threads to prefetch using `QueueRunner` objects

The short version: many of the `tf.train` functions listed above add `QueueRunner` objects to your graph. These require that you call `tf.train.start_queue_runners` before running any training or inference steps, or it will hang forever. This will start threads that run the input pipeline, filling the example queue so that the dequeue to get the examples will succeed. This is best combined with a `tf.train.Coordinator` to cleanly shut down these threads when there are errors. If you set a limit on the number of epochs, that will use an epoch counter that will need to be initialized. The recommended code pattern combining these is:

```

Create the graph, etc.
init_op = tf.initialize_all_variables()

Create a session for running operations in the Graph.
sess = tf.Session()

Initialize the variables (like the epoch counter).
sess.run(init_op)

Start input enqueue threads.
coord = tf.train.Coordinator()
threads = tf.train.start_queue_runners(sess=sess, coord=coord)

try:
 while not coord.should_stop():
 # Run training steps or whatever
 sess.run(train_op)

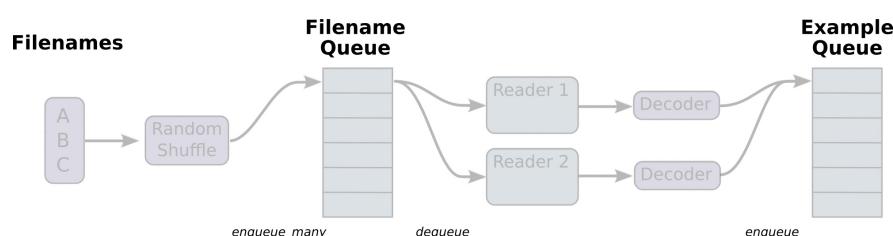
except tf.errors.OutOfRangeError:
 print('Done training -- epoch limit reached')
finally:
 # When done, ask the threads to stop.
 coord.request_stop()

Wait for threads to finish.
coord.join(threads)
sess.close()

```

## Aside: What is happening here?

First we create the graph. It will have a few pipeline stages that are connected by queues. The first stage will generate filenames to read and enqueue them in the filename queue. The second stage consumes filenames (using a `Reader`), produces examples, and enqueues them in an example queue. Depending on how you have set things up, you may actually have a few independent copies of the second stage, so that you can read from multiple files in parallel. At the end of these stages is an enqueue operation, which enqueues into a queue that the next stage dequeues from. We want to start threads running these enqueueing operations, so that our training loop can dequeue examples from the example queue.



The helpers in `tf.train` that create these queues and enqueueing operations add a `tf.train.QueueRunner` to the graph using the `tf.train.add_queue_runner` function. Each `QueueRunner` is responsible for one stage, and holds the list of enqueue operations that need to be run in threads. Once the graph is constructed, the `tf.train.start_queue_runners` function asks each `QueueRunner` in the graph to start its threads running the enqueueing operations.

If all goes well, you can now run your training steps and the queues will be filled by the background threads. If you have set an epoch limit, at some point an attempt to dequeue examples will get an `tf.errors.OutOfRangeError`. This is the TensorFlow equivalent of "end of file" (EOF) -- this means the epoch limit has been reached and no more examples are available.

The last ingredient is the `Coordinator`. This is responsible for letting all the threads know if anything has signalled a shut down. Most commonly this would be because an exception was raised, for example one of the threads got an error when running some operation (or an ordinary Python exception).

For more about threading, queues, QueueRunners, and Coordinators [see here](#).

## Aside: How clean shut-down when limiting epochs works

Imagine you have a model that has set a limit on the number of epochs to train on. That means that the thread generating filenames will only run that many times before generating an `OutOfRange` error. The `QueueRunner` will catch that error, close the filename queue, and exit the thread. Closing the queue does two things:

- Any future attempt to enqueue in the filename queue will generate an error. At this point there shouldn't be any threads trying to do that, but this is helpful when queues are closed due to other errors.
- Any current or future dequeue will either succeed (if there are enough elements left) or fail (with an `OutOfRange` error) immediately. They won't block waiting for more elements to be enqueued, since by the previous point that can't happen.

The point is that when the filename queue is closed, there will likely still be many filenames in that queue, so the next stage of the pipeline (with the reader and other preprocessing) may continue running for some time. Once the filename queue is exhausted, though, the next attempt to dequeue a filename (e.g. from a reader that has finished with the file it was working on) will trigger an `OutOfRange` error. In this case, though, you might have multiple threads associated with a single `QueueRunner`. If this isn't the last thread in the `QueueRunner`, the `OutOfRange` error just causes the one thread to exit. This allows the other threads, which are still finishing up their last file, to proceed until they finish as well.

(Assuming you are using a `tf.train.Coordinator`, other types of errors will cause all the threads to stop.) Once all the reader threads hit the `OutOfRange` error, only then does the next queue, the example queue, gets closed.

Again, the example queue will have some elements queued, so training will continue until those are exhausted. If the example queue is a `RandomShuffleQueue`, say because you are using `shuffle_batch` or `shuffle_batch_join`, it normally will avoid ever going having fewer than its `min_after_dequeue` attr elements buffered. However, once the queue is closed that restriction will be lifted and the queue will eventually empty. At that point the actual training threads, when they try and dequeue from example queue, will start getting `OutOfRange` errors and exiting. Once all the training threads are done, `tf.train.Coordinator.join` will return and you can exit cleanly.

## Filtering records or producing multiple examples per record

Instead of examples with shapes `[x, y, z]`, you will produce a batch of examples with shape `[batch, x, y, z]`. The batch size can be 0 if you want to filter this record out (maybe it is in a hold-out set?), or bigger than 1 if you are producing multiple examples per record. Then simply set `enqueue_many=True` when calling one of the batching functions (such as `shuffle_batch` or `shuffle_batch_join`).

## Sparse input data

SparseTensors don't play well with queues. If you use SparseTensors you have to decode the string records using `tf.parse_example` **after** batching (instead of using `tf.parse_single_example` **before** batching).

## Preloaded data

This is only used for small data sets that can be loaded entirely in memory. There are two approaches:

- Store the data in a constant.
- Store the data in a variable, that you initialize and then never change.

Using a constant is a bit simpler, but uses more memory (since the constant is stored inline in the graph data structure, which may be duplicated a few times).

```

training_data = ...
training_labels = ...
with tf.Session():
 input_data = tf.constant(training_data)
 input_labels = tf.constant(training_labels)
 ...

```

To instead use a variable, you need to also initialize it after the graph has been built.

```

training_data = ...
training_labels = ...
with tf.Session() as sess:
 data_initializer = tf.placeholder(dtype=training_data.dtype,
 shape=training_data.shape)
 label_initializer = tf.placeholder(dtype=training_labels.dtype,
 shape=training_labels.shape)
 input_data = tf.Variable(data_initializer, trainable=False, collections[])
 input_labels = tf.Variable(label_initializer, trainable=False, collections[])
 ...
 sess.run(input_data.initializer,
 feed_dict={data_initializer: training_data})
 sess.run(input_labels.initializer,
 feed_dict={label_initializer: training_labels})

```

Setting `trainable=False` keeps the variable out of the `GraphKeys.TRAINABLE_VARIABLES` collection in the graph, so we won't try and update it when training. Setting `collections=[]` keeps the variable out of the `GraphKeys.VARIABLES` collection used for saving and restoring checkpoints.

Either way, `tf.train.slice_input_producer` function can be used to produce a slice at a time. This shuffles the examples across an entire epoch, so further shuffling when batching is undesirable. So instead of using the `shuffle_batch` functions, we use the plain `tf.train.batch` function. To use multiple preprocessing threads, set the `num_threads` parameter to a number bigger than 1.

An MNIST example that preloads the data using constants can be found in

[tensorflow/examples/how\\_tos/reading\\_data/fully\\_connected\\_preloaded.py](#), and one that preloads the data using variables can be found in

[tensorflow/examples/how\\_tos/reading\\_data/fully\\_connected\\_preloaded\\_var.py](#), You can compare these with the `fully_connected_feed` and `fully_connected_reader` versions above.

## Multiple input pipelines

Commonly you will want to train on one dataset and evaluate (or "eval") on another. One way to do this is to actually have two separate processes:

- The training process reads training input data and periodically writes checkpoint files with all the trained variables.
- The evaluation process restores the checkpoint files into an inference model that reads validation input data.

This is what is done in [the example CIFAR-10 model](#). This has a couple of benefits:

- The eval is performed on a single snapshot of the trained variables.
- You can perform the eval even after training has completed and exited.

You can have the train and eval in the same graph in the same process, and share their trained variables. See [the shared variables tutorial](#).

# 쓰레딩(Threading)과 큐(Queues)

큐는 TensorFlow 를 사용하는 비동기 계산에 대해 강력한 메커니즘이다.

TensorFlow 의 다른 모든 것들처럼, 하나의 큐는 TensorFlow 그래프에서 하나의 노드다. 이는 변수(variable)와 비슷한, 상태저장 노드다: 다른 노드들은 그 저장물(콘텐츠)의 수정이 가능하다. 특히, 노드들은 큐에 새로운 아이템들을 추가할 수 있거나 큐에 존재하는 아이템들을 해제할 수 있다.

큐에 대한 감을 잡기 위해, 간단한 예제를 생각해보자. 우리는 "first in, first out" 큐( FIFOQueue ) 를 만들어볼 것이고, 이를 0 으로 채울 것이다. 다음으로 큐에서 아이템을 제거, 아이템을 추가, 그리고 큐 끝에 이를 다시 넣는 그래프를 만들것이다. 큐의 숫자들은 천천히 증가한다.

## Client

```

q = tf.FIFOQueue(3, "float")
init = q.enqueue_many(([0., 0., 0.],))
x = q.dequeue()
y = x+1
q_inc = q.enqueue([y])

init.run()
q_inc.run()
q_inc.run()
q_inc.run()
q_inc.run()

```

`Enqueue` , `EnqueueMany` , 그리고 `Dequeue` 는 특별 노드들이다. 이들은 평범한 값 대신 큐에 대한 포인터를 가지며, 이들은 이 포인터를 변경할 수 있다. 우리는 큐의 매쏘드(method)들과 같은 것에 대해 생각해보길 권한다. 사실, Python API 에서, 이들은 큐 객체에 대한 매쏘드(method)들이다 (`q.enqueue(...)` ).

주의 큐 매쏘드들(method)( `q.enqueue(...)` 와 같은)은, 큐 처럼, 반드시 같은 장치에서 실행해야 한다. 이들 연산이 생성될 때 호환성이 없는 장치 배치 지시들(Incompatible device placement directives) 은 무시될 것이다.

이제 큐에 대한 감을 조금 가졌을 것이고, 자세한 부분으로 들어가보자...

## 큐 사용 개요

큐들은, `FIFOQueue` 와 `RandomShuffleQueue` 와 같은, 그래프에서 비동기로 `tensor` 들을 계산하기 위한 중요한 TensorFlow 객체들이다.

예를 들어, 대표적인 입력 아키텍쳐는 모델 학습을 위한 입력들을 준비하기 위해 `RandomShuffleQueue` 를 사용해야 한다:

- 다수의 쓰레드들은 학습 예제들을 대비하고 이들을 큐에 넣는다.

- 학습하는 쓰레드는 큐에서 mini-batches 를 빼내는 학습 연산을 실행한다.

이 아키텍처는, 입력 파이프라인들의 구성을 간략화하는 함수들에 대한 개요를 알려주는 [Reading data how to](#) 에서 강조되었던 것처럼, 많은 이점들을 가진다.

TensorFlow `Session` 객체는 멀티 쓰레드화 되어있다. 그리고 다수 쓰레드들은 같은 세션 사용을 쉽게할 수 있고 병렬로 연산들을 실행할 수 있다. 그러나, 위에서 묘사된 것처럼 쓰레드들을 다루는 Python 프로그램을 구현하는 것이 항상 쉽지만은 않다. 모든 쓰레드들은 함께 멈춰질 수 있어야 하며, 예외처리들은 처리되어야 하고 알려져야 한다. 그리고 큐는 멈췄을 때 적절하게 종료되어야 한다.

TensorFlow 는 도움을 주는 두 클래스들을 제공한다: `tf.Coordinator` 와 `tf.QueueRunner`. 이들 두 클래스들은 함께 사용되기 위해 디자인되었다. `Coordinator` 클래스는 멀티 쓰레드들이 함께 정지되도록 돋고 예외처리들을 그들이 정지되기 위해 대기하는 프로그램에 알린다. `QueueRunner` 클래스는 같은 큐 안의 `tensors` 를 추가하기 위해 협력하는 많은 쓰레드들을 생성한다.

## 조정자(Coordinator)

조정자 클래스는 멀티 쓰레드들이 같이 정지되도록 한다.

이것의 핵심 매쏘드들은 아래와 같다:

- `should_stop()` : 쓰레드들이 정지되어야 한다면 `True` 값을 반환한다.
- `request_stop(<exception>)` : 쓰레드들이 정지되어야 함을 요청한다.
- `'join()'`: 특정 쓰레드들이 멈출 때 까지 대기한다.

당신은 우선 `Coordinator` 객체를 생성하고, 다음으로 `coordinator` 를 사용하는 쓰레드들을 생성한다. 일반적으로 쓰레드들은 `should_stop()` 이 `True` 를 반환할 때 멈추는 루프를 실행한다.

어떤 쓰레드는 멈춰야 하는 계산을 결정할 수 있다. 이것은 `request_stop()` 함수를 부르는 것이고 다른 쓰레드들은 `should_stop()` 함수가 `True` 값을 반환한 다음 정지된다.

```

쓰레드 body: coordinator 가 정지가 요청됨을 알릴 때까지 반복
어떤 조건이 true 가 이면, coordinator 가 멈출 것을 요청
def MyLoop(coord):
 while not coord.should_stop():
 ...do something...
 if ...some condition...:
 coord.request_stop()

Main code: coordinator 생성
coord = Coordinator()

'MyLoop()' 를 실행하는 10개의 쓰레드를 생성
threads = [threading.Thread(target=MyLoop, args=(coord,)) for i in xrange(10)]

쓰레드들을 시작하고 그들 모두의 정지를 위한 대기
for t in threads: t.start()
coord.join(threads)

```

분명히, `coordinator` 는 다양한 처리를 하는 쓰레드들을 관리할 수 있다. 이들은 위 예제와 같이 모두 같지는 않다. 또한 `coordinator` 는 예외처리들을 감지하고 알린다. 자세한 것은 [Coordinator class](#) 문서를 살펴보자.

## QueueRunner

`QueueRunner` 클래스는 `enqueue` 연산을 반복적으로 실행하는 쓰레드들을 생성한다. 이들 쓰레드들은 `coordinator` 를 이용해 함께 정지하도록 할 수 있다. 추가로, `queue runner` 는 예외처리가 `coordinator` 에 보고하면 자동적으로 `queue` 를 종료하는 *closer thread* 를 실행한다.

당신은 위에 설명된 아키텍처 구현을 위해 `queue runner` 를 사용할 수 있다.

우선 입력 예제들에 대해 `Queue` 를 사용하는 그래프를 만든다. 예제들을 처리하고 큐에 이들을 추가하는 연산을 추가한다. 큐에서 해제하는 것을 시작하는 학습 연산들을 추가한다.

```

example = ... 예제를 생성하는 연산들...
큐와 큐에 차례로 예제들을 추가하는 연산을 생성한다.
queue = tf.RandomShuffleQueue(...)
enqueue_op = queue.enqueue(example)
예제들을 큐에서 해제하도록 하는 학습 그래프를 생성한다.
inputs = queue.dequeue_many(batch_size)
train_op = ...그래프의 학습 부분을 만들기 위한 'inputs' 을 사용

```

Python 학습 프로그램에서, 예제들을 처리하고 큐에 추가하기 위한 쓰레드들을 실행할 `QueueRunner` 를 생성한다. `coordinator` 를 생성하고 `queue runner` 가 `coordinator` 와 함께 이들의 쓰레드들을 시작하는 것을 요청한다. `coordinator` 를 사용하는 학습 루프를 적어보자.

```
예제들을 큐에 병렬로 추가하기 위한 4 쓰레드를 실행할 queue runner 를 생성한다.
qr = tf.train.QueueRunner(queue, [enqueue_op] * 4)

그래프 시작
sess = tf.Session()
queue runner 쓰레드들을 실행하는 coordinator 생성
coord = tf.train.Coordinator()
enqueue_threads = qr.create_threads(sess, coord=coord, start=True)
coordinator 와 함께 종료를 제어하는 학습 루프를 실행
for step in xrange(1000000):
 if coord.should_stop():
 break
 sess.run(train_op)
완료 후, 쓰레드에 정지 요청
coord.request_stop()
실제 정지하기를 대기
coord.join(threads)
```

## 예외처리 다루기

queue runner 에 의해 시작된 쓰레드들은 그냥 연산들을 큐에 추가하여 실행하는 것보다 더 많은 일을 수행한다. 또한 이들은 큐가 닫혔음을 알리기 위해 사용되는 `OutOfRangeException` 를 포함하여, 큐에 의해 생성된 예외처리들을 포착하고 처리한다.

`coordinator` 를 사용하는 학습 프로그램은 마찬가지로 이들의 메인 루프에서 예외처리들을 포착하고 알려야 한다.

아래는 위 학습 루프의 향상된 버전이다.

```
try:
 for step in xrange(1000000):
 if coord.should_stop():
 break
 sess.run(train_op)
except Exception, e:
 # Report exceptions to the coordinator.
 coord.request_stop(e)
finally:
 # Terminate as usual. It is innocuous to request stop twice.
 coord.request_stop()
 coord.join(threads)
```

# 분산환경 텐서플로우

(v1.0)

이 문서는 텐서플로우 서버 클러스터를 생성하고, 클러스터 상에서 분산 처리를 수행하는 방법에 대하여 설명한다. 이 문서의 독자들은 텐서플로우를 이용한 [기본적인 프로그래밍 개념](#)은 숙지가 되어있다고 가정한다.

## 안녕 분산환경 텐서플로우!

텐서플로우 클러스터의 간단한 동작을 살펴보기 위해서는 아래 예제를 실행해보아라.

```
싱글 프로세스 클러스터에서 텐서플로우 서버 실행하기.
$ python
>>> import tensorflow as tf
>>> c = tf.constant("Hello, distributed TensorFlow!")
>>> server = tf.train.Server.create_local_server()
>>> sess = tf.Session(server.target) # 서버에서 세션 생성하기.
>>> sess.run(c)
'Hello, distributed TensorFlow!'
```

`tf.train.Server.create_local_server()` 메소드는 in-process 서버 형태로 단일 프로세스 클러스터를 생성한다.

## 클러스터 생성하기

텐서플로우에서 "클러스터"란 텐서플로우 그래프 상에서의 분산 연산의 일부로서 "작업(Task)"의 집합을 의미한다. 각각의 작업은 텐서플로우의 "서버"에 연관되어 있으며, 각 서버는 세션을 생성할 수 있는 "마스터"와 그래프상에서 연산을 수행하는 "작업자"로 구성된다. 각 클러스터는 복수개의 "직무(Jobs)"로 구성되어 있으며, 각각의 직무는 복수개의 작업으로 이루어져 있다.

클러스터를 생성하기 위해서는, 작업 하나당 하나의 텐서플로우 서버를 실행해야한다. 각 작업은 보통 서로 다른 머신에서 실행되지만, 하나의 머신에서 여러개의 작업을 실행하는 것도 가능하다. (예를 들어 복수개의 GPU를 사용하는 경우). 각 작업마다 아래의 절차를 따라서 진행된다.

1. 클러스터에 할당된 작업을 설명하는 `tf.train.ClusterSpec` 을 생성하라. 이 것은 각 작업마다 동일해야 한다.
2. `tf.train.ClusterSpec` 를 생성자 인자로 넘겨주어 `tf.train.Server` 를 생성하고, 로컬 작업을 직무 이름과 작업 인덱스로 식별하라.

## 클러스터를 설명하는 `tf.train.ClusterSpec` 생성하기

클러스터 명세 딕셔너리는 직무 이름과 네트워크 주소를 매핑한다. 아래와 같이 딕셔너리를 `tf.train.ClusterSpec` 의 생성자의 인자로 넘겨 주어라.

| <code>tf.train.ClusterSpec</code> 생성자                                                                                                                                                                                                                     | 사용 가능 작업                                                                                                                                                            |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>tf.train.ClusterSpec({"local": ["localhost:2222", "localhost:2223"]})</code>                                                                                                                                                                        | <code>/job:local/task:0</code> / <code>job:local/task:1</code>                                                                                                      |
| <code>tf.train.ClusterSpec({     "worker": [         "worker0.example.com:2222",         "worker1.example.com:2222",         "worker2.example.com:2222"     ],     "ps": [         "ps0.example.com:2222",         "ps1.example.com:2222"     ] })</code> | <code>/job:worker/task:0</code><br><code>/job:worker/task:1</code><br><code>/job:worker/task:2</code><br><code>/job:ps/task:0</code><br><code>/job:ps/task:1</code> |

## 각 작업마다 `tf.train.Server` 객체 생성하기

`tf.train.Server` 객체는 여러개의 로컬 디바이스 정보와, 각 작업과 디바이스를 연결해주는 정보인 `tf.train.ClusterSpec`, 분산 연산 수행에 이용되는 "session target" 을 포함하고 있다.

각각의 서버는 특정한 이름이 부여된 직무의 멤버이며, 해당 직무에서의 작업 인덱스를 가지고 있다. 서버는 클러스터내에 있는 다른 서버와 통신이 가능하다.

예를 들어, `localhost:2222` `localhost:2223` 두 개의 서버를 가진 클러스터를 구동하려면 밑의 두 코드를 로컬머신의 다른 두 개의 프로세스에서 실행하면 된다.

```
0번 작업:
cluster = tf.train.ClusterSpec({"local": ["localhost:2222", "localhost:2223"]})
server = tf.train.Server(cluster, job_name="local", task_index=0)
```

```
1번 작업:
cluster = tf.train.ClusterSpec({"local": ["localhost:2222", "localhost:2223"]})
server = tf.train.Server(cluster, job_name="local", task_index=1)
```

**Note:** 거대한 클러스터를 생성하기 위해서 클러스터 명세를 일일히 수동으로 작성하는 것은 지루한 일일 수 있다. 우리는 이러한 작업을 프로그래밍으로 구동하기 위한 도구를 고민중이다. 예를 들어, [Kubernetes](#) 와 같은 클러스터 매니저를 사용하는 방법이다. 만약 우리가 지원하기를 원하는 클러스터 매니저가 있다면, [GitHub issue](#) 에 제보해주길 바란다.

## 모델 내에서 디바이스 명시하기

연산을 CPU 혹은 GPU에서 수행할지 선택할 때와 마찬가지로, 특정 연산에 연산을 수행할 디바이스를 명시할 때는 `tf.device()` 를 사용하면 된다.

```
with tf.device("/job:ps/task:0"):
 weights_1 = tf.Variable(...)
 biases_1 = tf.Variable(...)

with tf.device("/job:ps/task:1"):
 weights_2 = tf.Variable(...)
 biases_2 = tf.Variable(...)

with tf.device("/job:worker/task:7"):
 input, labels = ...
 layer_1 = tf.nn.relu(tf.matmul(input, weights_1) + biases_1)
 logits = tf.nn.relu(tf.matmul(layer_1, weights_2) + biases_2)
 # ...
 train_op = ...

with tf.Session("grpc://worker7.example.com:2222") as sess:
 for _ in range(10000):
 sess.run(train_op)
```

위의 예제에서, 변수들은 `ps` 직무(job)에서 생성이 되고, 연산이 집중적으로 일어나는 모델은 `worker` 에서 생성이 된다. 텐서플로우는 각각의 직무간에 적절하게 데이터를 이동시켜준다.(정방향 연산시에는 `ps` 에서 `worker` 로 gradients 전파시에는 `worker` 에서 `ps` 로 전달한다.)

## 훈련 복제

일반적으로 "데이터 병렬화(data parallelism)"로 명명되는 훈련 방식은 `worker` 직무의 여러개의 작업이 하나의 모델에 대하여, 데이터의 각기 다른 일부를 이용하여 `ps` 에서 생성된 공유변수를 병렬적으로 업데이트 시키는 방식이다. 모든 작업들은 각각 다른 머신에서 동작한다. 텐서플로우에

서 이러한 훈련 방식을 구현하는 방법은 여러가지가 있는데, 우리는 복제된 모델을 간단하게 생성 할 수 있도록 도와주는 라이브러리를 구축하였다. 시도 가능한 방법은 아래와 같다:

- **그래프내 복제(In-graph replication)**. 이 방법에서 클라이언트는 한 세트의 변수( `/job:ps` 에 연관된 `tf.Variable` )가 포함된 `tf.Graph` 하나를 구축하고, `/job:worker` 에 소속된 서로 다른 작업에 각각 연관된 여러개의 연산 집중 모델을 복제하여 구축한다.
- **그래프간 복제(Between-graph replication)**. 이 방법에서는 각 `/job:worker` 작업마다 별도의 클라이언트가 존재하며 일반적으로 연산수행 작업과 동일한 클라이언트에 있다. 각 클라이언트는 변수를 포함하는 유사한 그래프를 구축한다. (각 변수는 `tf.train.replica_device_setter()` 를 사용하기 전에 `/job:ps` 에 연관되어있고, 사용후에 동일한 작업에 매핑 된다.) 연산 집중 모델의 하나의 복사본은 `/job:worker` 의 로컬 작업에 연관되어 있다.
- **비동기식 훈련(Asynchronous training)**. 이 방법에서는 각 그래프의 복제품이 독립적으로 각자 고유의 훈련 루프를 가지고 있다. 이 방법은 위의 두 복제방식과 호환이 가능하다.
- **동기식 훈련(Synchronous training)**. 이 방식에서는 각 그래프의 복제품이 현재의 변수에서 값을 읽어오고, 병렬적으로 gradient를 계산한뒤 병렬적으로 모델에 반영한다. 이 방법은 위의 두 복제방식과 호환이 가능하다. 예를 들어 [CIFAR-10 multi-GPU trainer](#) 에서 와 같이 gradient 평균을 활용하여 그래프내 복제를 하거나, `tf.train.SyncReplicasOptimizer` 를 활용해서 그래프간 복제를 활용하는 방법이 있다.

## 총 정리 : 훈련 프로그램 예시

아래 코드는 그래프간 복제와 비동기식 훈련을 활용한 분산 훈련 프로그램의 뼈대 코드이다. 아래 코드에는 변수 서버(ps)와 연산 수행(worker)작업을 구현한 코드도 포함되어 있다.

```
import argparse
import sys
import tensorflow as tf

FLAGS=None

def main(_):
 ps_hosts = FLAGS.ps_hosts.split(",")
 worker_hosts = FLAGS.worker_hosts.split(",")

 # 변수 서버와 작업자 클러스터를 생성한다.
 cluster = tf.train.ClusterSpec({"ps": ps_hosts, "worker": worker_hosts})

 # 로컬 작업 수행을 위해 서버를 생성하고 구동한다.
 server = tf.train.Server(cluster,
 job_name=FLAGS.job_name,
 task_index=FLAGS.task_index)

 if FLAGS.job_name == "ps":
 server.join()
```

```

elif FLAGS.job_name == "worker":

 # 작업자에 연산을 분배한다.
 with tf.device(tf.train.replica_device_setter(
 worker_device="/job:worker/task:%d" % FLAGS.task_index,
 cluster=cluster)):

 # 모델 구축...
 loss = ...
 global_step = tf.Variable(0)

 train_op = tf.train.AdagradOptimizer(0.01).minimize(
 loss, global_step=global_step)

 saver = tf.train.Saver()
 summary_op = tf.merge_all_summaries()
 init_op = tf.initialize_all_variables()

 # 훈련 과정을 살펴보기 위해 "supervisor"를 생성한다.
 sv = tf.train.Supervisor(is_chief=(FLAGS.task_index == 0),
 logdir="/tmp/train_logs",
 init_op=init_op,
 summary_op=summary_op,
 saver=saver,
 global_step=global_step,
 save_model_secs=600)

 # supervisor는 세션 초기화를 관리하고, checkpoint로부터 모델을 복원하고
 # 에러가 발생하거나 연산이 완료되면 프로그램을 종료한다.
 with sv.managed_session(server.target) as sess:
 # "supervisor"가 종료되거나 1000000 step이 수행 될 때까지 반복한다.
 step = 0
 while not sv.should_stop() and step < 1000000:
 # 훈련 과정을 비동기식으로 실행한다.Run a training step asynchronously.
 # 동기식 훈련 수행을 위해서는 `tf.train.SyncReplicasOptimizer`를 참조하라.
 _, step = sess.run([train_op, global_step])

 # 모든 서비스 중단.
 sv.stop()

if __name__ == "__main__":
 parser = argparse.ArgumentParser()
 parser.register("type", "bool", lambda v: v.lower() == "true")
 # Flags for defining the tf.train.ClusterSpec
 parser.add_argument(
 "--ps_hosts",
 type=str,
 default="",
 help="Comma-separated list of hostname:port pairs"
)
 parser.add_argument(
 "--worker_hosts",
 type=str,

```

```

 default="",
 help="Comma-separated list of hostname:port pairs"
)
parser.add_argument(
 "--job_name",
 type=str,
 default="",
 help="One of 'ps', 'worker'"
)
Flags for defining the tf.train.Server
parser.add_argument(
 "--task_index",
 type=int,
 default=0,
 help="Index of task within the job"
)
FLAGS, unparsed = parser.parse_known_args()
tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)

```

두 개의 변수 서버와 두 개의 연산 수행 작업으로 구성된 훈련용 프로그램을 구동하기 위해서는, 아래 커맨드 라인을 실행하면 된다.(스크립트 파일명이 `train.py` 라고 가정한다.) :

```

On ps0.example.com:
$ python trainer.py \
 --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
 --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
 --job_name=ps --task_index=0

On ps1.example.com:
$ python trainer.py \
 --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
 --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
 --job_name=ps --task_index=1

On worker0.example.com:
$ python trainer.py \
 --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
 --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
 --job_name=worker --task_index=0

On worker1.example.com:
$ python trainer.py \
 --ps_hosts=ps0.example.com:2222,ps1.example.com:2222 \
 --worker_hosts=worker0.example.com:2222,worker1.example.com:2222 \
 --job_name=worker --task_index=1

```

## 용어 사전

### 클라이언트

클라이언트는 일반적으로 텐서플로우 그래프를 정의하고, 클러스터와 상호작용하기 위해 `'tensorflow::Session'`을 구축하는 프로그램이다. 클라이언트는 보통 파이썬 또는 C++로 작성

된다. 하나의 클라이언트 프로세스는 여러개의 텐서플로우 서버와 직접적으로 상호작용할 수 있으며(위의 "훈련 복제" 단원을 참조하라), 하나의 서버는 여러개의 클라이언트의 요청을 처리할 수 있다.

## 클러스터

텐서플로우 클러스터는 하나 혹은 복수개의 "직무(job)"로 구성 되며, 각 직무는 하나 혹은 복수 개의 "작업(task)"으로 이루어 진다. 하나의 클러스터는 특정한 하나의 목표에 기여한다. 예를 들어 여러개의 머신을 활용하여 병렬적으로 신경망 회로를 훈련시키는 것을 들 수 있다.. 클러스터는 `tf.train.ClusterSpec` 객체를 이용하여 정의된다.

## 직무(Job)

하나의 직무(Job)은 여러 개의 작업(task)으로 이루어진다. 각 작업은 하나의 목적을 처리한다. 예를 들어 `ps(parameter server)`란 이름의 직무는 일반적으로 변수를 저장하는 주체이며, `worker`란 이름의 직무는 연산 집중 업무를 수행하는 노드를 담당한다. 직무에 포함되는 각 작업은 일반적으로 다른 머신에서 실행 된다. 각 직무가 담당하는 범위는 유동적으로 결정 된다. 예를 들어 `worker`가 변수를 저장하는 경우가 있을 수도 있다.

## 마스터 서비스(Master service)

분산된 디바이스에 원격 접근을 제공하고, 세션의 타겟으로 작동하는 RPC 서비스이다. 마스터 서비스는 `tensorflow::Session` 인터페이스를 구현하며, 복수개의 연산 수행 작업간의 조정을 담당하고 있다. 모든 텐서플로우 서버는 마스터 서비스를 구현해야 한다.

## 작업(Task)

하나의 작업은 하나의 텐서플로우 서버에 대응이 되고, 역시 하나의 프로세스와 대응된다. 하나의 작업은 특정한 직무에 소속되며, 직무 내의 작업 리스트에서는 작업 인덱스로 구분이 된다.

## 텐서플로우 서버(TensorFlow server)

`tf.train.Server` 를 수행하는 프로세스이다. 각 프로세스는 클러스터의 멤버이며 마스터 서비스와 작업 서비스를 제공한다.

## 작업 서비스(Worker service)

로컬 디바이스를 이용하여 텐서플로우 그래프의 한 부분 연산을 수행하는 RPC 서비스이다. 각 작업 서비스는 `worker_service.proto` 를 구현한다. 모든 텐서플로우 서버는 작업서비스를 구현 한다.

# 새 작업을 추가하세요

전제조건:

- C++에 어느정도 친숙할 것.
- 반드시 [TensorFlow binary](#)가 설치되어 있거나, [downloaded TensorFlow source](#)가 있어야만, 빌드 할 수 있음

만약 당신이 존재하는 라이브러리로 감싸져있지 않은 작업을 포함 하길 원한다면, custom Op를 생성할 수 있습니다. 당신의 custom Op를 포함하기 위해서, '이하'의 항목을 충족해야 합니다.

이하:

- C++ 파일에서 새로운 작업을 등록하세요. 그 작업 등록은 실행에서 독립적이고, 그 작업이 들먹여 지는 방법의 의미론을 말합니다.(?) 예를들어, 이것은 작업의 이름을 정의하고 입력과 출력들을 구체적으로 명시합니다.
- C++안에서 그 작업을 실행하세요. 이 실행은 "커널"이라고 불립니다. 그리고 각색의 구조들(CPUs, GPUs) 또는 입출력 형태들을 위한 다양한 커널들이 존재 할 수 있습니다.
- 경우에 따라, 파이썬 래퍼(wrapper)를 만드세요. 이 래퍼는 작업을 생성하는 공용의 API입니다. 기본적인 래퍼는 작업 등록으로부터 발생되어집니다. 그리고 그것은 직접적으로 사용되어질 수 있거나 추가 되어질 수 있습니다.
- 경우에 따라, 그 작업을 위해 경사도(gradients)를 계산할 함수를 써넣으세요.
- 경우에 따라, 그 작업을 위해 입출력 모양들을 설명할 함수를 써넣으세요. 이것이 작업 추론으로 하여금 당신의 작업을 다룰 수 있도록 허락합니다.
- 전형적으로, 파이썬에서 그 작업을 테스트 하세요. 만약 당신이 기울기들을 정의한다면, 파이썬으로 그 것들을 식별할 수 있을 것입니다. [GradientChecker](#).

[TOC]

## 작업의 인터페이스를 정의하세요

텐서플로우 시스템으로 작업을 등록함으로, 당신은 그 작업의 인터페이스를 정의할 수 있습니다. 등록에서, 당신의 작업 이름과 그 작업의 입력들(형태들과 이름들)과 출력들(형태들과 이름들) 그리고 'docstrings' 과 그 작업이 요구할지도 모를 어떤 속성들을 명시합니다.

이것이 어떻게 동작할지 보기 위해서는, 당신이 'int32'들의 텐서를 챙겨서, 그것의 복사본을 출력하는 작업을 만들고 싶어함에도 불구하고 그 첫번째 요소는 0으로 세트한다고 가정해보세요.

`tensorflow/core/user_ops /zero_out.cc` 파일을 생성하세요. 그리고 Create file

`tensorflow/core/user_ops /zero_out.cc` and '이하'의 작업을 위한 인터페이스를 정의하는 `REGISTER_OP` macro 에의 요청을 추가하세요.

이하 :

```
#"tensorflow/core/framework/op.h"를 포함하세요 (Include)

REGISTER_OP("ZeroOut")
 .Input("to_zero: int32")
 .Output("zeroed: int32");
```

This `ZeroOut` Op takes one tensor `to_zero` of 32-bit integers as input, and 이 `zeroout` 작업은 텐서 한개를 32비트 정수의 `to_zero` 를 입력으로 이해한다. 그리고 텐서 한개를 32비트 정수의 `zeroed` 로 출력한다.

A note on naming: The name of the Op should be unique and CamelCase. Names 이름 명명에 관한 주목할 점 : 작업의 이름은 유일해야하고 'CamelCase'여야 합니다. 밑줄 (`_`)로 시작하는 이름들은 내부 사용을 위해 예약되어집니다. starting with an underscore (`_`) are reserved for internal use.

## 작업을 하기 위해 커널을 실행

당신이 인터페이스를 정의한 후에, 하나 혹은 더 많은 작업의 실행을 제공하세요. 이 커널들 중 한 개를 생성하기 위해서, `OpKernel` 를 확장하는 클래스 한개를 생성하고 `compute` 메소드를 오버라이드 하세요. `Compute` 메소드는 입출력 텐서와 같은 유용한 것들에 접근하게 하는 `OpKernelContext*` 타입의 `context` 매개변수 한 개를 제공합니다.

Important note: Instances of your OpKernel may be accessed concurrently. Your 중요한 메모 : 당신의 작업커널(OpKernel)의 인스턴스들은 동시에 접근되어질지도 모릅니다. 당신의 Compute 메소드는 다양한 쓰레드들로부터 안전하게 연결되어질 것임에 틀림없습니다. 럭스의 클래스 멤버와의 연결을 지키세요.(아니면 클래스 맴버를 통한 상태를 공유하지 않는게 낫습니다!

작업 상태를 계속 파악 하기 위해서 `ResourceMgr` 의 사용을 고려하세요.

당신의 커널을 당신이 먼저 만들어 놓은 파일에 추가하세요. 그 커널은 '이하'의 것과 같이 보일 것입니다.

이하:

```
#include "tensorflow/core/framework/op_kernel.h"

using namespace tensorflow;

class ZeroOutOp : public OpKernel {
public:
 explicit ZeroOutOp(OpKernelConstruction* context) : OpKernel(context) {}

 void Compute(OpKernelContext* context) override {
 // Grab the input tensor
 const Tensor& input_tensor = context->input(0);
 auto input = input_tensor.flat<int32>();

 // Create an output tensor
 Tensor* output_tensor = NULL;
 OP_REQUIRES_OK(context, context->allocate_output(0, input_tensor.shape(),
 &output_tensor));
 auto output = output_tensor->template flat<int32>();

 // Set all but the first element of the output tensor to 0.
 const int N = input.size();
 for (int i = 1; i < N; i++) {
 output(i) = 0;
 }

 // Preserve the first input value if possible.
 if (N > 0) output(0) = input(0);
 }
};


```

커널을 실행 한 뒤, 텐서플로우 시스템에 그것을 등록합니다. 등록할 때, 당신은 이 커널이 동작하게 될 다른 제약사항들을 명시 합니다. 예를 들어, 당신이 하나의 커널을 CPUs를 위해 그리고 다른 하나는 GPUs를 위해 만들수도 있습니다. `zeroout` 작업을 위한 이 일을 하기 위해서, `zero_out.cc` 를 따라서 추가하세요.

```
REGISTER_KERNEL_BUILDER(Name("ZeroOut").Device(DEVICE_CPU), ZeroOutOp);
```

## 작업 라이브러리를 빌드

### 텐써플로우 바이너리 설치도 병행

당신의 시스템에서 동작 할 수 있는 `g++` 또는 `clang` 과 같은 `c++` 컴파일러로 `zero_out.cc` 을 컴파일 할 수 있어야 합니다. 바이너리 PIP 패키지는 당신의 작업을 시스템이 명시한 곳에서 컴파일 해야만 하는 라이브러리와 헤더파일을 설치합니다. 그러나, 텐써플로우 파이썬 라이브러리는

`get_include` 함수를 제공합니다. 이 함수는 헤더 디렉토리를 얻게 합니다. 여기 이 함수의 출력된 값을 우분투 머신에서 볼 수 있습니다.

```
$ python
>>> import tensorflow as tf
>>> tf.sysconfig.get_include()
'/usr/local/lib/python2.7/site-packages/tensorflow/include'
```

가령 당신이 설치된 `g++` 를 가졌다고 가정해 본다면, 당신이 다이나믹 라이브러리 안에서 작업을 컴파일 하는 것을 가능하도록 해주는 커맨드들의 흐름들이 '이하'에 있습니다.

이하:

```
TF_INC=$(python -c 'import tensorflow as tf; print(tf.sysconfig.get_include())')

g++ -std=c++11 -shared zero_out.cc -o zero_out.so -fPIC -I $TF_INC
```

맥 OS에서, "-undefined dynamic\_lookup"라는 추가적인 표시사항은 `.so` 파일을 빌드할 때 필수적으로 필요합니다.

gcc 5버전에서의 주의사항 : gcc 5는 새로운 C++을 사용합니다. [ABI](#) 텐서플로우 웹사이트에서 이용 가능한 바이너리 pip 패키지들은 더 오래된 ABI를 사용하는 gcc4로 빌드되어졌습니다. 만약 당신이 gcc5로 작업 라이브러리를 컴파일 한다면, `-D_GLIBCXX_USE_CXX11_ABI=0` 를 커맨드라인에 추가해야합니다. 왜냐하면, 그 라이브러리를 오래된 abi와 호환가능하게 해야하기 때문입니다.

## 텐서플로우 소스 설치와 함께

만약 당신이 텐서플로우를 다 설치 했다면, 당신의 작업을 컴파일하는 텐서플로우의 빌드 시스템을 이용할 수 있습니다. `Bazel` 빌드 규칙(`tensorflow/core/user_ops` 디렉토리)을 따라 빌드 파일을 가져다 놓으세요.

```
load("//tensorflow:tensorflow.bzl", "tf_custom_op_library")

tf_custom_op_library(
 name = "zero_out.so",
 srcs = ["zero_out.cc"],
)
```

'이하'의 `zero_out.so` 를 빌드하는 명령을 실행하세요.

이하:

```
$ bazel build -c opt //tensorflow/core/user_ops:zero_out.so
```

알림: 표준 `cc_library` 규칙으로, 당신이 공유된 라이브러리 (`.so` 파일)를 생성할 수 있음에도 불구하고, `tf_custom_op_library` 매크로를 사용할 것을 강력하게 권고합니다. 이것이 어떤 의존들(dependencies)을 추가하고, 공유된 라이브러리가 텐서플로우의 플러그인 로딩 구조와 호환이 되는지 점검합니다.

## 파이썬에서의 작업 실행

텐서플로우 파이썬 API는 역동적인 라이브러리를 로드하는 것과 텐서플로우 프레임워크에 작업을 등록하기 위해서 `load_op_library` 함수를 제공합니다. `load_op_library` 는 작업을 위한 파이썬 래퍼들을 담고 있는 파이썬 모듈을 반환합니다. 게다가, 당신이 그 작업을 빌드 했다면, 파이썬으로부터 이하의 작업을 실행 할 수 있습니다. 이하:

```
import tensorflow as tf
zero_out_module = tf.load_op_library('zero_out.so')
with tf.Session(''):
 zero_out_module.zero_out([[1, 2], [3, 4]]).eval()

프린트들
array([[1, 0],
 [0, 0]], dtype=int32)
```

알림: 발생된 함수는 ([PEP8](#))을 준수하기 위해서 뱀형(snake\_case)이름을 받을 것입니다. 그래서 만약 C++ 파일에서 `zeroout` 으로 작업이름을 명명한다면, 파이썬 함수는 `zero_out` 로 쓰여질 것입니다.

파이썬 모듈로 부터 일반적인 함수 `import -able`로써, 그 작업이 사용 가능해지도록 하기 위해서, 파이썬 소스파일(이하 참조 : [zero\\_out\\_op\\_1.py](#))에서 `load_op_library` 을 가지고 있는 것이 유용 할지도 모릅니다.

이하:

```
tf로써 텐서플로우를 임포트

_zero_out_module = tf.load_op_library('zero_out_op_kernel_1.so')
zero_out = _zero_out_module.zero_out
```

## 이것이 작동하는지 확인하세요.

당신이 성공적으로 작업을 수행했다는 것을 확인할 좋은 방법은 테스트를 작성하는 것입니다.

`tensorflow/python/kernel_tests/zero_out_op_test.py` 파일을 '이하'의 내용으로 작성하세요.

이하:

```
import tensorflow as tf

class ZeroOutTest(tf.test.TestCase):
 def testZeroOut(self):
 zero_out_module = tf.load_op_library('zero_out.so')
 with self.test_session():
 result = zero_out_module.zero_out([5, 4, 3, 2, 1])
 self.assertAllEqual(result.eval(), [5, 0, 0, 0, 0])
```

이렇게 한 이후에 당신의 테스트를 실행해 보세요.

```
$ bazel test tensorflow/python:zero_out_op_test
```

## 유효성

# Validation

이 예제는 작업이 먼저 어떤 모양의 텐서에 적용했다는 것을 가정합니다. 만약에 이것을 오직 벡터에만 적용한다면 어떻게 될까요? 이 말은 확인(check)을 OpKernel 구현 위에 추가한다는 것을 의미 합니다.

```
void Compute(OpKernelContext* context) override {
 // Grab the input tensor
 const Tensor& input_tensor = context->input(0);

 OP_REQUIRES(context, TensorShapeUtils::IsVector(input_tensor.shape()),
 errors::InvalidArgument("ZeroOut expects a 1-D vector."));
 // ...
}
```

이것은 '입력은 한 벡터다'라고 주장합니다. 만약 입력이 있지 않다면, 이것은 `InvalidArgument` 상태를 세팅한 것을 되돌립니다. `OP_REQUIRES` macro은 3요소를 가지고 있습니다.

- `context` 의 `SetStatus()` 메소드를 위해서 `OpKernelContext` 혹은 `OpKernelConstruction` 포인터(참조: `tensorflow/core/framework/op_kernel.h`) 둘 중 하나가 될수 있는 `context`.
- '상태(condition)'. 예를들어, `tensorflow/core/framework/tensor_shape.h`에서의 텐서 모양을 확인하기 위한 함수들이 있습니다.

- `Status` 객체에 의해 보여지는 '에러 그자체' (참조 : [tensorflow/core/lib/core/status.h](#))  
`Status` 는 타입(종종 `InvalidArgument` 이긴 하나, 타입들의 리스트를 봅니다)과 메시지를 가집니다. 에러를 구성하는 것을 위한 함수들은 [tensorflow/core/lib/core/errors.h](#) 에서 찾을지도 모릅니다.

그렇지 않으면, 만약 당신이 어떤 함수로 부터 반환되어진 `status` 객체가 오류인지 아닌지를 테스트하고, 그것을 반환하기를 원한다면 `OP_REQUIRES_OK` 를 사용하세요. 이 두가지 매크로들은 오류에 걸린 함수로 부터 되돌아 옵니다.

## 작업 등록

### 속성들

작업들은 속성을 가질 수 있습니다. 그리고, 속성의 값은 작업이 그래프에 추가되어질 때 할당되어집니다. 이 속성들은 작업의 환경 설정을 위해 사용되어 지며, 속성들의 값은 커널 구현과 작업 등록의 입출력 형태안에서 접근되어 질 수 있습니다. 입력이 가능 할 때, 속성보다 입력들이 좀 더 유연하기 때문에, 속성보단 입력을 사용할 것을 권장합니다. 입력들은 모든 단계들을 바꿀 수 있고, `feed`를 사용할 준비 등을 할 수 있습니다. 속성들은 특징(숫자 혹은 입출력의 형태)에 영향을 주거나 단계별로 변경할 수 없는 환경설정들과 같은 입력을 끝마칠 수 없는 것들을 위해 사용되어집니다.

당신은 작업을 등록 할 때, `Attr` 메소드를 사용하는 속성의 이름과 타입을 명시함으로 속성을 정의합니다.

`Attr` 메소드에서 예상할 수 있는 형태:

```
<name>: <attr-type-expr>
```

`<name>` 이 한 글자로 시작하고, 글자와 숫자로 쓴 문자와 밑줄, 그리고 `<attr-type-expr>` 로 구성되어 질 수 있는 곳은 '이하'에 표현된 품의 형태입니다. (#attr-types)

이하: 예를 들어, 만약 당신이 `ZeroOut` 작업이 사용자 지정 색인을 보존하기 원한다면, 단지 0번째 요소 대신에 다음과 같은 작업을 등록 할 수 있습니다. `REGISTER\_OP("ZeroOut") .Attr("preserve\_index: int") .Input("to\_zero: int32") .Output("zeroed: int32");`

당신의 커널은 `context` 파라미터를 통해서 이것의 `constructor` 안에 있는 이 속성에 접근 할 수 있습니다.

```
class ZeroOutOp : public OpKernel { public: explicit ZeroOutOp(OpKernelConstruction* context) : OpKernel(context) { // Get the index of the value to preserve OP_REQUIRES_OK(context, context->GetAttr("preserve_index", &preserve_index_)); // Check that preserve_index is positive OP_REQUIRES(context, preserve_index_ >= 0, errors::InvalidArgument("Need preserve_index >= 0, got ", preserve_index_)); } void Compute(OpKernelContext* context) override { // ... } private: int preserve_index_; }
```

Compute 메소드에서 사용 가능 한 것 :

```
void Compute(OpKernelContext* context) override { // ... // Check that preserve_index
is in range OP_REQUIRES(context, preserve_index_ < input.dimension(0),
errors::InvalidArgument("preserve_index out of range")); // Set all the elements of the
output tensor to 0 const int N = input.size(); for (int i = 0; i < N; i++) {
output_flat(i) = 0; } // Preserve the requested input value
output_flat(preserve_index_) = input(preserve_index_); }
```

**backwards compatibility** 를 보호하기 위해서, 당신이 '이하'존재하는 작업에 속성을 추가할 때 **default value**를 명시해야 합니다. 이하: REGISTER\\_OP("ZeroOut")
.Attr("preserve\\_index: int = 0") .Input("to\\_zero: int32") .Output("zeroed: int32");

## 속성 타입들

'이하'의 타입들은 속성에서 지원됩니다.

이하:

- `string` : 바이트들의 연속 (UTF8이 필수는 아님).
- `int` : 부호가 붙은 정수형.
- `float` : 부동 소수점 숫자.
- `bool` : 참 혹은 거짓.
- `type` : `DataType` 의 불 참조 값들 중에 하나.
- `shape` : `TensorShapeProto` .
- `tensor` : `TensorProto` .
- `list(<type>)` : `<type>` 이 상위 타입들 중 하나인 곳에서 `<type>` 의 리스트.
`list(list(<type>))` 가 유효하지 않음을 주의하세요 .

참조 : 최종 리스트를 위한 `op_def_builder.cc:FinalizeAttr`

## 기본 값 & 제약사항

속성들은 기본 값을 가지고 있을 것이다. 그리고 속성의 어떤 타입들은 제약사항을 가질 수 있다. 제약사항이 있는 속성을 정의하기 위해선, '아래'의 `<attr-type-expr>` 를 이용 할 수 있습니다. 이하:

- `{'<string1>', '<string2>'}` : 같은 `<string1>` 혹은 `<string2>` 둘중 하나를 가지고 있는 'string'이여야만 합니다. 당신이 이 문법을 사용할 때, 타입의 이름인 `string` 은 암시되어집니다.

이것은 'enum'을 모방합니다 :

```
REGISTER_OP("EnumExample")
.Attr("e: {'apple', 'orange'}");
```

- {<type1>, <type2>} : 같은 type 타입이고, <type1> 이나 <type2> 이 'tensor types'에 의해 지원되는 곳에서 <type1> 혹은 <type2> 중 하나여야만 합니다.

당신은 '속성의 타입이 `type`이다.'라고 명시하지 않습니다. 이것은 당신이 `...`안에서 타입의 리스트를 가질때 암시되어집니다.

예를 들어 이 경우엔, `t` 속성이 `int32`, `float`, `bool` 중 하나여야만 하는 타입입니다 :

```
REGISTER_OP("RestrictedTypeExample")
 .Attr("t: {int32, float, bool}");
```

- 일반적인 타입의 제약사항들을 위해 여기 몇가지 손쉬운 방법이 있습니다 :

- numbertype : type 타입은 숫자형으로 제한됩니다. (non-string and non-bool)
- realnumbertype : 복잡한 타입이 없이 numbertype 와 같습니다.
- quantizedtype : quantized 숫자를 제외한 numbertype 와 같습니다.

이러한 것들로 허가되어진 타입들의 구체적인 리스트들은 함수들('이하'참조)에 의해 정의 되어집니다. 이하 : [tensorflow/core/framework/types.h](#) 에 있는 NumberTypes() 와 같습니다.

이 사례에서 t 속성은 반드시 숫자 타입들중 하나여야만 합니다 :

```
REGISTER_OP("NumberType")
 .Attr("t: numbertype");
```

이 작업(op)을 위해서:

```
tf.number_type(t=tf.int32) # Valid
tf.number_type(t=tf.bool) # Invalid
```

- int >= <n> : 이 같은 자연수인 <n> 보다 크거나 같은 값이어야 합니다.

예를 들어, '아래'의 작업 등록은 a 속성이 최소 2 인 값을 가지고 있다는 것을 명시합니다. 아래:

```
REGISTER_OP("MinIntExample")
 .Attr("a: int >= 2");
```

- list(<type>) >= <n> : <n> 보다 크거나 같은 길이를 가진 <type> 타입의 리스트입니다.

예를 들어, '아래'의 작업등록은 a 속성이 int32 혹은 float 둘중 하나의 타입의 리스트이고, 적어도 그것 들 중에서 3이있어야만 한다는 것을 명시합니다. 아래:

```
REGISTER_OP("TypeListExample")
 .Attr("a: list({int32, float}) >= 3");
```

발생된 코드에 값을 선택적으로 하는 속성을 위한 기본 값을 할당하기 위해서는, 끝 부분에 `= <default>` 를 추가하세요. 예:

```
REGISTER_OP("AttrDefaultExample")
 .Attr("i: int = 0");
```

기본값에 대해 지원 되는 문법은 GraphDef 의미를 결과로 내는 것들 중에서 프로토(proto)표시에 사용 되어진다.

모든 타입의 기본값을 명시하는 방법에 대한 예 :

```
REGISTER_OP("AttrDefaultExampleForAllTypes")
 .Attr("s: string = 'foo'")
 .Attr("i: int = 0")
 .Attr("f: float = 1.0")
 .Attr("b: bool = true")
 .Attr("ty: type = DT_INT32")
 .Attr("sh: shape = { dim { size: 1 } dim { size: 2 } }")
 .Attr("te: tensor = { dtype: DT_INT32 int_val: 5 }")
 .Attr("l_empty: list(int) = []")
 .Attr("l_int: list(int) = [2, 3, 5, 7]");
```

특히 `type` 타입의 값을 사용하는 것에 주의하세요. 타입을 위한 `DT_*` 이름들.

## Polymorphism

### Type Polymorphism

For ops that can take different types as input or produce different output types, you can specify [an attr](#) in [an input or output type](#) in the Op registration. Typically you would then register an `OpKernel` for each supported type.

For instance, if you'd like the `ZeroOut` Op to work on `float` `s` in addition to `int32` `s`, your Op registration might look like:

```
REGISTER_OP("ZeroOut") .Attr("T: {float, int32}") .Input("to_zero: T") .Output("zeroed: T");
```

Your Op registration now specifies that the input's type must be `float`, or `int32`, and that its output will be the same type, since both have type `T`.

A note on naming: Inputs, outputs, and attrs generally should be given snake\_case names. The one exception is attrs that are used as the type of an input or in the type of an input. Those attrs can be inferred when the op is added to the graph and so don't appear in the op's function. For example, this last definition of ZeroOut will generate a Python function that looks like:

```
def zero_out(to_zero, name=None):
 """
 ...
 Args:
 to_zero: A `Tensor`. Must be one of the following types:
 `float32`, `int32`.
 name: A name for the operation (optional).

 Returns:
 A `Tensor`. Has the same type as `to_zero`.
 """

```

If `to_zero` is passed an `int32` tensor, then `T` is automatically set to `int32` (well, actually `DT_INT32`). Those inferred attrs are given Capitalized or CamelCase names.

Compare this with an op that has a type attr that determines the output type:

```
REGISTER_OP("StringToNumber")
 .Input("string_tensor: string")
 .Output("output: out_type")
 .Attr("out_type: {float, int32}");
 .Doc(R"doc(
Converts each string in the input Tensor to the specified numeric type.
)doc");
```

In this case, the user has to specify the output type, as in the generated Python:

```
def string_to_number(string_tensor, out_type=None, name=None):
 """
 Converts each string in the input Tensor to the specified numeric type.

 Args:
 string_tensor: A `Tensor` of type `string`.
 out_type: An optional `tf.DType` from: `tf.float32, tf.int32`.
 Defaults to `tf.float32`.
 name: A name for the operation (optional).

 Returns:
 A `Tensor` of type `out_type`.
 """

```

```
\#include "tensorflow/core/framework/op_kernel.h"
class ZeroOutInt32Op : public OpKernel
{ // as before };
class ZeroOutFloatOp : public OpKernel { public: explicit
ZeroOutFloatOp(OpKernelConstruction* context) : OpKernel(context) {} void
Compute(OpKernelContext* context) override { // Grab the input tensor const Tensor&
input_tensor = context->input(0); auto input = input_tensor.flat<float>(); // Create an
output tensor Tensor* output = NULL; OP_REQUIRES_OK(context, context->allocate_output(0,
input_tensor.shape(), &output)); auto output_flat = output->template flat<float>(); // Set
all the elements of the output tensor to 0 const int N = input.size(); for (int i = 0; i <
N; i++) { output_flat(i) = 0; } // Preserve the first input value if (N > 0)
output_flat(0) = input(0); } }; // Note that TypeConstraint<int32>"T" means that attr
"T" (defined // in the Op registration above) must be "int32" to use this template //
instantiation. REGISTER_KERNEL_BUILDER(Name("ZeroOut") .Device(DEVICE_CPU)
.TypeConstraint<int32>"T"), ZeroOutOp<int32>); REGISTER_KERNEL_BUILDER(Name("ZeroOut")
.Device(DEVICE_CPU) .TypeConstraint<float>"T"), ZeroOutFloatOp);
```

To preserve [backwards compatibility](#), you should specify a [default value](#) when adding an attr to an existing op:

```
REGISTER_OP("ZeroOut") .Attr("T: {float, int32} = DT_INT32") .Input("to_zero: T")
.Output("zeroed: T")
```

Lets say you wanted to add more types, say double :

```
REGISTER_OP("ZeroOut") .Attr("T: {float, double, int32}") .Input("to_zero: T")
.Output("zeroed: T");
```

Instead of writing another `OpKernel` with redundant code as above, often you will be able to use a C++ template instead. You will still have one kernel registration (`REGISTER\_KERNEL\_BUILDER` call) per overload.

```
template <typename T> class ZeroOutOp : public OpKernel { public: explicit
ZeroOutOp(OpKernelConstruction* context) : OpKernel(context) {} void
Compute(OpKernelContext* context) override { // Grab the input tensor const Tensor&
input_tensor = context->input(0); auto input = input_tensor.flat<T>(); // Create an
output tensor Tensor* output = NULL; OP_REQUIRES_OK(context, context->allocate_output(0,
input_tensor.shape(), &output)); auto output_flat = output->template flat<T>(); // Set all
the elements of the output tensor to 0 const int N = input.size(); for (int i = 0; i < N;
i++) { output_flat(i) = 0; } // Preserve the first input value if (N > 0) output_flat(0)
= input(0); } }; // Note that TypeConstraint<int32>"T" means that attr "T" (defined // in
the Op registration above) must be "int32" to use this template // instantiation.
REGISTER_KERNEL_BUILDER(Name("ZeroOut") .Device(DEVICE_CPU) .TypeConstraint<int32>
("T"), ZeroOutOp<int32>); REGISTER_KERNEL_BUILDER(Name("ZeroOut") .Device(DEVICE_CPU)
.TypeConstraint<float>"T"), ZeroOutOp<float>); REGISTER_KERNEL_BUILDER(Name("ZeroOut")
.Device(DEVICE_CPU) .TypeConstraint<double>"T"), ZeroOutOp<double>);
```

If you have more than a couple overloads, you can put the registration in a macro.

```
#include "tensorflow/core/framework/op_kernel.h"

#define REGISTER_KERNEL(type) \
REGISTER_KERNEL_BUILDER(\
 Name("ZeroOut").Device(DEVICE_CPU).TypeConstraint<type>("T"), \
 ZeroOutOp<type>)

REGISTER_KERNEL(int32);
REGISTER_KERNEL(float);
REGISTER_KERNEL(double);

#undef REGISTER_KERNEL
```

Depending on the list of types you are registering the kernel for, you may be able to use a macro provided by `tensorflow/core/framework/register_types.h` :

```
#include "tensorflow/core/framework/op_kernel.h"
#include "tensorflow/core/framework/register_types.h"

REGISTER_OP("ZeroOut")
 .Attr("T: realnumbertype")
 .Input("to_zero: T")
 .Output("zeroed: T");

template <typename T>
class ZeroOutOp : public OpKernel { ... };

#define REGISTER_KERNEL(type) \
REGISTER_KERNEL_BUILDER(\
 Name("ZeroOut").Device(DEVICE_CPU).TypeConstraint<type>("T"), \
 ZeroOutOp<type>)

TF_CALL_REAL_NUMBER_TYPES(REGISTER_KERNEL);

#undef REGISTER_KERNEL
```

## List Inputs and Outputs

In addition to being able to accept or produce different types, ops can consume or produce a variable number of tensors.

In the next example, the attr `T` holds a *list* of types, and is used as the type of both the input `in` and the output `out`. The input and output are lists of tensors of that type (and the number and types of tensors in the output are the same as the input, since both have type `T`).

```
REGISTER_OP("PolymorphicListExample")
 .Attr("T: list(type)")
 .Input("in: T")
 .Output("out: T");
```

You can also place restrictions on what types can be specified in the list. In this next case, the input is a list of `float` and `double` tensors. The Op accepts, for example, input types `(float, double, float)` and in that case the output type would also be `(float, double, float)`.

```
REGISTER_OP("ListTypeRestrictionExample")
 .Attr("T: list({float, double})")
 .Input("in: T")
 .Output("out: T");
```

If you want all the tensors in a list to be of the same type, you might do something like:

```
REGISTER_OP("IntListInputExample")
 .Attr("N: int")
 .Input("in: N * int32")
 .Output("out: int32");
```

This accepts a list of `int32` tensors, and uses an `int` attr `N` to specify the length of the list.

This can be made [type polymorphic](#) as well. In the next example, the input is a list of tensors (with length `"N"`) of the same (but unspecified) type (`"T"`), and the output is a single tensor of matching type:

```
REGISTER_OP("SameListInputExample")
 .Attr("N: int")
 .Attr("T: type")
 .Input("in: N * T")
 .Output("out: T");
```

By default, tensor lists have a minimum length of 1. You can change that default using a ["`>=`" constraint on the corresponding attr](#). In this next example, the input is a list of at least 2 `int32` tensors:

```
REGISTER_OP("MinLengthIntListExample")
 .Attr("N: int >= 2")
 .Input("in: N * int32")
 .Output("out: int32");
```

The same syntax works with `"list(type)"` attrs:

```
REGISTER_OP("MinimumLengthPolymorphicListExample")
 .Attr("T: list(type) >= 3")
 .Input("in: T")
 .Output("out: T");
```

## Inputs and Outputs

To summarize the above, an Op registration can have multiple inputs and outputs:

```
REGISTER_OP("MultipleInsAndOuts")
 .Input("y: int32")
 .Input("z: float")
 .Output("a: string")
 .Output("b: int32");
```

Each input or output spec is of the form:

```
<name>: <io-type-expr>
```

where `<name>` begins with a letter and can be composed of alphanumeric characters and underscores. `<io-type-expr>` is one of the following type expressions:

- `<type>`, where `<type>` is a supported input type (e.g. `float`, `int32`, `string`). This specifies a single tensor of the given type.

See [the list of supported Tensor types](#).

```
REGISTER_OP("BuiltInTypesExample")
 .Input("integers: int32")
 .Input("complex_numbers: complex64");
```

- `<attr-type>`, where `<attr-type>` is the name of an `Attr` with type `type` or `list(type)` (with a possible type restriction). This syntax allows for [polymorphic ops](#).

```
REGISTER_OP("PolymorphicSingleInput")
 .Attr("T: type")
 .Input("in: T");

REGISTER_OP("RestrictedPolymorphicSingleInput")
 .Attr("T: {int32, int64}")
 .Input("in: T");
```

Referencing an attr of type `list(type)` allows you to accept a sequence of tensors.

```
REGISTER_OP("ArbitraryTensorSequenceExample")
 .Attr("T: list(type)")
 .Input("in: T")
 .Output("out: T");

REGISTER_OP("RestrictedTensorSequenceExample")
 .Attr("T: list({int32, int64})")
 .Input("in: T")
 .Output("out: T");
```

Note that the number and types of tensors in the output `out` is the same as in the input `in`, since both are of type `T`.

- For a sequence of tensors with the same type: `<number> * <type>`, where `<number>` is the name of an `Attr` with type `int`. The `<type>` can either be a specific type like `int32` or `float`, or the name of an attr with type `type`. As an example of the first, this Op accepts a list of `int32` tensors:

```
REGISTER_OP("Int32SequenceExample")
 .Attr("NumTensors: int")
 .Input("in: NumTensors * int32")
```

Whereas this Op accepts a list of tensors of any type, as long as they are all the same:

```
REGISTER_OP("SameTypeSequenceExample")
 .Attr("NumTensors: int")
 .Attr("T: type")
 .Input("in: NumTensors * T")
```

- For a reference to a tensor: `Ref(<type>)`, where `<type>` is one of the previous types.

A note on naming: Any attr used in the type of an input will be inferred. By convention those inferred attrs use capital names (like `T` or `N`). Otherwise inputs, outputs, and attrs have names like function parameters (e.g. `num_outputs`). For more details, see the [earlier note on naming](#).

For more details, see [tensorflow/core/framework/op\\_def\\_builder.h](#).

## Backwards compatibility

In general, changes to specifications must be backwards-compatible: changing the specification of an Op must not break prior serialized `GraphDef` protocol buffers constructed from older specifications. The details of `GraphDef` compatibility are [described here](#).

There are several ways to preserve backwards-compatibility.

1. Any new attrs added to an operation must have default values defined, and with that default value the Op must have the original behavior. To change an operation from not polymorphic to polymorphic, you *must* give a default value to the new type attr to preserve the original signature by default. For example, if your operation was:

```
REGISTER_OP("MyGeneralUnaryOp")
 .Input("in: float")
 .Output("out: float");
```

you can make it polymorphic in a backwards-compatible way using:

```
REGISTER_OP("MyGeneralUnaryOp")
 .Input("in: T")
 .Output("out: T")
 .Attr("T: numerictype = DT_FLOAT");
```

2. You can safely make a constraint on an attr less restrictive. For example, you can change from `{int32, int64}` to `{int32, int64, float}` or `type`. Or you may change from `{"apple", "orange"}` to `{"apple", "banana", "orange"}` or `string`.
3. You can change single inputs / outputs into list inputs / outputs, as long as the default for the list type matches the old signature.
4. You can add a new list input / output, if it defaults to empty.
5. Namespace any new Ops you create, by prefixing the Op names with something unique to your project. This avoids having your Op colliding with any Ops that might be included in future versions of Tensorflow.
6. Plan ahead! Try to anticipate future uses for the Op. Some signature changes can't be done in a compatible way (for example, making a list of the same type into a list of varying types).

The full list of safe and unsafe changes can be found in

[tensorflow/core/framework/op\\_compatibility\\_test.cc](#). If you cannot make your change to an operation backwards compatible, then create a new operation with a new name with the new semantics.

Also note that while these changes can maintain `GraphDef` compatibility, the generated Python code may change in a way that isn't compatible with old callers. The Python API may be kept compatible by careful changes in a hand-written Python wrapper, by keeping the old signature except possibly adding new optional arguments to the end. Generally incompatible changes may only be made when TensorFlow's changes major versions, and must conform to the [GraphDef version semantics](#).

## GPU Support

You can implement different OpKernels and register one for CPU and another for GPU, just like you can [register kernels for different types](#). There are several examples of kernels with GPU support in [tensorflow/core/kernels/](#). Notice some kernels have a CPU version in a `.cc` file, a GPU version in a file ending in `_gpu.cc`, and some code shared in common in a `.h` file.

For example, the `pad` op has everything but the GPU kernel in

[tensorflow/core/kernels/pad\\_op.cc](#). The GPU kernel is in [tensorflow/core/kernels/pad\\_op\\_gpu.cc](#), and the shared code is a templated class defined in [tensorflow/core/kernels/pad\\_op.h](#). One thing to note, even when the GPU kernel version of `pad` is used, it still needs its `"paddings"` input in CPU memory. To mark that inputs or outputs are kept on the CPU, add a `HostMemory()` call to the kernel registration, e.g.:

```
#define REGISTER_GPU_KERNEL(T) \
 REGISTER_KERNEL_BUILDER(Name("Pad") \
 .Device(DEVICE_GPU) \
 .TypeConstraint<T>("T") \
 .HostMemory("paddings"), \
 PadOp<GPUDevice, T>)
```

## Compiling the kernel for the GPU device

Look at [cuda\\_op\\_kernel.cu.cc](#) for an example that uses a CUDA kernel to implement an op. The `tf_custom_op_library` accepts a `gpu_srcs` argument in which the list of source files containing the CUDA kernels (`*.cu.cc` files) can be specified. For use with a binary installation of TensorFlow, the CUDA kernels have to be compiled with NVIDIA's `nvcc` compiler. Here is the sequence of commands you can use to compile the `cuda_op_kernel.cu.cc` and `cuda_op_kernel.cc` into a single dynamically loadable library:

```
nvcc -std=c++11 -c -o cuda_op_kernel.cu.o cuda_op_kernel.cu.cc \
-I $TF_INC -D GOOGLE_CUDA=1 -x cu -Xcompiler -fPIC

g++ -std=c++11 -shared -o cuda_op_kernel.so cuda_op_kernel.cc \
cuda_op_kernel.cu.o -I $TF_INC -fPIC -lcudart
```

`cuda_op_kernel.so` produced above can be loaded as usual in Python, using the `tf.load_op_library` function.

## Implement the gradient in Python

Given a graph of ops, TensorFlow uses automatic differentiation (backpropagation) to add new ops representing gradients with respect to the existing ops (see [Gradient Computation](#)). To make automatic differentiation work for new ops, you must register a gradient function which computes gradients with respect to the ops' inputs given gradients with respect to the ops' outputs.

Mathematically, if an op computes  $y = f(x)$  the registered gradient op converts gradients  $(\partial / \partial y)$  with respect to  $y$  into gradients  $(\partial / \partial x)$  with respect to  $x$  via the chain rule:

$$\frac{\partial}{\partial x} = \frac{\partial}{\partial y} \frac{\partial y}{\partial x} = \frac{\partial}{\partial y} \frac{\partial f}{\partial x}.$$

In the case of `zeroout`, only one entry in the input affects the output, so the gradient with respect to the input is a sparse "one hot" tensor. This is expressed as follows:

```

from tensorflow.python.framework import ops
from tensorflow.python.ops import array_ops
from tensorflow.python.ops import sparse_ops

@ops.RegisterGradient("ZeroOut")
def _zero_out_grad(op, grad):
 """The gradients for `zero_out`.

Args:
 op: The `zero_out` `Operation` that we are differentiating, which we can use
 to find the inputs and outputs of the original op.
 grad: Gradient with respect to the output of the `zero_out` op.

Returns:
 Gradients with respect to the input of `zero_out`.
"""
 to_zero = op.inputs[0]
 shape = array_ops.shape(to_zero)
 index = array_ops.zeros_like(shape)
 first_grad = array_ops.reshape(grad, [-1])[0]
 to_zero_grad = sparse_ops.sparse_to_dense(index, shape, first_grad, 0)
 return [to_zero_grad] # List of one Tensor, since we have one input

```

Details about registering gradient functions with `ops.RegisterGradient` :

- For an op with one output, the gradient function will take an `Operation` `op` and a `Tensor` `grad` and build new ops out of the tensors `op.inputs[i]` , `op.outputs[i]` , and `grad` . Information about any attrs can be found via `op.get_attr` .
- If the op has multiple outputs, the gradient function will take `op` and `grads` , where `grads` is a list of gradients with respect to each output. The result of the gradient function must be a list of `Tensor` objects representing the gradients with respect to each input.
- If there is no well-defined gradient for some input, such as for integer inputs used as indices, the corresponding returned gradient should be `None` . For example, for an op taking a floating point tensor `x` and an integer index `i` , the gradient function would  
`return [x_grad, None]` .
- If there is no meaningful gradient for the op at all, use `ops.NoGradient("OpName")` to disable automatic differentiation.

Note that at the time the gradient function is called, only the data flow graph of ops is available, not the tensor data itself. Thus, all computation must be performed using other tensorflow ops, to be run at graph execution time.

# Implement a shape function in Python

The TensorFlow Python API has a feature called "shape inference" that provides information about the shapes of tensors without having to execute the graph. Shape inference is supported by "shape functions" that are registered for each op type, and perform two roles: asserting that the shapes of the inputs are compatible, and specifying the shapes for the outputs. A shape function is a Python function that takes an `operation` as input, and returns a list of `TensorShape` objects (one per output of the op). To register a shape function, apply the `tf.RegisterShape` decorator to a shape function. For example, the `ZeroOut` op defined above would have a shape function like the following:

```
@tf.RegisterShape("ZeroOut")
def _zero_out_shape(op):
 """Shape function for the ZeroOut op.

 This is the unconstrained version of ZeroOut, which produces an output
 with the same shape as its input.
 """
 return [op.inputs[0].get_shape()]
```

A shape function can also constrain the shape of an input. For the version of `ZeroOut` with a [vector shape constraint](#), the shape function would be as follows:

```
@tf.RegisterShape("ZeroOut")
def _zero_out_shape(op):
 """Shape function for the ZeroOut op.

 This is the constrained version of ZeroOut, which requires the input to
 have rank 1 (a vector).
 """
 input_shape = op.inputs[0].get_shape().with_rank(1)
 return [input_shape]
```

If your op is [polymorphic with multiple inputs](#), use the properties of the operation to determine the number of shapes to check:

```
@tf.RegisterShape("IntListInputExample")
def _int_list_input_example_shape(op):
 """Shape function for the "IntListInputExample" op.

 All inputs and the output are matrices of the same size.
 """
 output_shape = tf.TensorShape(None)
 for input in op.inputs:
 output_shape = output_shape.merge_with(input.get_shape().with_rank(2))
 return [output_shape]
```

Since shape inference is an optional feature, and the shapes of tensors may vary dynamically, shape functions must be robust to incomplete shape information for any of the inputs. The `merge_with` method allows the caller to assert that two shapes are the same, even if either or both of them do not have complete information. Shape functions are defined for all of the standard Python ops, and provide many different usage examples.

# TensorFlow 문서 작성

TensorFlow의 문서는 [Markdown](#)에서 유지되고 있고 `g3doc/` 디렉토리에 존재합니다.

*Introduction, Overview, Tutorials, How-Tos* 부분은 수작업으로 수정됩니다.

`g3doc/api_docs` 디렉토리에 있는 어떤 것이든 코드에 있는 주석으로부터 생성되기 때문에 직접 수정해선 안됩니다. `tools/docs/gen_docs.sh` 스크립트는 API문서를 생성합니다. 스크립트를 인자 없이 호출하면 Python API 문서만을 재작성합니다. (Ops에 관한 문서는 Python, C++ 상관없이 작성됩니다.). `-a` 를 인자로 전달하면 C++ 문서 또한 재작성합니다. 이것은 반드시 `tools/docs` 디렉토리에서 호출되어야 합니다. 그리고 `-a` 를 인자로 전달하려면 `doxygen` 의 설치가 요구됩니다.

## Python API 문서

Ops, classes, utility 함수는 `image_ops.py` 와 같은 Python 모듈에 정의되어 있습니다. 그 모듈의 docstring은 해당 Python 파일에 대해 생성되는 마크다운 파일의 시작부분에 삽입됩니다. 그래서 `image_ops.md` 는 `image_ops.py` 모듈에 있는 docstring으로 시작합니다.

`python/framework/gen_docs_combined.py` 는 마크다운 파일이 생성되는 모든 *libraries*(라이브러리) 목록을 포함하고 있습니다. 만약에 새로운 라이브러리를 더한다면 (API 문서에 독립된 섹션을 생성하는 것), 반드시 `gen_docs_combined.py` 에 있는 라이브러리 목록에 추가해야 합니다. C++ API는 하나의 라이브러리 파일만 존재합니다, 그것의 마크다운은 `api_docs/cc/index.md` 에서 생성된 `gen_cc_md.py` 의 문자열입니다. 나머지 C++문서들은 `doxygen`에 의해 만들어진 XML파일로부터 생성됩니다.

라이브러리로 등록된 파일의 모듈 docstring에서, (빈 줄의 시작 부분에) `@@<python-name>` 문법으로 Ops, classes, functions를 호출해 그것들을 위해 생성된 문서를 삽입할 수 있습니다. 호출한 op, function 또는 class는 같은 파일에서 정의될 필요가 없습니다. 이것은 Ops, classes, functions가 기록되는 순서를 정할 수 있게 해줍니다. 고수준의 문서를 적절히 배치해서 논리적인 순서로 나누세요.

모든 공개된 op, class, function 은 반드시 라이브러리의 서두에서 `@@` 로 호출되어야 합니다. 그렇게 하지 않을 경우 `doc_gen_test` 가 실패하게 됩니다.

Ops를 위한 문서는 자동적으로 Python wrapper 또는 C++ Ops registrations로 부터 발췌합니다. 둘 중 Python wrappers를 우선적으로 가져옵니다.

- Python wrappers는 `python/ops/*.py` 에 있습니다.
- C++ Ops registrations는 `core/ops/*.cc` 에 있습니다.

Classes와 Utility Functions를 위한 문서는 docstring에서 발췌합니다.

# Op 문서 스타일 가이드

이상적으로는, 제시된 순서를 따라 아래와 같은 정보를 제공해야 합니다:

- op가 무엇을 하는지를 설명하는 짧은 문장.
- op에 인자를 전달할 때 어떤 일이 생기는지에 대한 짧은 설명.
- op가 어떻게 작동하는지를 보여주는 예시(수도코드가 가장 좋음).
- 요구사항, 경고, 중요한 내용 (하나라도 있다면).
- 입력, 출력, 속성, op 생성자의 다른 변수에 대한 설명.

위 항목에 대해 설명된 자세한 정보는 [이곳](#).

글을 마크다운 (.md) 포맷으로 적으세요. 기본적인 문법에 관한 레퍼런스는 [여기](#)에 있습니다. 방정식 표기를 위해 [MathJax](#)을 사용할 수 있습니다. 이것들은 [tensorflow.org](#)에서 적절히 표현됩니다. 하지만 [github](#)에 나타나지는 않습니다.

## 코드에 대해서 적을 때

문서에서 아래와 같은 것들을 적을 때  (backticks)로 감싸야 합니다:

- 인자의 이름 (e.g. `input` , `x` , `tensor` )
- 반환되는 Tensor의 이름 (e.g. `output` , `idx` , `out` )
- 데이터 타입 (e.g. `int32` , `float` , `uint8` )
- 문서에서 언급되는 다른 op의 이름 (e.g. `list_diff()` , `shuffle()` )
- 클래스 이름 (e.g. 실제로 `Tensor` 객체를 의미할 때만 `Tensor` 를 사용하세요. op가 `tensor`, 그래프, 실행에서 보통 어떻게 작동하는지 설명할 때는 대문자나 backticks를 사용하면 안됩니다.)
- 파일 이름 (e.g. `image_ops.py` , `/path-to-your-data/xml/example-name` )

예시 코드와 수도코드는 backticks를 세번 써서 감싸야 합니다. op가 무엇을 반환하는지 보여주고 싶을 때는 single equal sign( '=' 기호 )대신 `==>`를 사용해야 합니다. 예시:

```
```
# 'input' is a tensor of shape [2, 3, 5]
(tf.expand_dims(input, 0)) ==> [1, 2, 3, 5]
```
```

Python 코드 예시를 제공하고자 한다면 문법 강조가 적절히 되도록 Python 스타일 라벨을 추가하세요:

```
```python
# some Python code
```

수식이나 조건은 하나의 backticks로 감싸세요. 예시:

```
```markdown
This operation requires that `^-1-input.dims() <= dim <= input.dims()` .
```

## Tensor의 차원

보통 tensor에 대해 말할 때는 tensor라는 단어의 첫 글자를 대문자로 쓰지 마세요. op에 인자로 전달하거나 op에서 반환하는 특정한 객체를 말할 때는 Tensor의 첫 글자를 대문자로 쓰고 backticks로 감싸세요. 왜냐하면 전달되어지는 `Tensor` 객체에 대해 말하고 있기 때문입니다.

진짜 `Tensors`라는 객체에 대해 이야기하지 않는다면 `Tensors`를 여러개의 Tensor 객체를 서술할 때 사용하지 마세요. "a list of `Tensor` objects." (`Tensor` 객체 리스트) 또는 "`Tensor`s" (`Tensor` 들)로 부르는게 낫습니다.

`tensor`의 크기에 대해 이야기 할 때는 아래의 가이드라인을 참고하세요:

`tensor`의 크기를 말할 때는 "dimension"(차원)이라는 단어를 사용하세요. 크기를 특정화할 필요가 있다면 아래의 규칙을 사용하세요:

- scalar(스칼라)는 "0-D tensor"로 부른다
- vector(벡터)는 "1-D tensor"로 부른다
- matrix(메트릭스)는 "2-D tensor"로 부른다
- 3차원 이상인 `tensors`는 "3-D tensors" 또는 n-D tensors"로 부른다. 이해가 될 경우에는 "rank"라는 단어를 사용한다. 하지만 "dimension"을 대신 쓰도록 노력하라. 절대 `tensor`의 크기를 표현하기 위해 "order"라는 단어를 사용하지 말라.

`tensor`의 dimensions를 자세히 설명 하려면 "shape"라는 단어를 사용하세요. 그리고 backticks와 꺽쇠 괄호를 사용해서 모양을 보여주세요. 예시:

```
If `input` is a 3-D tensor with shape `[3, 4, 3]`, this operation will return
a 3-D tensor with shape `[6, 8, 6]` .
```

## 링크

`g3docs` 트리에 있는 다른 것들에 링크를 걸려면 `[tf.parse_example]`

(`.../api_docs/python/ops.md#parse_example`) 처럼 상대 경로를 사용하세요. 내부 링크에 절대 경로를 사용하지 마세요. 왜냐하면 웹사이트 생성기를 손상시키기 때문입니다.

소스코드에 링크를 걸려면 <https://www.tensorflow.org/code/>로 시작하고 깃허브 루트에서 시작하는 파일 이름으로 이어지는 링크를 사용하세요. 예를 들면, 이 파일로 연결된 링크는

[https://www.tensorflow.org/code/tensorflow/g3doc/how\\_tos/documentation/index.md](https://www.tensorflow.org/code/tensorflow/g3doc/how_tos/documentation/index.md) 처럼 쓰여

져야 합니다. 이것은 [tensorflow.org](https://tensorflow.org) 가 당신이 보고 있는 문서의 버전에 대응한 코드의 브랜치를 가리킬 수 있도록 보장해 줍니다. url 파라미터를 URL에 포함하지 마세요.

## C++에서 정의된 Ops

C++에 정의된 모든 Ops는 반드시 `REGISTER_OP` 선언 부분에 기록되어 있어야 합니다. C++ 파일에 있는 docstring은 입력 타입, 출력 타입, Attr 타입, default 값 정보를 자동적으로 추가하기 위해 처리됩니다.

예시:

```
REGISTER_OP("PngDecode")
 .Input("contents: string")
 .Attr("channels: int = 0")
 .Output("image: uint8")
 .Doc(R"doc(
Decodes the contents of a PNG file into a uint8 tensor.

contents: PNG file contents.
channels: Number of color channels, or 0 to autodetect based on the input.
Must be 0 for autodetect, 1 for grayscale, 3 for RGB, or 4 for RGBA.
If the input has a different number of channels, it will be transformed
accordingly.
image:= A 3-D uint8 tensor of shape `[height, width, channels]`.
If `channels` is 0, the last dimension is determined
from the png contents.
)doc");
```

이 부분의 마크다운의 결과:

```
tf.image.png_decode(contents, channels=None, name=None) {#png_decode}

Decodes the contents of a PNG file into a uint8 tensor.

Args:

* contents: A string Tensor. PNG file contents.
* channels: An optional int. Defaults to 0.
Number of color channels, or 0 to autodetect based on the input.
Must be 0 for autodetect, 1 for grayscale, 3 for RGB, or 4 for RGBA. If the
input has a different number of channels, it will be transformed accordingly.
* name: A name for the operation (optional).

Returns:

A 3-D uint8 tensor of shape `[height, width, channels]`.
If `channels` is 0, the last dimension is determined
from the png contents.
```

인자에 대한 설명은 대부분 자동적으로 추가됩니다. 특히 doc generator(문서 생성기)는 자동적으로 모든 입력, attrs, 출력의 이름과 타입을 추가합니다. 위의 예시에서 `<b>contents</b>: A string Tensor.` 는 자동으로 추가되었습니다. 글이 자연스럽게 흘러가도록 추가적인 문장을 해당 설명 뒤에 써야 합니다.

입,출력에 대해서, equal sign( = 기호)을 추가적인 문장의 서두에 붙여서 이름과 타입을 자동으로 추가하는 것을 막을 수 있습니다. 위의 예시에서, 이름이 `image` 인 출력에 대한 설명은 우리가 입력한 글 `A 3-D uint8 Tensor...` 이전에 `A uint8 Tensor.` 가 추가되는 것을 방지하기 위해 `=`로 시작합니다. 이 방법으로는 attrs의 이름, 타입, default 값이 추가되는 것을 막을 수는 없기 때문에 글을 적을 때 신중해야 합니다.

## Python에서 정의된 Ops

op가 `python/ops/*.py` 파일에 정의되어 있다면, 모든 인자와 출력값(반환값) tensor에 관한 글을 제공해야 합니다.

Python docstring 규칙에 따라야 하고 docstring에서 마크다운을 사용해야 합니다. doc generator(문서 생성기)는 Python에 정의된 ops에 관한 글은 어떤 것도 자동으로 생성하지 않기 때문에 당신이 적은 것을 사용합니다.

간단한 예시:

```
def foo(x, y, name="bar"):
 """Computes foo.

 Given two 1-D tensors `x` and `y`, this operation computes the foo.

 For example:

```

**x is [1, 1]**

**y is [2, 2]**

`tf.foo(x, y) ==> [3, 3]`

```

Args:
 x: A `Tensor` of type `int32`.
 y: A `Tensor` of type `int32`.
 name: A name for the operation (optional).

Returns:
 A `Tensor` of type `int32` that is the foo of `x` and `y`.

Raises:
 ValueError: If `x` or `y` are not of type `int32`.
"""

...

```

## Docstring 부분에 관한 설명

여기에서 더 자세한 내용과 docstring의 각 속성에 대한 예시가 있습니다.

### **op**가 무엇을 하는지 설명하는 짧은 문장

예시:

```
Concatenates tensors.
```

```
Flips an image horizontally from left to right.
```

```
Computes the Levenshtein distance between two sequences.
```

```
Saves a list of tensors to a file.
```

```
Extracts a slice from a tensor.
```

### **op**에 인자를 전달했을 때 무엇이 일어나는지에 대한 짧은 설명.

예시:

Given a tensor input of numerical type, this operation returns a tensor of the same type and size with values reversed along dimension `seq\_dim`. A vector `seq\_lengths` determines which elements are reversed for each index within dimension 0 (usually the batch dimension).

This operation returns a tensor of type `dtype` and dimensions `shape`, with all elements set to zero.

## op가 어떻게 작동하는지를 보여주는 예시.

`squeeze()` op에 좋은 수도코드 예시가 있습니다:

```
shape(input) => `[1, 2, 1, 3, 1, 1]`
shape(squeeze(input)) => `[2, 3]`
```

`tile()` op는 좋은 설명문의 예시를 제공합니다:

```
For example, tiling `[a, b, c, d]` by 2 produces
`[[a, b, c, d], [a, b, c, d]]`.
```

Python에서 코드 예시를 보여주는 것은 도움이 됩니다. 절대 그것을 C++ Ops 파일에 넣지 마세요. 그리고 Python Ops 문서에 넣는 것도 피하세요. Ops 생성자가 호출되는 모듈이나 클래스의 docstring에 삽입하세요.

여기 `image_ops.py`에 있는 모듈 docstring 예제가 있습니다:

```
Tensorflow can convert between images in RGB or HSV. The conversion
functions work only on `float` images, so you need to convert images in
other formats using [`convert_image_dtype`](#convert-image-dtype).
```

Example:

```
```python
# Decode an image and convert it to HSV.
rgb_image = tf.image.decode_png(..., channels=3)
rgb_image_float = tf.image.convert_image_dtype(rgb_image, tf.float32)
hsv_image = tf.image.rgb_to_hsv(rgb_image)
```
```

## 필요조건, 경고, 중요한 사항들.

예시:

```
This operation requires that: `^-1-input.dims() <= dim <= input.dims()`
```

Note: This tensor will produce an error if evaluated. Its value must be fed using the `feed\_dict` optional argument to `Session.run()`, `Tensor.eval()`, or `Operation.run()`.

## 인자와 출력(반환) tensors에 관한 설명.

설명은 요점만 간단히 하세요. 인자 부분에서는 어떻게 실행되는지를 설명할 필요가 없습니다.

op가 입력 혹은 출력 tensors의 dimension(차원)에 강한 제한이 있으면 언급해야 합니다. C++ Ops는 tensor의 타입이 자동으로 "A ..type.. Tensor" 또는 "A Tensor with type in {...list of types...}" 형태로 더해집니다. 이런 경우에, Op가 차원에 제한이 있으면 "Must be 4-D"라는 글을 더하거나 설명의 처음에 `=`를 추가하고(tensor의 타입이 자동으로 추가되는 것을 막기 위해) "A 4-D float tensor"와 같이 적으세요.

예를 들어, 여기에 C++ op의 이미지 인자를 문서화하는 두 가지 방법이 있습니다 ("=" 기호를 주목):

```
image: Must be 4-D. The image to resize.
```

```
image:= A 4-D `float` tensor. The image to resize.
```

문서에서 이와 같은 마크다운으로 렌더링됩니다

```
image: A `float` Tensor. Must be 4-D. The image to resize.
```

```
image: A 4-D `float` Tensor. The image to resize.
```

## 선택적인 인자 설명 ("attrs")

doc generator(문서 생성기)는 항상 attrs의 타입과 default 값을 서술합니다. C++와 Python doc generator에서 생성되는 설명이 매우 다르기 때문에 equal sign( = 기호)으로 override(오버라이드) 할 수 없습니다.

타입과 default 값 뒤에 흐름이 잘 이어지도록 추가적인 attr 설명을 표현하세요.

`image_ops.py` 에 있는 예시입니다:

```
REGISTER_OP("PngDecode")
 .Input("contents: string")
 .Attr("channels: int = 0")
 .Output("image: uint8")
 .Doc(R"doc(
Decode a PNG-encoded image to a uint8 tensor.

The attr `channels` indicates the desired number of color channels for the
decoded image.

Accepted values are:

* 0: Use the number of channels in the PNG-encoded image.
* 1: output a grayscale image.

...
contents: 0-D. The PNG-encoded image.
channels: Number of color channels for the decoded image.
image: 3-D with shape `[height, width, channels]`.
)doc");
```

이것은 아래와 같은 "Args" 부분을 만들어 냅니다:

```
contents: A string Tensor. 0-D. The PNG-encoded image.
channels: An optional `int`. Defaults to 0. Number of color channels for the
 decoded image.
name: A name for the operation (optional).
```

# 사용자 지정 데이터 리더

전제 조건:

- C++에 대한 약간의 익숙함
- [다운로드 받은 TensorFlow 소스](#)가 있어야 하고, 빌드 할 수 있어야 합니다.

파일에 지원하는 작업은 두 가지 형식으로 나뉩니다:

- 파일 형식: 파일에서 레코드(모든 문자열이 될 수 있음)를 읽는데 리더(*Reader*) 오퍼레이션(Op)을 사용합니다.
- 레코드 형식 : TensorFlow에서 사용할 수 있는 텐서(tensors)로 문자열 레코드를 설정하는데 디코더 또는 구문 분석 오퍼레이션(Ops)을 사용합니다.

예를 들어, [CSV 파일](#)을 읽는데, [텍스트 파일에 대한 리더](#) 이후에 [텍스트 라인으로부터 CSV 데이터를 분석하는 오퍼레이션\(Op\)](#)을 사용합니다.

[TOC]

## 파일 형식의 리더 작성

리더는 파일에서 레코드를 읽는 것입니다. 리더 오퍼레이션(Ops)에 대한 몇 가지 예제는 이미 TensorFlow에 내장되어 있습니다.

- [tf.TFRecordReader](#) ([kernels/tf\\_record\\_reader\\_op.cc](#) 소스)
- [tf.FixedLengthRecordReader](#) ([kernels/fixed\\_length\\_record\\_reader\\_op.cc](#) 소스)
- [tf.TextLineReader](#) ([kernels/text\\_line\\_reader\\_op.cc](#) 소스)

여러분은 이것들이 모두 같은 인터페이스를 노출하는 것을 확인할 수 있으며, 유일한 차이점은 생성자에 있습니다. 가장 중요한 메소드는 `read`입니다. `read`는 큐를 인자로 사용해서 필요할 때마다 언제든지 읽을 파일 이름을 알 수 있습니다. (e.g. `read` 오퍼레이션(op)이 처음 실행되거나 이전 `read` 가 파일에서 마지막 레코드를 읽을 때). 그것은 두 개의 스칼라 텐서(tensors)를 생성합니다: 문자열 키와 문자열 값.

`SomeReader`라는 새로운 리더를 생성하려면 다음을 수행해야 합니다 :

1. C++에서, `tensorflow::ReaderBase`에 `SomeReader`라는 서브클래스 정의.
2. C++에서, "SomeReader"라는 이름으로 새로운 리더 오퍼레이션(op)과 커널 등록.
3. Python에서, `tf.ReaderBase`에 `SomeReader`라는 서브클래스 정의.

여러분은 모든 C++ 코드를 `tensorflow/core/user_ops/some_reader_op.cc` 파일에 집어 넣을 수 있습니다. 파일을 읽는 코드는 C++ `ReaderBase` 의 서브 클래스에서 실행되고, `tensorflow/core/kernels/reader_base.h` 에 정의되어 있습니다. 여러분은 다음 메소드를 구현해야 합니다.

- `OnWorkStartedLocked` : 다음 파일 열기
- `ReadLocked` : 레코드를 읽거나 EOF/error 리포트
- `OnWorkFinishedLocked` : 현재 파일을 닫고
- `ResetLocked` : 예를 들어 에러가 발생한 후에 깨끗하게 만듭니다.

`ReaderBase` 내에 "Locked"으로 끝나는 이름을 갖는 메소드는 호출되기 전에 확실하게 뮤텍스 (mutex)를 습득하므로, 여러분은 일반적으로 쓰레드 세이프(thread safety)를 걱정할 필요가 없습니다. (클래스의 멤버로만 보호됩니다, 전역으로 보호되는 것이 아닙니다).

`OnWorkStartedLocked`의 경우, 열려 있는 파일의 이름은 `current_work()` 메소드에서 반환된 값입니다. `ReadLocked`은 이런 시그니처를 갖습니다.

```
Status ReadLocked(string* key, string* value, bool* produced, bool* at_end)
```

`ReadLocked`이 파일의 레코드를 성공적으로 읽으면, 아래 값을 채웁니다:

- `*key` : 레코드의 식별자로, 사용자는 이 레코드를 다시 찾을 때 사용할 수 있습니다. 여러분은 `current_work()`에 파일 이름을 포함할 수도 있고, 레코드 번호나, 또는 무엇이든 추가할 수 있습니다.
- `*value` : 레코드의 내용.
- `*produced` : `true`로 설정.

파일의 끝(EOF)에 도달하면, `*at_end`를 `true`로 설정합니다. 어떤 경우에도 `Status::OK()`를 반환합니다. 만약 에러가 발생하면, 매개변수를 변경하지 않고

`tensorflow/core/lib/core/errors.h`에 있는 도움말 함수 중 하나를 단순히 반환하는데 사용합니다.

다음으로 여러분은 실제로 리더 오퍼레이션(op)을 만들 것입니다. 여러분이 [오퍼레이션을 추가하는 방법](#)에 익숙하다면 도움이 될 겁니다. 주요 단계는 다음과 같습니다:

- 오퍼레이션(op) 등록.
- `OpKernel`을 정의하고 등록.

오퍼레이션(op) 등록은, `tensorflow/core/framework/op.h`에 정의되어 있는 `REGISTER_OP`을 호출해서 사용할 수 있습니다. 리더 오퍼레이션(ops)은 다른 입력을 받을 수 없고 항상 `Ref(string)` 타입의 단일 출력만 합니다. 항상 `SetIsStateful()`를 호출해야 하고, `container`와 `shared_name` 속성(attrs)은 문자열을 갖습니다. 선택적으로 configuration에 대한 추가적인 속성을 정의하거나 `Doc`에 문서를 포함할 수 있습니다. 예를 들어, `tensorflow/core/ops/io_ops.cc`를 예로 들어봅시다.

```
#include "tensorflow/core/framework/op.h"

REGISTER_OP("TextLineReader")
 .Output("reader_handle: Ref(string)")
 .Attr("skip_header_lines: int = 0")
 .Attr("container: string = ''")
 .Attr("shared_name: string = ''")
 .SetIsStateful()
 .Doc(R"doc(
A Reader that outputs the lines of a file delimited by '\n'.
)doc");
```

OpKernel 를 정의하기 위해, 리더(Readers)는 `tensorflow/core/framework/reader_op_kernel.h`에 정의된 `ReaderOpKernel`에서 shortcut of descending을 사용할 수 있고, `SetReaderFactory`를 호출하는 생성자를 구현합니다. 클래스를 정의하고 난 후에, `REGISTER_KERNEL_BUILDER(...)` 을 사용해서 등록해야 합니다. 속성이 없는 예제:

```
#include "tensorflow/core/framework/reader_op_kernel.h"

class TFRecordReaderOp : public ReaderOpKernel {
public:
 explicit TFRecordReaderOp(OpKernelConstruction* context)
 : ReaderOpKernel(context) {
 Env* env = context->env();
 SetReaderFactory([this, env]() { return new TFRecordReader(name(), env); });
 }
};

REGISTER_KERNEL_BUILDER(Name("TFRecordReader").Device(DEVICE_CPU),
 TFRecordReaderOp);
```

속성이 있는 예제:

```
#include "tensorflow/core/framework/reader_op_kernel.h"

class TextLineReaderOp : public ReaderOpKernel {
public:
 explicit TextLineReaderOp(OpKernelConstruction* context)
 : ReaderOpKernel(context) {
 int skip_header_lines = -1;
 OP_REQUIRES_OK(context,
 context->GetAttr("skip_header_lines", &skip_header_lines));
 OP_REQUIRES(context, skip_header_lines >= 0,
 errors::InvalidArgument("skip_header_lines must be >= 0 not ",
 skip_header_lines));
 Env* env = context->env();
 SetReaderFactory([this, skip_header_lines, env]() {
 return new TextLineReader(name(), skip_header_lines, env);
 });
 }
};

REGISTER_KERNEL_BUILDER(Name("TextLineReader").Device(DEVICE_CPU),
 TextLineReaderOp);
```

마지막 단계는 파이썬 wrapper를 추가하는 것입니다. [tensorflow/python/user\\_ops/user\\_ops.py](#)에 있는 `tensorflow.python.ops.io_ops` 를 임포트할 수 있고, `io_ops.ReaderBase` 의 서브 클래스를 추가할 수 있습니다.

```
from tensorflow.python.framework import ops
from tensorflow.python.ops import common_shapes
from tensorflow.python.ops import io_ops

class SomeReader(io_ops.ReaderBase):

 def __init__(self, name=None):
 rr = gen_user_ops.some_reader(name=name)
 super(SomeReader, self).__init__(rr)

 ops.NoGradient("SomeReader")
 ops.RegisterShape("SomeReader")(common_shapes.scalar_shape)
```

몇 가지 예제는 [tensorflow/python/ops/io\\_ops.py](#) .에서 확인할 수 있습니다.

## 레코드 형식 오퍼레이션 작성

일반적으로 이것은 스칼라 문자열 레코드를 입력으로 받고, [오퍼레이션\(Op\)](#)을 추가하는 지침에 따르는 보통의 오퍼레이션(op)입니다. 선택적으로 스칼라 문자열 키를 입력하고, 적절하지 않은 형식의 데이터는 에러 메시지에 포함되어 리포팅 됩니다. 그런 방식으로 사용자들은 잘못된 데이터가 어디서 발생했는지 더 쉽게 추적할 수 있습니다.

디코딩 레코드에 대한 유용한 오퍼레이션(Ops)의 예:

- `tf.parse_single_example` (and `tf.parse_example`)
- `tf.decode_csv`
- `tf.decode_raw`

특정 레코드 형식을 디코딩하기 위해 여러 개의 오퍼레이션(Ops)을 사용하는 것은 유용합니다. 예를 들어, 여러분은 `tf.train.Example`에 `protocol buffer`를 통해 문자열로 저장 된 이미지를 가질 수 있습니다. 이미지의 형식에 따라, `tf.parse_single_example` 오퍼레이션(op)과 `tf.image.decode_jpeg`, `tf.image.decode_png`, 또는 `tf.decode_raw` 를 호출해서 대응하는 출력을 얻을 수 있을 겁니다. `tf.decode_raw` 의 출력을 가져가는 것이 일반적이고 `tf.slice` 와 `tf.reshape` 의 조각을 추출하는 데 사용합니다.

# GPU 사용하기

## 지원되는 디바이스

일반적인 시스템에는 여러 개의 계산 디바이스가 존재합니다. TensorFlow에서는 `CPU` 와 `GPU` 디바이스를 지원합니다. 디바이스는 다음과 같이 `string` 으로 표현됩니다.

- `"/cpu:0"` : 시스템의 CPU를 지정함
- `"/gpu:0"` : 시스템의 첫 번째 GPU를 지정함(있는 경우)
- `"/gpu:1"` : 시스템의 두 번째 GPU를 지정함(세 번째 이후도 `:2, :3, ...` 으로 지정 가능)

만약 사용된 TensorFlow 계산 과정이 CPU와 GPU를 모두 지원한다면, 해당 계산 과정은 GPU 디바이스에 우선적으로 배치됩니다. 예로, `matmul` 은 CPU와 GPU 커널 모두 가지고 있습니다. 따라서 `cpu:0` 과 `gpu:0` 디바이스가 있는 시스템에서는, `gpu:0` 이 `matmul` 을 실행하도록 선택될 것입니다.

## 디바이스 배치 로깅

연산과 텐서가 어떤 디바이스에 배치되었는지 알아보기 위해, 세션을 만들 때 `log_device_placement` 옵션을 `True` 로 설정할 수 있습니다.

```
그래프를 생성합니다.
a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
c = tf.matmul(a, b)
log_device_placement을 True로 설정하여 세션을 만듭니다.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
op를 실행합니다.
print sess.run(c)
```

다음과 같은 출력 결과를 얻을 수 있습니다:

```
Device mapping:
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: Tesla K40c, pci bus
id: 0000:05:00.0
b: /job:localhost/replica:0/task:0/gpu:0
a: /job:localhost/replica:0/task:0/gpu:0
MatMul: /job:localhost/replica:0/task:0/gpu:0
[[22. 28.]
 [49. 64.]]
```

## 수동으로 디바이스에 배치

만약 특정 작업을 자동으로 디바이스에 배치하지 않고, 원하는 디바이스에 배치하고 싶다면, `with tf.device` 를 사용할 수 있습니다. 이를 사용하여 만들어진 디바이스 컨텍스트 내에서 이루어지는 모든 연산 과정은 명시된 디바이스 내에서 일어납니다.

```
그래프를 생성합니다.
with tf.device('/cpu:0'):
 a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
 b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
 c = tf.matmul(a, b)
log_device_placement을 True로 설정하여 세션을 만듭니다.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
op를 실행합니다.
print sess.run(c)
```

다음과 같이 `a` 와 `b` 가 `cpu:0` 에 배치된 것을 확인할 수 있습니다.

```
Device mapping:
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: Tesla K40c, pci bus
id: 0000:05:00.0
b: /job:localhost/replica:0/task:0/cpu:0
a: /job:localhost/replica:0/task:0/cpu:0
MatMul: /job:localhost/replica:0/task:0/gpu:0
[[22. 28.]
 [49. 64.]]
```

## GPU 메모리 증가 허용하기

기본적으로, TensorFlow는 GPU를 쓸 때 거의 대부분의 GPU 메모리를 사용합니다. 이는 디바이스의 [메모리 단편화](#)를 방지해 메모리를 효율적으로 사용할 수 있기 때문입니다.

때때로 디바이스에서 사용 가능한 메모리의 일부분만 할당하거나, 실행 과정에서 메모리를 추가로 할당하는 것이 더 나을 수도 있습니다. TensorFlow는 이러한 과정을 위해 세션에 두 가지 설정 옵션을 제공합니다.

첫 번째는 `allow_growth` 옵션입니다. 이는 실행 과정에서 요구되는 만큼의 GPU 메모리만 할당하게 합니다. 처음에는 매우 작은 메모리만 할당합니다. 그리고 세션이 실행되면서 더 많은 GPU 메모리가 필요해지면, TensorFlow에서 필요한 메모리 영역을 증가시켜 추가로 할당하게 됩니다. 메모리 할당을 해제하면 메모리 단편화 현상이 발생할 수 있으므로 증가된 메모리는 해제되지 않습니다. 이 옵션을 사용하기 위해서, 다음과 같은 코드로 `ConfigProto`의 옵션을 설정할 수 있습니다.

```
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
session = tf.Session(config=config, ...)
```

두 번째 방법은 `per_process_gpu_memory_fraction` 옵션입니다. 이는 각각의 사용 가능한 GPU에 대해 GPU 메모리의 일정 부분만 할당하게 합니다. 예로, 다음과 같은 코드로 TensorFlow에서 각각의 GPU 메모리의 40%만 할당하도록 설정할 수 있습니다.

```
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.4
session = tf.Session(config=config, ...)
```

이 설정은 TensorFlow 프로세스에 의해 사용되는 GPU 메모리의 양에 상한이 필요할 때 유용합니다.

## 멀티 GPU 시스템에서 한 개의 GPU만 사용하기

만약 시스템에 하나 이상의 GPU 디바이스가 있다면, 기본적으로는 가장 ID가 작은 GPU가 자동으로 선택됩니다. 만약 다른 GPU에서 실행하고 싶다면, 다음과 같이 디바이스를 명시해 주어야 합니다.

```
그래프를 생성합니다.
with tf.device('/gpu:2'):
 a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
 b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
 c = tf.matmul(a, b)
log_device_placement을 True로 설정하여 세션을 만듭니다.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
op를 실행합니다.
print sess.run(c)
```

만약 명시된 디바이스가 존재하지 않는다면, 다음과 같이 `InvalidArgumentError` 오류가 나게 됩니다.

```
InvalidArgumentError: Invalid argument: Cannot assign a device to node 'b':
Could not satisfy explicit device specification '/gpu:2'
[[Node: b = Const[dtype=DT_FLOAT, value=Tensor<type: float shape: [3,2]
values: 1 2 3...>, _device="/gpu:2"]()]]
```

명시된 디바이스가 존재하지 않는 경우 실행 디바이스를 TensorFlow가 자동으로 존재하는 디바이스 중 선택하게 하려면, 세션을 만들 때 `allow_soft_placement` 옵션을 `True`로 설정하면 됩니다.

```
그래프를 생성합니다.
with tf.device('/gpu:2'):
 a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
 b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
 c = tf.matmul(a, b)
allow_soft_placement와 log_device_placement 옵션을
True로 설정하여 세션을 만듭니다.
sess = tf.Session(config=tf.ConfigProto(
 allow_soft_placement=True, log_device_placement=True))
op를 실행합니다.
print sess.run(c)
```

## 여러 개의 GPU 사용하기

TensorFlow를 멀티 GPU 시스템에서 사용하는 경우, 다음의 코드와 같이 각각의 GPU에 모델을 분산시켜 구성할 수 있습니다.

```
그래프를 생성합니다.
c = []
for d in ['/gpu:2', '/gpu:3']:
 with tf.device(d):
 a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3])
 b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2])
 c.append(tf.matmul(a, b))
with tf.device('/cpu:0'):
 sum = tf.add_n(c)
log_device_placement을 True로 설정하여 세션을 만듭니다.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
op를 실행합니다.
print sess.run(sum)
```

다음의 결과를 얻을 수 있습니다.

```
Device mapping:
/job:localhost/replica:0/task:0/gpu:0 -> device: 0, name: Tesla K20m, pci bus
id: 0000:02:00.0
/job:localhost/replica:0/task:0/gpu:1 -> device: 1, name: Tesla K20m, pci bus
id: 0000:03:00.0
/job:localhost/replica:0/task:0/gpu:2 -> device: 2, name: Tesla K20m, pci bus
id: 0000:83:00.0
/job:localhost/replica:0/task:0/gpu:3 -> device: 3, name: Tesla K20m, pci bus
id: 0000:84:00.0
Const_3: /job:localhost/replica:0/task:0/gpu:3
Const_2: /job:localhost/replica:0/task:0/gpu:3
MatMul_1: /job:localhost/replica:0/task:0/gpu:3
Const_1: /job:localhost/replica:0/task:0/gpu:2
Const: /job:localhost/replica:0/task:0/gpu:2
MatMul: /job:localhost/replica:0/task:0/gpu:2
AddN: /job:localhost/replica:0/task:0/cpu:0
[[44. 56.]
 [98. 128.]]
```

여러 개의 GPU를 사용하여 모델을 훈련하는 과정의 좋은 예시로 [cifar10 튜토리얼](#)이 있습니다.

# 변수 공유

(v1.0)

[Variables HowTo](#)에 설명된 방법으로 단일 변수를 생성, 초기화, 저장 및 불러오기를 할 수 있습니다. 하지만 복잡한 모델을 구축할 때는 가끔 큰 변수 세트를 공유할 필요가 있고 한 곳에서 모든 변수의 초기화를 해야 할 수도 있습니다. 이번 튜토리얼은 `tf.variable_scope()` 와 `tf.get_variable()` 를 사용해서 이것을 어떻게 할 수 있는지 보여줍니다.

## 문제점

우리의 [Convolutional Neural Networks Tutorial](#) 모델과 유사하지만 2개의 콘볼루션(예제의 간단함을 위해)만 사용하는 이미지 필터에 대한 간단한 모델을 만든다고 가정하십시오. [Variables HowTo](#)의 설명대로 `tf.Variable` 을 바르게 사용했다면 여러분의 모델은 아래와 같을 겁니다.

```
def my_image_filter(input_images):
 conv1_weights = tf.Variable(tf.random_normal([5, 5, 32, 32]),
 name="conv1_weights")
 conv1_biases = tf.Variable(tf.zeros([32]), name="conv1_biases")
 conv1 = tf.nn.conv2d(input_images, conv1_weights,
 strides=[1, 1, 1, 1], padding='SAME')
 relu1 = tf.nn.relu(conv1 + conv1_biases)

 conv2_weights = tf.Variable(tf.random_normal([5, 5, 32, 32]),
 name="conv2_weights")
 conv2_biases = tf.Variable(tf.zeros([32]), name="conv2_biases")
 conv2 = tf.nn.conv2d(relu1, conv2_weights,
 strides=[1, 1, 1, 1], padding='SAME')
 return tf.nn.relu(conv2 + conv2_biases)
```

여러분이 쉽게 상상할 수 있듯이, 모델은 이것보다 훨씬 더 복잡하며, 여기에도 이미 4개의 다른 변수가 있습니다: `conv1_weights` , `conv1_biases` , `conv2_weights` , 그리고 `conv2_biases` .

문제는 이 모델을 다시 사용하고자 할 때 발생합니다. 2개의 다른 이미지, `image1` 과 `image2` 를 여러분의 이미지 필터에 적용하기를 원한다고 가정하십시오. 여러분은 같은 파라미터로 같은 필터에서 처리된 이미지가 필요합니다. `my_image_filter()` 를 두 번 호출할 수 있지만, 이것은 두 세트의 변수를 생성합니다:

```
First call creates one set of variables.
result1 = my_image_filter(image1)
Another set is created in the second call.
result2 = my_image_filter(image2)
```

변수를 공유하는 일반적인 방법은 별도의 코드로 작성하여 이를 사용하는 함수에 전달하는 것입니다. `dictionary`를 사용하는 예를 들어 봅시다:

```
variables_dict = {
 "conv1_weights": tf.Variable(tf.random_normal([5, 5, 32, 32])),
 name="conv1_weights")
 "conv1_biases": tf.Variable(tf.zeros([32]), name="conv1_biases")
 ... etc. ...
}

def my_image_filter(input_images, variables_dict):
 conv1 = tf.nn.conv2d(input_images, variables_dict["conv1_weights"],
 strides=[1, 1, 1, 1], padding='SAME')
 relu1 = tf.nn.relu(conv1 + variables_dict["conv1_biases"])

 conv2 = tf.nn.conv2d(relu1, variables_dict["conv2_weights"],
 strides=[1, 1, 1, 1], padding='SAME')
 return tf.nn.relu(conv2 + variables_dict["conv2_biases"])

The 2 calls to my_image_filter() now use the same variables
result1 = my_image_filter(image1, variables_dict)
result2 = my_image_filter(image2, variables_dict)
```

위와 같은 변수 생성은 편리하지만, 코드 밖에서 캡슐화가 중단됩니다.

- 그래프를 빌드하는 코드는 생성할 변수의 이름, 타입, 그리고 모양(`shape`)을 반드시 기록해야 합니다.
- 코드가 변경되면 호출자(`callers`)는 어느 정도 다른 변수를 생성해야 할지도 모릅니다.

문제를 해결하는 한 가지 방법은 모델을 생성할 때 필요한 변수를 관리하는 클래스(`classes`)를 사용하는 것입니다. TensorFlow는 클래스를 포함하지 않는 더 가벼운 솔루션을 위해 그래프를 구성하는 동안 명명된 변수를 쉽게 공유할 수 있는 *Variable Scope* 메커니즘을 제공합니다.

## 변수 범위(**Variable scope**) 예제

TensorFlow의 Variable Scope 메커니즘은 두 개의 메인 함수로 되어 있습니다.

- `tf.get_variable(<name>, <shape>, <initializer>)` : 입력된 이름의 변수를 생성하거나 반환합니다.
- `tf.variable_scope(<scope_name>)` : Manages namespaces for names passed to `tf.get_variable()`. `tf.get_variable()`에 전달 된 이름의 네임스페이스를 관리합니다.

`tf.get_variable()` 함수는 `tf.Variable` 을 직접호출 대신 변수를 가져오거나 생성하는 데 사용합니다. `tf.Variable` 처럼 직접 값을 전달하는 대신 *initializer*를 사용합니다. *initializer*는 모양(`shape`)을 가져와서 텐서를 제공하는 함수입니다. 여기 TensorFlow에서 사용 가능한 몇 개의 *initializer*가 있습니다.

- `tf.constant_initializer(value)` 제공된 값으로 모든 것을 초기화합니다,
- `tf.random_uniform_initializer(a, b)`  $[a, b]$ 를 균일하게 초기화 합니다,
- `tf.random_normal_initializer(mean, stddev)` 주어진 평균 및 표준 편차로 정규 분포에서 초기화합니다.

`tf.get_variable()` 이 앞에서 논의한 문제를 어떻게 해결하는지 보시려면, 하나의 convolution을 생성한 코드를 `conv_relu`라는 별개의 함수로 리펙토링합시다:

```
def conv_relu(input, kernel_shape, bias_shape):
 # Create variable named "weights".
 weights = tf.get_variable("weights", kernel_shape,
 initializer=tf.random_normal_initializer())
 # Create variable named "biases".
 biases = tf.get_variable("biases", bias_shape,
 initializer=tf.constant_initializer(0.0))
 conv = tf.nn.conv2d(input, weights,
 strides=[1, 1, 1, 1], padding='SAME')
 return tf.nn.relu(conv + biases)
```

이 함수는 짧은 이름 "weights" 와 "biases" 를 사용합니다. 우리는 그것을 `conv1` 과 `conv2` 둘다에서 사용하기를 원하지만, 변수들은 다른 이름을 가질 필요가 있습니다. 이곳은 `tf.variable_scope()` 가 동작하는 곳입니다 : 변수에 대한 네임 스페이스를 푸시(pushes)합니다.

```
def my_image_filter(input_images):
 with tf.variable_scope("conv1"):
 # Variables created here will be named "conv1/weights", "conv1/biases".
 relu1 = conv_relu(input_images, [5, 5, 32, 32], [32])
 with tf.variable_scope("conv2"):
 # Variables created here will be named "conv2/weights", "conv2/biases".
 return conv_relu(relu1, [5, 5, 32, 32], [32])
```

이제, 우리가 `my_image_filter()` 를 두 번 호출할 때 벌어지는 일을 봅시다.

```
result1 = my_image_filter(image1)
result2 = my_image_filter(image2)
Raises ValueError(... conv1/weights already exists ...)
```

보다시피, `tf.get_variable()` 은 이미 존재하는 변수가 우연히 공유된 것이 아닌지 확인합니다. 만약 공유하기를 원하면, 여러분은 다음과 같이 `reuse_variables()` 를 설정해야 합니다.

```
with tf.variable_scope("image_filters") as scope:
 result1 = my_image_filter(image1)
 scope.reuse_variables()
 result2 = my_image_filter(image2)
```

이것은 가볍고 안전하게 변수를 공유하는 좋은 방법입니다.

## 변수 범위는 어떻게 동작합니까?

### `tf.get_variable()` 이해하기

변수 범위를 이해하기 위해서는 첫 번째로 `tf.get_variable()` 이 어떻게 동작하는지 완전히 이해하는 것이 필요합니다. 다음은 `tf.get_variable()` 가 일반적으로 호출되는 방법입니다.

```
v = tf.get_variable(name, shape, dtype, initializer)
```

호출되는 범위에 따라 두 가지 중 하나를 호출합니다. 다음은 두 가지 옵션입니다.

- **Case 1:** 범위는 `tf.get_variable_scope().reuse == False`에서 증명된 것처럼 새로운 변수를 생성하기 위해 설정됩니다.

이 경우, `v` 는 제공된 모양(`shape`)과 데이터 타입을 가지는 새롭게 만들어진 `tf.Variable` 이 됩니다. 생성된 변수의 전체 이름은 현재 변수 범위 이름 + 제공된 `name` 으로 설정되고, 이 전체 이름을 가진 변수가 아직 존재하지 않는지 보장하기 위한 검사를 하게 됩니다. 이 전체 이름이 변수로 사용 중이라면, 함수는 `ValueError` 를 발생시킵니다. 만약 새로운 변수를 생성하면, 변수는 `initializer(shape)` 값으로 초기화될 것입니다. 예를 들어:

```
with tf.variable_scope("foo"):
 v = tf.get_variable("v", [1])
assert v.name == "foo/v:0"
```

- **Case 2:** `tf.get_variable_scope().reuse == True`에서 증명된 것처럼 변수를 재사용하기 위한 범위가 설정됩니다.

이 경우, 호출은 현재 변수 범위 이름 + 제공된 `name` 과 같은 이름으로 이미 존재하는 변수를 검색 합니다. 만약 변수가 없으면 `ValueError` 가 발생할 겁니다. 변수가 발견된다면 반환됩니다. 예를 들어:

```
with tf.variable_scope("foo"):
 v = tf.get_variable("v", [1])
with tf.variable_scope("foo", reuse=True):
 v1 = tf.get_variable("v", [1])
assert v1 == v
```

### `tf.variable_scope()` 의 기본

`tf.get_variable()` 가 어떻게 동작하는지 알면 변수 범위를 쉽게 이해하고 만들 수 있습니다. 변수 범위의 주요 기능은 변수 이름에 대한 접두사로 사용되는 이름을 들고 있는 것이고, 위에서 설명한 두 가지 경우를 구별하기 위한 재사용-플래그입니다. 중첩 변수 범위는 딕터리가 동작하는 방식과 유사한 방법으로 그것들의 이름을 추가합니다.

```
with tf.variable_scope("foo"):
 with tf.variable_scope("bar"):
 v = tf.get_variable("v", [1])
assert v.name == "foo/bar/v:0"
```

현재 변수 범위는 `tf.get_variable_scope()` 를 사용해서 회수할 수 있으며, 현재 변수 범위의 `reuse` 플래그는 `tf.get_variable_scope().reuse_variables()` 를 호출해서 `True` 로 설정할 수 있습니다.

```
with tf.variable_scope("foo"):
 v = tf.get_variable("v", [1])
 tf.get_variable_scope().reuse_variables()
 v1 = tf.get_variable("v", [1])
assert v1 == v
```

여러분은 `reuse` 플래그를 `False` 로 설정할 수는 없습니다. 이유는 모델을 생성하는 함수를 구성하는 것을 허용하기 위해서입니다. 이전처럼 `my_image_filter(inputs)` 함수를 작성한다고 상상해 보십시오. 변수 범위에서 `reuse=True` 와 함께 함수를 호출하는 누군가는 모든 내부 변수가 재사용될 것으로 예상합니다. 함수 내부에서 `reuse=False` 를 강제로 허용하면 이 계약이 깨지게 되고, 이런 방법으로 파라미터를 공유하는 것을 어렵게 만듭니다.

`reuse` 를 명시적으로 `False` 로 설정할 수는 없지만, 재사용 가능한 변수 범위를 입력한 다음 종료하고 재사용하지 않는 변수 범위로 돌아갈 수 있습니다. 변수 범위를 열 때 `reuse = True` 파라미터를 사용하여 이 작업을 수행할 수 있습니다. 또한, 위와 같은 이유로 `reuse` 파라미터가 상속된다는 점에 유의하십시오. 따라서 재사용 가능한 변수 범위를 열면 모든 하위 범위(sub-scopes)도 재사용됩니다.

```

with tf.variable_scope("root"):
 # At start, the scope is not reusing.
 assert tf.get_variable_scope().reuse == False
with tf.variable_scope("foo"):
 # Opened a sub-scope, still not reusing.
 assert tf.get_variable_scope().reuse == False
with tf.variable_scope("foo", reuse=True):
 # Explicitly opened a reusing scope.
 assert tf.get_variable_scope().reuse == True
 with tf.variable_scope("bar"):
 # Now sub-scope inherits the reuse flag.
 assert tf.get_variable_scope().reuse == True
Exited the reusing scope, back to a non-reusing one.
assert tf.get_variable_scope().reuse == False

```

## 변수 범위 캡처(Capturing)

위에 제시된 모든 예제에서, 우리는 변수들의 이름이 일치했기 때문에 파라미터를 공유할 수 있었는데, 그건, 정확히 같은 문자열로 재사용 변수 범위를 열었기 때문입니다. 더 복잡한 경우에는 이를 올바르게 가져와서 전달하는 것보다 VariableScope 객체를 통과시키는 것이 유용할 수도 있습니다. 이를 위해, 변수 범위는 새로운 변수 범위를 열 때 이름 대신 캡처해서 사용할 수 있습니다.

```

with tf.variable_scope("foo") as foo_scope:
 v = tf.get_variable("v", [1])
with tf.variable_scope(foo_scope):
 w = tf.get_variable("w", [1])
with tf.variable_scope(foo_scope, reuse=True):
 v1 = tf.get_variable("v", [1])
 w1 = tf.get_variable("w", [1])
assert v1 == v
assert w1 == w

```

변수 범위를 열 때 이전에 존재하는 범위를 사용하면 현재 변수 범위 접두사에서 완전히 다른 곳으로 넘어갑니다. 이것은 우리가 하는 곳과 완전히 독립적입니다.

```

with tf.variable_scope("foo") as foo_scope:
 assert foo_scope.name == "foo"
with tf.variable_scope("bar"):
 with tf.variable_scope("baz") as other_scope:
 assert other_scope.name == "bar/baz"
 with tf.variable_scope(foo_scope) as foo_scope2:
 assert foo_scope2.name == "foo" # Not changed.

```

## 변수 범위의 Initializers

`tf.get_variable()` 을 사용하면 변수를 생성하거나 재사용하는 함수를 작성하는 것이 허용되며 외부에서 투명하게 호출할 수 있습니다. 하지만 생성된 변수의 `initializer`를 변경하려면 어떻게 해야 할까요? 변수를 만드는 모든 함수에 추가인수를 전달하는 것이 필요할까요? 모든 변수의 기본 `initializer`를 모든 함수의 위의 한 곳에서 설정하기를 원할 때 일반적인 때는 어렵습니까? 이런 경우를 돋기 위해, 변수 범위는 기본 `initializer`를 들고 있을 수 있습니다. 하위 범위(sub-scopes)에 의해 상속받고 각각 `tf.get_variable()` 호출에 전달됩니다. 그러나 다른 `initializer`를 명시적으로 지정하면 오버라이드(overridden) 될 겁니다.

```
with tf.variable_scope("foo", initializer=tf.constant_initializer(0.4)):
 v = tf.get_variable("v", [1])
 assert v.eval() == 0.4 # Default initializer as set above.
 w = tf.get_variable("w", [1], initializer=tf.constant_initializer(0.3)):
 assert w.eval() == 0.3 # Specific initializer overrides the default.
 with tf.variable_scope("bar"):
 v = tf.get_variable("v", [1])
 assert v.eval() == 0.4 # Inherited default initializer.
 with tf.variable_scope("baz", initializer=tf.constant_initializer(0.2)):
 v = tf.get_variable("v", [1])
 assert v.eval() == 0.2 # Changed default initializer.
```

## tf.variable\_scope() 오퍼레이션(ops)의 이름

우리는 어떻게 `tf.variable_scope` 가 변수들의 이름을 관리하는지 논의했습니다. 하지만 그것이 범위 내에서 다른 오퍼레이션(ops)의 이름에 영향을 미치나요? 변수 범위 내에서 생성된 오퍼레이션(ops)은 이름이 공유되는 것이 정상적입니다. 이런 이유로, 우리가 `with tf.variable_scope("name")` 를 사용할 때 이것은 무조건 `tf.name_scope("name")` 를 엽니다. 예를 들어:

```
with tf.variable_scope("foo"):
 x = 1.0 + tf.get_variable("v", [1])
assert x.op.name == "foo/add"
```

이름 변수는 변수 범위를 추가로 열 수 있으며, 그것들은 변수가 아닌 오직 오퍼레이션(ops)의 이름에만 영향을 미칩니다.

```
with tf.variable_scope("foo"):
 with tf.name_scope("bar"):
 v = tf.get_variable("v", [1])
 x = 1.0 + v
 assert v.name == "foo/v:0"
 assert x.op.name == "foo/bar/add"
```

문자열 대신 캡처된 객체를 사용해서 변수 범위를 열 때, 우리는 오퍼레이션(ops)의 현재 이름 범위를 바꾸지 않습니다.

## 사용 예시

다음은 변수 범위를 사용하는 몇 가지 파일에 대한 포인터(pointers)입니다. [TensorFlow models repo](#)에서 모두 찾을수있다. 특히, 변수 범위는 RNN(recurrent neural networks)과 seq2seq(sequence-to-sequence) 모델에서 많이 사용됩니다.

| 파일                                | 내용                      |
|-----------------------------------|-------------------------|
| models/tutorials/image/cifar10.py | 이미지 내에서 객체를 찾는 모델.      |
| models/tutorials/rnn/rnn_cell.py  | RNN을 위한 Cell 함수.        |
| models/tutorials/rnn/seq2seq.py   | seq2seq 모델을 구축하기 위한 함수. |

# 툴 개발자의 TensorFlow 모델 파일에 대한 가이드

(v1.0)

대부분의 사용자는 TensorFlow가 디스크에 데이터를 어떻게 저장하는지에 대한 내부적인 세부 사항들에 대해 신경쓸 필요가 없지만, 만약 당신이 툴 개발자라면 그럴 필요가 있습니다. 예를 들면, 당신은 모델을 분석하고 싶다거나 TensorFlow와 다른 포맷 사이에서 상호 변환을 하고싶을 수 있습니다. 이 가이드는 각종 툴들을 좀 더 쉽게 개발할 수 있도록 모델 데이터를 가진 메인 파일들을 어떻게 다룰 수 있는지에 대한 상세 내용들을 설명합니다.

## 프로토콜 버퍼 (Protocol Buffers)

TensorFlow의 모든 파일 포맷은 [Protocol Buffers](#)에 기반하므로, 그들이 작동하는 방법에 대해 알아가는 것이 좋습니다. 요약하면 텍스트 파일에 데이터 구조를 정의하면, 프로토콜 버퍼 툴이 C, Python, 그리고 익숙한 방법으로 데이터를 로드, 저장 그리고 접근할 수 있는 다른 언어로 클래스를 생성합니다. 우리는 종종 프로토콜 버퍼를 `protobufs`라 부르며, 이 가이드에서는 이 컨벤션을 사용할 것입니다.

## GraphDef

TensorFlow에서 계산의 기본은 `Graph` 객체입니다. 이는 각각이 연산을 나타내고, 입력과 출력으로써 서로 연결되어 있는 노드들의 네트워크를 가지고 있습니다. `Graph` 객체를 생성한 후엔 `GraphDef` 객체를 반환하는 `as_graph_def()` 를 호출함으로써 이를 저장할 수 있습니다.

`GraphDef` 클래스는 [tensorflow/core/framework/graph.proto](#)에 정의된 프로토콜 버퍼 라이브러리에 의해 생성되는 객체입니다. 프로토콜 버퍼 툴은 이 텍스트 파일을 파싱하고 그래프 정의를 로딩, 저장 및 조작하는 코드를 생성합니다. 모델을 나타내는 단일 TensorFlow 파일이 있다면, 이는 프로토콜 버퍼 코드에 의해 저장된 `GraphDef` 객체들중 하나를 직렬화한 버전을 포함할 가능성이 높습니다.

이 생성된 코드는 디스크로부터 `GraphDef` 파일들을 저장하고 로드하는데 쓰입니다. 이 코드는 실제로 모델을 다음과 같이 로드 합니다.

```
graph_def = graph_pb2.GraphDef()
```

이 라인은 `graph.proto`에 텍스트로 정의되며 생성된 `GraphDef` 클래스의 빈 객체를 생성합니다. 이는 우리가 가진 파일의 데이터를 채울 객체입니다.

```
with open(FLAGS.graph, "rb") as f:
```

여기서 스크립트에 전달한 경로에 대한 파일 핸들을 얻습니다.

```
if FLAGS.input_binary:
 graph_def.ParseFromString(f.read())
else:
 text_format.Merge(f.read(), graph_def)
```

## 텍스트 또는 바이너리?

프로토콜 버퍼가 저장할 수 있는 포맷은 두 가지 있습니다. 텍스트 포맷 (Text Format)은 사람이 읽을 수 있는 형태이며, 디버깅과 편집이 편리하지만, 가중치와 같은 수치 데이터가 저장될 경우 커질 수 있습니다. 이에 대한 간단한 예제는 [graph\\_run\\_run2.pbtxt](#)에서 볼 수 있습니다.

바이너리 포맷 (Binary Format) 파일은 비록 읽기는 어렵지만, 같은 내용의 텍스트 포맷보다 훨씬 작은 크기를 갖습니다. 우리는 적절한 함수를 호출할 수 있도록 사용자에게 입력 파일이 바이너리인지 텍스트인지 구분할 수 있는 플래그를 제공하도록 요청합니다. 큰 바이너리 파일에 대한 예제는 [inception\\_dec\\_2015.ziparchive](#)의 `tensorflow_inception_graph.pb`에서 볼 수 있습니다.

API 자체는 조금 혼란스러울 수 있습니다 - 텍스트 파일을 로드 할 때에는 `text_format` 모듈에 있는 유ти리티 함수를 사용하는 반면, 바이너리 호출은 실제로 `ParseFromString()`를 사용합니다.

## 노드 (Nodes)

`graph_def` 변수로 파일을 로드했다면, 이제 파일 데이터에 접근할 수 있습니다. 대부분의 실용적인 목적을 위한, 중요한 부분은 노드의 리스트를 노드 멤버에 저장하는 것입니다. 여기에 노드를 순회하는 코드가 있습니다.

```
for node in graph_def.node
```

각 노드는 `NodeDef` 객체이며, 이 또한 `graph.proto`에 정의되어 있습니다. 이들은 TensorFlow 그래프의 기본적인 빌딩 블록이고 각각은 입력 커넥션과 함께 하나의 연산을 정의합니다. 아래에 `NodeDef`의 멤버들이 있으며, 그들의 의미도 설명합니다.

### name

모든 노드는 그래프의 다른 어떤 노드들에 의해서도 사용되지 않는 유일한 식별자를 가져야 합니다. 만약 파이썬 API를 사용해 그래프를 생성하면서 "MatMul"과 같이 연산의 명칭과 "5"와 같이 단조 증가 숫자와 연결하는 등의 식별자를 지정하지 않으면, 이는 대신 선택해줄 것입니다. 임의의 식별자를 대신 선택해줄 것입니다. 명칭은 노드끼리의 연결을 정의할 때와 그래프를 실행할 때 전체 그래프를 위한 입력과 출력을 설정할 때에 사용됩니다.

## op

이는 실행될 연산을 정의하는데, 예를 들면 `Add`, `MatMul`, 또는 `"Conv2D"` 가 있습니다. 그래프가 실행될 때, 이 연산 명칭은 구현을 찾기위해 레지스트리에서 조회됩니다. 레지스트리는 [tensorflow/core/ops/nn\\_ops.cc](#)에 있는것들처럼, `REGISTER_OP()` 매크로를 호출함으로써 채워집니다.

## input

문자열의 리스트, 문자열들은 각각 다른 노드의 명칭이며, 선택적으로 콜론(:)과 출력 포트 번호가 따라붙습니다. 예를 들면, 두 개의 입력을 갖는 노드는 `["some_node_name", "another_node_name"]` (이는 `["some_node_name:0", "another_node_name:0"]` 와 동일합니다.) 형태의 리스트를 가질 것이며, 노드의 첫번째 입력을 `"some_node_name"` 의 명칭을 갖는 노드의 첫번째 출력으로, 그리고 `"another_node_name"` 의 첫번째 출력으로 두번째 입력을 정의할 것입니다.

## device

이는 분산된 환경 또는 강제로 연산을 CPU나 GPU 위에 놓고 싶을 때 노드가 어디에서 실행될 것 인지를 정의하기때문에, 대부분의 상황에선 이를 무시할 수 있습니다.

## attr

이는 노드의 모든 속성들을 가지고 있는 key/value 저장소입니다. 이들은 컨볼루션에 대한 필터 사이즈나 상수 연산의 값처럼 런타임때 변경할 수 없는 노드의 영속적인 프로퍼티입니다. 문자열부 터, 정수, 텐서값의 배열까지 속성값의 타입들이 매우 많을 수 있기 때문에, [tensorflow/core/framework/attr\\_value.proto](#)에 그것들을 가지고 있는 데이터 구조를 정의하는 별도의 프로토콜 버퍼 파일이 있습니다.

각 속성은 유일한 명칭을 가지고 있으며, 예상되는 속성들은 연산이 정의될 때 리스트됩니다. 만약 한 속성이 노드엔 존재하지 않지만 연산 정의에서 기본값을 가지고 있을 때, 그래프 생성시 그 기본값이 사용됩니다.

파이썬에서 `node.name`, `node.op` 등을 호출함으로써 이 모든 멤버들에 접근할 수 있습니다. `GraphDef` 에 저장된 노드의 리스트는 모델 아키텍처의 전체 정의입니다.

## 프리징 (Freezing)

이에 대한 하나 혼란스러운 부분은 가중치는 보통 트레이닝 중 파일 포맷에 저장되지 않는다는 것입니다. 대신에, 그들은 분리된 체크포인트 파일에서 유지되며, 그래프에는 가중치들이 초기화될 때 최신값을 로드하는 `variable` 연산이 있습니다. 종종 프로덕션에 배포할 때 나뉘어진 파일들을 갖고 있다는건 매우 불편하기 때문에, 그래프 정의와 체크포인트셋을 받아 하나의 파일로 묶어주는 `freeze_graph.py` 스크립트가 있습니다.

프리징이 하는 일은 `GraphDef` 을 로드하고, 최신 체크포인트 파일로부터 모든 변수들의 값을 받아온 후, 각 `variable` 연산을 각 속성에 저장된 가중치들의 수치 데이터를 가진 `Const` 로 교체합니다. 그 다음에 이는 정방향 추론에 쓰이지 않는 관계 없는 노드들은 모두 제거하고, 하나의 출력 파일에 결과 `GraphDef` 를 저장합니다.

## 가중치 포맷 (Weight Formats)

만약 신경망에 대한 TensorFlow 모델을 다루고 있다면, 가장 일반적인 문제 중 하나는 가중치값을 추출 및 해석하는 것입니다. 그들을 저장하는 일반적인 방법은, 예시로 `freeze_graph` 스크립트를 통해 생성된 그래프에서, 가중치 `Tensors` 를 포함하고 있는 `Const` 연산으로 저장하는 것입니다. 이들은 [tensorflow/core/framework/tensor.proto](#)에 정의되어 있고, 값 뿐만 아니라 데이터의 사이즈와 타입에 대한 정보도 포함하고 있습니다. 파일에서 `some_node_def.attr['value'].tensor` 와 같은 것을 호출함으로써 `Const` 연산을 나타내는 `NodeDef`로부터 `TensorProto` 객체를 얻을 수 있습니다.

이는 가중치 데이터를 나타내는 객체를 줄 것입니다. 데이터 자체는 객체의 타입에 의해 가리켜지는 형태로 `_val` 접미어와 함께 리스트의 하나로 저장될 것입니다, 예로 32 비트 실수 데이터 타입은 `float_val`입니다.

서로 다른 프레임워크간 변환 작업시 컨볼루션 가중치 값의 순서는 종종 다루기가 까다롭습니다. TensorFlow에서, `conv2D` 연산을 위한 필터 가중치들은 두 번째 입력에 저장되며, `[filter_height, filter_width, input_depth, output_depth]` 의 순서가 될 것으로 예상됩니다. 여기서 1씩 증가하는 `filter_count`는 메모리에서 인접한 값으로 이동함을 의미합니다.

이 개요가 TensorFlow 모델 파일에서 무슨 일들이 일어나고 있는지에 대한 더 나은 아이디어를 제공하기 바라며, 당신이 그들을 다루어야 할 경우 도움이 될 것입니다.

# How to Retrain Inception's Final Layer for New Categories

Modern object recognition models have millions of parameters and can take weeks to fully train. Transfer learning is a technique that shortcuts a lot of this work by taking a fully-trained model for a set of categories like ImageNet, and retrains from the existing weights for new classes. In this example we'll be retraining the final layer from scratch, while leaving all the others untouched. For more information on the approach you can see [this paper on Decaf](#).

Though it's not as good as a full training run, this is surprisingly effective for many applications, and can be run in as little as thirty minutes on a laptop, without requiring a GPU. This tutorial will show you how to run the example script on your own images, and will explain some of the options you have to help control the training process.

[TOC]

## Training on Flowers

 Image by Kelly Sikkema

Before you start any training, you'll need a set of images to teach the network about the new classes you want to recognize. There's a later section that explains how to prepare your own images, but to make it easy we've created an archive of creative-commons licensed flower photos to use initially. To get the set of flower photos, run these commands:

```
cd ~
curl -O http://download.tensorflow.org/example_images/flower_photos.tgz
tar xzf flower_photos.tgz
```

Once you have the images, you can build the retrainer like this, from the root of your TensorFlow source directory:

```
bazel build tensorflow/examples/image_retraining:retrain
```

If you have a machine which supports [the AVX instruction set](#) (common in x86 CPUs produced in the last few years) you can improve the running speed of the retraining by building for that architecture, like this:

```
bazel build -c opt --copt=-mavx tensorflow/examples/image_retraining:retrain
```

The retrainer can then be run like this:

```
bazel-bin/tensorflow/examples/image_retraining/retrain --image_dir ~/flower_photos
```

This script loads the pre-trained Inception v3 model, removes the old top layer, and trains a new one on the flower photos you've downloaded. None of the flower species were in the original ImageNet classes the full network was trained on. The magic of transfer learning is that lower layers that have been trained to distinguish between some objects can be reused for many recognition tasks without any alteration.

## Bottlenecks

The script can take thirty minutes or more to complete, depending on the speed of your machine. The first phase analyzes all the images on disk and calculates the bottleneck values for each of them. 'Bottleneck' is an informal term we often use for the layer just before the final output layer that actually does the classification. This penultimate layer has been trained to output a set of values that's good enough for the classifier to use to distinguish between all the classes it's been asked to recognize. That means it has to be a meaningful and compact summary of the images, since it has to contain enough information for the classifier to make a good choice in a very small set of values. The reason our final layer retraining can work on new classes is that it turns out the kind of information needed to distinguish between all the 1,000 classes in ImageNet is often also useful to distinguish between new kinds of objects.

Because every image is reused multiple times during training and calculating each bottleneck takes a significant amount of time, it speeds things up to cache these bottleneck values on disk so they don't have to be repeatedly recalculated. By default they're stored in the `/tmp/bottleneck` directory, and if you rerun the script they'll be reused so you don't have to wait for this part again.

## Training

Once the bottlenecks are complete, the actual training of the top layer of the network begins. You'll see a series of step outputs, each one showing training accuracy, validation accuracy, and the cross entropy. The training accuracy shows what percent of the images used in the current training batch were labeled with the correct class. The validation accuracy is the

precision on a randomly-selected group of images from a different set. The key difference is that the training accuracy is based on images that the network has been able to learn from so the network can overfit to the noise in the training data. A true measure of the performance of the network is to measure its performance on a data set not contained in the training data -- this is measured by the validation accuracy. If the train accuracy is high but the validation accuracy remains low, that means the network is overfitting and memorizing particular features in the training images that aren't helpful more generally. Cross entropy is a loss function which gives a glimpse into how well the learning process is progressing. The training's objective is to make the loss as small as possible, so you can tell if the learning is working by keeping an eye on whether the loss keeps trending downwards, ignoring the short-term noise.

By default this script will run 4,000 training steps. Each step chooses ten images at random from the training set, finds their bottlenecks from the cache, and feeds them into the final layer to get predictions. Those predictions are then compared against the actual labels to update the final layer's weights through the back-propagation process. As the process continues you should see the reported accuracy improve, and after all the steps are done, a final test accuracy evaluation is run on a set of images kept separate from the training and validation pictures. This test evaluation is the best estimate of how the trained model will perform on the classification task. You should see an accuracy value of between 90% and 95%, though the exact value will vary from run to run since there's randomness in the training process. This number is based on the percent of the images in the test set that are given the correct label after the model is fully trained.

## Using the Retrained Model

The script will write out a version of the Inception v3 network with a final layer retrained to your categories to `/tmp/output_graph.pb`, and a text file containing the labels to `/tmp/output_labels.txt`. These are both in a format that the [C++ and Python image classification examples](#) can read in, so you can start using your new model immediately. Since you've replaced the top layer, you will need to specify the new name in the script, for example with the flag `--output_layer=final_result` if you're using `label_image`.

Here's an example of how to build and run the `label_image` example with your retrained graphs:

```
bazel build tensorflow/examples/label_image:label_image && \
bazel-bin/tensorflow/examples/label_image/label_image \
--graph=/tmp/output_graph.pb --labels=/tmp/output_labels.txt \
--output_layer=final_result \
--image=$HOME/flower_photos/daisy/21652746_cc379e0eea_m.jpg
```

You should see a list of flower labels, in most cases with daisy on top (though each retrained model may be slightly different). You can replace the `--image` parameter with your own images to try those out, and use the C++ code as a template to integrate with your own applications.

If you'd like to use the retrained model in a Python program [this example from @eldor4do shows what you'll need to do](#).

## Training on Your Own Categories

If you've managed to get the script working on the flower example images, you can start looking at teaching it to recognize categories you care about instead. In theory all you'll need to do is point it at a set of sub-folders, each named after one of your categories and containing only images from that category. If you do that and pass the root folder of the subdirectories as the argument to `--image_dir`, the script should train just like it did for the flowers.

Here's what the folder structure of the flowers archive looks like, to give you an example of the kind of layout the script is looking for:



In practice it may take some work to get the accuracy you want. I'll try to guide you through some of the common problems you might encounter below.

## Creating a Set of Training Images

The first place to start is by looking at the images you've gathered, since the most common issues we see with training come from the data that's being fed in.

For training to work well, you should gather at least a hundred photos of each kind of object you want to recognize. The more you can gather, the better the accuracy of your trained model is likely to be. You also need to make sure that the photos are a good representation of what your application will actually encounter. For example, if you take all your photos indoors against a blank wall and your users are trying to recognize objects outdoors, you probably won't see good results when you deploy.

Another pitfall to avoid is that the learning process will pick up on anything that the labeled images have in common with each other, and if you're not careful that might be something that's not useful. For example if you photograph one kind of object in a blue room, and another in a green one, then the model will end up basing its prediction on the background color, not the features of the object you actually care about. To avoid this, try to take pictures

in as wide a variety of situations as you can, at different times, and with different devices. If you want to know more about this problem, you can read about the classic (and possibly apocryphal) [tank recognition problem] (<http://www.jefftk.com/p/detecting-tanks>).

You may also want to think about the categories you use. It might be worth splitting big categories that cover a lot of different physical forms into smaller ones that are more visually distinct. For example instead of 'vehicle' you might use 'car', 'motorbike', and 'truck'. It's also worth thinking about whether you have a 'closed world' or an 'open world' problem. In a closed world, the only things you'll ever be asked to categorize are the classes of object you know about. This might apply to a plant recognition app where you know the user is likely to be taking a picture of a flower, so all you have to do is decide which species. By contrast a roaming robot might see all sorts of different things through its camera as it wanders around the world. In that case you'd want the classifier to report if it wasn't sure what it was seeing. This can be hard to do well, but often if you collect a large number of typical 'background' photos with no relevant objects in them, you can add them to an extra 'unknown' class in your image folders.

It's also worth checking to make sure that all of your images are labeled correctly. Often user-generated tags are unreliable for our purposes, for example using #daisy for pictures of a person named Daisy. If you go through your images and weed out any mistakes it can do wonders for your overall accuracy.

## Training Steps

If you're happy with your images, you can take a look at improving your results by altering the details of the learning process. The simplest one to try is `--how_many_training_steps`. This defaults to 4,000, but if you increase it to 8,000 it will train for twice as long. The rate of improvement in the accuracy slows the longer you train for, and at some point will stop altogether, but you can experiment to see when you hit that limit for your model.

## Distortions

A common way of improving the results of image training is by deforming, cropping, or brightening the training inputs in random ways. This has the advantage of expanding the effective size of the training data thanks to all the possible variations of the same images, and tends to help the network learn to cope with all the distortions that will occur in real-life uses of the classifier. The biggest disadvantage of enabling these distortions in our script is that the bottleneck caching is no longer useful, since input images are never reused exactly. This means the training process takes a lot longer, so I recommend trying this as a way of fine-tuning your model once you've got one that you're reasonably happy with.

You enable these distortions by passing `--random_crop`, `--random_scale` and `--random_brightness` to the script. These are all percentage values that control how much of each of the distortions is applied to each image. It's reasonable to start with values of 5 or 10 for each of them and then experiment to see which of them help with your application. `--flip_left_right` will randomly mirror half of the images horizontally, which makes sense as long as those inversions are likely to happen in your application. For example it wouldn't be a good idea if you were trying to recognize letters, since flipping them destroys their meaning.

## Hyper-parameters

There are several other parameters you can try adjusting to see if they help your results. The `--learning_rate` controls the magnitude of the updates to the final layer during training. Intuitively if this is smaller then the learning will take longer, but it can end up helping the overall precision. That's not always the case though, so you need to experiment carefully to see what works for your case. The `--train_batch_size` controls how many images are examined during one training step, and because the learning rate is applied per batch you'll need to reduce it if you have larger batches to get the same overall effect.

## Training, Validation, and Testing Sets

One of the things the script does under the hood when you point it at a folder of images is divide them up into three different sets. The largest is usually the training set, which are all the images fed into the network during training, with the results used to update the model's weights. You might wonder why we don't use all the images for training? A big potential problem when we're doing machine learning is that our model may just be memorizing irrelevant details of the training images to come up with the right answers. For example, you could imagine a network remembering a pattern in the background of each photo it was shown, and using that to match labels with objects. It could produce good results on all the images it's seen before during training, but then fail on new images because it's not learned general characteristics of the objects, just memorized unimportant details of the training images.

This problem is known as overfitting, and to avoid it we keep some of our data out of the training process, so that the model can't memorize them. We then use those images as a check to make sure that overfitting isn't occurring, since if we see good accuracy on them it's a good sign the network isn't overfitting. The usual split is to put 80% of the images into the main training set, keep 10% aside to run as validation frequently during training, and then have a final 10% that are used less often as a testing set to predict the real-world

performance of the classifier. These ratios can be controlled using the `--testing_percentage` and `--validation_percentage` flags. One subtle thing that the script does is it uses the filename of the image to determine which set it is put into. This is designed to ensure that images don't get moved between training and testing sets on different runs, since that could be a problem if images that had been used for training a model were subsequently used in a validation set. In general you should be able to leave these values at their defaults, since you won't usually find any advantage to training to adjusting them.

# 메타 그래프를 내보내고 가져오기

(v1.0)

`MetaGraph` 는 텐서플로우 `GraphDef` 뿐만 아니라 프로세스 경계를 교차할 때 그래프에서 연산을 실행하는 데 필요한 관련된 `metadata`도 포함합니다. 그것은 또한 그래프의 장기간 보관에도 사용될 수 있습니다. `MetaGraph`는 훈련을 계속하는 데 필요한 정보를 포함하고 있으며, 이전에 훈련된 그래프에 대한 평가를 수행하거나 실행하는 데 필요한 정보를 수록하고 있습니다.

전체 모델을 내보내고 가져오는 API는 `tf.train.Saver` class에 `export_meta_graph` 와 `import_meta_graph` 입니다.

## MetaGraph에 있는 것

MetaGraph에 수록된 정보는 `MetaGraphDef` 프로토콜 버퍼를 나타냅니다. 다음 필드가 포함됩니다.

- 버전과 기타 사용자 정보 같은 메타 정보에 대한 `MetaInfoDef`.
- 그래프를 묘사하기 위한 `GraphDef`.
- 세이버(saver)에 대한 `SaverDef`.
- `CollectionDef map`은 모델의 `Variables`, `QueueRunners`, etc와 같은 추가적인 요소를 더 자세히 설명합니다. Python 오브젝트를 `MetaGraphDef`로부터 직렬화하기 위해서, Python 클래스는 `to_proto()` 와 `from_proto()` 메소드를 실행하고, `register_proto_function` 를 사용해서 시스템에 등록합니다.

예를 들어,

```

def to_proto(self):
 """Converts a `Variable` to a `VariableDef` protocol buffer.

 Returns:
 A `VariableDef` protocol buffer.
 """
 var_def = variable_pb2.VariableDef()
 var_def.variable_name = self._variable.name
 var_def.initializer_name = self.initializer.name
 var_def.snapshot_name = self._snapshot.name
 if self._save_slice_info:
 var_def.save_slice_info_def.MergeFrom(self._save_slice_info.to_proto())
 return var_def

@staticmethod
def from_proto(variable_def):
 """Returns a `Variable` object created from `variable_def`."""
 return Variable(variable_def=variable_def)

ops.register_proto_function(ops.GraphKeys.VARIABLES,
 proto_type=variable_pb2.VariableDef,
 to_proto=Variable.to_proto,
 from_proto=Variable.from_proto)

```

## 전체 모델을 MetaGraph로 내보내기

실행 중인 모델을 MetaGraph로 내보내는 API는 `export_meta_graph()` 입니다.

```

def export_meta_graph(filename=None, collection_list=None, as_text=False):
 """Writes `MetaGraphDef` to save_path/filename.

 Args:
 filename: Optional meta_graph filename including the path.
 collection_list: List of string keys to collect.
 as_text: If `True`, writes the meta_graph as an ASCII proto.

 Returns:
 A `MetaGraphDef` proto.
 """

```

`collection` 은 사용자가 고유하게 식별하고 쉽게 검색할 수 있는 Python 객체를 포함할 수 있습니다. 이 객체들은 그래프 안에서 `train_op` 나 하이퍼 파라미터(hyper parameters), "learning rate"처럼 특별한 연산을 할 수 있습니다. 사용자는 내보내려는 컬렉션 목록을 지정할 수 있습니다. 만약 `collection_list` 가 지정되지 않으면, 모델 안에 모든 컬렉션이 내 보내어질 겁니다.

API는 직렬화된 프로토콜 버퍼를 반환합니다. 만약 `filename` 이 지정됐다면, 프로토콜 버퍼는 파일에 쓰여질 겁니다.

다음은 일반적인 사용 모델의 일부입니다.

- 기본 실행 그래프 내보내기:

```
Build the model
...
with tf.Session() as sess:
 # Use the model
 ...
Export the model to /tmp/my-model.meta.
meta_graph_def = tf.train.export_meta_graph(filename='/tmp/my-model.meta')
```

- 기본 실행 그래프와 컬렉션의 일부분만 내보냅니다.

```
meta_graph_def = tf.train.export_meta_graph(
 filename='/tmp/my-model.meta',
 collection_list=["input_tensor", "output_tensor"])
```

MetaGraph는 또한 `tf.train.Saver`에 `save()` API를 통해 자동으로 내보내기 됩니다.

## MetaGraph 가져오기

MetaGraph 파일을 그래프로 가져오기 위한 API는 `import_meta_graph()` 입니다.

다음은 일반적인 사용 모델의 일부입니다:

- 처음부터 모델을 구축하지 않고 가져와서 계속 훈련합니다.

```
...
Create a saver.
saver = tf.train.Saver(...variables...)
Remember the training_op we want to run by adding it to a collection.
tf.add_to_collection('train_op', train_op)
sess = tf.Session()
for step in xrange(1000000):
 sess.run(train_op)
 if step % 1000 == 0:
 # Saves checkpoint, which by default also exports a meta_graph
 # named 'my-model-global_step.meta'.
 saver.save(sess, 'my-model', global_step=step)
```

나중에 우리는 처음부터 모델을 구축하지 않고 저장된 meta\_graph로부터 계속 훈련할 수 있습니다.

```
with tf.Session() as sess:
 new_saver = tf.train.import_meta_graph('my-save-dir/my-model-10000.meta')
 new_saver.restore(sess, 'my-save-dir/my-model-10000')
 # tf.get_collection() returns a list. In this example we only want the
 # first one.
 train_op = tf.get_collection('train_op')[0]
 for step in xrange(1000000):
 sess.run(train_op)
```

- 그래프를 가져오고 확장하십시오.

예를 들어, 먼저 추론 그래프를 작성하여 메타 그래프로 내보낼 수 있습니다.

```

Creates an inference graph.

Hidden 1
images = tf.constant(1.2, tf.float32, shape=[100, 28])
with tf.name_scope("hidden1"):
 weights = tf.Variable(
 tf.truncated_normal([28, 128],
 stddev=1.0 / math.sqrt(float(28))),
 name="weights")
 biases = tf.Variable(tf.zeros([128]),
 name="biases")
 hidden1 = tf.nn.relu(tf.matmul(images, weights) + biases)

Hidden 2
with tf.name_scope("hidden2"):
 weights = tf.Variable(
 tf.truncated_normal([128, 32],
 stddev=1.0 / math.sqrt(float(128))),
 name="weights")
 biases = tf.Variable(tf.zeros([32]),
 name="biases")
 hidden2 = tf.nn.relu(tf.matmul(hidden1, weights) + biases)

Linear
with tf.name_scope("softmax_linear"):
 weights = tf.Variable(
 tf.truncated_normal([32, 10],
 stddev=1.0 / math.sqrt(float(32))),
 name="weights")
 biases = tf.Variable(tf.zeros([10]),
 name="biases")
 logits = tf.matmul(hidden2, weights) + biases
 tf.add_to_collection("logits", logits)

init_all_op = tf.initialize_all_variables()

with tf.Session() as sess:
 # Initializes all the variables.
 sess.run(init_all_op)
 # Runs to logit.
 sess.run(logits)
 # Creates a saver.
 saver0 = tf.train.Saver()
 saver0.save(sess, saver0_ckpt)
 # Generates MetaGraphDef.
 saver0.export_meta_graph('my-save-dir/my-model-10000.meta')

```

그런 다음 나중에 가져와서 훈련 그래프로 확장합니다.

```

with tf.Session() as sess:
 new_saver = tf.train.import_meta_graph('my-save-dir/my-model-10000.meta')
 new_saver.restore(sess, 'my-save-dir/my-model-10000')
 # Addes loss and train.
 labels = tf.constant(0, tf.int32, shape=[100], name="labels")
 batch_size = tf.size(labels)
 labels = tf.expand_dims(labels, 1)
 indices = tf.expand_dims(tf.range(0, batch_size), 1)
 concated = tf.concat([indices, labels], 1)
 onehot_labels = tf.sparse_to_dense(
 concated, tf.stack([batch_size, 10]), 1.0, 0.0)
 logits = tf.get_collection("logits")[0]
 cross_entropy = tf.nn.softmax_cross_entropy_with_logits(
 labels=onehot_labels, logits=logits, name="xentropy")
 loss = tf.reduce_mean(cross_entropy, name="xentropy_mean")

 tf.scalar_summary(loss.op.name, loss)
 # Creates the gradient descent optimizer with the given learning rate.
 optimizer = tf.train.GradientDescentOptimizer(0.01)

 # Runs train_op.
 train_op = optimizer.minimize(loss)
 sess.run(train_op)

```

- 하이퍼 파라미터(Hyper Parameters) 검색

```

filename = ".".join([tf.latest_checkpoint(train_dir), "meta"])
tf.train.import_meta_graph(filename)
hparams = tf.get_collection("hparams")

```

# 텐서플로우를 이용하여 신경망 양자화 (Quantize) 하는 방법

(v1.0)

최신 신경망이 개발될 때, 가장 큰 도전은 어떻게든 일을 하게 하는 것이였다. 이것은 학습에서 정확도와 속도가 가장 중요했다는 것을 의미한다. 부동소수점을 이용한 연산은 정확도를 유지하기 가장 쉬운 방법이었다, 그리고 GPU들은 이런 부동소수점 계산 가속에 특화되어 있다, 따라서 다른 형태의 연산 타입에는 많은 관심이 없었다.

최근에, 많은 모델들이 적용된 상용 제품들을 다수 보유하고 있다. The computation demands of training grow with the number of researchers, but the cycles needed for inference expand in proportion to users. 이는 많은 팀에서 순수한 추론 효율이 상당히 중요한 이슈라는 것을 의미합니다.

이런 이유로 quantization이 발생한 것입니다. Quantization은 숫자를 저장하고, 계산을 위한 32bit 부동 소수점 형식보다 컴팩트하고, 많은 부분을 커버하는 포괄적 용어입니다. 나는 앞으로 8bit 고정소수점에 초점을 맞추고 풀어갈 예정입니다.

[TOC]

## 양자화는 왜 동작할까?

신경망을 학습시키기 위해서는, 가중치들을 여러번 조금씩 조정해야 합니다. 그리고 조금씩 증가 혹은 감소를 통해 동작 시키기 위해서는 부동소수점이 필요하게 됩니다 (양자화된 표현을 사용하려는 노력이 있기는 하지만 말이죠).

미리 학습된 모델을 이용하여 추론하는 것은 매우 다릅니다. 깊은 신경망에서 대단한 성능을 보이는 것은 그 deep network가 입력의 노이즈를 잘 감추기 때문입니다. 만약 당신이 방금 찍은 사진에서 물체를 인식하기를 바란다면, 신경망은 모든 CCD 잡음, 빛 변화, 그리고 학습예제로 사용했던 것과 다른 불필요한 것들을 무시할 필요가 있습니다, 그리고 중요하다고 생각되는 것에 집중하게 되죠. 이 능력은 낮은 정밀도의 연산은 위에서 언급한 것처럼 단순히 잡음으로 취급될 수 있다는 것을 의미합니다. 그리고 적은 정보를 가지고 있는 숫자 표현으로도 정확한 결과를 생산해냅니다.

## 왜 양자일까?

신경망 모델들은 디스크의 많은 공간을 차지합니다, 예를 들어 AlexNet의 경우 200MB 가 넘는 부동 소수점 변수들이 필요합니다. 그 크기의 대부분은 신경망 연결을 위한 가중치에 할애되고 있습니다. 그 이유는 신경망 연결이 수백만개가 하나의 모델에 존재하기 때문입니다. 왜냐하면, 모든

변수들은 다 조금씩 다른 부동 소수점이기 때문입니다, zip과 같이 단순한 압축 형식은 이 변수들을 잘 압축시키지 못합니다. 많은 층에 위치해 있고, 각각의 층의 가중치들은 대부분 특정 영역에 분산되어 있습니다. 예를 들어 -3.0부터 +6.0까지.

Quantization의 가장 단순한 동기는 파일의 크기를 줄이는 것입니다. 크기를 줄이기 위해서 각 층의 최대, 최소 값을 저장하고, 각각의 부동 소수점 값을 256개의 범위 내에서 가장 가까운 실수 값을 8bit 정수형으로 압축시키는 것입니다. 예를 들어 -3.0에서 6.0까지의 범위에서, 0x0은 -3.0을 의미합니다, 그리고 0xff는 6.0을 표현 하겠죠, 그리고 0xef는 1.5정도를 표현 하겠죠. 정확한 계산은 뒤에서 합시다, 왜냐하면 좀 미묘한 문제가 있습니다, 하지만 이런 방식을 사용하면 75%정도의 디스크 크기를 아낄 수 있습니다, 그리고 다시 부동 소수점으로 바꾸면 우리의 모델은 변경 없이 부동 소수점을 사용할 수 있습니다.

Quantize의 다른 이유는 연산을 통한 추론 과정에서 하드웨어 연산기를 줄이기 위함입니다, 전체를 8bit 입력과 8bit 출력으로 사용하여서 말이죠. 계산이 필요한 모든 곳에서 변화가 필요하기 때문에 상당히 어렵습니다. 하지만 매우 높은 보상이 있습니다. 8bit의 값을 읽어 오는 것은 부동 소수점을 읽어오는 것의 25%에 해당하는 메모리 대역폭만을 요구합니다, 따라서 RAM에 접근에 따른 병목 현상이나, 캐싱을 함에 있어 훨씬 더 좋은 성능을 낼 수 있습니다. 또한 SIMD(Single Instruction Multiple Data) 명령어를 이용하여 한 클럭당 더욱 많은 명령어를 수행할 수도 있습니다. 몇몇 경우에는 DSP(Digital Signal Processing)칩을 이용하여 8bit 연산을 가속할 수도 있습니다.

8bit를 이용한 연산으로 옮겨갈 경우, 모델들을 더욱 빠르고, 저전력(휴대기기에서 중요한 조건)으로 동작시킬 수 있습니다. 또한 부동 소수점 연산이 불가능한 많은 embed ded 시스템에 적용될 수 있습니다, 따라서 IoT 세상에 많은 애플리케이션에 적용 될 수 있습니다.

## 왜 낮은 정밀도로 바로 학습시키지 않는 것인가요?

적은 bit를 이용하여 학습시키는 실험들이 몇번 시도 되었었습니다, 하지만 결과는 역전파(backpropagation)과 기울기(gradient)를 위해서는 더욱 정밀한 bit 폭이 필요한 것으로 파악되었습니다. 이런 문제는 학습을 더욱 복잡하게 만들고, 추론을 더욱 힘들게 합니다. 우리는 이미 부동 소수점으로 이루어진 잘알려진 모델들에 대해 실험 해보고, quantize로의 변환을 즉각적으로 매우 쉽게 할 수 있는 것을 확인 했습니다.

## 어떻게 당신의 모델을 Quantize하나요?

TensorFlow는 제품화 단계 등급의 8bit 연산기능을 지원하고 있습니다. 또한 부동 소수 점으로 학습된 많은 모델들을 양자화된 연산을 통해 동일한 추론이 가능하도록 변환할 수 있습니다. 예를 들어 가장 최근의 GoogLeNet 모델을 8bit 연산으로 변경하는 방법을 소개하고 있습니다:

```

curl http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz -o
/tmp/inceptionv3.tgz
tar xzf /tmp/inceptionv3.tgz -C /tmp/
bazel build tensorflow/tools/quantization/tools:quantize_graph
bazel-bin/tensorflow/tools/quantization/tools/quantize_graph \
--input=/tmp/classify_image_graph_def.pb \
--output_node_names="softmax" --output=/tmp/quantized_graph.pb \
--mode=eightbit

```

이것은 내부적으로는 8bit 연산을 사용하고 가중치 또한 양자화 되었지만, 원본 모델과 동일하게 동작합니다. 파일의 크기를 본다면, 원본과 비교하여 25%밖에 되지 않습니까(원본은 91MB인 반면 새로 생성된 모델은 23MB이다). 같은 입력과 출력을 갖고, 같은 결과를 얻을 수 있습니다. 아래에 예제가 있습니다:

```

bazel build tensorflow/examples/label_image:label_image
bazel-bin/tensorflow/examples/label_image/label_image \
--input_graph=/tmp/quantized_graph.pb \
--input_width=299 \
--input_height=299 \
--mean_value=128 \
--std_value=128 \
--input_layer_name="Mul:0" \
--output_layer_name="softmax:0"

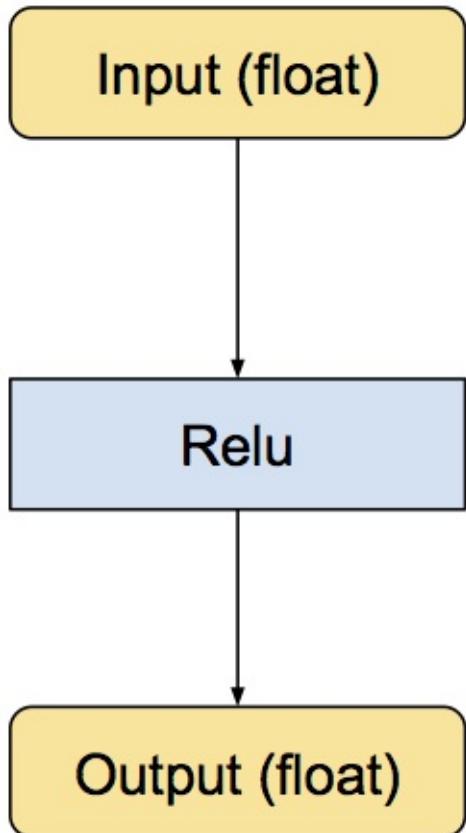
```

새롭게 양자화(quantized)된 그래프를 볼 수 있으며, 출력이 원본과 굉장히 비슷한 것을 볼 수 있습니다.

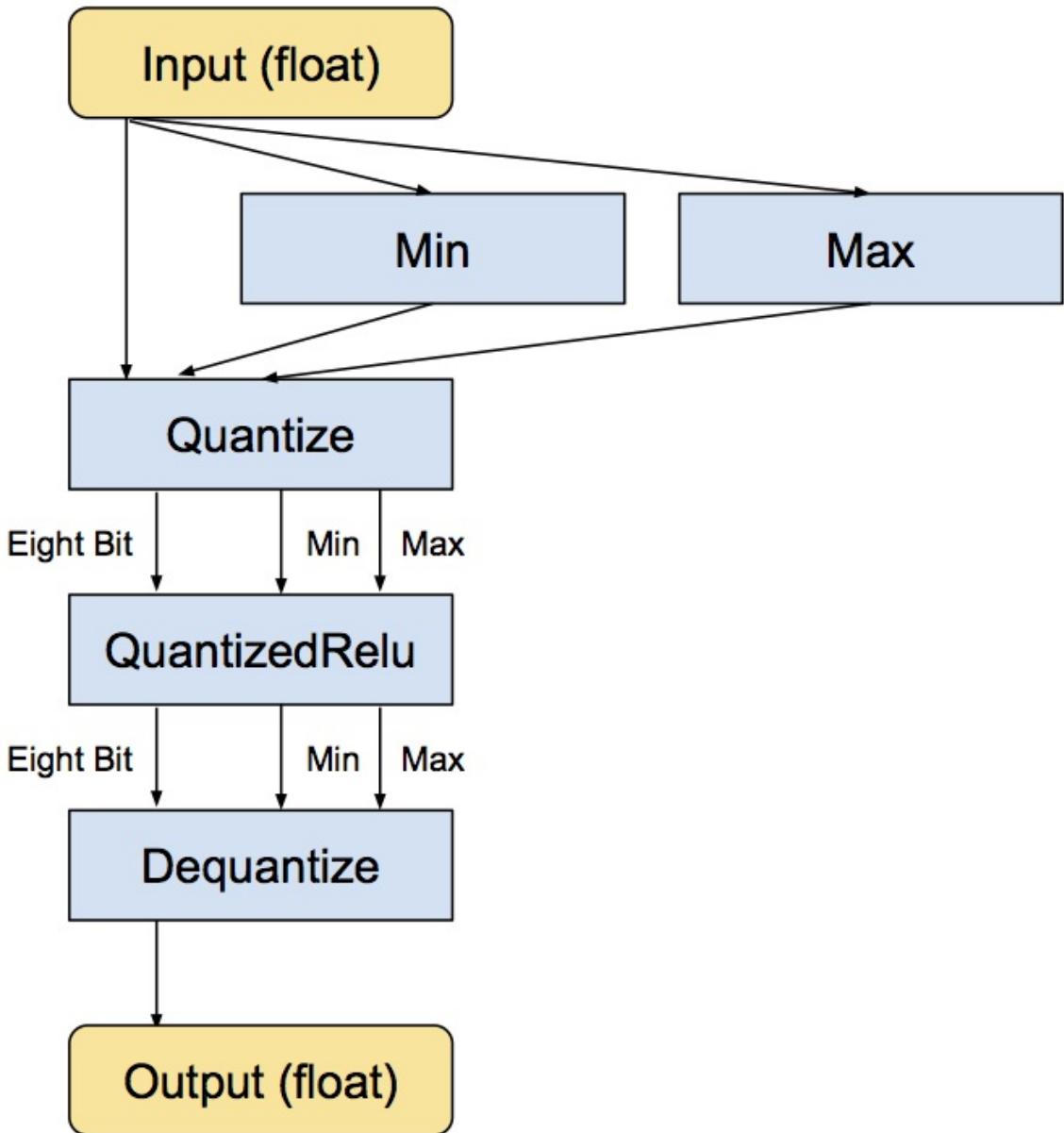
입력과 출력의 이름을 당신의 네트워크의 맞게 바꾸어 GraphDefs로 저장함으로서, 당신의 모델도 같은 처리를 할 수 있습니다. 상수값들이 저장되어 있는 파일을 변환하기 위해 `freeze_graph` 스크립트를 수행하는 것을 추천합니다.

## 양자화 프로세스가 어떻게 동작하나요?

추론과정에 주로 사용되는 연산(operations)들을 8bit 버전의 동일한 동작을 할 수 있는 코드를 구현하였습니다. Convolution, 행렬 곱셈기, 활성 함수, pooling 연산 그리고 행렬 합산기를 포함하고 있습니다. 기존 연산과 양자화 연산이 동일하게 각각의 연산 (TensorFlow의 ops; 번역)들에 변환 스크립트를 적용하였습니다. 부동 소수점과 8bit간의 이동을 위한 작은 부-그래프(sub-graph) 변환기를 위한 스크립트입니다. 아래 예제는 그것들이 어떻게 생겼는지에 대한 그림입니다. 처음으로 소개할 것은 입력과 출력이 부동 소수점으로 이루어진 원본 ReLU 연산입니다:

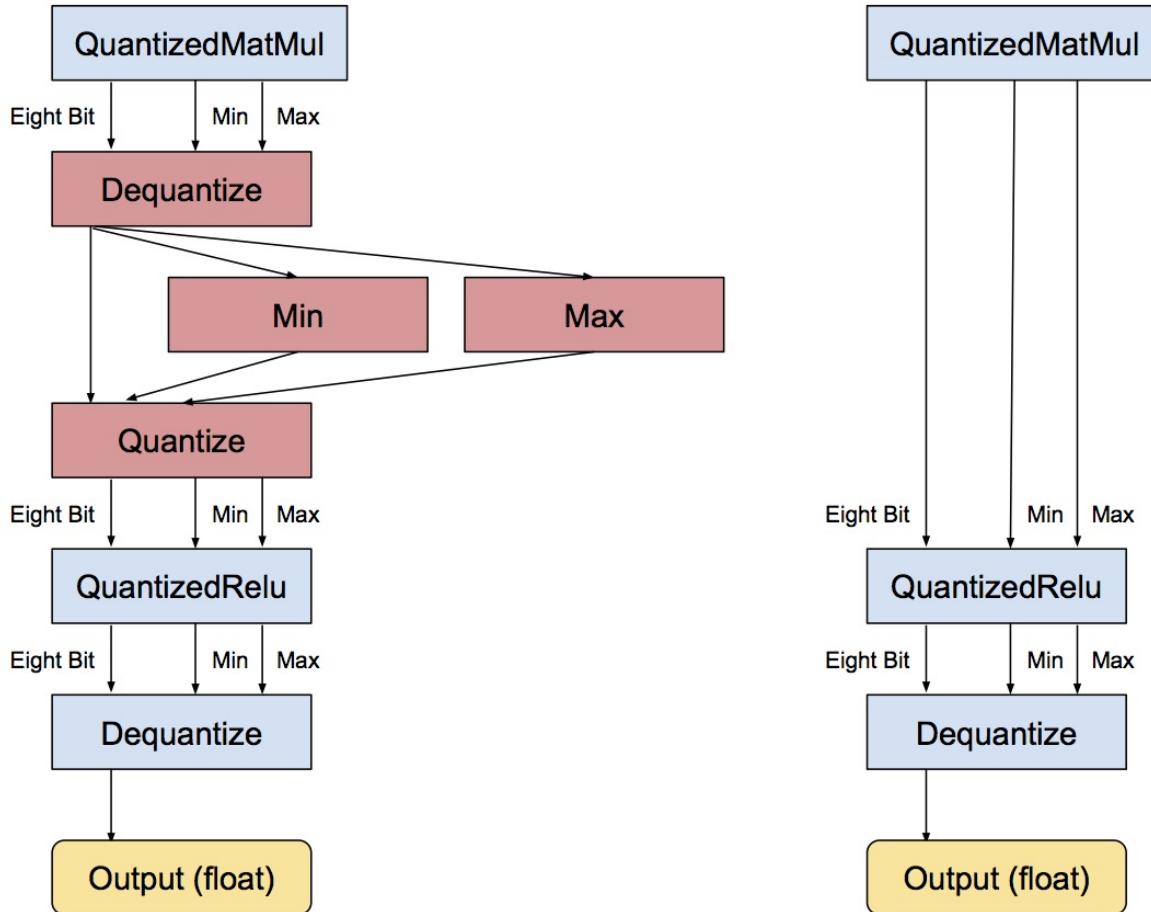


그리고, 아래 그림은 동일하지만 변환된 subgraph입니다, 여전히 부동 소수점 입력과 출력을 갖고 있지만 내부 변환을 통해 연산은 8bit로 진행되는 것을 알 수 있습니다.



최대, 최소값 연산은 처음의 부동 소수점 tensor를 확인하고 역양자화(dequantize)연 산을 위해 공급됩니다. 양자화 표현 방법에 관한 설명은 나중에 언급하도록 하겠습니다.

처음 각각의 연산(ReLU, convolution 등)이 변환 되었으면, 다음 단계는 부동 소수점으로(부터, to/from)의 불필요한 변환을 없애는 것입니다. 만약 연속된 연산들이 부동 소수점과 동일(여기서는 8bit quantize를 의미)하다면, 양자화와 역 양자화는 필요가 없게 되고 서로 상쇄될 것입니다 아래와 같아요:



모든 연산(graph in tensorflow)들이 양자화 된 거대한 모델에 적용된다면 tensor들의 연산은 모두 부동 소수점으로의 변환 없이 8bit로 끝낼 수 있습니다.

## 양자화된 Tensor를 위해 사용된 표현방법은 무엇인가요?

우리는 부동소수점 숫자의 집합들을 8bit 표현으로 변경하는 방법을 압축 문제로 접근 했습니다. 우리는 학습되어 있는 신경망 모델들의 가중치와 활성 tensor들이 비교적 작은 범위로 분포되어 있다는 것을 알았습니다(예를 들어 영상 모델에서 -15부터 15까 지는 가중치, -500부터 1000까 지는 활성 함수). 또한 우리는 신경망은 잡음에 상당히 강하고, 잡음과 같은 에러는 양자화 되어 결과에 영향을 미치기 힘들정도로 작아지는 것을 실험으로 확인하였습니다. 또한 거대한 행렬 곱과 같이 복잡한 연산에서 쉬운 방 법을 통한 변환을 선택하기를 원했습니다.

이런 아이디어는 두 개의 부동 소수점을 통해 최대, 최소값을 양자화 값으로 정하고 표현하는 방법을 선택하게 하였습니다. 각각의 입력에는 최대 최소값 사이에 일정한 분포를 가지고 있는 부동 소수점 집합이 존재합니다. 예를 들어 만약 우리가 -10을 최소 값으로 가지고 30을 최대값으로 가지면 8-bit 집합에서는 다음과 같이 양자화된 값이 표현될 수 있습니다:

| Quantized |  | Float |
|-----------|--|-------|
| -----     |  | ----- |
| 0         |  | -10.0 |
| 255       |  | 30.0  |
| 128       |  | 10.0  |

이런 형식의 장점은 범위를 통해 그 값을 표현할 수 있다는 점이고, 대칭일 필요가 없습니다, 또한 signed 형식 혹은 unsigned 형식에 구애받지 않습니다, 선형 분포는 계산을 바로 수행할 수 있습니다. 다음 [Song Han's code books](#) 에서와 같이 다른 양자 표현 방식 또한 존재합니다. 비 선형적 분포를 이용하여 부동 소수점을 표현하는 방법이지만, 계산은 복잡해지는 경향이 있습니다.

이런 방식으로 양자화 하는 것의 장점은 언제든지 이리저리 부동 소수점에서 양자화 값으로 변환 가능하다는 점입니다, 또는 디버깅을 위해 tensor를 분석할 때도 항상 변환이 가능합니다. One implementation detail in TensorFlow that we're hoping to improve in the future is that the minimum and maximum float values need to be passed as separate tensors to the one holding the quantized values, so graphs can get a bit dense!

최대 최소값을 이용한 변환이 좋은 이유중 하나는 그 값을 미리 알고 있을 수 있다는 점입니다. 가중치 파라미터들은 저장된 값을 읽을 때 알 수 있습니다, 따라서 그 값들의 범위는 저장할 때 상수로 저장될 수 있습니다. 입력의 범위는 미리 알고 있는 경우가 대부분이고(예를 들어 RGB는 0부터 255사이의 값만을 갖고 있습니다), 또한 많은 활성 함수들의 범위도 알고 있습니다. 이것은 행렬곱이나 convolution 같이 8bit 입력으로부터 누적되어 생성되는 32-bit 출력 값의 결과도 미리 알 수 있어 분석해야 하는 경우도 피할 수 있습니다.

## 다음으로는 무엇 인가요?

우리는 embedded 기기에서 8bit 수학적 연산을 통한 결과가 부동 소수점보다 훨씬 우수한 성능을 보인다는 사실을 알게 되었습니다(연산 속도인것 같습니다.;번역자). 우리가 사용하고 최적화한 행렬곱 framework를 [gemmlowp](#)에서 확인 할 수 있습니다. TensorFlow의 ops들의 최대 성능을 모바일 기기에서 얻기 위해 우리가 이번 실험등을 통해 얻은 결과들을 이용하여 활발히 연구하고 있습니다. 8-bit 모델을 더욱 폭넓고 다양한 기기들을 지원하기를 희망하고 있습니다. 그러기 위해 지금은 양자화 구현이 빠르고 정확한 reference 구현이 되어야 할 것입니다. 또한 우리는 이번 데모가 낮은 수준의 정확도를 가진 신경망을 연구하는 그룹에게 힘을 주기를 바라고 있습니다.

<번역 완료일: 2016. 06.30. 번역자: jjjhoon@gmail.com 의역 다수 존재> <중간 중간 의미 전달이 어려운 부분은 영어로 대체 및 번역을 하지 않았습니다> <피드백 및 오류 지적은 언제든 환영합니다>

# API 개요

텐서플로우(TensorFlow)는 그래프를 설계하고 실행하기 위해 여러 개발언어에서 사용 가능한 API들을 가지고 있습니다. 파이썬 API는 현재 가장 완벽하며 사용하기 쉽고, 안드로이드와 같은 모바일 기기에 배포하는 것을 지원합니다. 하지만 C++ API가 그래프를 실행하는데 있어 더 성능이 뛰어날 수 있습니다.

시간이 지나게 되면, 우리는 텐서플로우 커뮤니티 유저들이 Go, Java, Javascript, Lua, R 같은 언어에서도 프론트엔드 개발할 수 있기를 바랍니다. [SWIG](#)을 사용한다면, 비교적 쉽게 텐서플로우 인터페이스를 가장 좋아하는 언어로 개발할 수 있습니다.

노트: 대부분의 실제 사용사례는 [Mechanics](#) 탭에 수록되어 있으며, 특정 언어 API에 제한되어 있지 않은 추가적인 문서는 [Resources](#) 탭에서 확인하실 수 있습니다.

- [Python API](#)
- [C++ API](#)

# TensorFlow Python reference documentation

- **Building Graphs:**

- `add_to_collection`
- `as_dtype`
- `bytes`
- `control_dependencies`
- `convert_to_tensor`
- `convert_to_tensor_or_indexed_slices`
- `device`
- `DeviceSpec`
- `Dimension`
- `DTType`
- `get_collection`
- `get_collection_ref`
- `get_default_graph`
- `get_seed`
- `Graph`
- `GraphKeys`
- `import_graph_def`
- `load_file_system_library`
- `load_op_library`
- `name_scope`
- `NoGradient`
- `op_scope`
- `Operation`
- `register_tensor_conversion_function`
- `RegisterGradient`
- `RegisterShape`
- `reset_default_graph`
- `Tensor`
- `TensorShape`

- **Asserts and boolean checks:**

- `assert_equal`
- `assert_integer`

- `assert_less`
- `assert_less_equal`
- `assert_negative`
- `assert_non_negative`
- `assert_non_positive`
- `assert_positive`
- `assert_proper_iterable`
- `assert_rank`
- `assert_rank_at_least`
- `assert_type`
- `is_non_decreasing`
- `is_numeric_tensor`
- `is_strictly_increasing`

- **Constants, Sequences, and Random Values:**

- `constant`
- `fill`
- `linspace`
- `multinomial`
- `ones`
- `ones_like`
- `random_crop`
- `random_normal`
- `random_shuffle`
- `random_uniform`
- `range`
- `set_random_seed`
- `truncated_normal`
- `zeros`
- `zeros_like`

- **Variables:**

- `all_variables`
- `assert_variables_initialized`
- `assign`
- `assign_add`
- `assign_sub`
- `constant_initializer`
- `count_up_to`
- `device`
- `export_meta_graph`

- `get_checkpoint_state`
- `get_variable`
- `get_variable_scope`
- `import_meta_graph`
- `IndexedSlices`
- `initialize_all_variables`
- `initialize_local_variables`
- `initialize_variables`
- `is_variable_initialized`
- `latest_checkpoint`
- `local_variables`
- `make_template`
- `moving_average_variables`
- `no_regularizer`
- `ones_initializer`
- `random_normal_initializer`
- `random_uniform_initializer`
- `report_uninitialized_variables`
- `Saver`
- `scatter_add`
- `scatter_sub`
- `scatter_update`
- `sparse_mask`
- `trainable_variables`
- `truncated_normal_initializer`
- `uniform_unit_scaling_initializer`
- `update_checkpoint_state`
- `Variable`
- `variable_axis_size_partitioner`
- `variable_op_scope`
- `variable_scope`
- `VariableScope`
- `zeros_initializer`

- **Tensor Transformations:**

- `batch_to_space`
- `bitcast`
- `boolean_mask`
- `cast`
- `concat`
- `depth_to_space`

- `dynamic_partition`
- `dynamic_stitch`
- `expand_dims`
- `extract_image_patches`
- `gather`
- `gather_nd`
- `one_hot`
- `pack`
- `pad`
- `rank`
- `reshape`
- `reverse`
- `reverse_sequence`
- `saturate_cast`
- `shape`
- `shape_n`
- `size`
- `slice`
- `space_to_batch`
- `space_to_depth`
- `split`
- `squeeze`
- `string_to_number`
- `tile`
- `to_bfloat16`
- `to_double`
- `to_float`
- `to_int32`
- `to_int64`
- `transpose`
- `unique_with_counts`
- `unpack`

- **Math:**

- `abs`
- `accumulate_n`
- `acos`
- `add`
- `add_n`
- `argmax`
- `argmin`

- o [asin](#)
- o [atan](#)
- o [batch\\_cholesky](#)
- o [batch\\_cholesky\\_solve](#)
- o [batch\\_fft](#)
- o [batch\\_fft2d](#)
- o [batch\\_fft3d](#)
- o [batch\\_ifft](#)
- o [batch\\_ifft2d](#)
- o [batch\\_ifft3d](#)
- o [batch\\_matmul](#)
- o [batch\\_matrix\\_band\\_part](#)
- o [batch\\_matrix\\_determinant](#)
- o [batch\\_matrix\\_diag](#)
- o [batch\\_matrix\\_diag\\_part](#)
- o [batch\\_matrix\\_inverse](#)
- o [batch\\_matrix\\_solve](#)
- o [batch\\_matrix\\_solve\\_ls](#)
- o [batch\\_matrix\\_triangular\\_solve](#)
- o [batch\\_self\\_adjoint\\_eig](#)
- o [ceil](#)
- o [cholesky](#)
- o [cholesky\\_solve](#)
- o [complex](#)
- o [complex\\_abs](#)
- o [conj](#)
- o [cos](#)
- o [cross](#)
- o [diag](#)
- o [diag\\_part](#)
- o [digamma](#)
- o [div](#)
- o [edit\\_distance](#)
- o [erf](#)
- o [erfc](#)
- o [exp](#)
- o [fft](#)
- o [fft2d](#)
- o [fft3d](#)
- o [floor](#)

- [floordiv](#)
- [ifft](#)
- [ifft2d](#)
- [ifft3d](#)
- [igamma](#)
- [igammac](#)
- [imag](#)
- [inv](#)
- [invert\\_permutation](#)
- [lbeta](#)
- [lgamma](#)
- [listdiff](#)
- [log](#)
- [matmul](#)
- [matrix\\_determinant](#)
- [matrix\\_inverse](#)
- [matrix\\_solve](#)
- [matrix\\_solve\\_ls](#)
- [matrix\\_triangular\\_solve](#)
- [maximum](#)
- [minimum](#)
- [mod](#)
- [mul](#)
- [neg](#)
- [polygamma](#)
- [pow](#)
- [real](#)
- [reduce\\_all](#)
- [reduce\\_any](#)
- [reduce\\_max](#)
- [reduce\\_mean](#)
- [reduce\\_min](#)
- [reduce\\_prod](#)
- [reduce\\_sum](#)
- [round](#)
- [rsqrt](#)
- [scalar\\_mul](#)
- [segment\\_max](#)
- [segment\\_mean](#)
- [segment\\_min](#)

- `segment_prod`
- `segment_sum`
- `self_adjoint_eig`
- `sign`
- `sin`
- `sparse_segment_mean`
- `sparse_segment_sqrt_n`
- `sparse_segment_sqrt_n_grad`
- `sparse_segment_sum`
- `sqrt`
- `square`
- `squared_difference`
- `sub`
- `tan`
- `trace`
- `transpose`
- `truediv`
- `unique`
- `unsorted_segment_sum`
- `where`
- `zeta`

- **Strings:**

- `reduce_join`
- `string_to_hash_bucket`
- `string_to_hash_bucket_fast`
- `string_to_hash_bucket_strong`

- **Histograms:**

- `histogram_fixed_width`

- **Control Flow:**

- `add_check_numerics_ops`
- `Assert`
- `case`
- `check_numerics`
- `cond`
- `count_up_to`
- `equal`
- `greater`
- `greater_equal`
- `group`

- `identity`
- `is_finite`
- `is_inf`
- `is_nan`
- `less`
- `less_equal`
- `logical_and`
- `logical_not`
- `logical_or`
- `logical_xor`
- `no_op`
- `not_equal`
- `Print`
- `select`
- `tuple`
- `verify_tensor_all_finite`
- `where`
- `while_loop`

- **Higher Order Functions:**

- `foldl`
- `foldr`
- `map_fn`
- `scan`

- **TensorArray Operations:**

- `concat`
- `pack`
- `split`
- `TensorArray`
- `unpack`

- **Tensor Handle Operations:**

- `delete_session_tensor`
- `get_session_handle`
- `get_session_tensor`

- **Images:**

- `adjust_brightness`
- `adjust_contrast`
- `adjust_hue`
- `adjust_saturation`

- `central_crop`
- `convert_image_dtype`
- `crop_to_bounding_box`
- `decode_jpeg`
- `decode_png`
- `draw_bounding_boxes`
- `encode_jpeg`
- `encode_png`
- `extract_glimpse`
- `flip_left_right`
- `flip_up_down`
- `grayscale_to_rgb`
- `hsv_to_rgb`
- `pad_to_bounding_box`
- `per_image_whitening`
- `random_brightness`
- `random_contrast`
- `random_flip_left_right`
- `random_flip_up_down`
- `random_hue`
- `random_saturation`
- `resize_area`
- `resize_bicubic`
- `resize_bilinear`
- `resize_image_with_crop_or_pad`
- `resize_images`
- `resize_nearest_neighbor`
- `rgb_to_grayscale`
- `rgb_to_hsv`
- `sample_distorted_bounding_box`
- `transpose_image`

- **Sparse Tensors:**

- `shape`
- `sparse_add`
- `sparse_concat`
- `sparse_fill_empty_rows`
- `sparse_merge`
- `sparse_reduce_sum`
- `sparse_reorder`
- `sparse_reset_shape`

- [sparse\\_retain](#)
- [sparse\\_softmax](#)
- [sparse\\_split](#)
- [sparse\\_tensor\\_dense\\_matmul](#)
- [sparse\\_tensor\\_to\\_dense](#)
- [sparse\\_to\\_dense](#)
- [sparse\\_to\\_indicator](#)
- [SparseTensor](#)
- [SparseTensorValue](#)

- **[Inputs and Readers:](#)**

- [batch](#)
- [batch\\_join](#)
- [decode\\_csv](#)
- [decode\\_json\\_example](#)
- [decode\\_raw](#)
- [FIFOQueue](#)
- [FixedLenFeature](#)
- [FixedLengthRecordReader](#)
- [FixedLenSequenceFeature](#)
- [IdentityReader](#)
- [input\\_producer](#)
- [limit\\_epochs](#)
- [match\\_filenames\\_once](#)
- [matching\\_files](#)
- [PaddingFIFOQueue](#)
- [parse\\_example](#)
- [parse\\_single\\_example](#)
- [placeholder](#)
- [placeholder\\_with\\_default](#)
- [QueueBase](#)
- [RandomShuffleQueue](#)
- [range\\_input\\_producer](#)
- [read\\_file](#)
- [ReaderBase](#)
- [shuffle\\_batch](#)
- [shuffle\\_batch\\_join](#)
- [size](#)
- [slice\\_input\\_producer](#)
- [sparse\\_placeholder](#)
- [string\\_input\\_producer](#)

- `TextLineReader`
- `TFRecordReader`
- `VarLenFeature`
- `WholeFileReader`

- **Data IO (Python functions):**

- `tf_record_iterator`
- `TFRecordWriter`

- **Neural Network:**

- `atrous_conv2d`
- `avg_pool`
- `avg_pool3d`
- `batch_normalization`
- `bias_add`
- `bidirectional_rnn`
- `compute_accidental_hits`
- `conv2d`
- `conv2d_transpose`
- `conv3d`
- `depthwise_conv2d`
- `depthwise_conv2d_native`
- `dilation2d`
- `dropout`
- `dynamic_rnn`
- `elu`
- `embedding_lookup`
- `embedding_lookup_sparse`
- `erosion2d`
- `fixed_unigram_candidate_sampler`
- `in_top_k`
- `l2_loss`
- `l2_normalize`
- `learned_unigram_candidate_sampler`
- `local_response_normalization`
- `log_softmax`
- `log_uniform_candidate_sampler`
- `max_pool`
- `max_pool3d`
- `max_pool_with_argmax`
- `moments`

- `nce_loss`
- `normalize_moments`
- `relu`
- `relu6`
- `rnn`
- `sampled_softmax_loss`
- `separable_conv2d`
- `sigmoid`
- `sigmoid_cross_entropy_with_logits`
- `softmax`
- `softmax_cross_entropy_with_logits`
- `softplus`
- `softsign`
- `sparse_softmax_cross_entropy_with_logits`
- `state_saving_rnn`
- `sufficient_statistics`
- `tanh`
- `top_k`
- `uniform_candidate_sampler`
- `weighted_cross_entropy_with_logits`

- **Neural Network RNN Cells:**

- `BasicLSTMCell`
- `BasicRNNCell`
- `DropoutWrapper`
- `EmbeddingWrapper`
- `GRUCell`
- `InputProjectionWrapper`
- `LSTMCell`
- `LSTMStateTuple`
- `MultiRNNCell`
- `OutputProjectionWrapper`
- `RNNCell`

- **Running Graphs:**

- `AbortedError`
- `AlreadyExistsError`
- `CancelledError`
- `DataLossError`
- `DeadlineExceededError`
- `FailedPreconditionError`

- `get_default_session`
- `InteractiveSession`
- `InternalError`
- `InvalidArgumentError`
- `NotFoundError`
- `OpError`
- `OutOfRangeError`
- `PermissionDeniedError`
- `ResourceExhaustedError`
- `Session`
- `UnauthenticatedError`
- `UnavailableError`
- `UnimplementedError`
- `UnknownError`

- **Training:**

- `AdadeltaOptimizer`
- `AdagradOptimizer`
- `AdamOptimizer`
- `add_queue_runner`
- `AggregationMethod`
- `audio_summary`
- `clip_by_average_norm`
- `clip_by_global_norm`
- `clip_by_norm`
- `clip_by_value`
- `ClusterSpec`
- `Coordinator`
- `exponential_decay`
- `ExponentialMovingAverage`
- `FtrlOptimizer`
- `generate_checkpoint_state_proto`
- `global_norm`
- `global_step`
- `GradientDescentOptimizer`
- `gradients`
- `histogram_summary`
- `image_summary`
- `LooperThread`
- `merge_all_summaries`
- `merge_summary`

- `MomentumOptimizer`
- `Optimizer`
- `QueueRunner`
- `replica_device_setter`
- `RMSPropOptimizer`
- `scalar_summary`
- `Server`
- `SessionManager`
- `start_queue_runners`
- `stop_gradient`
- `summary_iterator`
- `SummaryWriter`
- `Supervisor`
- `write_graph`
- `zero_fraction`

- **Wraps python functions:**

- `py_func`

- **Testing:**

- `assert_equal_graph_def`
- `compute_gradient`
- `compute_gradient_error`
- `get_temp_dir`
- `is_built_with_cuda`
- `main`

- **Statistical distributions (contrib):**

- `BaseDistribution`
- `Chi2`
- `ContinuousDistribution`
- `DirichletMultinomial`
- `DiscreteDistribution`
- `Exponential`
- `Gamma`
- `MultivariateNormal`
- `Normal`
- `normal_conjugates_known_sigma_predictive`
- `normal_conjugates_known_sigma_posterior`
- `StudentT`
- `Uniform`

- **FFmpeg (contrib):**

- `decode_audio`
- `encode_audio`

- **Framework (contrib):**

- `add_arg_scope`
- `add_model_variable`
- `arg_scope`
- `arg_scoped_arguments`
- `assert_global_step`
- `assert_or_get_global_step`
- `assert_same_float_dtype`
- `assert_scalar_int`
- `convert_to_tensor_or_sparse_tensor`
- `create_global_step`
- `get_global_step`
- `get_graph_from_inputs`
- `get_local_variables`
- `get_model_variables`
- `get_or_create_global_step`
- `get_unique_variable`
- `get_variables`
- `get_variables_by_name`
- `get_variables_by_suffix`
- `get_variables_to_restore`
- `has_arg_scope`
- `is_non_decreasing`
- `is_numeric_tensor`
- `is_strictly_increasing`
- `local_variable`
- `model_variable`
- `reduce_sum_n`
- `safe_embedding_lookup_sparse`
- `variable`
- `VariableDeviceChooser`
- `with_same_shape`
- `with_shape`

- **Layers (contrib):**

- `apply_regularization`
- `convolution2d`
- `fully_connected`

- [l1\\_regularizer](#)
- [l2\\_regularizer](#)
- [optimize\\_loss](#)
- [sum\\_regularizer](#)
- [summarize\\_activation](#)
- [summarize\\_activations](#)
- [summarize\\_collection](#)
- [summarize\\_tensor](#)
- [summarize\\_tensors](#)
- [variance\\_scaling\\_initializer](#)
- [xavier\\_initializer](#)
- [xavier\\_initializer\\_conv2d](#)

- **Learn (contrib):**

- [BaseEstimator](#)
- [DNNClassifier](#)
- [DNNRegressor](#)
- [Estimator](#)
- [evaluate](#)
- [extract\\_task\\_data](#)
- [extract\\_task\\_labels](#)
- [extract\\_pandas\\_data](#)
- [extract\\_pandas\\_labels](#)
- [extract\\_pandas\\_matrix](#)
- [infer](#)
- [LinearClassifier](#)
- [LinearRegressor](#)
- [ModeKeys](#)
- [NanLossDuringTrainingError](#)
- [read\\_batch\\_examples](#)
- [read\\_batch\\_features](#)
- [read\\_batch\\_record\\_features](#)
- [run\\_feeds](#)
- [run\\_n](#)
- [RunConfig](#)
- [TensorFlowClassifier](#)
- [TensorFlowDNNClassifier](#)
- [TensorFlowDNNRegressor](#)
- [TensorFlowEstimator](#)
- [TensorFlowLinearClassifier](#)
- [TensorFlowLinearRegressor](#)

- `TensorFlowRegressor`
- `TensorFlowRNNClassifier`
- `TensorFlowRNNRegressor`
- `train`

- **Losses (contrib):**

- `absolute_difference`
- `add_loss`
- `cosine_distance`
- `get_losses`
- `get_regularization_losses`
- `get_total_loss`
- `log_loss`
- `sigmoid_cross_entropy`
- `softmax_cross_entropy`
- `sum_of_pairwise_squares`
- `sum_of_squares`

- **Metrics (contrib):**

- `accuracy`
- `auc_using_histogram`
- `confusion_matrix`
- `set_difference`
- `set_intersection`
- `set_size`
- `set_union`
- `streaming_accuracy`
- `streaming_auc`
- `streaming_mean`
- `streaming_mean_absolute_error`
- `streaming_mean_cosine_distance`
- `streaming_mean_relative_error`
- `streaming_mean_squared_error`
- `streaming_percentage_less`
- `streaming_precision`
- `streaming_recall`
- `streaming_recall_at_k`
- `streaming_root_mean_squared_error`
- `streaming_sparse_precision_at_k`
- `streaming_sparse_recall_at_k`

- **Utilities (contrib):**

- `constant_value`
- `make_ndarray`
- `make_tensor_proto`
- `ops_used_by_graph_def`
- `stripped_op_list_for_graph`

- **Copying Graph Elements (contrib):**

- `copy_op_to_graph`
- `copy_variable_to_graph`
- `get_copied_op`

# Building Graphs

[TOC]

Classes and functions for building TensorFlow graphs.

## Core graph data structures

---

### class tf.Graph

A TensorFlow computation, represented as a dataflow graph.

A `Graph` contains a set of `operation` objects, which represent units of computation; and `Tensor` objects, which represent the units of data that flow between operations.

A default `Graph` is always registered, and accessible by calling `tf.get_default_graph()`. To add an operation to the default graph, simply call one of the functions that defines a new `Operation`:

```
c = tf.constant(4.0)
assert c.graph is tf.get_default_graph()
```

Another typical usage involves the `Graph.as_default()` context manager, which overrides the current default graph for the lifetime of the context:

```
g = tf.Graph()
with g.as_default():
 # Define operations and tensors in `g`.
 c = tf.constant(30.0)
 assert c.graph is g
```

Important note: This class *is not* thread-safe for graph construction. All operations should be created from a single thread, or external synchronization must be provided. Unless otherwise specified, all methods are not thread-safe.

---

### tf.Graph.\_\_init\_\_()

Creates a new, empty Graph.

## `tf.Graph.as_default()`

Returns a context manager that makes this `Graph` the default graph.

This method should be used if you want to create multiple graphs in the same process. For convenience, a global default graph is provided, and all ops will be added to this graph if you do not create a new graph explicitly. Use this method with the `with` keyword to specify that ops created within the scope of a block should be added to this graph.

The default graph is a property of the current thread. If you create a new thread, and wish to use the default graph in that thread, you must explicitly add a `with g.as_default():` in that thread's function.

The following code examples are equivalent:

```
1. Using Graph.as_default():
g = tf.Graph()
with g.as_default():
 c = tf.constant(5.0)
 assert c.graph is g

2. Constructing and making default:
with tf.Graph().as_default() as g:
 c = tf.constant(5.0)
 assert c.graph is g
```

### Returns:

A context manager for using this graph as the default graph.

## `tf.Graph.as_graph_def(from_version=None, add_shapes=False)`

Returns a serialized `GraphDef` representation of this graph.

The serialized `GraphDef` can be imported into another `Graph` (using `import_graph_def()`) or used with the [C++ Session API](#).

This method is thread-safe.

### Args:

- `from_version` : Optional. If this is set, returns a `GraphDef` containing only the nodes that were added to this graph since its `version` property had the given value.
- `add_shapes` : If true, adds an `"_output_shapes"` list attr to each node with the inferred

shapes of each of its outputs.

**Returns:**

A [GraphDef](#) protocol buffer.

**Raises:**

- `ValueError` : If the `graph_def` would be too large.

**`tf.Graph.finalize()`**

Finalizes this graph, making it read-only.

After calling `g.finalize()`, no new operations can be added to `g`. This method is used to ensure that no operations are added to a graph when it is shared between multiple threads, for example when using a [QueueRunner](#).

**`tf.Graph.finalized`**

True if this graph has been finalized.

**`tf.Graph.control_dependencies(control_inputs)`**

Returns a context manager that specifies control dependencies.

Use with the `with` keyword to specify that all operations constructed within the context should have control dependencies on `control_inputs`. For example:

```
with g.control_dependencies([a, b, c]):
 # `d` and `e` will only run after `a`, `b`, and `c` have executed.
 d = ...
 e = ...
```

Multiple calls to `control_dependencies()` can be nested, and in that case a new `Operation` will have control dependencies on the union of `control_inputs` from all active contexts.

```
with g.control_dependencies([a, b]):
 # Ops constructed here run after `a` and `b`.
 with g.control_dependencies([c, d]):
 # Ops constructed here run after `a`, `b`, `c`, and `d`.
```

You can pass `None` to clear the control dependencies:

```
with g.control_dependencies([a, b]):
 # Ops constructed here run after `a` and `b`.
 with g.control_dependencies(None):
 # Ops constructed here run normally, not waiting for either `a` or `b`.
 with g.control_dependencies([c, d]):
 # Ops constructed here run after `c` and `d`, also not waiting
 # for either `a` or `b`.
```

*N.B.* The control dependencies context applies *only* to ops that are constructed within the context. Merely using an op or tensor in the context does not add a control dependency. The following example illustrates this point:

```
WRONG
def my_func(pred, tensor):
 t = tf.matmul(tensor, tensor)
 with tf.control_dependencies([pred]):
 # The matmul op is created outside the context, so no control
 # dependency will be added.
 return t

RIGHT
def my_func(pred, tensor):
 with tf.control_dependencies([pred]):
 # The matmul op is created in the context, so a control dependency
 # will be added.
 return tf.matmul(tensor, tensor)
```

### Args:

- `control_inputs` : A list of `Operation` or `Tensor` objects which must be executed or computed before running the operations defined in the context. Can also be `None` to clear the control dependencies.

### Returns:

A context manager that specifies control dependencies for all operations constructed within the context.

### Raises:

- `TypeError` : If `control_inputs` is not a list of `Operation` or `Tensor` objects.

---

**`tf.Graph.device(device_name_or_function)`**

---

Returns a context manager that specifies the default device to use.

The `device_name_or_function` argument may either be a device name string, a device function, or `None`:

- If it is a device name string, all operations constructed in this context will be assigned to the device with that name, unless overridden by a nested `device()` context.
- If it is a function, it will be treated as a function from Operation objects to device name strings, and invoked each time a new Operation is created. The Operation will be assigned to the device with the returned name.
- If it is `None`, all `device()` invocations from the enclosing context will be ignored.

For information about the valid syntax of device name strings, see the documentation in

[DeviceNameUtils](#).

For example:

```
with g.device('/gpu:0'):
 # All operations constructed in this context will be placed
 # on GPU 0.
 with g.device(None):
 # All operations constructed in this context will have no
 # assigned device.

 # Defines a function from `Operation` to device string.
 def matmul_on_gpu(n):
 if n.type == "MatMul":
 return "/gpu:0"
 else:
 return "/cpu:0"

with g.device(matmul_on_gpu):
 # All operations of type "MatMul" constructed in this context
 # will be placed on GPU 0; all other operations will be placed
 # on CPU 0.
```

**N.B.** The device scope may be overridden by op wrappers or other library code. For example, a variable assignment op `v.assign()` must be colocated with the `tf.Variable v`, and incompatible device scopes will be ignored.

### Args:

- `device_name_or_function` : The device name or function to use in the context.

### Returns:

A context manager that specifies the default device to use for newly created ops.

## **tf.Graph.name\_scope(name)**

Returns a context manager that creates hierarchical names for operations.

A graph maintains a stack of name scopes. A `with name_scope(...):` statement pushes a new name onto the stack for the lifetime of the context.

The `name` argument will be interpreted as follows:

- A string (not ending with '/') will create a new name scope, in which `name` is appended to the prefix of all operations created in the context. If `name` has been used before, it will be made unique by calling `self.unique_name(name)`.
- A scope previously captured from a `with g.name_scope(...) as scope:` statement will be treated as an "absolute" name scope, which makes it possible to re-enter existing scopes.
- A value of `None` or the empty string will reset the current name scope to the top-level (empty) name scope.

For example:

```

with tf.Graph().as_default() as g:
 c = tf.constant(5.0, name="c")
 assert c.op.name == "c"
 c_1 = tf.constant(6.0, name="c")
 assert c_1.op.name == "c_1"

 # Creates a scope called "nested"
 with g.name_scope("nested") as scope:
 nested_c = tf.constant(10.0, name="c")
 assert nested_c.op.name == "nested/c"

 # Creates a nested scope called "inner".
 with g.name_scope("inner"):
 nested_inner_c = tf.constant(20.0, name="c")
 assert nested_inner_c.op.name == "nested/inner/c"

 # Create a nested scope called "inner_1".
 with g.name_scope("inner"):
 nested_inner_1_c = tf.constant(30.0, name="c")
 assert nested_inner_1_c.op.name == "nested/inner_1/c"

 # Treats `scope` as an absolute name scope, and
 # switches to the "nested/" scope.
 with g.name_scope(scope):
 nested_d = tf.constant(40.0, name="d")
 assert nested_d.op.name == "nested/d"

 with g.name_scope("") as scope:
 e = tf.constant(50.0, name="e")
 assert e.op.name == "e"

```

The name of the scope itself can be captured by `with g.name_scope(...)` as `scope`, which stores the name of the scope in the variable `scope`. This value can be used to name an operation that represents the overall result of executing the ops in a scope. For example:

```

inputs = tf.constant(...)
with g.name_scope('my_layer') as scope:
 weights = tf.Variable(..., name="weights")
 biases = tf.Variable(..., name="biases")
 affine = tf.matmul(inputs, weights) + biases
 output = tf.nn.relu(affine, name=scope)

```

## Args:

- `name` : A name for the scope.

## Returns:

A context manager that installs `name` as a new name scope.

A `Graph` instance supports an arbitrary number of "collections" that are identified by name. For convenience when building a large graph, collections can store groups of related objects: for example, the `tf.Variable` uses a collection (named `tf.GraphKeys.VARIABLES`) for all variables that are created during the construction of a graph. The caller may define additional collections by specifying a new name.

---

### `tf.Graph.add_to_collection(name, value)`

Stores `value` in the collection with the given `name`.

Note that collections are not sets, so it is possible to add a value to a collection several times.

#### Args:

- `name` : The key for the collection. The `GraphKeys` class contains many standard names for collections.
  - `value` : The value to add to the collection.
- 

### `tf.Graph.add_to_collections(names, value)`

Stores `value` in the collections given by `names`.

Note that collections are not sets, so it is possible to add a value to a collection several times. This function makes sure that duplicates in `names` are ignored, but it will not check for pre-existing membership of `value` in any of the collections in `names`.

`names` can be any iterable, but if `names` is a string, it is treated as a single collection name.

#### Args:

- `names` : The keys for the collections to add to. The `GraphKeys` class contains many standard names for collections.
  - `value` : The value to add to the collections.
- 

### `tf.Graph.get_collection(name, scope=None)`

Returns a list of values in the collection with the given `name`.

This is different from `get_collection_ref()` which always returns the actual collection list if it exists in that it returns a new list each time it is called.

**Args:**

- `name` : The key for the collection. For example, the `GraphKeys` class contains many standard names for collections.
- `scope` : (Optional.) If supplied, the resulting list is filtered to include only items whose `name` attribute matches using `re.match`. Items without a `name` attribute are never returned if a scope is supplied and the choice of `re.match` means that a `scope` without special tokens filters by prefix.

**Returns:**

The list of values in the collection with the given `name`, or an empty list if no value has been added to that collection. The list contains the values in the order under which they were collected.

---

**`tf.Graph.get_collection_ref(name)`**

Returns a list of values in the collection with the given `name`.

If the collection exists, this returns the list itself, which can be modified in place to change the collection. If the collection does not exist, it is created as an empty list and the list is returned.

This is different from `get_collection()` which always returns a copy of the collection list if it exists and never creates an empty collection.

**Args:**

- `name` : The key for the collection. For example, the `GraphKeys` class contains many standard names for collections.

**Returns:**

The list of values in the collection with the given `name`, or an empty list if no value has been added to that collection.

---

**`tf.Graph.as_graph_element(obj, allow_tensor=True, allow_operation=True)`**

Returns the object referred to by `obj`, as an `Operation` or `Tensor`.

This function validates that `obj` represents an element of this graph, and gives an informative error message if it is not.

This function is the canonical way to get/validate an object of one of the allowed types from an external argument reference in the Session API.

This method may be called concurrently from multiple threads.

**Args:**

- `obj` : A `Tensor`, an `Operation`, or the name of a tensor or operation. Can also be any object with an `_as_graph_element()` method that returns a value of one of these types.
- `allow_tensor` : If true, `obj` may refer to a `Tensor`.
- `allow_operation` : If true, `obj` may refer to an `Operation`.

**Returns:**

The `Tensor` or `Operation` in the Graph corresponding to `obj`.

**Raises:**

- `TypeError` : If `obj` is not a type we support attempting to convert to types.
  - `ValueError` : If `obj` is of an appropriate type but invalid. For example, an invalid string.
  - `KeyError` : If `obj` is not an object in the graph.
- 

**`tf.Graph.get_operation_by_name(name)`**

Returns the `Operation` with the given `name`.

This method may be called concurrently from multiple threads.

**Args:**

- `name` : The name of the `Operation` to return.

**Returns:**

The `Operation` with the given `name`.

**Raises:**

- `TypeError` : If `name` is not a string.
  - `KeyError` : If `name` does not correspond to an operation in this graph.
- 

**`tf.Graph.get_tensor_by_name(name)`**

Returns the `Tensor` with the given `name`.

This method may be called concurrently from multiple threads.

**Args:**

- `name` : The name of the `Tensor` to return.

**Returns:**

The `Tensor` with the given `name`.

**Raises:**

- `TypeError` : If `name` is not a string.
- `KeyError` : If `name` does not correspond to a tensor in this graph.

---

**`tf.Graph.get_operations()`**

Return the list of operations in the graph.

You can modify the operations in place, but modifications to the list such as inserts/delete have no effect on the list of operations known to the graph.

This method may be called concurrently from multiple threads.

**Returns:**

A list of Operations.

---

**`tf.Graph.seed`**

The graph-level random seed of this graph.

---

**`tf.Graph.unique_name(name, mark_as_used=True)`**

Return a unique operation name for `name`.

Note: You rarely need to call `unique_name()` directly. Most of the time you just need to create `with g.name_scope()` blocks to generate structured names.

`unique_name` is used to generate structured names, separated by `/`, to help identify operations when debugging a graph. Operation names are displayed in error messages reported by the TensorFlow runtime, and in various visualization tools such as TensorBoard.

If `mark_as_used` is set to `True`, which is the default, a new unique name is created and marked as in use. If it's set to `False`, the unique name is returned without actually being marked as used. This is useful when the caller simply wants to know what the name to be created will be.

**Args:**

- `name` : The name for an operation.
- `mark_as_used` : Whether to mark this name as being used.

**Returns:**

A string to be passed to `create_op()` that will be used to name the operation being created.

---

**`tf.Graph.version`**

Returns a version number that increases as ops are added to the graph.

Note that this is unrelated to the [GraphDef version](#).

---

**`tf.Graph.graph_def_versions`**

The GraphDef version information of this graph.

For details on the meaning of each version, see [ [GraphDef](#) ] (<https://www.tensorflow.org/code/tensorflow/core/framework/graph.proto>).

**Returns:**

A `versionDef`.

---

**`tf.Graph.create_op(op_type, inputs, dtypes, input_types=None, name=None, attrs=None, op_def=None, compute_shapes=True, compute_device=True)`**

Creates an `Operation` in this graph.

This is a low-level interface for creating an `operation`. Most programs will not call this method directly, and instead use the Python op constructors, such as `tf.constant()`, which add ops to the default graph.

**Args:**

- `op_type` : The `Operation` type to create. This corresponds to the `OpDef.name` field for the proto that defines the operation.
- `inputs` : A list of `Tensor` objects that will be inputs to the `Operation`.
- `dtypes` : A list of `DType` objects that will be the types of the tensors that the operation produces.
- `input_types` : (Optional.) A list of `DType`s that will be the types of the tensors that the operation consumes. By default, uses the base `DType` of each input in `inputs`. Operations that expect reference-typed inputs must specify `input_types` explicitly.
- `name` : (Optional.) A string name for the operation. If not specified, a name is generated based on `op_type`.
- `attrs` : (Optional.) A dictionary where the key is the attribute name (a string) and the value is the respective `attr` attribute of the `NodeDef` proto that will represent the operation (an `AttrValue` proto).
- `op_def` : (Optional.) The `OpDef` proto that describes the `op_type` that the operation will have.
- `compute_shapes` : (Optional.) If True, shape inference will be performed to compute the shapes of the outputs.
- `compute_device` : (Optional.) If True, device functions will be executed to compute the device property of the Operation.

#### Raises:

- `TypeError` : if any of the inputs is not a `Tensor`.
- `ValueError` : if colocation conflicts with existing device assignment.

#### Returns:

An `Operation` object.

---

## `tf.Graph.gradient_override_map(op_type_map)`

EXPERIMENTAL: A context manager for overriding gradient functions.

This context manager can be used to override the gradient function that will be used for ops within the scope of the context.

For example:

```

@tf.RegisterGradient("CustomSquare")
def _custom_square_grad(op, grad):
 # ...

 with tf.Graph().as_default() as g:
 c = tf.constant(5.0)
 s_1 = tf.square(c) # Uses the default gradient for tf.square.
 with g.gradient_override_map({"Square": "CustomSquare"}):
 s_2 = tf.square(s_1) # Uses _custom_square_grad to compute the
 # gradient of s_2.

```

**Args:**

- `op_type_map` : A dictionary mapping op type strings to alternative op type strings.

**Returns:**

A context manager that sets the alternative op type to be used for one or more ops created in that context.

**Raises:**

- `TypeError` : If `op_type_map` is not a dictionary mapping strings to strings.

## Other Methods

---

### `tf.Graph.colocate_with(op, ignore_existing=False)`

Returns a context manager that specifies an op to colocate with.

Note: this function is not for public use, only for internal libraries.

For example:

```

a = tf.Variable([1.0])
with g.colocate_with(a):
 b = tf.constant(1.0)
 c = tf.add(a, b)

```

`b` and `c` will always be colocated with `a`, no matter where `a` is eventually placed.

**Args:**

- `op` : The op to colocate all created ops with.
- `ignore_existing` : If true, only applies colocation of this op within the context, rather

than applying all colocation properties on the stack.

**Raises:**

- `ValueError` : if op is None.

**Yields:**

A context manager that specifies the op with which to colocate newly created ops.

---

**`tf.Graph.get_all_collection_keys()`**

Returns a list of collections used in this graph.

---

**`tf.Graph.is_feedable(tensor)`**

Returns `True` if and only if `tensor` is feedable.

---

**`tf.Graph.is_fetchable(tensor_or_op)`**

Returns `True` if and only if `tensor_or_op` is fetchable.

---

**`tf.Graph.prevent_feeding(tensor)`**

Marks the given `tensor` as unfeedable in this graph.

---

**`tf.Graph.prevent_fetching(op)`**

Marks the given `op` as unfetchable in this graph.

---

**`class tf.Operation`**

Represents a graph node that performs computation on tensors.

An `Operation` is a node in a TensorFlow `Graph` that takes zero or more `Tensor` objects as input, and produces zero or more `Tensor` objects as output. Objects of type `Operation` are created by calling a Python op constructor (such as `tf.matmul()` ) or `Graph.create_op()` .

---

For example `c = tf.matmul(a, b)` creates an `Operation` of type "MatMul" that takes tensors `a` and `b` as input, and produces `c` as output.

After the graph has been launched in a session, an `operation` can be executed by passing it to `Session.run()`. `op.run()` is a shortcut for calling `tf.get_default_session().run(op)`.

---

### **tf.Operation.name**

The full name of this operation.

---

### **tf.Operation.type**

The type of the op (e.g. `"MatMul"`).

---

### **tf.Operation.inputs**

The list of `Tensor` objects representing the data inputs of this op.

---

### **tf.Operation.control\_inputs**

The `Operation` objects on which this op has a control dependency.

Before this op is executed, TensorFlow will ensure that the operations in `self.control_inputs` have finished executing. This mechanism can be used to run ops sequentially for performance reasons, or to ensure that the side effects of an op are observed in the correct order.

#### **Returns:**

A list of `Operation` objects.

---

### **tf.Operation.outputs**

The list of `Tensor` objects representing the outputs of this op.

---

### **tf.Operation.device**

The name of the device to which this op has been assigned, if any.

**Returns:**

The string name of the device to which this op has been assigned, or an empty string if it has not been assigned to a device.

---

**tf.Operation.graph**

The `Graph` that contains this operation.

---

**tf.Operation.run(feed\_dict=None, session=None)**

Runs this operation in a `Session`.

Calling this method will execute all preceding operations that produce the inputs needed for this operation.

*N.B.* Before invoking `operation.run()`, its graph must have been launched in a session, and either a default session must be available, or `session` must be specified explicitly.

**Args:**

- `feed_dict` : A dictionary that maps `Tensor` objects to feed values. See [`Session.run\(\)`](#) for a description of the valid feed values.
  - `session` : (Optional.) The `Session` to be used to run to this operation. If none, the default session will be used.
- 

**tf.Operation.get\_attr(name)**

Returns the value of the attr of this op with the given `name`.

**Args:**

- `name` : The name of the attr to fetch.

**Returns:**

The value of the attr, as a Python object.

**Raises:**

- `ValueError` : If this op does not have an attr with the given `name` .

## `tf.Operation.traceback`

Returns the call stack from when this operation was constructed.

## Other Methods

```
tf.Operation.__init__(node_def, g, inputs=None,
output_types=None, control_inputs=None, input_types=None,
original_op=None, op_def=None)
```

Creates an `Operation` .

NOTE: This constructor validates the name of the `operation` (passed as `node_def.name` ). Valid `operation` names match the following regular expression:

```
[A-Za-z0-9.][A-Za-z0-9_.\-/]*
```

### Args:

- `node_def` : `graph_pb2.NodeDef` . `NodeDef` for the `Operation` . Used for attributes of `graph_pb2.NodeDef` , typically `name` , `op` , and `device` . The `input` attribute is irrelevant here as it will be computed when generating the model.
- `g` : `Graph` . The parent graph.
- `inputs` : list of `Tensor` objects. The inputs to this `operation` .
- `output_types` : list of `DType` objects. List of the types of the `Tensors` computed by this operation. The length of this list indicates the number of output endpoints of the `Operation` .
- `control_inputs` : list of operations or tensors from which to have a control dependency.
- `input_types` : List of `DType` objects representing the types of the tensors accepted by the `operation` . By default uses `[x.dtype.base_dtype for x in inputs]` . Operations that expect reference-typed inputs must specify these explicitly.
- `original_op` : Optional. Used to associate the new `operation` with an existing `Operation` (for example, a replica with the op that was replicated).
- `op_def` : Optional. The `op_def_pb2.OpDef` proto that describes the op type that this `Operation` represents.

### Raises:

- `TypeError` : if control inputs are not Operations or Tensors, or if `node_def` is not a `NodeDef`, or if `g` is not a `Graph`, or if `inputs` are not tensors, or if `inputs` and `input_types` are incompatible.
- `ValueError` : if the `node_def` name is not valid.

---

## `tf.Operation.colocation_groups()`

Returns the list of colocation groups of the op.

---

## `tf.Operation.node_def`

Returns a serialized `NodeDef` representation of this operation.

**Returns:**

A `NodeDef` protocol buffer.

---

## `tf.Operation.op_def`

Returns the `OpDef` proto that represents the type of this op.

**Returns:**

An `OpDef` protocol buffer.

---

## `tf.Operation.values()`

DEPRECATED: Use `outputs`.

---

## `class tf.Tensor`

Represents a value produced by an `Operation`.

A `Tensor` is a symbolic handle to one of the outputs of an `operation`. It does not hold the values of that operation's output, but instead provides a means of computing those values in a TensorFlow `Session`.

This class has two primary purposes:

1. A `Tensor` can be passed as an input to another `Operation`. This builds a dataflow connection between operations, which enables TensorFlow to execute an entire `Graph` that represents a large, multi-step computation.
2. After the graph has been launched in a session, the value of the `Tensor` can be computed by passing it to `Session.run()`. `t.eval()` is a shortcut for calling `tf.get_default_session().run(t)`.

In the following example, `c`, `d`, and `e` are symbolic `Tensor` objects, whereas `result` is a numpy array that stores a concrete value:

```
Build a dataflow graph.
c = tf.constant([[1.0, 2.0], [3.0, 4.0]])
d = tf.constant([[1.0, 1.0], [0.0, 1.0]])
e = tf.matmul(c, d)

Construct a `Session` to execute the graph.
sess = tf.Session()

Execute the graph and store the value that `e` represents in `result`.
result = sess.run(e)
```

---

## tf.Tensor.dtype

The `DType` of elements in this tensor.

---

## tf.Tensor.name

The string name of this tensor.

---

## tf.Tensor.value\_index

The index of this tensor in the outputs of its `Operation`.

---

## tf.Tensor.graph

The `Graph` that contains this tensor.

---

## tf.Tensor.op

The `Operation` that produces this tensor as an output.

## tf.Tensor.consumers()

Returns a list of `Operation`s that consume this tensor.

### Returns:

A list of `Operation`s.

## tf.Tensor.eval(feed\_dict=None, session=None)

Evaluates this tensor in a `Session`.

Calling this method will execute all preceding operations that produce the inputs needed for the operation that produces this tensor.

*N.B.* Before invoking `Tensor.eval()`, its graph must have been launched in a session, and either a default session must be available, or `session` must be specified explicitly.

### Args:

- `feed_dict` : A dictionary that maps `Tensor` objects to feed values. See `Session.run()` for a description of the valid feed values.
- `session` : (Optional.) The `Session` to be used to evaluate this tensor. If none, the default session will be used.

### Returns:

A numpy array corresponding to the value of this tensor.

## tf.Tensor.get\_shape()

Returns the `TensorShape` that represents the shape of this tensor.

The shape is computed using shape inference functions that are registered for each `Operation` type using `tf.RegisterShape`. See `TensorShape` for more details of what a shape represents.

The inferred shape of a tensor is used to provide shape information without having to launch the graph in a session. This can be used for debugging, and providing early error messages. For example:

```
c = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])

print(c.get_shape())
==> TensorShape([Dimension(2), Dimension(3)])

d = tf.constant([[1.0, 0.0], [0.0, 1.0], [1.0, 0.0], [0.0, 1.0]])

print(d.get_shape())
==> TensorShape([Dimension(4), Dimension(2)])

Raises a ValueError, because `c` and `d` do not have compatible
inner dimensions.
e = tf.matmul(c, d)

f = tf.matmul(c, d, transpose_a=True, transpose_b=True)

print(f.get_shape())
==> TensorShape([Dimension(3), Dimension(4)])
```

In some cases, the inferred shape may have unknown dimensions. If the caller has additional information about the values of these dimensions, `Tensor.set_shape()` can be used to augment the inferred shape.

### Returns:

A `TensorShape` representing the shape of this tensor.

---

### `tf.Tensor.set_shape(shape)`

Updates the shape of this tensor.

This method can be called multiple times, and will merge the given `shape` with the current shape of this tensor. It can be used to provide additional information about the shape of this tensor that cannot be inferred from the graph alone. For example, this can be used to provide additional information about the shapes of images:

```

_, image_data = tf.TFRecordReader(...).read(...)
image = tf.image.decode_png(image_data, channels=3)

The height and width dimensions of `image` are data dependent, and
cannot be computed without executing the op.
print(image.get_shape())
==> TensorShape([Dimension(None), Dimension(None), Dimension(3)])

We know that each image in this dataset is 28 x 28 pixels.
image.set_shape([28, 28, 3])
print(image.get_shape())
==> TensorShape([Dimension(28), Dimension(28), Dimension(3)])

```

**Args:**

- `shape` : A `TensorShape` representing the shape of this tensor.

**Raises:**

- `ValueError` : If `shape` is not compatible with the current shape of this tensor.

## Other Methods

**`tf.Tensor.__init__(op, value_index, dtype)`**

Creates a new `Tensor`.

**Args:**

- `op` : An `Operation`. `Operation` that computes this tensor.
- `value_index` : An `int`. Index of the operation's endpoint that produces this tensor.
- `dtype` : A `DTType`. Type of elements stored in this tensor.

**Raises:**

- `TypeError` : If the op is not an `operation`.

**`tf.Tensor.device`**

The name of the device on which this tensor will be produced, or `None`.

## Tensor types

## class tf.DType

Represents the type of the elements in a `Tensor`.

The following `DType` objects are defined:

- `tf.float16` : 16-bit half-precision floating-point.
- `tf.float32` : 32-bit single-precision floating-point.
- `tf.float64` : 64-bit double-precision floating-point.
- `tf.bfloat16` : 16-bit truncated floating-point.
- `tf.complex64` : 64-bit single-precision complex.
- `tf.complex128` : 128-bit double-precision complex.
- `tf.int8` : 8-bit signed integer.
- `tf.uint8` : 8-bit unsigned integer.
- `tf.uint16` : 16-bit unsigned integer.
- `tf.int16` : 16-bit signed integer.
- `tf.int32` : 32-bit signed integer.
- `tf.int64` : 64-bit signed integer.
- `tf.bool` : Boolean.
- `tf.string` : String.
- `tf.qint8` : Quantized 8-bit signed integer.
- `tf.quint8` : Quantized 8-bit unsigned integer.
- `tf.qint16` : Quantized 16-bit signed integer.
- `tf.quint16` : Quantized 16-bit unsigned integer.
- `tf.qint32` : Quantized 32-bit signed integer.

In addition, variants of these types with the `_ref` suffix are defined for reference-typed tensors.

The `tf.as_dtype()` function converts numpy types and string type names to a `DType` object.

### `tf.DType.is_compatible_with(other)`

Returns True if the `other` `DType` will be converted to this `DType`.

The conversion rules are as follows:

```
DTType(T).is_compatible_with(DTType(T)) == True
DTType(T).is_compatible_with(DTType(T).as_ref) == True
DTType(T).as_ref.is_compatible_with(DTType(T)) == False
DTType(T).as_ref.is_compatible_with(DTType(T).as_ref) == True
```

**Args:**

- `other` : A `DTType` (or object that may be converted to a `DTType`).

**Returns:**

True if a Tensor of the `other` `DTType` will be implicitly converted to this `DTType`.

**`tf.DType.name`**

Returns the string name for this `DTType`.

**`tf.DType.base_dtype`**

Returns a non-reference `DTType` based on this `DTType`.

**`tf.DType.real_dtype`**

Returns the dtype correspond to this dtype's real part.

**`tf.DType.is_ref_dtype`**

Returns `True` if this `DTType` represents a reference type.

**`tf.DType.as_ref`**

Returns a reference `DTType` based on this `DTType`.

**`tf.DType.is_floating`**

Returns whether this is a (real) floating point type.

### **tf.DType.is\_complex**

Returns whether this is a complex floating point type.

---

### **tf.DType.is\_integer**

Returns whether this is a (non-quantized) integer type.

---

### **tf.DType.is\_quantized**

Returns whether this is a quantized data type.

---

### **tf.DType.is\_unsigned**

Returns whether this type is unsigned.

Non-numeric, unordered, and quantized types are not considered unsigned, and this function returns `False`.

#### **Returns:**

Whether a `DType` is unsigned.

---

### **tf.DType.as\_numpy\_dtype**

Returns a `numpy.dtype` based on this `DType`.

---

### **tf.DType.as\_datatype\_enum**

Returns a `types_pb2.DataType` enum value based on this `DType`.

---

## **Other Methods**

---

### **tf.DType.\_\_init\_\_(type\_enum)**

Creates a new `(DataType)`.

NOTE(mrry): In normal circumstances, you should not need to construct a `DataType` object directly. Instead, use the `tf.as_dtype()` function.

**Args:**

- `type_enum` : A `types_pb2.DataType` enum value.

**Raises:**

- `TypeError` : If `type_enum` is not a value `types_pb2.DataType`.
- 

## `tf.DType.max`

Returns the maximum representable value in this data type.

**Raises:**

- `TypeError` : if this is a non-numeric, unordered, or quantized type.
- 

## `tf.DType.min`

Returns the minimum representable value in this data type.

**Raises:**

- `TypeError` : if this is a non-numeric, unordered, or quantized type.
- 

## `tf.DType.size`

---

## `tf.as_dtype(type_value)`

Converts the given `type_value` to a `DType`.

**Args:**

- `type_value` : A value that can be converted to a `tf.DType` object. This may currently be a `tf.DType` object, a `DataType` enum, a string type name, or a `numpy.dtype`.

**Returns:**

---

A `dtype` corresponding to `type_value`.

**Raises:**

- `TypeError` : If `type_value` cannot be converted to a `DType`.

## Utility functions

---

### `tf.device(device_name_or_function)`

Wrapper for `Graph.device()` using the default graph.

See [Graph.device\(\)](#) for more details.

**Args:**

- `device_name_or_function` : The device name or function to use in the context.

**Returns:**

A context manager that specifies the default device to use for newly created ops.

---

### `tf.name_scope(name)`

Wrapper for `Graph.name_scope()` using the default graph.

See [Graph.name\\_scope\(\)](#) for more details.

**Args:**

- `name` : A name for the scope.

**Returns:**

A context manager that installs `name` as a new name scope in the default graph.

---

### `tf.control_dependencies(control_inputs)`

Wrapper for `Graph.control_dependencies()` using the default graph.

See [Graph.control\\_dependencies\(\)](#) for more details.

**Args:**

- `control_inputs` : A list of `Operation` or `Tensor` objects which must be executed or computed before running the operations defined in the context. Can also be `None` to clear the control dependencies.

**Returns:**

A context manager that specifies control dependencies for all operations constructed within the context.

## `tf.convert_to_tensor(value, dtype=None, name=None, as_ref=False)`

Converts the given `value` to a `Tensor`.

This function converts Python objects of various types to `Tensor` objects. It accepts `Tensor` objects, numpy arrays, Python lists, and Python scalars. For example:

```
import numpy as np

def my_func(arg):
 arg = tf.convert_to_tensor(arg, dtype=tf.float32)
 return tf.matmul(arg, arg) + arg

The following calls are equivalent.
value_1 = my_func(tf.constant([[1.0, 2.0], [3.0, 4.0]]))
value_2 = my_func([[1.0, 2.0], [3.0, 4.0]])
value_3 = my_func(np.array([[1.0, 2.0], [3.0, 4.0]]), dtype=np.float32)
```

This function can be useful when composing a new operation in Python (such as `my_func` in the example above). All standard Python op constructors apply this function to each of their Tensor-valued inputs, which allows those ops to accept numpy arrays, Python lists, and scalars in addition to `Tensor` objects.

**Args:**

- `value` : An object whose type has a registered `Tensor` conversion function.
- `dtype` : Optional element type for the returned tensor. If missing, the type is inferred from the type of `value`.
- `name` : Optional name to use if a new `Tensor` is created.
- `as_ref` : True if we want the result as a ref tensor. Only used if a new `Tensor` is created.

**Returns:**

A `Tensor` based on `value`.

**Raises:**

- `TypeError` : If no conversion function is registered for `value`.
- `RuntimeError` : If a registered conversion function returns an invalid value.

**`tf.convert_to_tensor_or_indexed_slices(value, dtype=None, name=None, as_ref=False)`**

Converts the given object to a `Tensor` or an `IndexedSlices`.

If `value` is an `IndexedSlices` or `SparseTensor` it is returned unmodified. Otherwise, it is converted to a `Tensor` using `convert_to_tensor()`.

**Args:**

- `value` : An `IndexedSlices`, `SparseTensor`, or an object that can be consumed by `convert_to_tensor()`.
- `dtype` : (Optional.) The required `dtype` of the returned `Tensor` or `IndexedSlices`.
- `name` : (Optional.) A name to use if a new `Tensor` is created.
- `as_ref` : True if the caller wants the results as ref tensors.

**Returns:**

An `Tensor`, `IndexedSlices`, or `SparseTensor` based on `value`.

**Raises:**

- `ValueError` : If `dtype` does not match the element type of `value`.

**`tf.get_default_graph()`**

Returns the default graph for the current thread.

The returned graph will be the innermost graph on which a `Graph.as_default()` context has been entered, or a global default graph if none has been explicitly created.

NOTE: The default graph is a property of the current thread. If you create a new thread, and wish to use the default graph in that thread, you must explicitly add a `with g.as_default():` in that thread's function.

**Returns:**

The default `Graph` being used in the current thread.

**`tf.reset_default_graph()`**

Clears the default graph stack and resets the global default graph.

NOTE: The default graph is a property of the current thread. This function applies only to the current thread. Calling this function while a `tf.Session` or `tf.InteractiveSession` is active will result in undefined behavior. Using any previously created `tf.Operation` or `tf.Tensor` objects after calling this function will result in undefined behavior.

**`tf.import_graph_def(graph_def, input_map=None, return_elements=None, name=None, op_dict=None, producer_op_list=None)`**

Imports the TensorFlow graph in `graph_def` into the Python `Graph`.

This function provides a way to import a serialized TensorFlow `GraphDef` protocol buffer, and extract individual objects in the `GraphDef` as `Tensor` and `Operation` objects. See `Graph.as_graph_def()` for a way to create a `GraphDef` proto.

**Args:**

- `graph_def` : A `GraphDef` proto containing operations to be imported into the default graph.
- `input_map` : A dictionary mapping input names (as strings) in `graph_def` to `Tensor` objects. The values of the named input tensors in the imported graph will be re-mapped to the respective `Tensor` values.
- `return_elements` : A list of strings containing operation names in `graph_def` that will be returned as `Operation` objects; and/or tensor names in `graph_def` that will be returned as `Tensor` objects.
- `name` : (Optional.) A prefix that will be prepended to the names in `graph_def`. Defaults to "import".
- `op_dict` : (Optional.) A dictionary mapping op type names to `OpDef` protos. Must contain an `OpDef` proto for each op type named in `graph_def`. If omitted, uses the `OpDef` protos registered in the global registry.
- `producer_op_list` : (Optional.) An `OpList` proto with the (possibly stripped) list of `OpDef`s used by the producer of the graph. If provided, attrs for ops in `graph_def` that

are not in `op_dict` that have their default value according to `producer_op_list` will be removed. This will allow some more `GraphDef`s produced by later binaries to be accepted by earlier binaries.

**Returns:**

A list of `Operation` and/or `Tensor` objects from the imported graph, corresponding to the names in `return_elements`.

**Raises:**

- `TypeError` : If `graph_def` is not a `GraphDef` proto, `input_map` is not a dictionary mapping strings to `Tensor` objects, or `return_elements` is not a list of strings.
  - `ValueError` : If `input_map`, or `return_elements` contains names that do not appear in `graph_def`, or `graph_def` is not well-formed (e.g. it refers to an unknown tensor).
- 

## **tf.load\_file\_system\_library(library\_filename)**

Loads a TensorFlow plugin, containing file system implementation.

Pass `library_filename` to a platform-specific mechanism for dynamically loading a library. The rules for determining the exact location of the library are platform-specific and are not documented here.

**Args:**

- `library_filename` : Path to the plugin. Relative or absolute filesystem path to a dynamic library file.

**Returns:**

None.

**Raises:**

- `RuntimeError` : when unable to load the library.
- 

## **tf.load\_op\_library(library\_filename)**

Loads a TensorFlow plugin, containing custom ops and kernels.

Pass "library\_filename" to a platform-specific mechanism for dynamically loading a library. The rules for determining the exact location of the library are platform-specific and are not documented here.

**Args:**

- `library_filename` : Path to the plugin. Relative or absolute filesystem path to a dynamic library file.

**Returns:**

A python module containing the Python wrappers for Ops defined in the plugin.

**Raises:**

- `RuntimeError` : when unable to load the library or get the python wrappers.

## Graph collections

---

### `tf.add_to_collection(name, value)`

Wrapper for `Graph.add_to_collection()` using the default graph.

See [Graph.add\\_to\\_collection\(\)](#) for more details.

**Args:**

- `name` : The key for the collection. For example, the `GraphKeys` class contains many standard names for collections.
- `value` : The value to add to the collection.

### `tf.get_collection(key, scope=None)`

Wrapper for `Graph.get_collection()` using the default graph.

See [Graph.get\\_collection\(\)](#) for more details.

**Args:**

- `key` : The key for the collection. For example, the `GraphKeys` class contains many standard names for collections.
- `scope` : (Optional.) If supplied, the resulting list is filtered to include only items whose

---

`name` attribute matches using `re.match`. Items without a `name` attribute are never returned if a scope is supplied and the choice or `re.match` means that a `scope` without special tokens filters by prefix.

**Returns:**

The list of values in the collection with the given `name`, or an empty list if no value has been added to that collection. The list contains the values in the order under which they were collected.

---

## **tf.get\_collection\_ref(key)**

Wrapper for `Graph.get_collection_ref()` using the default graph.

See [`Graph.get\_collection\_ref\(\)`](#) for more details.

**Args:**

- `key` : The key for the collection. For example, the `GraphKeys` class contains many standard names for collections.

**Returns:**

The list of values in the collection with the given `name`, or an empty list if no value has been added to that collection. Note that this returns the collection list itself, which can be modified in place to change the collection.

---

## **class tf.GraphKeys**

Standard names to use for graph collections.

The standard library uses various well-known names to collect and retrieve values associated with a graph. For example, the `tf.optimizer` subclasses default to optimizing the variables collected under `tf.GraphKeys.TRAINABLE_VARIABLES` if none is specified, but it is also possible to pass an explicit list of variables.

The following standard keys are defined:

- `VARIABLES` : the `Variable` objects that comprise a model, and must be saved and restored together. See [`tf.all\_variables\(\)`](#) for more details.
- `TRAINABLE_VARIABLES` : the subset of `Variable` objects that will be trained by an optimizer. See [`tf.trainable\_variables\(\)`](#) for more details.

- `SUMMARIES` : the summary `Tensor` objects that have been created in the graph. See `tf.merge_all_summaries()` for more details.
- `QUEUE_RUNNERS` : the `QueueRunner` objects that are used to produce input for a computation. See `tf.start_queue_runners()` for more details.
- `MOVING_AVERAGE_VARIABLES` : the subset of `Variable` objects that will also keep moving averages. See `tf.moving_average_variables()` for more details.
- `REGULARIZATION_LOSSES` : regularization losses collected during graph construction.
- `WEIGHTS` : weights inside neural network layers
- `BIASES` : biases inside neural network layers
- `ACTIVATIONS` : activations of neural network layers

## Defining new operations

### `class tf.RegisterGradient`

A decorator for registering the gradient function for an op type.

This decorator is only used when defining a new op type. For an op with `m` inputs and `n` outputs, the gradient function is a function that takes the original `operation` and `n` `Tensor` objects (representing the gradients with respect to each output of the op), and returns `m` `Tensor` objects (representing the partial gradients with respect to each input of the op).

For example, assuming that operations of type `"Sub"` take two inputs `x` and `y`, and return a single output `x - y`, the following gradient function would be registered:

```
@tf.RegisterGradient("Sub")
def _sub_grad(unused_op, grad):
 return grad, tf.neg(grad)
```

The decorator argument `op_type` is the string type of an operation. This corresponds to the `opDef.name` field for the proto that defines the operation.

### `tf.RegisterGradient.__init__(op_type)`

Creates a new decorator with `op_type` as the Operation type.

#### Args:

- `op_type` : The string type of an operation. This corresponds to the `opDef.name` field for

---

the proto that defines the operation.

---

## **tf.NoGradient(op\_type)**

Specifies that ops of type `op_type` do not have a defined gradient.

This function is only used when defining a new op type. It may be used for ops such as `tf.size()` that are not differentiable. For example:

```
tf.NoGradient("Size")
```

### Args:

- `op_type` : The string type of an operation. This corresponds to the `OpDef.name` field for the proto that defines the operation.

### Raises:

- `TypeError` : If `op_type` is not a string.
- 

## **class tf.RegisterShape**

A decorator for registering the shape function for an op type.

This decorator is only used when defining a new op type. A shape function is a function from an `Operation` object to a list of `TensorShape` objects, with one `TensorShape` for each output of the operation.

For example, assuming that operations of type `"Sub"` take two inputs `x` and `y`, and return a single output `x - y`, all with the same shape, the following shape function would be registered:

```
@tf.RegisterShape("Sub")
def _sub_shape(op):
 return [op.inputs[0].get_shape().merge_with(op.inputs[1].get_shape())]
```

---

The decorator argument `op_type` is the string type of an operation. This corresponds to the `OpDef.name` field for the proto that defines the operation.

---

## **tf.RegisterShape.\_\_init\_\_(op\_type)**

Saves the `op_type` as the `operation type`.

---

## class `tf.TensorShape`

Represents the shape of a `Tensor`.

A `TensorShape` represents a possibly-partial shape specification for a `Tensor`. It may be one of the following:

- *Fully-known shape*: has a known number of dimensions and a known size for each dimension.
- *Partially-known shape*: has a known number of dimensions, and an unknown size for one or more dimension.
- *Unknown shape*: has an unknown number of dimensions, and an unknown size in all dimensions.

If a tensor is produced by an operation of type `"Foo"`, its shape may be inferred if there is a registered shape function for `"Foo"`. See [`tf.RegisterShape\(\)`](#) for details of shape functions and how to register them. Alternatively, the shape may be set explicitly using

[`Tensor.set\_shape\(\)`](#).

---

## `tf.TensorShape.merge_with(other)`

Returns a `TensorShape` combining the information in `self` and `other`.

The dimensions in `self` and `other` are merged elementwise, according to the rules defined for `Dimension.merge_with()`.

### Args:

- `other` : Another `TensorShape`.

### Returns:

A `TensorShape` containing the combined information of `self` and `other`.

### Raises:

- `ValueError` : If `self` and `other` are not compatible.

---

## `tf.TensorShape.concatenate(other)`

Returns the concatenation of the dimension in `self` and `other`.

*N.B.* If either `self` or `other` is completely unknown, concatenation will discard information about the other shape. In future, we might support concatenation that preserves this information for use with slicing.

**Args:**

- `other` : Another `TensorShape`.

**Returns:**

A `TensorShape` whose dimensions are the concatenation of the dimensions in `self` and `other`.

---

**`tf.TensorShape.ndims`**

Returns the rank of this shape, or None if it is unspecified.

---

**`tf.TensorShape.dims`**

Returns a list of Dimensions, or None if the shape is unspecified.

---

**`tf.TensorShape.as_list()`**

Returns a list of integers or None for each dimension.

**Returns:**

A list of integers or None for each dimension.

---

**`tf.TensorShape.as_proto()`**

Returns this shape as a `TensorShapeProto`.

---

**`tf.TensorShape.is_compatible_with(other)`**

Returns True iff `self` is compatible with `other`.

Two possibly-partially-defined shapes are compatible if there exists a fully-defined shape that both shapes can represent. Thus, compatibility allows the shape inference code to reason about partially-defined shapes. For example:

- `TensorShape(None)` is compatible with all shapes.
- `TensorShape([None, None])` is compatible with all two-dimensional shapes, such as `TensorShape([32, 784])`, and also `TensorShape(None)`. It is not compatible with, for example, `TensorShape([None])` or `TensorShape([None, None, None])`.
- `TensorShape([32, None])` is compatible with all two-dimensional shapes with size 32 in the 0th dimension, and also `TensorShape([None, None])` and `TensorShape(None)`. It is not compatible with, for example, `TensorShape([32])`, `TensorShape([32, None, 1])` or `TensorShape([64, None])`.
- `TensorShape([32, 784])` is compatible with itself, and also `TensorShape([32, None])`, `TensorShape([None, 784])`, `TensorShape([None, None])` and `TensorShape(None)`. It is not compatible with, for example, `TensorShape([32, 1, 784])` or `TensorShape([None])`.

The compatibility relation is reflexive and symmetric, but not transitive. For example, `TensorShape([32, 784])` is compatible with `TensorShape(None)`, and `TensorShape(None)` is compatible with `TensorShape([4, 4])`, but `TensorShape([32, 784])` is not compatible with `TensorShape([4, 4])`.

#### Args:

- `other` : Another `TensorShape`.

#### Returns:

True iff `self` is compatible with `other`.

---

### `tf.TensorShape.is_fully_defined()`

Returns True iff `self` is fully defined in every dimension.

---

### `tf.TensorShape.with_rank(rank)`

Returns a shape based on `self` with the given rank.

This method promotes a completely unknown shape to one with a known rank.

#### Args:

- `rank` : An integer.

**Returns:**

A shape that is at least as specific as `self` with the given rank.

**Raises:**

- `ValueError` : If `self` does not represent a shape with the given `rank`.
- 

**`tf.TensorShape.with_rank_at_least(rank)`**

Returns a shape based on `self` with at least the given rank.

**Args:**

- `rank` : An integer.

**Returns:**

A shape that is at least as specific as `self` with at least the given rank.

**Raises:**

- `ValueError` : If `self` does not represent a shape with at least the given `rank`.
- 

**`tf.TensorShape.with_rank_at_most(rank)`**

Returns a shape based on `self` with at most the given rank.

**Args:**

- `rank` : An integer.

**Returns:**

A shape that is at least as specific as `self` with at most the given rank.

**Raises:**

- `ValueError` : If `self` does not represent a shape with at most the given `rank`.
- 

**`tf.TensorShape.assert_has_rank(rank)`**

Raises an exception if `self` is not compatible with the given `rank`.

**Args:**

- `rank` : An integer.

**Raises:**

- `ValueError` : If `self` does not represent a shape with the given `rank`.

**`tf.TensorShape.assert_same_rank(other)`**

Raises an exception if `self` and `other` do not have compatible ranks.

**Args:**

- `other` : Another `TensorShape`.

**Raises:**

- `ValueError` : If `self` and `other` do not represent shapes with the same rank.

**`tf.TensorShape.assert_is_compatible_with(other)`**

Raises exception if `self` and `other` do not represent the same shape.

This method can be used to assert that there exists a shape that both `self` and `other` represent.

**Args:**

- `other` : Another `TensorShape`.

**Raises:**

- `ValueError` : If `self` and `other` do not represent the same shape.

**`tf.TensorShape.assert_is_fully_defined()`**

Raises an exception if `self` is not fully defined in every dimension.

**Raises:**

- `ValueError` : If `self` does not have a known value for every dimension.

## Other Methods

---

### `tf.TensorShape.__init__(dims)`

Creates a new TensorShape with the given dimensions.

#### Args:

- `dims` : A list of Dimensions, or None if the shape is unspecified.
- `DEPRECATED` : A single integer is treated as a singleton list.

#### Raises:

- `TypeError` : If dims cannot be converted to a list of dimensions.

### `tf.TensorShape.num_elements()`

Returns the total number of elements, or none for incomplete shapes.

---

## `class tf.Dimension`

Represents the value of one dimension in a TensorShape.

---

### `tf.Dimension.__init__(value)`

Creates a new Dimension with the given value.

---

### `tf.Dimension.assert_is_compatible_with(other)`

Raises an exception if `other` is not compatible with this Dimension.

#### Args:

- `other` : Another Dimension.

#### Raises:

- `ValueError` : If `self` and `other` are not compatible (see `is_compatible_with`).
- 

## `tf.Dimension.is_compatible_with(other)`

Returns true if `other` is compatible with this Dimension.

Two known Dimensions are compatible if they have the same value. An unknown Dimension is compatible with all other Dimensions.

### Args:

- `other` : Another Dimension.

### Returns:

True if this Dimension and `other` are compatible.

---

## `tf.Dimension.merge_with(other)`

Returns a Dimension that combines the information in `self` and `other`.

Dimensions are combined as follows:

```
Dimension(n) .merge_with(Dimension(n)) == Dimension(n)
Dimension(n) .merge_with(Dimension(None)) == Dimension(n)
Dimension(None).merge_with(Dimension(n)) == Dimension(n)
Dimension(None).merge_with(Dimension(None)) == Dimension(None)
Dimension(n) .merge_with(Dimension(m)) raises ValueError for n != m
```

### Args:

- `other` : Another Dimension.

### Returns:

A Dimension containing the combined information of `self` and `other`.

### Raises:

- `ValueError` : If `self` and `other` are not compatible (see `is_compatible_with`).
- 

## `tf.Dimension.value`

The value of this dimension, or `None` if it is unknown.

## `tf.op_scope(values, name, default_name=None)`

Returns a context manager for use when defining a Python op.

This context manager validates that the given `values` are from the same graph, ensures that graph is the default graph, and pushes a name scope.

For example, to define a new Python op called `my_op` :

```
def my_op(a, b, c, name=None):
 with tf.op_scope([a, b, c], name, "MyOp") as scope:
 a = tf.convert_to_tensor(a, name="a")
 b = tf.convert_to_tensor(b, name="b")
 c = tf.convert_to_tensor(c, name="c")
 # Define some computation that uses `a`, `b`, and `c`.
 return foo_op(..., name=scope)
```

### Args:

- `values` : The list of `Tensor` arguments that are passed to the op function.
- `name` : The name argument that is passed to the op function.
- `default_name` : The default name to use if the `name` argument is `None`.

### Returns:

A context manager for use in defining Python ops. Yields the name scope.

### Raises:

- `ValueError` : if neither `name` nor `default_name` is provided.

## `tf.get_seed(op_seed)`

Returns the local seeds an operation should use given an op-specific seed.

Given operation-specific seed, `op_seed`, this helper function returns two seeds derived from graph-level and op-level seeds. Many random operations internally use the two seeds to allow user to change the seed globally for a graph, or for only specific operations.

For details on how the graph-level seed interacts with op seeds, see [set\\_random\\_seed](#).

**Args:**

- `op_seed` : integer.

**Returns:**

A tuple of two integers that should be used for the local seed of this operation.

## For libraries building on TensorFlow

### `tf.register_tensor_conversion_function(base_type, conversion_func, priority=100)`

Registers a function for converting objects of `base_type` to `Tensor`.

The conversion function must have the following signature:

```
def conversion_func(value, dtype=None, name=None, as_ref=False):
 # ...
```

It must return a `Tensor` with the given `dtype` if specified. If the conversion function creates a new `Tensor`, it should use the given `name` if specified. All exceptions will be propagated to the caller.

The conversion function may return `NotImplemented` for some inputs. In this case, the conversion process will continue to try subsequent conversion functions.

If `as_ref` is true, the function must return a `Tensor` reference, such as a `Variable`.

NOTE: The conversion functions will execute in order of priority, followed by order of registration. To ensure that a conversion function `F` runs before another conversion function `G`, ensure that `F` is registered with a smaller priority than `G`.

**Args:**

- `base_type` : The base type or tuple of base types for all objects that `conversion_func` accepts.
- `conversion_func` : A function that converts instances of `base_type` to `Tensor`.
- `priority` : Optional integer that indicates the priority for applying this conversion function. Conversion functions with smaller priority values run earlier than conversion functions with larger priority values. Defaults to 100.

**Raises:**

- `TypeError` : If the arguments do not have the appropriate type.

## Other Functions and Classes

---

### `class tf.DeviceSpec`

Represents a (possibly partial) specification for a TensorFlow device.

`DeviceSpec`s are used throughout TensorFlow to describe where state is stored and computations occur. Using `DeviceSpec` allows you to parse device spec strings to verify their validity, merge them or compose them programmatically.

Example:

```
Place the operations on device "GPU:0" in the "ps" job.
device_spec = DeviceSpec(job="ps", device_type="GPU", device_index=0)
with tf.device(device_spec):
 # Both my_var and squared_var will be placed on /job:ps/device:GPU:0.
 my_var = tf.Variable(..., name="my_variable")
 squared_var = tf.square(my_var)
```

If a `DeviceSpec` is partially specified, it will be merged with other `DeviceSpec`s according to the scope in which it is defined. `DeviceSpec` components defined in inner scopes take precedence over those defined in outer scopes.

```
with tf.device(DeviceSpec(job="train",)):
 with tf.device(DeviceSpec(job="ps", device_type="GPU", device_index=0)):
 # Nodes created here will be assigned to /job:ps/device:GPU:0.
 with tf.device(DeviceSpec(device_type="GPU", device_index=1)):
 # Nodes created here will be assigned to /job:train/device:GPU:1.
```

A `DeviceSpec` consists of 5 components -- each of which is optionally specified:

- Job: The job name.
- Replica: The replica index.
- Task: The task index.
- Device type: The device type string (e.g. "CPU" or "GPU").
- Device index: The device index.

---

```
tf.DeviceSpec.__init__(job=None, replica=None, task=None,
device_type=None, device_index=None)
```

Create a new `DeviceSpec` object.

#### Args:

- `job` : string. Optional job name.
  - `replica` : int. Optional replica index.
  - `task` : int. Optional task index.
  - `device_type` : Optional device type string (e.g. "CPU" or "GPU")
  - `device_index` : int. Optional device index. If left unspecified, device represents 'any' device\_index.
- 

```
tf.DeviceSpec.from_string(spec)
```

Construct a `DeviceSpec` from a string.

#### Args:

- `spec` : a string of the form /job:/replica:/task:/device:CPU: or /job:/replica:/task:/device:GPU: as cpu and gpu are mutually exclusive. All entries are optional.

#### Returns:

A `DeviceSpec`.

---

```
tf.DeviceSpec.job
```

---

```
tf.DeviceSpec.merge_from(dev)
```

Merge the properties of "dev" into this `DeviceSpec`.

#### Args:

- `dev` : a `DeviceSpec`.
- 

```
tf.DeviceSpec.parse_from_string(spec)
```

Parse a `DeviceSpec` name into its components.

**Args:**

- `spec` : a string of the form `/job:/replica:/task:/device:CPU:` or `/job:/replica:/task:/device:GPU:` as cpu and gpu are mutually exclusive. All entries are optional.

**Returns:**

The `DeviceSpec`.

**Raises:**

- `ValueError` : if the spec was not valid.
- 

**`tf.DeviceSpec.replica`**

---

**`tf.DeviceSpec.task`**

---

**`tf.DeviceSpec.to_string()`**

Return a string representation of this `DeviceSpec`.

**Returns:**

a string of the form `/job:/replica:/task:/device::`.

---

**`class tf.bytes`**

`str(object="") -> string`

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

# 상수, 시퀀스, 그리고 난수

참고 : `Tensor` 를 인자로 받는 함수들은 `tf.convert_to_tensor` 의 인자로 들어갈 수 있는 값들 또한 받을 수 있습니다.

[TOC]

## 상수값 텐서

TensorFlow는 상수를 생성할 수 있는 몇가지 연산을 제공합니다.

---

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

모든 원소의 값이 0인 텐서를 생성합니다.

이 연산은 모든 원소의 값이 0이고, `shape` `shape`을 가진 `dtype` 타입의 텐서를 반환합니다.

예시:

```
tf.zeros([3, 4], int32) ==> [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

인자:

- `shape` : 정수 리스트 또는 `int32` 타입의 1-D(1-Dimension) `Tensor`.
- `dtype` : 반환되는 `Tensor` 의 원소 타입.
- `name` : 연산의 명칭 (선택사항).

반환값:

모든 원소의 값이 0인 `Tensor`.

---

### `tf.zeros_like(tensor, dtype=None, name=None)`

모든 원소의 값이 0인 텐서를 생성합니다.

하나의 텐서(`tensor`)가 주어졌을 때, 이 연산은 모든 원소의 값이 0이고 `tensor` 와 같은 타입과 `shape`을 가진 텐서를 반환합니다. 선택적으로, `dtype` 을 사용해서 새로운 타입을 지정할 수도 있습니다.

예시:

```
'tensor' is [[1, 2, 3], [4, 5, 6]]
tf.zeros_like(tensor) ==> [[0, 0, 0], [0, 0, 0]]
```

인자:

- **tensor** : 하나의 `Tensor` .
- **dtype** : 반환되는 `Tensor` 의 타입. `float32`, `float64`, `int8`, `int16`, `int32`, `int64`, `uint8`, `complex64`, or `complex128` 만 가능합니다.
- **name** : 연산의 명칭 (선택사항).

반환값:

모든 원소의 값이 0인 `Tensor` .

---

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

모든 원소의 값이 1인 텐서를 생성합니다.

이 연산은 모든 원소의 값이 1이고, `shape` `shape`을 가진 `dtype` 타입의 텐서를 반환합니다.

예시:

```
tf.ones([2, 3], int32) ==> [[1, 1, 1], [1, 1, 1]]
```

인자:

- **shape** : 정수 리스트 또는 `int32` 타입의 1-D(1-Dimension) `Tensor` .
- **dtype** : 반환되는 `Tensor` 의 원소 타입.
- **name** : 연산의 명칭 (선택사항).

반환값:

모든 원소의 값이 1인 `Tensor` .

---

## **tf.ones\_like(tensor, dtype=None, name=None)**

모든 원소의 값이 1인 텐서를 생성합니다.

하나의 텐서( `tensor` )가 주어졌을 때, 이 연산은 모든 원소의 값이 1이고 `tensor` 와 같은 타입과 `shape`을 가진 텐서를 반환합니다. 선택적으로, `dtype` 을 사용해서 새로운 타입을 지정할 수도 있습니다.

예시:

```
'tensor' is [[1, 2, 3], [4, 5, 6]]
tf.zeros_like(tensor) ==> [[1, 1, 1], [1, 1, 1]]
```

인자:

- `tensor` : 하나의 `Tensor` .
- `dtype` : 반환되는 `Tensor` 의 타입. `float32` , `float64` , `int8` , `int16` , `int32` , `int64` , `uint8` , `complex64` , or `complex128` 만 가능합니다.
- `name` : 연산의 명칭 (선택사항).

반환값:

모든 원소의 값이 1인 `Tensor` .

---

## `tf.fill(dims, value, name=None)`

스칼라값으로 채워진 텐서를 생성합니다.

이 연산은 `dims` `shape`의 텐서를 만들고 `value` 로 값을 채웁니다.

예시:

```
Output tensor has shape [2, 3].
fill([2, 3], 9) ==> [[9, 9, 9]
 [9, 9, 9]]
```

인자:

- `dims` : `int32` 타입의 `Tensor` . 1-D(1-Dimension)이며 반환값 텐서의 `shape`을 나타냅니다.
- `value` : 스칼라 값을 갖는 `Tensor` . 반환값 텐서에 채워지는 값입니다.
- `name` : 연산의 명칭 (선택사항).

반환값:

`value` 와 같은 타입을 가진 `Tensor` .

---

```
tf.constant(value, dtype=None, shape=None, name='Const')
```

상수 텐서를 생성합니다.

결과값 텐서는 `value` 인자와 (선택적인) `shape`에 의해 결정됨으로써 `dtype` 타입의 값으로 채워집니다. (아래 예시를 보세요.)

인자 `value` 는 상수 또는 `dtype` 타입을 가진 값들의 리스트가 될 수 있습니다. 만약 `value` 가 리스트라면, 리스트의 길이는 `shape` 인자에 의해 나올 수 있는 원소들의 갯수와 같거나 작아야 합니다. 리스트의 길이가 `shape`에 의해 정해지는 원소들의 갯수보다 적을 경우, 리스트의 마지막 원소가 나머지 엔트리를 채우는데 사용됩니다.

`shape` 인자는 선택사항입니다. 만약 이 인자가 존재할 경우, 이는 결과값 텐서의 차원을 결정합니다. 그 외에는, `value` 의 `shape`을 그대로 사용합니다.

만약 `dtype` 인자가 결정되지 않을 경우에는, `value`로부터 타입을 추론하여 사용합니다.

예시:

```
Constant 1-D Tensor populated with value list.
tensor = tf.constant([1, 2, 3, 4, 5, 6, 7]) => [1 2 3 4 5 6 7]

Constant 2-D tensor populated with scalar value -1.
tensor = tf.constant(-1.0, shape=[2, 3]) => [[-1. -1. -1.]
 [-1. -1. -1.]]
```

인자:

- `value` : 반환 타입 `dtype`의 상수값 (또는 리스트).
- `dtype` : 결과값 텐서 원소들의 타입.
- `shape` : 결과값 텐서의 차원 (선택사항).
- `name` : 텐서의 명칭 (선택사항).

반환값:

상수 `Tensor`.

## 시퀀스

```
tf.linspace(start, stop, num, name=None)
```

구간 사이의 값들을 생성합니다.

`start` 부터 시작해서 생성된 `num` 개의 고르게 분포된 값들의 시퀀스입니다. 만약 `num > 1` 이면, 시퀀스의 값들은 `stop - start / num - 1` 씩 증가되며, 마지막 원소는 `stop` 값과 같아집니다.

예시:

```
tf.linspace(10.0, 12.0, 3, name="linspace") => [10.0 11.0 12.0]
```

인자:

- `start` : `float32` 또는 `float64` 타입의 `Tensor`. 구간의 첫번째 엔트리입니다.
- `stop` : `start` 와 같은 타입을 가진 `Tensor`. 구간의 마지막 엔트리입니다.
- `num` : `int32` 타입의 `Tensor`. 생성할 값들의 갯수입니다.
- `name` : 연산의 명칭 (선택사항).

반환값:

`start` 와 같은 타입을 가진 `Tensor`. 생성된 값들은 1-D입니다.

---

**tf.range(start, limit=None, delta=1, name='range')**

정수 시퀀스를 생성합니다.

`start` 부터 시작하여 `limit` 까지 (`limit` 는 포함하지 않음) `delta` 의 증가량만큼 확장하며 정수 리스트를 생성합니다.

파이썬의 내장 함수인 `range` 와 유사하며, `start` 의 기본값은 0이고, 즉 `range(n) = range(0, n)` 입니다.

예시:

```
'start' is 3
'limit' is 18
'delta' is 3
tf.range(start, limit, delta) ==> [3, 6, 9, 12, 15]

'limit' is 5
tf.range(limit) ==> [0, 1, 2, 3, 4]
```

인자:

- `start` : `int32` 타입의 스칼라 값(0-D)입니다. 시퀀스의 첫번째 엔트리이며, 기본값은 0입니다.

다.

- **limit** : `int32` 타입의 스칼라 값(0-D)입니다. 시퀀스의 상한이며, 시퀀스에 포함되지 않습니다. (exclusive)
- **delta** : A 0-D `Tensor` (scalar) of type `int32`. Optional. Default is 1. Number that increments `start`. `int32` 타입의 스칼라(0-D) 텐서입니다. 선택적인 인자이며, 기본값은 1입니다. `start` 를 증가시키는 수입니다.
- **name** : 연산의 명칭 (선택사항).

**반환값:**

1-D의 `int32` 타입을 갖는 `Tensor`.

## 난수 텐서

TensorFlow는 서로 다른 분포를 가진 난수 텐서들을 생성하는 여러가지 연산들을 제공합니다. 난수 연산들은 상태를 가지며, 계산될 때마다 새로운 난수를 생성합니다.

이러한 함수들의 `seed` 키워드 인자는 그래프 수준의 난수 시드값과 함께 작용합니다.

`set_random_seed` 를 사용하는 그래프 수준의 시드 또는 연산 수준의 시드를 바꾸는 것은 이러한 연산들의 기본 시드값을 바꿀 것입니다. 연산 수준과 그래프 수준의 난수 시드 사이의 상호작용에 대해 자세히 알고 싶다면 `set_random_seed` 를 참고하십시오.

## 예시:

```
Create a tensor of shape [2, 3] consisting of random normal values, with mean
-1 and standard deviation 4.
norm = tf.random_normal([2, 3], mean=-1, stddev=4)

Shuffle the first dimension of a tensor
c = tf.constant([[1, 2], [3, 4], [5, 6]])
shuff = tf.random_shuffle(c)

Each time we run these ops, different results are generated
sess = tf.Session()
print(sess.run(norm))
print(sess.run(norm))

Set an op-level seed to generate repeatable sequences across sessions.
norm = tf.random_normal([2, 3], seed=1234)
sess = tf.Session()
print(sess.run(norm))
print(sess.run(norm))
sess = tf.Session()
print(sess.run(norm))
print(sess.run(norm))
```

또 다른 난수값을 사용하는 일반적인 사례는 변수들의 초기화입니다. 이 또한 [Variables How To](#)에서 볼 수 있습니다.

```
Use random uniform values in [0, 1) as the initializer for a variable of shape
[2, 3]. The default type is float32.
var = tf.Variable(tf.random_uniform([2, 3]), name="var")
init = tf.initialize_all_variables()

sess = tf.Session()
sess.run(init)
print(sess.run(var))
```

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

정규분포로부터의 난수값을 반환합니다.

인자:

- **shape** : 정수값의 1-D 텐서 또는 파이썬 배열. 반환값 텐서의 **shape**입니다.
- **mean** : 0-D 텐서 또는 **dtype** 타입의 파이썬 값. 정규분포의 평균값.
- **stddev** : 0-D 텐서 또는 **dtype** 타입의 파이썬 값. 정규분포의 표준 편차.
- **dtype** : 반환값의 타입.
- **seed** : 파이썬 정수. 분포의 난수 시드값을 생성하는데에 사용됩니다. 동작 방식은 [set\\_random\\_seed](#) 를 보십시오.
- **name** : 연산의 명칭 (선택사항).

반환값:

정규 난수값들로 채워진 **shape**으로 정해진 텐서.

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

절단정규분포로부터의 난수값을 반환합니다.

생성된 값들은 평균으로부터 떨어진 벼려지고 재선택된 두 개의 표준편차보다 큰 값을 제외한 지정된 평균과 표준 편차를 가진 정규 분포를 따릅니다.

인자:

- **shape** : 정수값의 D-1 텐서 또는 파이썬 배열. 반환값 텐서의 **shape**입니다.
- **mean** : 0-D 텐서 또는 **dtype** 타입의 파이썬 값. 절단정규분포의 평균값.
- **stddev** : 0-D 텐서 또는 파이썬 값. 절단정규분포의 표준 편차.
- **dtype** : 반환값의 타입.
- **seed** : 파이썬 정수. 분포의 난수 시드값을 생성하는데에 사용됩니다. 동작 방식은 [set\\_random\\_seed](#) 를 보십시오.
- **name** : 연산의 명칭 (선택사항).

#### 반환값:

절단 정규 난수값들로 채워진 **shape**으로 정해진 텐서.

---

```
tf.random_uniform(shape, minval=0,
maxval=None, dtype=tf.float32, seed=None,
name=None)
```

균등분포로부터의 난수값을 반환합니다.

생성된 값들은 `[minval, maxval]` 구간의 균등분포를 따릅니다. 하한 `minval` 은 구간에 포함(included)되는 반면, 상한인 `maxval` 은 포함되지 않습니다(excluded).

실수형의 경우, 기본 구간은 `[0, 1)` 입니다. 정수형의 경우, 적어도 `maxval` 은 명시적으로 지정되어야합니다.

정수형의 경우, `maxval - minval` 가 2의 제곱수가 아니라면 정수 난수들은 한쪽으로 약간 치우칩니다. 치우침의 정도는 `maxval - minval` 의 값이 반환값의 구간(`2**32` 또는 `2**64`)보다 훨씬 작을 경우엔 작습니다.

#### 인자:

- **shape** : 정수값의 D-1 텐서 또는 파이썬 배열. 반환값 텐서의 **shape**입니다.
- **minval** : 0-D 텐서 또는 **dtype** 타입의 파이썬 값. 난수값 생성 구간의 하한입니다. 기본값은 0입니다.
- **maxval** : 0-D 텐서 또는 **dtype** 타입의 파이썬 값. 난수값 생성 구간의 상한입니다. **dtype** 이 실수형일 경우 기본값은 1입니다.
- **dtype** : 반환값의 타입: `float32` , `float64` , `int32` , 또는 `int64` .
- **seed** : 파이썬 정수. 분포의 난수 시드값을 생성하는데에 사용됩니다. 동작 방식은 [set\\_random\\_seed](#) 를 보십시오.
- **name** : 연산의 명칭 (선택사항).

#### 반환값:

균등 난수값들로 채워진 shape으로 정해진 텐서.

예외:

- `ValueError` : `dtype` 이 정수형인데 `maxval` 이 지정되지 않을 경우 발생합니다.
- 

### **`tf.random_shuffle(value, seed=None, name=None)`**

값의 첫번째 차원을 기준으로 랜덤하게 섞어줍니다.

텐서는 0차원을 따라 섞이는데, 예를 들면 각 `value[j]` 는 `output[i]` 의 각 원소에 정확히 하나씩 매핑이됩니다. 예를 들면,  $3 \times 2$  텐서의 경우 다음과 같은 매핑을 가질 수 있습니다.

```
[[1, 2], [[5, 6],
 [3, 4], ==> [1, 2],
 [5, 6]] [3, 4]]
```

인자:

- `value` : 섞기 위한 텐서.
- `seed` : 파이썬 정수. 분포의 난수 시드값을 생성하는데 사용됩니다. 동작 방식은 `set_random_seed` 를 보십시오.
- `name` : 연산의 명칭 (선택사항).

반환값:

`value` 의 첫번째 차원을 따라 섞인 `value` 와 같은 타입과 `shape`을 가진 텐서.

---

### **`tf.random_crop(value, size, seed=None, name=None)`**

텐서를 주어진 사이즈만큼 랜덤하게 잘라냅니다.

균등하게 선택된 오프셋에서 `value` 의 일부분을 `size` `shape`으로 잘라냅니다. `value.shape >= size` 를 만족해야합니다.

만약 차원을 잘라낼 수 없다면 차원의 전체 크기를 보냅니다. 예를 들면, RGB 이미지는 `size = [crop_height, crop_width, 3]` 을 가지고 잘라낼 수 있습니다.

인자:

- `value` : 자르기 위한 입력 텐서.
- `size` : `value` 의 랭크값을 가진 1-D 텐서.
- `seed` : 파이썬 정수. 분포의 난수 시드값을 생성하는데에 사용됩니다. 동작 방식은 `set_random_seed` 를 보십시오.
- `name` : 연산의 명칭 (선택사항).

반환값:

`value` 와 같은 랭크값을 갖고 `size` `shape`을 갖는 잘려진 텐서

---

```
tf.multinomial(logits, num_samples, seed=None, name=None)
```

다항분포로부터 샘플을 뽑아줍니다.

예시:

```
samples = tf.multinomial(tf.log([[0.5, 0.5]]), 10)
```

**samples has shape [1, 10], where each value is either 0 or 1.**

```
samples = tf.multinomial([[1, -1, -1]], 10)
```

**samples is equivalent to tf.zeros([1, 10], dtype=tf.int64).**

인자:

- `logits` : `[batch_size, num_classes]` `shape`을 갖는 2-D 텐서. 각 슬라이스 `[i, :]` 는 모든 클래스에 대한 비정규화 로그 확률을 나타냅니다.
- `num_samples` : 0-D. Number of independent samples to draw for each row slice. 0-D. 각 행 슬라이스를 뽑기위한 독립적인 샘플의 갯수.
- `seed` : 파이썬 정수. 분포의 난수 시드값을 생성하는데에 사용됩니다. 동작 방식은 `set_random_seed` 를 보십시오.
- `name` : 연산의 명칭 (선택사항).

반환값:

[batch\_size, num\_samples] shape의 샘플들

## tf.set\_random\_seed(seed)

그래프 수준의 난수 시드를 설정합니다.

난수 시드에 의존하는 연산들은 실제로 그래프 수준과 연산 수준의 두 가지 시드로부터 시드를 얻어냅니다. 이 연산은 그래프 수준의 시드를 설정합니다.

연산 수준의 시드와의 상호작용은 다음과 같습니다.

1. 그래프 수준과 연산 시드가 모두 설정되어있지 않은 경우: 난수 시드는 이 연산을 위해 사용됩니다.
2. 그래프 수준의 시드가 설정되어있고, 연산 시드는 설정되어있지 않은 경우, 시스템은 유일한 난수 시퀀스를 얻기위해 결정론적으로 그래프 수준의 시드와 함께 사용할 연산 시드를 선택합니다.
3. 그래프 수준의 시드가 설정되어있지 않고 연산 시드만 설정되어있는 경우, 난수 시퀀스를 결정하기 위해 그래프 수준의 시드의 기본값과 지정된 연산 시드가 사용됩니다.
4. 두 시드 모두 설정되어있을 경우, 난수 시퀀스를 결정하기 위해 두 시드가 함께 사용됩니다.

눈에 보이는 효과를 설명하기위해, 다음과 같은 예시들을 생각해봅시다:

세션간에 다른 시퀀스를 생성하기위해 그래프 수준과 연산 수준의 시드를 모두 설정하지 않습니다.

```
a = tf.random_uniform([1])
b = tf.random_normal([1])

print("Session 1")
with tf.Session() as sess1:
 print(sess1.run(a)) # generates 'A1'
 print(sess1.run(a)) # generates 'A2'
 print(sess1.run(b)) # generates 'B1'
 print(sess1.run(b)) # generates 'B2'

print("Session 2")
with tf.Session() as sess2:
 print(sess2.run(a)) # generates 'A3'
 print(sess2.run(a)) # generates 'A4'
 print(sess2.run(b)) # generates 'B3'
 print(sess2.run(b)) # generates 'B4'
```

세션간에 하나의 연산이 똑같이 반복가능한 시퀀스를 생성할 수 있도록, 연산 시드를 설정합니다.

```

a = tf.random_uniform([1], seed=1)
b = tf.random_normal([1])

Repeatedly running this block with the same graph will generate the same
sequence of values for 'a', but different sequences of values for 'b'.
print("Session 1")
with tf.Session() as sess1:
 print(sess1.run(a)) # generates 'A1'
 print(sess1.run(a)) # generates 'A2'
 print(sess1.run(b)) # generates 'B1'
 print(sess1.run(b)) # generates 'B2'

print("Session 2")
with tf.Session() as sess2:
 print(sess2.run(a)) # generates 'A1'
 print(sess2.run(a)) # generates 'A2'
 print(sess2.run(b)) # generates 'B3'
 print(sess2.run(b)) # generates 'B4'

```

모든 연산에 의해 생성된 난수 시퀀스들이 세션간 반복이 가능하게 하기위해서, 그래프 수준의 시드를 설정합니다.

```

tf.set_random_seed(1234)
a = tf.random_uniform([1])
b = tf.random_normal([1])

Repeatedly running this block with the same graph will generate different
sequences of 'a' and 'b'.
print("Session 1")
with tf.Session() as sess1:
 print(sess1.run(a)) # generates 'A1'
 print(sess1.run(a)) # generates 'A2'
 print(sess1.run(b)) # generates 'B1'
 print(sess1.run(b)) # generates 'B2'

print("Session 2")
with tf.Session() as sess2:
 print(sess2.run(a)) # generates 'A1'
 print(sess2.run(a)) # generates 'A2'
 print(sess2.run(b)) # generates 'B1'
 print(sess2.run(b)) # generates 'B2'

```

인자:

- **seed** : 정수.

# Variables

Note: 함수의 `Tensor` 인자는 `tf.convert_to_tensor`에 의한 것도 가능합니다.

[TOC]

## Variables

### `class tf.Variable`

[Variables How To](#)에서 자세한 개요를 확인할 수 있습니다.

변수는 `graph`에서 `run()`의 호출로 상태를 유지합니다. `Variable`의 객체를 만들어 `graph`에 변수를 추가합니다.

`Variable()` 생성자는 변수의 초기값으로 `Tensor`의 `type`과 `shape`이 필요합니다. 초기값은 변수의 `type`과 `shape`을 정의합니다. 생성 후, 변수의 `type`과 `shape`은 고정됩니다. 변수의 값은 `assign` 메소드를 사용해 변경할 수 있습니다.

후에 변수의 `shape`을 변경하고 싶다면 `assign`에서 `validate_shape=False`로 해야합니다.

`Tensor`의 경우, `Variable()`로 만들어진 변수는 `graph`의 `ops`의 `input`으로 사용될 수 있습니다. 추가적으로, `Tensor` 클래스로 오버로드 되는 모든 연산(operators)은 변수로 넘겨집니다. 그래서 변수의 산술연산만으로도 `graph`에 노드를 추가할 수 있습니다.

```
import tensorflow as tf

Create a variable.
w = tf.Variable(<initial-value>, name=<optional-name>)

Use the variable in the graph like any Tensor.
y = tf.matmul(w, ...another variable or tensor...)

The overloaded operators are available too.
z = tf.sigmoid(w + y)

Assign a new value to the variable with `assign()` or a related method.
w.assign(w + 1.0)
w.assign_add(1.0)
```

graph를 실행할 때, 변수는 그 값을 사용하는 ops를 실행하기 전에 명시적으로 초기화해야 합니다. 변수는 1) *initializer op*을 실행하거나, 2) 저장된 파일로부터 변수를 다시 저장(restoring)하거나, 3) 간단하게 변수에 값을 할당하는 `assign` Op을 실행하여 초기화 할 수 있습니다. 사실, 변수 *initializer op*은 단지 변수의 초기값을 할당하는 `assign` Op입니다.

```
Launch the graph in a session.
with tf.Session() as sess:
 # Run the variable initializer.
 sess.run(w.initializer)
 # ...you now can run ops that use the value of 'w'...
```

가장 일반적인 초기화 방법은 모든 변수를 초기화 할 graph에 편의 함수인 `initialize_all_variables()` 으로 Op를 추가하는 것입니다. 그런 다음 graph를 실행한 후 Op를 실행합니다.

```
Add an Op to initialize all variables.
init_op = tf.initialize_all_variables()

Launch the graph in a session.
with tf.Session() as sess:
 # Run the Op that initializes all variables.
 sess.run(init_op)
 # ...you can now run any Op that uses variable values...
```

다른 변수의 결과로 초기되는 변수를 만들어야한다면, 다른 변수의 `initialized_value()` 를 사용 합니다. 이것은 변수가 올바는 순서로 초기화되도록 합니다.

모든 변수들은 자동적으로 그들이 만들어진 graph에 쌓입니다. 기본적으로, 생성자는 그래프 컬렉션(graph collection) `GraphKeys.VARIABLES` 에 변수를 추가합니다. 편의 함수인 `all_variables()` 은 컬렉션의 내용을 반환합니다.

머신 러닝 모델을 만들 때, 학습 가능한 모델 매개변수를 가지고 있는 변수와 `global step` 변수과 같이 학습 단계를 계산하기 위한 다른 변수로 구분하는 것은 종종 편리합니다. 이것을 더 쉽게 하기 위해, 변수 생성자는 `trainable=<bool>` 매개변수를 지원합니다. `True` 일 때 새로운 변수는 그래프 컬렉션 `GraphKeys.TRAINABLE_VARIABLES` 에 추가됩니다. 편의 함수 `trainable_variables()` 는 이 컬렉션의 내용을 반환합니다. 다양한 `Optimizer` 클래스는 이 컬렉션을 최적화(optimize) 변수의 기본 리스트로 사용합니다.

Creating a variable.

---

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

Creates a new variable with value `initial_value`.

The new variable is added to the graph collections listed in `collections`, which defaults to `[GraphKeys.VARIABLES]`.

If `trainable` is `True` the variable is also added to the graph collection `GraphKeys.TRAINABLE_VARIABLES`.

This constructor creates both a `variable` Op and an `assign` Op to set the variable to its initial value.

### Args:

- `initial_value` : A `Tensor`, or Python object convertible to a `Tensor`, which is the initial value for the Variable. The initial value must have a shape specified unless `validate_shape` is set to `False`. Can also be a callable with no argument that returns the initial value when called. In that case, `dtype` must be specified. (Note that initializer functions from `init_ops.py` must first be bound to a shape before being used here.)
- `trainable` : If `True`, the default, also adds the variable to the graph collection `GraphKeys.TRAINABLE_VARIABLES`. This collection is used as the default list of variables to use by the `Optimizer` classes.
- `collections` : List of graph collections keys. The new variable is added to these collections. Defaults to `[GraphKeys.VARIABLES]`.
- `validate_shape` : If `False`, allows the variable to be initialized with a value of unknown shape. If `True`, the default, the shape of `initial_value` must be known.
- `caching_device` : Optional device string describing where the Variable should be cached for reading. Defaults to the Variable's device. If not `None`, caches on another device. Typical use is to cache on the device where the Ops using the Variable reside, to deduplicate copying through `switch` and other conditional statements.
- `name` : Optional name for the variable. Defaults to `'Variable'` and gets uniquified automatically.
- `variable_def` : `VariableDef` protocol buffer. If not `None`, recreates the Variable object with its contents. `variable_def` and the other arguments are mutually exclusive.
- `dtype` : If set, `initial_value` will be converted to the given type. If `None`, either the datatype will be kept (if `initial_value` is a Tensor), or `convert_to_tensor` will decide.

### Returns:

A Variable.

### Raises:

- `ValueError` : If both `variable_def` and `initial_value` are specified.
- `ValueError` : If the initial value is not specified, or does not have a shape and

---

```
validate_shape is True .
```

---

## **tf.Variable.initialized\_value()**

Returns the value of the initialized variable.

You should use this instead of the variable itself to initialize another variable with a value that depends on the value of this variable.

```
Initialize 'v' with a random tensor.
v = tf.Variable(tf.truncated_normal([10, 40]))
Use `initialized_value` to guarantee that `v` has been
initialized before its value is used to initialize `w`.
The random values are picked only once.
w = tf.Variable(v.initialized_value() * 2.0)
```

### **Returns:**

A `Tensor` holding the value of this variable after its initializer has run.

Changing a variable value.

---

## **tf.Variable.assign(value, use\_locking=False)**

Assigns a new value to the variable.

This is essentially a shortcut for `assign(self, value)`.

### **Args:**

- `value` : A `Tensor`. The new value for this variable.
- `use_locking` : If `True`, use locking during the assignment.

### **Returns:**

A `Tensor` that will hold the new value of this variable after the assignment has completed.

---

## **tf.Variable.assign\_add(delta, use\_locking=False)**

Adds a value to this variable.

This is essentially a shortcut for `assign_add(self, delta)`.

**Args:**

- `delta` : A `Tensor`. The value to add to this variable.
- `use_locking` : If `True`, use locking during the operation.

**Returns:**

A `Tensor` that will hold the new value of this variable after the addition has completed.

---

**`tf.Variable.assign_sub(delta, use_locking=False)`**

Subtracts a value from this variable.

This is essentially a shortcut for `assign_sub(self, delta)`.

**Args:**

- `delta` : A `Tensor`. The value to subtract from this variable.
- `use_locking` : If `True`, use locking during the operation.

**Returns:**

A `Tensor` that will hold the new value of this variable after the subtraction has completed.

---

**`tf.Variable.scatter_sub(sparse_delta, use_locking=False)`**

Subtracts `IndexedSlices` from this variable.

This is essentially a shortcut for `scatter_sub(self, sparse_delta.indices, sparse_delta.values)`.

**Args:**

- `sparse_delta` : `IndexedSlices` to be subtracted from this variable.
- `use_locking` : If `True`, use locking during the operation.

**Returns:**

A `Tensor` that will hold the new value of this variable after the scattered subtraction has completed.

**Raises:**

- `ValueError` : if `sparse_delta` is not an `IndexedSlices`.

## tf.Variable.count\_up\_to(limit)

Increments this variable until it reaches `limit`.

When that Op is run it tries to increment the variable by `1`. If incrementing the variable would bring it above `limit` then the Op raises the exception `OutOfRangeError`.

If no error is raised, the Op outputs the value of the variable before the increment.

This is essentially a shortcut for `count_up_to(self, limit)`.

### Args:

- `limit` : value at which incrementing the variable raises an error.

### Returns:

A `Tensor` that will hold the variable value before the increment. If no other Op modifies this variable, the values produced will all be distinct.

## tf.Variable.eval(session=None)

In a session, computes and returns the value of this variable.

This is not a graph construction method, it does not add ops to the graph.

This convenience method requires a session where the graph containing this variable has been launched. If no session is passed, the default session is used. See the [Session class](#) for more information on launching a graph and on sessions.

```
v = tf.Variable([1, 2])
init = tf.initialize_all_variables()

with tf.Session() as sess:
 sess.run(init)
 # Usage passing the session explicitly.
 print(v.eval(sess))
 # Usage with the default session. The 'with' block
 # above makes 'sess' the default session.
 print(v.eval())
```

### Args:

- `session` : The session to use to evaluate this variable. If none, the default session is used.

**Returns:**

A numpy `ndarray` with a copy of the value of this variable.

Properties.

---

**`tf.Variable.name`**

The name of this variable.

---

**`tf.Variable.dtype`**

The `DType` of this variable.

---

**`tf.Variable.get_shape()`**

The `TensorShape` of this variable.

**Returns:**

A `TensorShape`.

---

**`tf.Variable.device`**

The device of this variable.

---

**`tf.Variable.initializer`**

The initializer operation for this variable.

---

**`tf.Variable.graph`**

The `Graph` of this variable.

---

**`tf.Variable.op`**

The `Operation` of this variable.

## Other Methods

---

### `tf.Variable.from_proto(variable_def)`

Returns a `variable` object created from `variable_def`.

---

### `tf.Variable.initial_value`

Returns the Tensor used as the initial value for the variable.

Note that this is different from `initialized_value()` which runs the op that initializes the variable before returning its value. This method returns the tensor that is used by the op that initializes the variable.

#### Returns:

A `Tensor`.

---

### `tf.Variable.ref()`

Returns a reference to this variable.

You usually do not need to call this method as all ops that need a reference to the variable call it automatically.

Returns is a `Tensor` which holds a reference to the variable. You can assign a new value to the variable by passing the tensor to an assign op. See `value()` if you want to get the value of the variable.

#### Returns:

A `Tensor` that is a reference to the variable.

---

### `tf.Variable.to_proto()`

Converts a `variable` to a `variableDef` protocol buffer.

#### Returns:

---

A `variableDef` protocol buffer.

---

### **tf.Variable.value()**

Returns the last snapshot of this variable.

You usually do not need to call this method as all ops that need the value of the variable call it automatically through a `convert_to_tensor()` call.

Returns a `Tensor` which holds the value of the variable. You can not assign a new value to this tensor as it is not a reference to the variable. See `ref()` if you want to get a reference to the variable.

To avoid copies, if the consumer of the returned value is on the same device as the variable, this actually returns the live value of the variable, not a copy. Updates to the variable are seen by the consumer. If the consumer is on a different device it will get a copy of the variable.

**Returns:**

A `Tensor` containing the value of the variable.

## **Variable helper functions**

TensorFlow provides a set of functions to help manage the set of variables collected in the graph.

---

### **tf.all\_variables()**

Returns all variables that must be saved/restored.

The `Variable()` constructor automatically adds new variables to the graph collection `GraphKeys.VARIABLES`. This convenience function returns the contents of that collection.

**Returns:**

A list of `Variable` objects.

---

### **tf.trainable\_variables()**

Returns all variables created with `trainable=True`.

When passed `trainable=True`, the `Variable()` constructor automatically adds new variables to the graph collection `GraphKeys.TRAINABLE_VARIABLES`. This convenience function returns the contents of that collection.

**Returns:**

A list of Variable objects.

---

## `tf.local_variables()`

Returns all variables created with `collection=[LOCAL_VARIABLES]`.

**Returns:**

A list of local Variable objects.

---

## `tf.moving_average_variables()`

Returns all variables that maintain their moving averages.

If an `ExponentialMovingAverage` object is created and the `apply()` method is called on a list of variables, these variables will be added to the `GraphKeys.MOVING_AVERAGE_VARIABLES` collection. This convenience function returns the contents of that collection.

**Returns:**

A list of Variable objects.

---

## `tf.initialize_all_variables()`

Returns an Op that initializes all variables.

This is just a shortcut for `initialize_variables(all_variables())`

**Returns:**

An Op that initializes all variables in the graph.

## tf.initialize\_variables(var\_list, name='init')

Returns an Op that initializes a list of variables.

After you launch the graph in a session, you can run the returned Op to initialize all the variables in `var_list`. This Op runs all the initializers of the variables in `var_list` in parallel.

Calling `initialize_variables()` is equivalent to passing the list of initializers to `Group()`.

If `var_list` is empty, however, the function still returns an Op that can be run. That Op just has no effect.

### Args:

- `var_list` : List of `Variable` objects to initialize.
- `name` : Optional name for the returned operation.

### Returns:

An Op that run the initializers of all the specified variables.

## tf.initialize\_local\_variables()

Returns an Op that initializes all local variables.

This is just a shortcut for `initialize_variables(local_variables())`

### Returns:

An Op that initializes all local variables in the graph.

## tf.is\_variable\_initialized(variable)

Tests if a variable has been initialized.

### Args:

- `variable` : A `Variable`.

### Returns:

---

Returns a scalar boolean Tensor, `True` if the variable has been initialized, `False` otherwise.

---

### `tf.report_uninitialized_variables(var_list=None, name='report_uninitialized_variables')`

Adds ops to list the names of uninitialized variables.

When run, it returns a 1-D tensor containing the names of uninitialized variables if there are any, or an empty array if there are none.

#### Args:

- `var_list` : List of `Variable` objects to check. Defaults to the value of `all_variables() + local_variables()`
- `name` : Optional name of the `Operation`.

#### Returns:

A 1-D tensor containing names of the uninitialized variables, or an empty 1-D tensor if there are no variables or no uninitialized variables.

---

### `tf.assert_variables_initialized(var_list=None)`

Returns an Op to check if variables are initialized.

NOTE: This function is obsolete and will be removed in 6 months. Please change your implementation to use `report_uninitialized_variables()`.

When run, the returned Op will raise the exception `FailedPreconditionError` if any of the variables has not yet been initialized.

Note: This function is implemented by trying to fetch the values of the variables. If one of the variables is not initialized a message may be logged by the C++ runtime. This is expected.

#### Args:

- `var_list` : List of `Variable` objects to check. Defaults to the value of `all_variables()`.

#### Returns:

An Op, or None if there are no variables.

# Saving and Restoring Variables

## `class tf.train.Saver`

Saves and restores variables.

See [Variables](#) for an overview of variables, saving and restoring.

The `Saver` class adds ops to save and restore variables to and from *checkpoints*. It also provides convenience methods to run these ops.

Checkpoints are binary files in a proprietary format which map variable names to tensor values. The best way to examine the contents of a checkpoint is to load it using a `Saver`.

Savers can automatically number checkpoint filenames with a provided counter. This lets you keep multiple checkpoints at different steps while training a model. For example you can number the checkpoint filenames with the training step number. To avoid filling up disks, savers manage checkpoint files automatically. For example, they can keep only the N most recent files, or one checkpoint for every N hours of training.

You number checkpoint filenames by passing a value to the optional `global_step` argument to `save()`:

```
saver.save(sess, 'my-model', global_step=0) ==> filename: 'my-model-0'
...
saver.save(sess, 'my-model', global_step=1000) ==> filename: 'my-model-1000'
```

Additionally, optional arguments to the `Saver()` constructor let you control the proliferation of checkpoint files on disk:

- `max_to_keep` indicates the maximum number of recent checkpoint files to keep. As new files are created, older files are deleted. If `None` or `0`, all checkpoint files are kept. Defaults to `5` (that is, the `5` most recent checkpoint files are kept.)
- `keep_checkpoint_every_n_hours` : In addition to keeping the most recent `max_to_keep` checkpoint files, you might want to keep one checkpoint file for every `N` hours of training. This can be useful if you want to later analyze how a model progressed during a long training session. For example, passing `keep_checkpoint_every_n_hours=2` ensures that you keep one checkpoint file for every `2` hours of training. The default value of `10,000` hours effectively disables the feature.

Note that you still have to call the `save()` method to save the model. Passing these arguments to the constructor will not save variables automatically for you.

A training program that saves regularly looks like:

```
...
Create a saver.
saver = tf.train.Saver(...variables...)
Launch the graph and train, saving the model every 1,000 steps.
sess = tf.Session()
for step in xrange(1000000):
 sess.run(..training_op..)
 if step % 1000 == 0:
 # Append the step number to the checkpoint name:
 saver.save(sess, 'my-model', global_step=step)
```

In addition to checkpoint files, savers keep a protocol buffer on disk with the list of recent checkpoints. This is used to manage numbered checkpoint files and by `latest_checkpoint()`, which makes it easy to discover the path to the most recent checkpoint. That protocol buffer is stored in a file named 'checkpoint' next to the checkpoint files.

If you create several savers, you can specify a different filename for the protocol buffer file in the call to `save()`.

```
tf.train.Saver.__init__(var_list=None, reshape=False,
sharded=False, max_to_keep=5,
keep_checkpoint_every_n_hours=10000.0, name=None,
restore_sequentially=False, saver_def=None, builder=None)
```

Creates a `Saver`.

The constructor adds ops to save and restore variables.

`var_list` specifies the variables that will be saved and restored. It can be passed as a `dict` or a `list`:

- A `dict` of names to variables: The keys are the names that will be used to save or restore the variables in the checkpoint files.
- A list of variables: The variables will be keyed with their op name in the checkpoint files.

For example:

```

v1 = tf.Variable(..., name='v1')
v2 = tf.Variable(..., name='v2')

Pass the variables as a dict:
saver = tf.train.Saver({'v1': v1, 'v2': v2})

Or pass them as a list.
saver = tf.train.Saver([v1, v2])
Passing a list is equivalent to passing a dict with the variable op names
as keys:
saver = tf.train.Saver({v.op.name: v for v in [v1, v2]})
```

The optional `reshape` argument, if `True`, allows restoring a variable from a save file where the variable had a different shape, but the same number of elements and type. This is useful if you have reshaped a variable and want to reload it from an older checkpoint.

The optional `sharded` argument, if `True`, instructs the saver to shard checkpoints per device.

### Args:

- `var_list` : A list of `Variable` objects or a dictionary mapping names to variables. If `None`, defaults to the list of all variables.
- `reshape` : If `True`, allows restoring parameters from a checkpoint where the variables have a different shape.
- `sharded` : If `True`, shard the checkpoints, one per device.
- `max_to_keep` : Maximum number of recent checkpoints to keep. Defaults to 5.
- `keep_checkpoint_every_n_hours` : How often to keep checkpoints. Defaults to 10,000 hours.
- `name` : String. Optional name to use as a prefix when adding operations.
- `restore_sequentially` : A `Bool`, which if true, causes restore of different variables to happen sequentially within each device. This can lower memory usage when restoring very large models.
- `saver_def` : Optional `SaverDef` proto to use instead of running the builder. This is only useful for specialty code that wants to recreate a `saver` object for a previously built `Graph` that had a `Saver`. The `saver_def` proto should be the one returned by the `as_saver_def()` call of the `Saver` that was created for that `Graph`.
- `builder` : Optional `SaverBuilder` to use if a `saver_def` was not provided. Defaults to `BaseSaverBuilder()`.

### Raises:

- `TypeError` : If `var_list` is invalid.
- `ValueError` : If any of the keys or values in `var_list` are not unique.

---

```
tf.train.Saver.save(sess, save_path, global_step=None,
latest_filename=None, meta_graph_suffix='meta',
write_meta_graph=True)
```

Saves variables.

This method runs the ops added by the constructor for saving variables. It requires a session in which the graph was launched. The variables to save must also have been initialized.

The method returns the path of the newly created checkpoint file. This path can be passed directly to a call to `restore()`.

#### Args:

- `sess` : A Session to use to save the variables.
- `save_path` : String. Path to the checkpoint filename. If the saver is `sharded`, this is the prefix of the sharded checkpoint filename.
- `global_step` : If provided the global step number is appended to `save_path` to create the checkpoint filename. The optional argument can be a `Tensor`, a `Tensor` name or an integer.
- `latest_filename` : Optional name for the protocol buffer file that will contain the list of most recent checkpoint filenames. That file, kept in the same directory as the checkpoint files, is automatically managed by the saver to keep track of recent checkpoints. Defaults to 'checkpoint'.
- `meta_graph_suffix` : Suffix for `MetaGraphDef` file. Defaults to 'meta'.
- `write_meta_graph` : Boolean indicating whether or not to write the meta graph file.

#### Returns:

A string: path at which the variables were saved. If the saver is sharded, this string ends with: '-?????-of-nnnnn' where 'nnnnn' is the number of shards created.

#### Raises:

- `TypeError` : If `sess` is not a `Session`.
- `ValueError` : If `latest_filename` contains path components.

---

```
tf.train.Saver.restore(sess, save_path)
```

Restores previously saved variables.

---

This method runs the ops added by the constructor for restoring variables. It requires a session in which the graph was launched. The variables to restore do not have to have been initialized, as restoring is itself a way to initialize variables.

The `save_path` argument is typically a value previously returned from a `save()` call, or a call to `latest_checkpoint()`.

#### Args:

- `sess` : A `Session` to use to restore the parameters.
- `save_path` : Path where parameters were previously saved.

#### Raises:

- `ValueError` : If the given `save_path` does not point to a file.

Other utility methods.

---

### `tf.train.Saver.last_checkpoints`

List of not-yet-deleted checkpoint filenames.

You can pass any of the returned values to `restore()`.

#### Returns:

A list of checkpoint filenames, sorted from oldest to newest.

---

### `tf.train.Saver.set_last_checkpoints(last_checkpoints)`

DEPRECATED: Use `set_last_checkpoints_with_time`.

Sets the list of old checkpoint filenames.

#### Args:

- `last_checkpoints` : A list of checkpoint filenames.

#### Raises:

- `AssertionError` : If `last_checkpoints` is not a list.
- 

### `tf.train.Saver.as_saver_def()`

Generates a `SaverDef` representation of this saver.

#### Returns:

A `saverDef` proto.

## Other Methods

---

```
tf.train.Saver.export_meta_graph(filename=None,
collection_list=None, as_text=False)
```

Writes `MetaGraphDef` to save\_path/filename.

#### Args:

- `filename` : Optional meta\_graph filename including the path.
- `collection_list` : List of string keys to collect.
- `as_text` : If `True`, writes the meta\_graph as an ASCII proto.

#### Returns:

A `MetaGraphDef` proto.

---

```
tf.train.Saver.from_proto(saver_def)
```

Returns a `saver` object created from `saver_def`.

---

```
tf.train.Saver.set_last_checkpoints_with_time(last_checkpoints_with_time)
```

Sets the list of old checkpoint filenames and timestamps.

#### Args:

- `last_checkpoints_with_time` : A list of tuples of checkpoint filenames and timestamps.

#### Raises:

- `AssertionError` : If `last_checkpoints_with_time` is not a list.

---

**tf.train.Saver.to\_proto()**

Converts this `Saver` to a `SaverDef` protocol buffer.

**Returns:**

A `SaverDef` protocol buffer.

---

**tf.train.latest\_checkpoint(checkpoint\_dir, latest\_filename=None)**

Finds the filename of latest saved checkpoint file.

**Args:**

- `checkpoint_dir` : Directory where the variables were saved.
- `latest_filename` : Optional name for the protocol buffer file that contains the list of most recent checkpoint filenames. See the corresponding argument to `Saver.save()`.

**Returns:**

The full path to the latest checkpoint or `None` if no checkpoint was found.

---

**tf.train.get\_checkpoint\_state(checkpoint\_dir, latest\_filename=None)**

Returns `CheckpointState` proto from the "checkpoint" file.

If the "checkpoint" file contains a valid `CheckpointState` proto, returns it.

**Args:**

- `checkpoint_dir` : The directory of checkpoints.
- `latest_filename` : Optional name of the checkpoint file. Default to 'checkpoint'.

**Returns:**

A `CheckpointState` if the state was available, `None` otherwise.

---

```
tf.train.update_checkpoint_state(save_dir,
model_checkpoint_path,
all_model_checkpoint_paths=None,
latest_filename=None)
```

Updates the content of the 'checkpoint' file.

This updates the checkpoint file containing a CheckpointState proto.

#### Args:

- `save_dir` : Directory where the model was saved.
- `model_checkpoint_path` : The checkpoint file.
- `all_model_checkpoint_paths` : List of strings. Paths to all not-yet-deleted checkpoints, sorted from oldest to newest. If this is a non-empty list, the last element must be equal to `model_checkpoint_path`. These paths are also saved in the CheckpointState proto.
- `latest_filename` : Optional name of the checkpoint file. Default to 'checkpoint'.

#### Raises:

- `RuntimeError` : If the save paths conflict.

## Sharing Variables

TensorFlow provides several classes and operations that you can use to create variables contingent on certain conditions.

```
tf.get_variable(name, shape=None,
dtype=tf.float32, initializer=None,
regularizer=None, trainable=True,
collections=None, caching_device=None,
partitioner=None, validate_shape=True)
```

Gets an existing variable with these parameters or create a new one.

This function prefixes the name with the current variable scope and performs reuse checks. See the [Variable Scope How To](#) for an extensive description of how reusing works. Here is a basic example:

```

with tf.variable_scope("foo"):
 v = tf.get_variable("v", [1]) # v.name == "foo/v:0"
 w = tf.get_variable("w", [1]) # w.name == "foo/w:0"
with tf.variable_scope("foo", reuse=True):
 v1 = tf.get_variable("v") # The same as v above.

```

If initializer is `None` (the default), the default initializer passed in the variable scope will be used. If that one is `None` too, a `UniformUnitScalingInitializer` will be used. The initializer can also be a Tensor, in which case the variable is initialized to this value and shape.

Similarly, if the regularizer is `None` (the default), the default regularizer passed in the variable scope will be used (if that is `None` too, then by default no regularization is performed).

If a partitioner is provided, first a sharded `variable` is created via `_get_partitioned_variable`, and the return value is a `Tensor` composed of the shards concatenated along the partition axis.

Some useful partitioners are available. See, e.g., `variable_axis_size_partitioner`.

### Args:

- `name` : The name of the new or existing variable.
- `shape` : Shape of the new or existing variable.
- `dtype` : Type of the new or existing variable (defaults to `DT_FLOAT` ).
- `initializer` : Initializer for the variable if one is created.
- `regularizer` : A (`Tensor` -> `Tensor` or `None`) function; the result of applying it on a newly created variable will be added to the collection `GraphKeys.REGULARIZATION_LOSSES` and can be used for regularization.
- `trainable` : If `True` also add the variable to the graph collection `GraphKeys.TRAINABLE_VARIABLES` (see `tf.Variable`).
- `collections` : List of graph collections keys to add the Variable to. Defaults to `[GraphKeys.VARIABLES]` (see `tf.Variable`).
- `caching_device` : Optional device string or function describing where the Variable should be cached for reading. Defaults to the Variable's device. If not `None`, caches on another device. Typical use is to cache on the device where the Ops using the Variable reside, to deduplicate copying through `Switch` and other conditional statements.
- `partitioner` : Optional callable that accepts a fully defined `TensorShape` and `dtype` of the Variable to be created, and returns a list of partitions for each axis (currently only one axis can be partitioned).
- `validate_shape` : If `False`, allows the variable to be initialized with a value of unknown shape. If `True`, the default, the shape of `initial_value` must be known.

**Returns:**

The created or existing variable.

**Raises:**

- `ValueError` : when creating a new variable and shape is not declared, or when violating reuse during variable creation. Reuse is set inside `variable_scope` .

---

**class tf.VariableScope**

Variable scope object to carry defaults to provide to `get_variable`.

Many of the arguments we need for `get_variable` in a variable store are most easily handled with a context. This object is used for the defaults.

Attributes: name: name of the current scope, used as prefix in `get_variable`. initializer: default initializer passed to `get_variable`. regularizer: default regularizer passed to `get_variable`. reuse: Boolean or None, setting the reuse in `get_variable`. caching\_device: string, callable, or None: the caching device passed to `get_variable`. partitioner: callable or None : the partitioner passed to `get_variable`. name\_scope: The name passed to `tf.name_scope` .

---

```
tf.VariableScope.__init__(reuse, name='', initializer=None,
regularizer=None, caching_device=None, partitioner=None,
name_scope='')
```

Creates a new VariableScope with the given properties.

---

**tf.VariableScope.caching\_device**

```
tf.VariableScope.get_variable(var_store, name, shape=None,
dtype=tf.float32, initializer=None, regularizer=None,
trainable=True, collections=None, caching_device=None,
partitioner=None, validate_shape=True)
```

Gets an existing variable with this name or create a new one.

---

**tf.VariableScope.initializer**

---

**tf.VariableScope.name**

---

**tf.VariableScope.partitioner**

---

**tf.VariableScope.regularizer**

---

**tf.VariableScope.reuse**

---

**tf.VariableScope.reuse\_variables()**

Reuse variables in this scope.

---

**tf.VariableScope.set\_caching\_device(caching\_device)**

Set caching\_device for this scope.

---

**tf.VariableScope.set\_initializer(initializer)**

Set initializer for this scope.

---

**tf.VariableScope.set\_partitioner(partitioner)**

Set partitioner for this scope.

---

**tf.VariableScope.set\_regularizer(regularizer)**

Set regularizer for this scope.

---

```
tf.variable_scope(name_or_scope, reuse=None,
initializer=None, regularizer=None,
caching_device=None, partitioner=None)
```

Returns a context for variable scope.

Variable scope allows to create new variables and to share already created ones while providing checks to not create or share by accident. For details, see the [Variable Scope How To](#), here we present only a few basic examples.

Simple example of how to create a new variable:

```
with tf.variable_scope("foo"):
 with tf.variable_scope("bar"):
 v = tf.get_variable("v", [1])
 assert v.name == "foo/bar/v:0"
```

Basic example of sharing a variable:

```
with tf.variable_scope("foo"):
 v = tf.get_variable("v", [1])
with tf.variable_scope("foo", reuse=True):
 v1 = tf.get_variable("v", [1])
assert v1 == v
```

Sharing a variable by capturing a scope and setting reuse:

```
with tf.variable_scope("foo") as scope:
 v = tf.get_variable("v", [1])
 scope.reuse_variables()
 v1 = tf.get_variable("v", [1])
assert v1 == v
```

To prevent accidental sharing of variables, we raise an exception when getting an existing variable in a non-reusing scope.

```
with tf.variable_scope("foo"):
 v = tf.get_variable("v", [1])
 v1 = tf.get_variable("v", [1])
 # Raises ValueError(... v already exists ...).
```

Similarly, we raise an exception when trying to get a variable that does not exist in reuse mode.

```
with tf.variable_scope("foo", reuse=True):
 v = tf.get_variable("v", [1])
 # Raises ValueError("... v does not exists ...").
```

Note that the `reuse` flag is inherited: if we open a reusing scope, then all its sub-scopes become reusing as well.

### Args:

- `name_or_scope` : string or `VariableScope` : the scope to open.
- `reuse` : `True` or `None` ; if `True`, we go into reuse mode for this scope as well as all sub-scopes; if `None`, we just inherit the parent scope reuse.
- `initializer` : default initializer for variables within this scope.
- `regularizer` : default regularizer for variables within this scope.
- `caching_device` : default caching device for variables within this scope.
- `partitioner` : default partitioner for variables within this scope.

### Returns:

A scope that can be captured and reused.

### Raises:

- `ValueError` : when trying to reuse within a create scope, or create within a reuse scope, or if reuse is not `None` or `True`.
- `TypeError` : when the types of some arguments are not appropriate.

```
tf.variable_op_scope(values, name_or_scope,
default_name=None, initializer=None,
regularizer=None, caching_device=None,
partitioner=None, reuse=None)
```

Returns a context manager for defining an op that creates variables.

This context manager validates that the given `values` are from the same graph, ensures that graph is the default graph, and pushes a name scope and a variable scope.

If `name_or_scope` is not `None`, it is used as is in the variable scope. If `scope` is `None`, then `default_name` is used. In that case, if the same name has been previously used in the same scope, it will be made unique by appending `_N` to it.

This is intended to be used when defining generic ops and so reuse is always inherited.

For example, to define a new Python op called `my_op_with_vars` :

```
def my_op_with_vars(a, b, scope=None):
 with tf.variable_scope([a, b], scope, "MyOp") as scope:
 a = tf.convert_to_tensor(a, name="a")
 b = tf.convert_to_tensor(b, name="b")
 c = tf.get_variable('c')
 # Define some computation that uses `a`, `b`, and `c`.
 return foo_op(..., name=scope)
```

## Args:

- `values` : The list of `Tensor` arguments that are passed to the op function.
- `name_or_scope` : The name argument that is passed to the op function, this `name_or_scope` is not unqualified in the variable scope.
- `default_name` : The default name to use if the `name_or_scope` argument is `None`, this name will be unqualified. If `name_or_scope` is provided it won't be used and therefore it is not required and can be `None`.
- `initializer` : The default initializer to pass to variable scope.
- `regularizer` : The default regularizer for variables within this scope.
- `caching_device` : The default caching device for variables within this scope.
- `partitioner` : The default partitioner for variables within this scope.
- `reuse` : `True` or `None`; if `True`, we go into reuse mode for this scope as well as all sub-scopes; if `None`, we just inherit the parent scope reuse.

## Returns:

A context manager for use in defining a Python op.

## Raises:

- `ValueError` : when trying to reuse within a create scope, or create within a reuse scope, or if reuse is not `None` or `True`.
- `TypeError` : when the types of some arguments are not appropriate.

## `tf.get_variable_scope()`

Returns the current variable scope.

---

`tf.make_template(name_, func_, create_scope_now_=False, **kwargs)`

Given an arbitrary function, wrap it so that it does variable sharing.

This wraps `func_` in a `Template` and partially evaluates it. Templates are functions that create variables the first time they are called and reuse them thereafter. In order for `func_` to be compatible with a `Template` it must have the following properties:

- The function should create all trainable variables and any variables that should be reused by calling `tf.get_variable`. If a trainable variable is created using `tf.variable`, then a `ValueError` will be thrown. Variables that are intended to be locals can be created by specifying `tf.Variable(..., trainable=False)`.
- The function may use variable scopes and other templates internally to create and reuse variables, but it shouldn't use `tf.get_variables` to capture variables that are defined outside of the scope of the function.
- Internal scopes and variable names should not depend on any arguments that are not supplied to `make_template`. In general you will get a `ValueError` telling you that you are trying to reuse a variable that doesn't exist if you make a mistake.

In the following example, both `z` and `w` will be scaled by the same `y`. It is important to note that if we didn't assign `scalar_name` and used a different name for `z` and `w` that a `ValueError` would be thrown because it couldn't reuse the variable.

```
def my_op(x, scalar_name):
 var1 = tf.get_variable(scalar_name,
 shape=[],
 initializer=tf.constant_initializer(1))
 return x * var1

scale_by_y = tf.make_template('scale_by_y', my_op, scalar_name='y')

z = scale_by_y(input1)
w = scale_by_y(input2)
```

As a safe-guard, the returned function will raise a `ValueError` after the first call if trainable variables are created by calling `tf.Variable`.

If all of these are true, then 2 properties are enforced by the template:

1. Calling the same template multiple times will share all non-local variables.
2. Two different templates are guaranteed to be unique, unless you reenter the same variable scope as the initial definition of a template and redefine it. An examples of this exception:

```

def my_op(x, scalar_name):
 var1 = tf.get_variable(scalar_name,
 shape=[],
 initializer=tf.constant_initializer(1))
 return x * var1

with tf.variable_scope('scope') as vs:
 scale_by_y = tf.make_template('scale_by_y', my_op, scalar_name='y')
 z = scale_by_y(input1)
 w = scale_by_y(input2)

Creates a template that reuses the variables above.
with tf.variable_scope(vs, reuse=True):
 scale_by_y2 = tf.make_template('scale_by_y', my_op, scalar_name='y')
 z2 = scale_by_y2(input1)
 w2 = scale_by_y2(input2)

```

Depending on the value of `create_scope_now_`, the full variable scope may be captured either at the time of first call or at the time of construction. If this option is set to True, then all Tensors created by repeated calls to the template will have an extra trailing `_N+1` to their name, as the first time the scope is entered in the Template constructor no Tensors are created.

Note: `name_`, `func_` and `create_scope_now_` have a trailing underscore to reduce the likelihood of collisions with kwargs.

### Args:

- `name_` : A name for the scope created by this template. If necessary, the name will be made unique by appending `_N` to the name.
- `func_` : The function to wrap.
- `create_scope_now_` : Boolean controlling whether the scope should be created when the template is constructed or when the template is called. Default is False, meaning the scope is created when the template is called.
- `**kwargs` : Keyword arguments to apply to `func_`.

### Returns:

A function to encapsulate a set of variables which should be created once and reused. An enclosing scope will be created, either where `make_template` is called, or wherever the result is called, depending on the value of `create_scope_now_`. Regardless of the value, the first time the template is called it will enter the scope with no reuse, and call `func_` to create variables, which are guaranteed to be unique. All subsequent calls will re-enter the scope and reuse those variables.

**Raises:**

- `ValueError` : if the name is None.

**`tf.no_regularizer(_)`**

Use this function to prevent regularization of variables.

**`tf.constant_initializer(value=0.0, dtype=tf.float32)`**

Returns an initializer that generates tensors with a single value.

**Args:**

- `value` : A Python scalar. All elements of the initialized variable will be set to this value.
- `dtype` : The data type. Only floating point types are supported.

**Returns:**

An initializer that generates tensors with a single value.

**Raises:**

- `ValueError` : if `dtype` is not a floating point type.

**`tf.random_normal_initializer(mean=0.0, stddev=1.0, seed=None, dtype=tf.float32)`**

Returns an initializer that generates tensors with a normal distribution.

**Args:**

- `mean` : a python scalar or a scalar tensor. Mean of the random values to generate.
- `stddev` : a python scalar or a scalar tensor. Standard deviation of the random values to generate.
- `seed` : A Python integer. Used to create random seeds. See [set\\_random\\_seed](#) for behavior.
- `dtype` : The data type. Only floating point types are supported.

**Returns:**

An initializer that generates tensors with a normal distribution.

**Raises:**

- `ValueError` : if `dtype` is not a floating point type.

---

```
tf.truncated_normal_initializer(mean=0.0,
stddev=1.0, seed=None, dtype=tf.float32)
```

Returns an initializer that generates a truncated normal distribution.

These values are similar to values from a `random_normal_initializer` except that values more than two standard deviations from the mean are discarded and re-drawn. This is the recommended initializer for neural network weights and filters.

**Args:**

- `mean` : a python scalar or a scalar tensor. Mean of the random values to generate.
- `stddev` : a python scalar or a scalar tensor. Standard deviation of the random values to generate.
- `seed` : A Python integer. Used to create random seeds. See `set_random_seed` for behavior.
- `dtype` : The data type. Only floating point types are supported.

**Returns:**

An initializer that generates tensors with a truncated normal distribution.

**Raises:**

- `ValueError` : if `dtype` is not a floating point type.

---

```
tf.random_uniform_initializer(minval=0.0,
maxval=1.0, seed=None, dtype=tf.float32)
```

Returns an initializer that generates tensors with a uniform distribution.

**Args:**

- `minval` : a python scalar or a scalar tensor. lower bound of the range of random values

to generate.

- `maxval` : a python scalar or a scalar tensor. upper bound of the range of random values to generate.
- `seed` : A Python integer. Used to create random seeds. See [set\\_random\\_seed](#) for behavior.
- `dtype` : The data type. Only floating point types are supported.

#### Returns:

An initializer that generates tensors with a uniform distribution.

#### Raises:

- `ValueError` : if `dtype` is not a floating point type.

---

```
tf.uniform_unit_scaling_initializer(factor=1.0, seed=None, dtype=tf.float32,
full_shape=None)
```

Returns an initializer that generates tensors without scaling variance.

When initializing a deep network, it is in principle advantageous to keep the scale of the input variance constant, so it does not explode or diminish by reaching the final layer. If the input is `x` and the operation `x * w`, and we want to initialize `w` uniformly at random, we need to pick `w` from

```
[-sqrt(3) / sqrt(dim), sqrt(3) / sqrt(dim)]
```

to keep the scale intact, where `dim = w.shape[0]` (the size of the input). A similar calculation for convolutional networks gives an analogous result with `dim` equal to the product of the first 3 dimensions. When nonlinearities are present, we need to multiply this by a constant factor . See [Sussillo et al., 2014 \(pdf\)](#) for deeper motivation, experiments and the calculation of constants. In section 2.3 there, the constants were numerically computed: for a linear layer it's 1.0, relu: ~1.43, tanh: ~1.15.

If the shape tuple `full_shape` is provided, the scale will be calculated from this predefined shape. This is useful when a `variable` is being partitioned across several shards, and each shard has a smaller shape than the whole. Since the shards are usually concatenated when used, the scale should be based on the shape of the whole.

#### Args:

- `factor` : Float. A multiplicative factor by which the values will be scaled.
- `seed` : A Python integer. Used to create random seeds. See [set\\_random\\_seed](#) for behavior.
- `dtype` : The data type. Only floating point types are supported.
- `full_shape` : Tuple or list of integers. The shape used for calculating scale normalization (instead of the shape passed at creation time). Useful when creating sharded variables via partitioning.

**Returns:**

An initializer that generates tensors with unit variance.

**Raises:**

- `ValueError` : if `dtype` is not a floating point type.

---

**`tf.zeros_initializer(shape, dtype=tf.float32)`**

An adaptor for zeros() to match the Initializer spec.

---

**`tf.ones_initializer(shape, dtype=tf.float32)`**

An adaptor for ones() to match the Initializer spec.

## Variable Partitioners for Sharding

---

**`tf.variable_axis_size_partitioner(max_shard_byt  
es, axis=0, bytes_per_string_element=16,  
max_shards=None)`**

Get a partitioner for VariableScope to keep shards below `max_shard_bytes`.

This partitioner will shard a Variable along one axis, attempting to keep the maximum shard size below `max_shard_bytes`. In practice, this is not always possible when sharding along only one axis. When this happens, this axis is sharded as much as possible (i.e., every dimension becomes a separate shard).

If the partitioner hits the `max_shards` limit, then each shard may end up larger than `max_shard_bytes`. By default `max_shards` equals `None` and no limit on the number of shards is enforced.

One reasonable value for `max_shard_bytes` is `(64 << 20) - 1`, or almost `64MB`, to keep below the protobuf byte limit.

#### Args:

- `max_shard_bytes` : The maximum size any given shard is allowed to be.
- `axis` : The axis to partition along. Default: outermost axis.
- `bytes_per_string_element` : If the `variable` is of type string, this provides an estimate of how large each scalar in the `variable` is.
- `max_shards` : The maximum number of shards in int created taking precedence over `max_shard_bytes`.

#### Returns:

A partition function usable as the `partitioner` argument to `variable_scope`, `get_variable`, and `get_partitioned_variable_list`.

#### Raises:

- `ValueError` : If any of the byte counts are non-positive.

## Sparse Variable Updates

The sparse update ops modify a subset of the entries in a dense `Variable`, either overwriting the entries or adding / subtracting a delta. These are useful for training embedding models and similar lookup-based networks, since only a small subset of embedding vectors change in any given step.

Since a sparse update of a large tensor may be generated automatically during gradient computation (as in the gradient of `tf.gather`), an `IndexedSlices` class is provided that encapsulates a set of sparse indices and values. `IndexedSlices` objects are detected and handled automatically by the optimizers in most cases.

---

```
tf.scatter_update(ref, indices, updates,
use_locking=None, name=None)
```

Applies sparse updates to a variable reference.

This operation computes

```
Scalar indices
ref[indices, ...] = updates[...]

Vector indices (for each i)
ref[indices[i], ...] = updates[i, ...]

High rank indices (for each i, ..., j)
ref[indices[i, ..., j], ...] = updates[i, ..., j, ...]
```

This operation outputs `ref` after the update is done. This makes it easier to chain operations that need to use the reset value.

If values in `ref` is to be updated more than once, because there are duplicate entires in `indices`, the order at which the updates happen for each value is undefined.

Requires `updates.shape = indices.shape + ref.shape[1:]`.

#### Args:

- `ref` : A mutable `Tensor`. Should be from a `Variable` node.
- `indices` : A `Tensor`. Must be one of the following types: `int32`, `int64`. A tensor of indices into the first dimension of `ref`.

- `updates` : A `Tensor`. Must have the same type as `ref`. A tensor of updated values to store in `ref`.
- `use_locking` : An optional `bool`. Defaults to `True`. If True, the assignment will be protected by a lock; otherwise the behavior is undefined, but may exhibit less contention.
- `name` : A name for the operation (optional).

**Returns:**

Same as `ref`. Returned as a convenience for operations that want to use the updated values after the update is done.

```
tf.scatter_add(ref, indices, updates,
use_locking=None, name=None)
```

Adds sparse updates to a variable reference.

This operation computes

```
Scalar indices
ref[indices, ...] += updates[...]

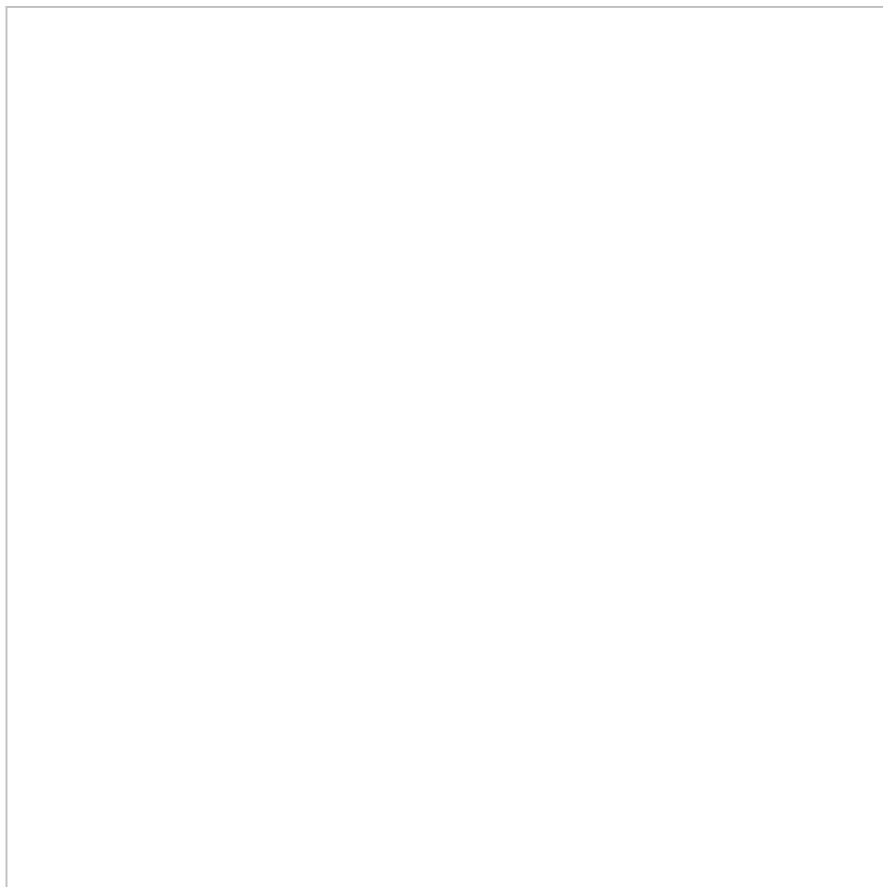
Vector indices (for each i)
ref[indices[i], ...] += updates[i, ...]

High rank indices (for each i, ..., j)
ref[indices[i, ..., j], ...] += updates[i, ..., j, ...]
```

This operation outputs `ref` after the update is done. This makes it easier to chain operations that need to use the reset value.

Duplicate entries are handled correctly: if multiple `indices` reference the same location, their contributions add.

Requires `updates.shape = indices.shape + ref.shape[1:]`.

**Args:**

- `ref` : A mutable `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`, `int16`, `int8`, `complex64`, `complex128`, `qint8`, `quint8`, `qint32`, `half`. Should be from a `Variable` node.
- `indices` : A `Tensor`. Must be one of the following types: `int32`, `int64`. A tensor of indices into the first dimension of `ref`.
- `updates` : A `Tensor`. Must have the same type as `ref`. A tensor of updated values to add to `ref`.
- `use_locking` : An optional `bool`. Defaults to `False`. If True, the addition will be protected by a lock; otherwise the behavior is undefined, but may exhibit less contention.
- `name` : A name for the operation (optional).

**Returns:**

Same as `ref`. Returned as a convenience for operations that want to use the updated values after the update is done.

---

```
tf.scatter_sub(ref, indices, updates,
use_locking=None, name=None)
```

Subtracts sparse updates to a variable reference.

```
Scalar indices
ref[indices, ...] -= updates[...]

Vector indices (for each i)
ref[indices[i], ...] -= updates[i, ...]

High rank indices (for each i, ..., j)
ref[indices[i, ..., j], ...] -= updates[i, ..., j, ...]
```

This operation outputs `ref` after the update is done. This makes it easier to chain operations that need to use the reset value.

Duplicate entries are handled correctly: if multiple `indices` reference the same location, their (negated) contributions add.

Requires `updates.shape = indices.shape + ref.shape[1:]`.

### Args:

- `ref` : A mutable `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`, `int16`, `int8`, `complex64`, `complex128`, `qint8`, `quint8`, `qint32`, `half`. Should be from a `Variable` node.

- `indices` : A `Tensor`. Must be one of the following types: `int32`, `int64`. A tensor of indices into the first dimension of `ref`.
- `updates` : A `Tensor`. Must have the same type as `ref`. A tensor of updated values to subtract from `ref`.
- `use_locking` : An optional `bool`. Defaults to `False`. If True, the subtraction will be protected by a lock; otherwise the behavior is undefined, but may exhibit less contention.
- `name` : A name for the operation (optional).

**Returns:**

Same as `ref`. Returned as a convenience for operations that want to use the updated values after the update is done.

**`tf.sparse_mask(a, mask_indices, name=None)`**

Masks elements of `IndexedSlices`.

Given an `IndexedSlices` instance `a`, returns another `IndexedSlices` that contains a subset of the slices of `a`. Only the slices at indices specified in `mask_indices` are returned.

This is useful when you need to extract a subset of slices in an `IndexedSlices` object.

For example:

```
`a` contains slices at indices [12, 26, 37, 45] from a large tensor
with shape [1000, 10]
a.indices => [12, 26, 37, 45]
tf.shape(a.values) => [4, 10]

`b` will be the subset of `a` slices at its second and third indices, so
we want to mask of its first and last indices (which are at absolute
indices 12, 45)
b = tf.sparse_mask(a, [12, 45])

b.indices => [26, 37]
tf.shape(b.values) => [2, 10]
```

**Args:**

- `a` : An `IndexedSlices` instance.
- `mask_indices` : Indices of elements to mask.
- `name` : A name for the operation (optional).

**Returns:**

The masked `IndexedSlices` instance.

## class `tf.IndexedSlices`

A sparse representation of a set of tensor slices at given indices.

This class is a simple wrapper for a pair of `Tensor` objects:

- `values` : A `Tensor` of any dtype with shape `[D0, D1, ..., Dn]` .
- `indices` : A 1-D integer `Tensor` with shape `[D0]` .

An `IndexedSlices` is typically used to represent a subset of a larger tensor `dense` of shape `[LARGE0, D1, ..., DN]` where `LARGE0 >> D0` . The values in `indices` are the indices in the first dimension of the slices that have been extracted from the larger tensor.

The dense tensor `dense` represented by an `IndexedSlices` `slices` has

```
dense[slices.indices[i], :, :, :, ..., ...] = slices.values[i, :, :, :, ..., ...]
```

The `IndexedSlices` class is used principally in the definition of gradients for operations that have sparse gradients (e.g. `tf.gather` ).

Contrast this representation with `SparseTensor` , which uses multi-dimensional indices and scalar values.

### `tf.IndexedSlices.__init__(values, indices, dense_shape=None)`

Creates an `IndexedSlices` .

### `tf.IndexedSlices.values`

A `Tensor` containing the values of the slices.

### `tf.IndexedSlices.indices`

A 1-D `Tensor` containing the indices of the slices.

---

**tf.IndexedSlices.dense\_shape**

A 1-D `Tensor` containing the shape of the corresponding dense tensor.

---

**tf.IndexedSlices.name**

The name of this `IndexedSlices`.

---

**tf.IndexedSlices.dtype**

The `DType` of elements in this tensor.

---

**tf.IndexedSlices.device**

The name of the device on which `values` will be produced, or `None`.

---

**tf.IndexedSlices.op**

The `Operation` that produces `values` as an output.

## Other Methods

---

**tf.IndexedSlices.graph**

The `Graph` that contains the values, indices, and shape tensors.

## Exporting and Importing Meta Graphs

---

```
tf.train.export_meta_graph(filename=None,
meta_info_def=None, graph_def=None,
saver_def=None, collection_list=None,
as_text=False)
```

Returns `MetaGraphDef` proto. Optionally writes it to filename.

This function exports the graph, saver, and collection objects into `MetaGraphDef` protocol buffer with the intention of it being imported at a later time or location to restart training, run inference, or be a subgraph.

#### Args:

- `filename` : Optional filename including the path for writing the generated `MetaGraphDef` protocol buffer.
- `meta_info_def` : `MetaInfoDef` protocol buffer.
- `graph_def` : `GraphDef` protocol buffer.
- `saver_def` : `SaverDef` protocol buffer.
- `collection_list` : List of string keys to collect.
- `as_text` : If `True`, writes the `MetaGraphDef` as an ASCII proto.

#### Returns:

A `MetaGraphDef` proto.

## `tf.train.import_meta_graph(meta_graph_or_file)`

Recreates a Graph saved in a `MetaGraphDef` proto.

This function takes a `MetaGraphDef` protocol buffer as input. If the argument is a file containing a `MetaGraphDef` protocol buffer, it constructs a protocol buffer from the file content. The function then adds all the nodes from the `graph_def` field to the current graph, recreates all the collections, and returns a saver constructed from the `saver_def` field.

In combination with `export_meta_graph()`, this function can be used to

- Serialize a graph along with other Python objects such as `QueueRunner`, `Variable` into a `MetaGraphDef`.
- Restart training from a saved graph and checkpoints.
- Run inference from a saved graph and checkpoints.

```

...
Create a saver.
saver = tf.train.Saver(...variables...)
Remember the training_op we want to run by adding it to a collection.
tf.add_to_collection('train_op', train_op)
sess = tf.Session()
for step in xrange(1000000):
 sess.run(train_op)
 if step % 1000 == 0:
 # Saves checkpoint, which by default also exports a meta_graph
 # named 'my-model-global_step.meta'.
 saver.save(sess, 'my-model', global_step=step)

```

Later we can continue training from this saved `meta_graph` without building the model from scratch.

```

with tf.Session() as sess:
 new_saver = tf.train.import_meta_graph('my-save-dir/my-model-10000.meta')
 new_saver.restore(sess, 'my-save-dir/my-model-10000')
 # tf.get_collection() returns a list. In this example we only want the
 # first one.
 train_op = tf.get_collection('train_op')[0]
 for step in xrange(1000000):
 sess.run(train_op)

```

NOTE: Restarting training from saved `meta_graph` only works if the device assignments have not changed.

### Args:

- `meta_graph_or_file` : `MetaGraphDef` protocol buffer or filename (including the path) containing a `MetaGraphDef`.

### Returns:

A saver constructed from `saver_def` in `MetaGraphDef` or None.

A None value is returned if no variables exist in the `MetaGraphDef` (i.e., there are no variables to restore).

# 텐서 변환

참고: `Tensor` 를 인자로 받는 함수들은, `tf.convert_to_tensor` 의 인자가 될 수 있는 것들 또한 인자로 받을 수 있습니다.

## 형변환 (Casting)

TensorFlow는 그래프에 사용되는 텐서 자료형들을 형변환(cast)할 수 있는 몇 가지 함수를 제공합니다.

### `tf.string_to_number(string_tensor, out_type=None, name=None)`

입력 텐서의 각 문자열(string)을 지정된 자료형의 값으로 변환합니다.

(참고로, `int32` 오버플로우는 에러를 내며, `float` 오버플로우는 반올림한 결과를 냅니다.)

인자:

- `string_tensor` : `string` 형 `Tensor`.
- `out_type` : `tf.DType` 오브젝트. `tf.float32` 또는 `tf.int32` 이어야 하며, 기본값은 `tf.float32`입니다. 이 자료형으로 `string_tensor` 의 문자열이 변환됩니다. (선택사항)
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

`out_type` 형의 `Tensor`. 출력 텐서는 입력 텐서 `string_tensor` 와 같은 구조(shape)를 가집니다.

### `tf.to_double(x, name='ToDouble')`

텐서를 `float64` 형으로 변환합니다.

인자:

- `x` : `Tensor` 또는 `SparseTensor`.
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

x 와 구조(shape)가 같은 float64 형의 Tensor 또는 SparseTensor .

예외:

- **TypeError** : x 가 float64 형으로 변환될 수 없는 경우.
- 

### **tf.to\_float(x, name='ToFloat')**

텐서를 float32 형으로 변환합니다.

인자:

- x : Tensor 또는 SparseTensor .
- name : 오퍼레이션의 명칭. (선택사항)

반환값:

x 와 구조(shape)가 같은 float32 형의 Tensor 또는 SparseTensor .

예외:

- **TypeError** : x 가 float32 형으로 변환될 수 없는 경우.
- 

### **tf.to\_bfloat16(x, name='ToBFloat16')**

텐서를 bfloat16 형으로 변환합니다.

인자:

- x : Tensor 또는 SparseTensor .
- name : 오퍼레이션의 명칭. (선택사항)

반환값:

x 와 구조(shape)가 같은 bfloat16 형의 Tensor 또는 SparseTensor .

예외:

- **TypeError** : x 가 bfloat16 형으로 변환될 수 없는 경우.
-

## **tf.to\_int32(x, name='ToInt32')**

텐서를 `int32` 형으로 변환합니다.

인자:

- `x` : `Tensor` 또는 `SparseTensor`.
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

`x` 와 구조(shape)가 같은 `int32` 형의 `Tensor` 또는 `SparseTensor`.

예외:

- `TypeError` : `x` 가 `int32` 형으로 변환될 수 없는 경우.
- 

## **tf.to\_int64(x, name='ToInt64')**

텐서를 `int64` 형으로 변환합니다.

인자:

- `x` : `Tensor` 또는 `SparseTensor`.
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

`x` 와 구조(shape)가 같은 `int64` 형의 `Tensor` 또는 `SparseTensor`.

예외:

- `TypeError` : `x` 가 `int64` 형으로 변환될 수 없는 경우.
- 

## **tf.cast(x, dtype, name=None)**

텐서를 새로운 자료형으로 변환합니다.

`x` (`Tensor` 의 경우) 또는 `x.values` (`SparseTensor` 의 경우)를 `dtype` 형으로 변환합니다.

예시:

```
텐서 `a`는 [1.8, 2.2], 자료형은 tf.float
tf.cast(a, tf.int32) ==> [1, 2] # dtype=tf.int32
```

**인자:**

- `x` : `Tensor` 또는 `SparseTensor`.
- `dtype` : 변환될 자료형.
- `name` : 오퍼레이션의 명칭. (선택사항)

**반환값:**

`x` 와 구조(shape)가 같은 `int64` 형의 `Tensor` 또는 `SparseTensor`.

**예외:**

- `TypeError` : `x` 가 `dtype` 형으로 변환될 수 없는 경우.

**`tf.saturate_cast(value, dtype, name=None)`**

`value` 를 `dtype` 형으로 안전하게 포화 형변환(saturating cast)합니다.

이 함수는 입력을 `dtype` 으로 스케일링(scaling) 없이 변환합니다. 형변환 시 오버플로우나 언더플로우가 발생할 수 있는 값들에 대해, 이 함수는 해당 값들을 허용되는 값 범위로 넣은 뒤 형변환을 진행합니다.

**인자:**

- `value` : `Tensor`.
- `dtype` : `DType` 오브젝트. 변환될 자료형.
- `name` : 오퍼레이션의 명칭. (선택사항)

**반환값:**

`dtype` 형으로 안전하게 변환된 `value`.

**구조(Shape) 및 구조 변형(Shaping)**

TensorFlow는 텐서의 구조(shape)를 확인하거나 구조를 변형하는 데 사용할 수 있는 몇 가지 함수를 제공합니다.

## tf.shape(input, name=None)

텐서의 구조(shape)를 반환합니다.

이 함수는 `input` 텐서의 구조(shape)를 1-D 정수형 텐서로 반환합니다.

예시:

```
't'는 [[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]]
shape(t) ==> [2, 2, 3]
```

인자:

- `input` : Tensor .
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

```
int32 형 Tensor .
```

---

## tf.size(input, name=None)

텐서의 크기(size)를 반환합니다.

이 함수는 `input` 텐서의 원소의 수를 정수로 반환합니다.

예시:

```
't'는 [[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]]
size(t) ==> 12
```

인자:

- `input` : Tensor .
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

```
int32 형 Tensor .
```

---

## tf.rank(input, name=None)

텐서의 랭크(rank)를 반환합니다.

이 함수는 `input` 텐서의 랭크를 정수로 반환합니다.

예시:

```
't' is [[[1, 1, 1], [2, 2, 2]], [[3, 3, 3], [4, 4, 4]]]
shape of tensor 't' is [2, 2, 3]
rank(t) ==> 3
```

참고: 텐서의 랭크는 행렬의 랭크와는 다른 개념입니다. 텐서의 랭크는 텐서의 각 원소를 선택하기 위해 필요한 인덱스의 수입니다. 랭크는 `order`, `degree`, `ndims` 등으로 부르기도 합니다.

인자:

- `input` : `Tensor` 또는 `SparseTensor`.
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

```
int32 형 Tensor .
```

---

## tf.reshape(tensor, shape, name=None)

텐서의 구조를 변형합니다.

`tensor` 가 주어졌을 때, 이 함수는 해당 텐서와 같은 원소들을 가지며 구조가 `shape` 인 텐서를 반환합니다.

만약 `shape` 의 한 원소가 -1이라면, 전체 크기가 일정하게 유지되도록 해당 차원의 길이가 자동으로 계산됩니다. 특별히, `shape` 가 `[-1]` 이라면, 텐서는 1-D로 펴지게 됩니다. `shape` 에서 최대한 개의 원소만 -1이 될 수 있습니다.

만약 `shape` 가 1-D이거나 그 이상이라면, 오퍼레이션은 `tensor` 의 원소로 `shape` 의 구조가 채워진 텐서를 반환합니다. 이 경우, `shape` 에 의해 지정된 원소의 전체 수는 `tensor` 의 원소의 전체 수와 동일해야 합니다.

예시:

```

tensor 't'는 [1, 2, 3, 4, 5, 6, 7, 8, 9]
tensor 't'의 구조(shape)는 [9]
reshape(t, [3, 3]) ==> [[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]

tensor 't'는 [[[1, 1], [2, 2]],
[[3, 3], [4, 4]]]
tensor 't'의 구조(shape)는 [2, 2, 2]
reshape(t, [2, 4]) ==> [[1, 1, 2, 2],
 [3, 3, 4, 4]]

tensor 't'는 [[[1, 1, 1],
[2, 2, 2]],
[[3, 3, 3],
[4, 4, 4]],
[[5, 5, 5],
[6, 6, 6]]]
tensor 't'의 구조(shape)는 [3, 2, 3]
shape를 '[-1]'로 하여 't'를 1-D로 펴기
reshape(t, [-1]) ==> [1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6]

구조를 암시(infer)하기 위한 -1의 사용

-1은 9를 의미:
reshape(t, [2, -1]) ==> [[1, 1, 1, 2, 2, 2, 3, 3, 3],
 [4, 4, 4, 5, 5, 5, 6, 6, 6]]
-1은 2를 의미:
reshape(t, [-1, 9]) ==> [[1, 1, 1, 2, 2, 2, 3, 3, 3],
 [4, 4, 4, 5, 5, 5, 6, 6, 6]]
-1은 3을 의미:
reshape(t, [2, -1, 3]) ==> [[[1, 1, 1],
 [2, 2, 2],
 [3, 3, 3]],
 [[4, 4, 4],
 [5, 5, 5],
 [6, 6, 6]]]

tensor 't'는 [7]
shape를 `[]`로 하면 스칼라(scalar)로 구조 변환
reshape(t, []) ==> 7

```

**인자:**

- **tensor** : Tensor .
- **shape** : int32 형 Tensor . 출력 텐서의 구조(shape) 지정.
- **name** : 오퍼레이션의 명칭. (선택사항)

**반환값:**

`tensor` 와 같은 자료형의 `Tensor`.

## `tf.squeeze(input, squeeze_dims=None, name=None)`

텐서에서 크기 1인 차원을 제거합니다.

`input` 텐서가 주어졌을 때, 이 함수는 그와 같은 자료형의 크기 1인 차원이 모두 제거된 텐서를 반환합니다. 만약 모든 크기 1인 차원을 제거하고 싶은 것이 아니라면, 제거하고 싶은 특정한 크기 1인 차원들을 `squeeze_dims` 으로 지정할 수 있습니다.

예시:

```
't'는 구조(shape) [1, 2, 1, 3, 1, 1]의 텐서
shape(squeeze(t)) ==> [2, 3]
```

제거할 크기 1인 차원들을 `squeeze_dims` 으로 지정하기:

```
't'는 구조(shape) [1, 2, 1, 3, 1, 1]의 텐서
shape(squeeze(t, [2, 4])) ==> [1, 2, 3, 1]
```

인자:

- `input` : `Tensor`.
- `squeeze_dims` : `int` 의 리스트. 기본값은 `[]`. 지정된 경우, 리스트 안의 차원만 제거합니다. 크기가 1이 아닌 차원을 제거하는 것은 오류입니다. (선택사항)
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

`input` 과 같은 자료형의 `Tensor`. `input` 과 같은 원소를 포함하지만, 하나 이상의 크기 1인 차원이 제거되어 있습니다.

## `tf.expand_dims(input, dim, name=None)`

크기 1인 차원을 텐서의 구조(shape)에 삽입합니다.

`input` 텐서가 주어졌을 때, 이 함수는 크기가 1인 차원을 `input` 의 구조에서 차원 인덱스 `dim` 에 삽입합니다. 차원 인덱스 `dim` 은 0부터 시작합니다. 만약 `dim` 에 음수가 지정된다면, 끝에서부터 역으로 계산됩니다.

이 함수는 단일 원소에 배치 차원(batch dimension)을 추가할 때 유용합니다. 예로, 만약 구조 [height, width, channels] 의 단일 이미지가 있는 경우, 이것에 `expand_dims(image, 0)` 을 적용해 구조 [1, height, width, channels] 인 하나의 이미지로 구성된 배치(batch)를 구성할 수 있습니다.

다른 예시들:

```
't'는 구조(shape) [2]의 텐서
shape(expand_dims(t, 0)) ==> [1, 2]
shape(expand_dims(t, 1)) ==> [2, 1]
shape(expand_dims(t, -1)) ==> [2, 1]

't2'는 구조(shape) [2, 3, 5]의 텐서
shape(expand_dims(t2, 0)) ==> [1, 2, 3, 5]
shape(expand_dims(t2, 2)) ==> [2, 3, 1, 5]
shape(expand_dims(t2, 3)) ==> [2, 3, 5, 1]
```

이 함수는 다음의 조건이 만족되어야 합니다:

```
-1 <= input.dims() <= dim <= input.dims()
```

이 함수는 크기 1인 차원을 제거하는 함수인 `squeeze()` 와 연관되어 있습니다.

인자:

- `input` : `Tensor` .
- `dim` : `int32` 형 `Tensor` . 0-D (스칼라). `input` 의 구조에서 어떤 차원 인덱스에 삽입할 것인지 지정합니다.
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

`input` 과 같은 자료형의 `Tensor` . `input` 과 같은 원소를 포함하지만, 하나 이상의 크기 1인 차원이 추가되어 있습니다.

## 자르고 붙이기

TensorFlow는 텐서를 자르고 특정 부분을 추출해내거나, 여러 텐서를 붙일 수 있는 몇 가지 함수를 제공합니다.

---

**`tf.slice(input_, begin, size, name=None)`**

텐서의 특정 부분을 추출합니다.

이 함수는 텐서 `input`에서 `begin` 위치에서 시작해 크기 `size`인 부분을 추출합니다. 추출할 부분의 `size`는 텐서 구조(shape)로 표현되는데, `size[i]`가 `input`에서 추출할 `i` 번째 차원의 원소의 수입니다. 추출 부분의 시작 위치 `begin`은 각 차원에서의 오프셋(offset)으로 표현됩니다. 즉, `begin[i]`는 `input`의 `i` 번째 차원에서 시작 위치의 오프셋입니다.

`begin`은 0부터 시작하고, `size`는 1부터 시작합니다. 만약 `size[i]`가 -1이라면, 차원 `i`의 모든 남은 원소들이 추출 부분에 포함됩니다. 즉, 이는 아래와 같이 설정하는 것과 동일합니다.

```
size[i] = input.dim_size(i) - begin[i]
```

이 함수는 다음의 조건이 만족되어야 합니다:

```
0 <= begin[i] <= begin[i] + size[i] <= Di for i in [0, n]
```

예시:

```
'input'은 [[[1, 1, 1], [2, 2, 2]],
[[3, 3, 3], [4, 4, 4]],
[[5, 5, 5], [6, 6, 6]]]
tf.slice(input, [1, 0, 0], [1, 1, 3]) ==> [[[3, 3, 3]]]
tf.slice(input, [1, 0, 0], [1, 2, 3]) ==> [[[3, 3, 3],
 [4, 4, 4]]]
tf.slice(input, [1, 0, 0], [2, 1, 3]) ==> [[[3, 3, 3]],
 [[5, 5, 5]]]
```

인자:

- `input_` : `Tensor`.
- `begin` : `int32` 또는 `int64` 형 `Tensor`.
- `size` : `int32` 또는 `int64` 형 `Tensor`.
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

`input` 과 같은 자료형의 `Tensor`.

```
tf.split(split_dim, num_split, value,
name='split')
```

한 차원을 따라서 입력된 텐서를 `num_split` 개의 텐서로 분리합니다.

`split_dim` 차원을 따라서 `value`를 `num_split` 개의 작은 텐서로 분리합니다. `num_split` 0이 `value.shape[split_dim]` 을 나눌 수 있어야 합니다.

예시:

```
'value'는 구조(shape) [5, 30]의 텐서
'value'를 차원 1을 따라 3개의 텐서로 분리
split0, split1, split2 = tf.split(1, 3, value)
tf.shape(split0) ==> [5, 10]
```

**인자:**

- **split\_dim** : 0-D `int32 Tensor`. 텐서를 분리할 차원. `[0, rank(value))` 범위 내에 있어야 합니다.
- **num\_split** : Python 정수. 텐서를 분리할 개수.
- **value** : 분리할 `Tensor`.
- **name** : 오퍼레이션의 명칭. (선택사항)

**반환값:**

`value` 를 분리하여 얻어진 `num_split` 개의 `Tensor`.

## **tf.tile(input, multiples, name=None)**

주어진 텐서를 타일링(tiling)하여 새로운 텐서를 만듭니다.

이 함수는 주어진 텐서 `input` 을 `multiples` 회 복사하여 새로운 텐서를 만듭니다. 출력 텐서의 `i` 번째 차원은 `input.dims(i) * multiples[i]` 개의 원소를 가지고, 이는 `input` 의 원소들이 `multiples[i]` 번 복사된 것에 해당합니다. 예로, `[a b c d]` 를 `[2]` 로 타일링하면 `[a b c d a b c d]` 를 얻습니다.

**인자:**

- **input** : 1-D 혹은 그 이상의 `Tensor`.
- **multiples** : `int32` 형 `Tensor`. 1-D. 길이는 `input` 의 차원의 수와 같아야 합니다.
- **name** : 오퍼레이션의 명칭. (선택사항)

**반환값:**

`input` 과 같은 자료형의 `Tensor`.

## **tf.pad(tensor, paddings, mode='CONSTANT', name=None)**

텐서에 패딩을 적용합니다.

이 함수는 지정한 `paddings`에 따라 `tensor`에 패딩을 적용합니다. `padding`은 구조(shape)가 `[n, 2]`인 정수형 텐서이고, 여기서 `n`은 `tensor`의 랭크(rank)입니다. `input`의 각각의 차원 `D`에 대해, `paddings[D, 0]`은 `tensor`의 원소 앞에 몇 개의 값을 넣을 것인지, `paddings[D, 1]`은 `tensor`의 원소 뒤에 몇 개의 값을 넣을 것인지를 나타냅니다. 만약 `mode`가 "REFLECT"라면, `paddings[D, 0]`과 `paddings[D, 1]` 모두 `tensor.dim_size(D) - 1` 보다 크지 않아야 합니다. `mode`가 "SYMMETRIC"이라면, `paddings[D, 0]`과 `paddings[D, 1]` 모두 `tensor.dim_size(D)` 보다 크지 않아야 합니다.

패딩이 이루어진 뒤 출력에서 차원 `D`의 길이는 다음과 같습니다.

```
paddings[D, 0] + tensor.dim_size(D) + paddings[D, 1]
```

예시:

```
't'는 [[1, 2, 3], [4, 5, 6]].
'paddings'는 [[1, 1], [2, 2]].
't'의 랭크(rank)는 2.
pad(t, paddings, "CONSTANT") ==> [[0, 0, 0, 0, 0, 0, 0],
 [0, 0, 1, 2, 3, 0, 0],
 [0, 0, 4, 5, 6, 0, 0],
 [0, 0, 0, 0, 0, 0, 0]]

pad(t, paddings, "REFLECT") ==> [[6, 5, 4, 5, 6, 5, 4],
 [3, 2, 1, 2, 3, 2, 1],
 [6, 5, 4, 5, 6, 5, 4],
 [3, 2, 1, 2, 3, 2, 1]]

pad(t, paddings, "SYMMETRIC") ==> [[2, 1, 1, 2, 3, 3, 2],
 [2, 1, 1, 2, 3, 3, 2],
 [5, 4, 4, 5, 6, 6, 5],
 [5, 4, 4, 5, 6, 6, 5]]
```

인자:

- `tensor` : Tensor .
- `paddings` : int32 형 Tensor .
- `mode` : "CONSTANT", "REFLECT", "SYMMETRIC" 중 하나.
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

```
tensor 와 같은 자료형의 Tensor .
```

예외:

- `ValueError` : 모드가 "CONSTANT", "REFLECT", or "SYMMETRIC" 중의 하나가 아닌 경우.

**tf.concat(concat\_dim, values, name='concat' )**

텐서들을 하나의 차원에서 이어붙입니다.

텐서들의 리스트 `values` 를 차원 `concat_dim` 에서 이어붙입니다. 만약 `values[i].shape = [D0, D1, ... Dconcat_dim(i), ... Dn]` 이면, 이어붙인 결과의 구조(shape)는 아래와 같습니다.

```
[D0, D1, ... Rconcat_dim, ... Dn]
```

여기서,

```
Rconcat_dim = sum(Dconcat_dim(i))
```

입니다. 즉, `concat_dim` 차원을 따라 입력된 텐서들의 데이터가 연결됩니다.

입력할 텐서들의 차원의 수는 모두 동일해야 하며, `concat_dim` 차원을 제외한 모든 차원의 길이가 동일해야 합니다.

예시:

```
t1 = [[1, 2, 3], [4, 5, 6]]
t2 = [[7, 8, 9], [10, 11, 12]]
tf.concat(0, [t1, t2]) ==> [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
tf.concat(1, [t1, t2]) ==> [[1, 2, 3, 7, 8, 9], [4, 5, 6, 10, 11, 12]]

tensor t3의 구조(shape)는 [2, 3]
tensor t4의 구조(shape)는 [2, 3]
tf.shape(tf.concat(0, [t3, t4])) ==> [4, 3]
tf.shape(tf.concat(1, [t3, t4])) ==> [2, 6]
```

인자:

- `concat_dim` : 0-D `int32` 형 `Tensor`. 텐서를 이어붙일 차원.
- `values` : `Tensor` 들의 리스트 또는 `Tensor`.
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

입력 텐서들을 이어붙여 만든 `Tensor`.

**tf.pack(values, name='pack' )**

랭크- R 텐서들을 묶어 하나의 랭크- (R+1) 텐서로 만듭니다.

`values` 안의 텐서들을 묶어 랭크가 `values`에 있는 각 텐서보다 1 높고, `[len(values)] + values[0].shape`의 구조(shape)를 가지는 텐서를 만듭니다. 출력 값이 `output`이라 하면, `output[i, ...] = values[i][...]`을 만족합니다.

`unpack` 과 반대 기능을 합니다. NumPy에서 `asarray` 와 같은 기능을 합니다.

```
tf.pack([x, y, z]) = np.asarray([x, y, z])
```

인자:

- `values` : 같은 구조(shape)와 자료형을 가지는 `Tensor`들의 리스트.
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

- `output` : `values`와 같은 자료형의 `Tensor`.

## tf.unpack(value, num=None, name='unpack')

랭크- R 텐서를 풀어 랭크- (R-1) 텐서들을 얻습니다.

`value`의 첫 번째 차원을 따라 텐서를 풀어 `num` 개의 텐서를 얻습니다. 만약 `num`을 지정하지 않았다면, `value`의 구조(shape)에서 자동으로 추정됩니다. 만약 `value.shape[0]`이 알려지지 않았다면, `ValueError` 예외가 발생합니다.

`output`의 `i` 번째 텐서는 `value[i, ...]` 가 됩니다. `output`의 각각의 텐서는 `value.shape[1:]`의 구조를 가지게 됩니다.

`pack` 함수와 반대 기능을 합니다. NumPy에서는 `list()`로 쓸 수 있습니다.

```
tf.unpack(x, n) = list(x)
```

인자:

- `value` : 랭크  $R > 0$ 인 `Tensor`.
- `num` : `int`. `value`의 첫 번째 차원. 기본값은 `None`이며, 이 경우 자동으로 추정됩니다. (선택사항)
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

`value`를 풀어 얻은 `Tensor`의 리스트.

## 예외:

- `ValueError` : `num` 이 지정되지 않았고 자동으로 추정할 수 없는 경우.

```
tf.reverse_sequence(input, seq_lengths,
seq_dim, batch_dim=None, name=None)
```

텐서를 특정한 부분을 반전시킵니다.

이 함수는 먼저 `input` 을 `batch_dim` 차원을 따라 여러 개의 슬라이스로 나눕니다. 그리고 각각의 슬라이스 `i` 에서 `seq_dim` 차원을 따라 처음 `seq_lengths[i]` 개의 원소들이 반전됩니다.

`seq_lengths` 의 원소들은 `seq_lengths[i] < input.dims[seq_dim]` 을 만족해야 하고, `seq_lengths` 는 길이 `input.dims[batch_dim]` 의 벡터여야 합니다.

반환될 텐서에서 `batch_dim` 차원의 `i` 번째 슬라이스는, 입력 텐서의 `i` 번째 슬라이스에서 `seq_dim` 차원을 따라 첫 `seq_lengths[i]` 개의 원소들이 반전된 것과 같습니다.

## 예시:

```
다음과 같이 설정합니다.
batch_dim = 0
seq_dim = 1
input.dims = (4, 8, ...)
seq_lengths = [7, 2, 3, 5]

입력 텐서의 각각의 슬라이스는 seq_dim 차원에서 seq_lengths 까지 반전됩니다.
output[0, 0:7, :, ...] = input[0, 7:0:-1, :, ...]
output[1, 0:2, :, ...] = input[1, 2:0:-1, :, ...]
output[2, 0:3, :, ...] = input[2, 3:0:-1, :, ...]
output[3, 0:5, :, ...] = input[3, 5:0:-1, :, ...]

seq_lengths 이후의 부분은 그대로 들어갑니다.
output[0, 7:, :, ...] = input[0, 7:, :, ...]
output[1, 2:, :, ...] = input[1, 2:, :, ...]
output[2, 3:, :, ...] = input[2, 3:, :, ...]
output[3, 2:, :, ...] = input[3, 2:, :, ...]
```

## 다른 예시:

```
다음과 같이 설정합니다.
batch_dim = 2
seq_dim = 0
input.dims = (8, ?, 4, ...)
seq_lengths = [7, 2, 3, 5]

입력 텐서의 각각의 슬라이스는 seq_dim 차원에서 seq_lengths 까지 반전됩니다.
output[0:7, :, 0, :, ...] = input[7:0:-1, :, 0, :, ...]
output[0:2, :, 1, :, ...] = input[2:0:-1, :, 1, :, ...]
output[0:3, :, 2, :, ...] = input[3:0:-1, :, 2, :, ...]
output[0:5, :, 3, :, ...] = input[5:0:-1, :, 3, :, ...]

seq_lengths 이후의 부분은 그대로 들어갑니다.
output[7:, :, 0, :, ...] = input[7:, :, 0, :, ...]
output[2:, :, 1, :, ...] = input[2:, :, 1, :, ...]
output[3:, :, 2, :, ...] = input[3:, :, 2, :, ...]
output[2:, :, 3, :, ...] = input[2:, :, 3, :, ...]
```

**인자:**

- `input` : `Tensor` . 반전시킬 텐서.
- `seq_lengths` : `int64` 형 1-D `Tensor` . 길이는 `input.dims(batch_dim)` 이며, `max(seq_lengths) < input.dims(seq_dim)` 을 만족합니다.
- `seq_dim` : `int` . (부분적으로) 반전되는 차원.
- `batch_dim` : `int` . 텐서의 반전이 이루어지는 차원, 기본값은 `0` . (선택사항)
- `name` : 오퍼레이션의 명칭. (선택사항)

**반환값:**

`input` 과 같은 자료형과 구조(shape)의 `Tensor` . `input` 의 일부분이 반전되어 있습니다.

**`tf.reverse(tensor, dims, name=None)`**

텐서의 특정 차원을 반전시킵니다.

`tensor` 와 그 텐서의 각 차원에 해당하는 `bool` 형 텐서 `dims` 가 주어졌을 때, 이 함수는 `dims[i]` 가 `True` 인 경우 `tensor` 의 차원 `i` 를 반전시킵니다.

`tensor` 는 차원을 8개까지 가질 수 있습니다. `tensor` 의 차원의 수는 `dims` 의 원소의 수와 동일해야 합니다. 즉, 다음의 식이 성립해야 합니다.

```
rank(tensor) = size(dims)
```

**예시:**

```

tensor 't'는 [[[[0, 1, 2, 3],
[4, 5, 6, 7],
[8, 9, 10, 11]],
[[12, 13, 14, 15],
[16, 17, 18, 19],
[20, 21, 22, 23]]]]
tensor 't'의 구조(shape)는 [1, 2, 3, 4]

'dims'가 [False, False, False, True] 일 때
reverse(t, dims) ==> [[[[3, 2, 1, 0],
 [7, 6, 5, 4],
 [11, 10, 9, 8]],
 [[15, 14, 13, 12],
 [19, 18, 17, 16],
 [23, 22, 21, 20]]]]

'dims'가 [False, True, False, False] 일 때
reverse(t, dims) ==> [[[[12, 13, 14, 15],
 [16, 17, 18, 19],
 [20, 21, 22, 23]],
 [[0, 1, 2, 3],
 [4, 5, 6, 7],
 [8, 9, 10, 11]]]]

'dims'가 [False, False, True, False] 일 때
reverse(t, dims) ==> [[[[8, 9, 10, 11],
 [4, 5, 6, 7],
 [0, 1, 2, 3]],
 [[20, 21, 22, 23],
 [16, 17, 18, 19],
 [12, 13, 14, 15]]]]

```

**인자:**

- `tensor` : `Tensor`. 자료형이 `uint8`, `int8`, `int32`, `bool`, `half`, `float32`, `float64` 중 하나여야 합니다. Up to 8-D.
- `dims` : `bool` 형 1-D `Tensor`. 반전시킬 차원을 나타냅니다.
- `name` : 오퍼레이션의 명칭. (선택사항)

**반환값:**

`tensor` 와 자료형과 구조(shape)가 같은 `Tensor`.

---

**`tf.transpose(a, perm=None, name='transpose')`**

`a` 를 전치합니다. `perm` 에 따라 차원의 순서를 구성합니다.

반환되는 텐서의 차원 `i` 는 입력되는 텐서의 차원 `perm[i]` 에 해당합니다. 만약 `perm` 이 주어지지 않을 경우, ( $n-1 \dots 0$ )으로 설정됩니다. 여기서  $n$ 은 입력 텐서의 랭크(rank)입니다. 따라서, 기본적으로 이 함수는 2-D 텐서가 입력될 경우 일반적인 행렬 전치를 수행합니다.

예시:

```
'x'는 [[1 2 3]
[4 5 6]]
tf.transpose(x) ==> [[1 4]
 [2 5]
 [3 6]]

perm의 기본값과 동일한 경우
tf.transpose(x, perm=[1, 0]) ==> [[1 4]
 [2 5]
 [3 6]]

'perm'은 차원이 n > 2인 텐서일 경우 더 유용합니다.
'x'는 [[[1 2 3]
[4 5 6]
[[7 8 9]
[10 11 12]]]
차원-0의 행렬들에 대해서 전치를 수행합니다.
tf.transpose(x, perm=[0, 2, 1]) ==> [[[1 4]
 [2 5]
 [3 6]

 [[7 10]
 [8 11]
 [9 12]]]]
```

인자:

- `a` : `Tensor`.
- `perm` : `a` 의 차원들의 순열.
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

전치된 `Tensor`.

```
tf.extract_image_patches(images, padding,
ksizes=None, strides=None, rates=None,
name=None)
```

`images`에서 패치(patch)들을 추출하여 출력의 "depth" 차원에 넣습니다.

**인자:**

- **images** : `Tensor` . 자료형이 `float32` , `float64` , `int32` , `int64` , `uint8` , `int16` , `int8` , `uint16` , `half` 중 하나여야 합니다. 구조(shape)가 `[batch, in_rows, in_cols, depth]` 인 4-D 텐서입니다.
- **padding** : `"SAME"` 또는 `"VALID"` . `string` . 사용할 패딩 알고리즘을 선택합니다.

크기와 관련된 인자는 다음과 같이 정해집니다:

```
ksizes = [1, ksize_rows, ksize_cols, 1]
strides = [1, strides_rows, strides_cols, 1]
rates = [1, rates_rows, rates_cols, 1]
```

- **ksizes** : `int` 들의 리스트. 기본값은 `[]` . `images` 의 각 차원에 대한 슬라이딩 윈도우의 크기를 지정합니다. (선택사항)
- **strides** : `int` 들의 리스트. 기본값은 `[]` . 길이 4의 1-D 텐서. 이미지에서 추출할 두 `patch` 사이의 중심 거리를 지정합니다. `[1, stride_rows, stride_cols, 1]` 와 같은 형태여야 합니다. (선택사항)
- **rates** : `int` 들의 리스트. 기본값은 `[]` . 길이 4의 1-D 텐서. 입력의 스트라이드로, 입력에서 두 연속된 `patch` 샘플들이 얼마나 멀리 떨어져 있어야 할지 지정합니다. `[1, rate_rows, rate_cols, 1]` 와 같은 형태여야 합니다. (선택사항) `patch` 를 추출할 때 `patch_sizes_eff = patch_sizes + (patch_sizes - 1) * (rates - 1)` 으로 놓고 공간적으로 `rates` 의 인자로 부차추출(subsampling)하는 것과 동일합니다.
- **name** : 오퍼레이션의 명칭. (선택사항)

**반환값:**

`images` 와 자료형이 같은 `Tensor` . 구조(shape)가 `[batch, out_rows, out_cols, ksize_rows * ksize_cols * depth]` 인 4-D 텐서입니다. 크기가 `ksize_rows x ksize_cols x depth` 인 "depth" 차원에서 벡터화된 이미지 패치(patch)들을 포함합니다.

```
tf.space_to_batch(input, paddings, block_size, name=None)
```

타입 T의 4-D 텐서에 대한 SpaceToBatch 함수입니다.

텐서에 제로 패딩을 붙이고 공간적인 데이터의 블록을 `batch`로 재배열합니다. 구체적으로, 이 함수는 입력 텐서의 `height` 와 `width` 차원이 `batch` 차원으로 옮겨진 복사본을 반환합니다. 제로 패딩이 행해진 뒤, 입력 텐서의 `height` 와 `width` 값 모두 블록 크기로 나눌 수 있어야 합니다.

**인자:**

- **input** : 구조(shape)가 [batch, height, width, depth] 인 4-D Tensor .
- **paddings** : int32 형 2-D Tensor . 구조(shape)가 [2, 2] 이며, 음이 아닌 정수로 구성됩니다. 입력을 공간 차원에서 어떻게 패딩할 것인지에 대해 다음과 같은 형태로 지정합니다.

```
paddings = [[pad_top, pad_bottom], [pad_left, pad_right]]
```

입력 텐서에서 패딩이 이루어지면 다음과 같은 공간 차원들을 가지게 됩니다.

```
height_pad = pad_top + height + pad_bottom
width_pad = pad_left + width + pad_right
```

**block\_size** 는 1보다 커야 합니다. 이는 블록의 크기를 지정합니다.

- **height** 와 **width** 차원에서 **block\_size** x **block\_size** 의 크기를 가지는 블록들은 각 위치에서 **batch** 차원으로 재배열됩니다.
- 출력 텐서의 배치(batch)는 **batch** \* **block\_size** \* **block\_size** 와 같습니다.
- **block\_size** 가 **height\_pad** 와 **width\_pad** 의 약수여야 합니다.

출력 텐서의 구조는 다음과 같습니다.

```
[batch*block_size*block_size, height_pad/block_size, width_pad/block_size,
depth]
```

- **block\_size** : int .
- **name** : 오퍼레이션의 명칭. (선택사항)

반환값:

**input** 과 같은 자료형의 Tensor .

## **tf.batch\_to\_space(input, crops, block\_size, name=None)**

타입 T의 4-D 텐서에 대한 BatchToSpace 함수입니다.

배치(batch)의 데이터를 공간적인 데이터의 블록으로 재배열하고 자릅니다. **SpaceToBatch** 함수의 역과정에 해당합니다. 더 구체적으로, 이 함수는 **input** 텐서의 **batch** 차원의 값들이 **height** 와 **width** 차원의 공간적인 블록으로 이동된 후, **height** 차원과 **width** 차원을 따라 잘린 텐서를 반환합니다.

인자:

- **input** : 4-D Tensor . [batch\*block\_size\*block\_size, height\_pad/block\_size,

`width_pad/block_size, depth]` 의 구조(shape)를 가집니다. 입력 텐서의 배치(batch) 크기는 `block_size * block_size` 의 배수여야 합니다.

- `crops` : `int32` 형 구조 `[2, 2]` 의 2-D 텐서. 음이 아닌 정수로 구성됩니다. 중간 결과에서 공간 차원을 따라 몇 개의 원소를 잘라낼 것인지를 다음과 같이 결정합니다.

```
crops = [[crop_top, crop_bottom], [crop_left, crop_right]]
```

- `block_size` : `int`.
- `name` : 오퍼레이션의 명칭. (선택사항)

#### 반환값:

`input` 과 같은 자료형의 구조 `[batch, height, width, depth]` 인 4-D `Tensor`.

```
height = height_pad - crop_top - crop_bottom
width = width_pad - crop_left - crop_right
```

을 만족합니다. `block_size` 는 1보다 커야 합니다.

## `tf.space_to_depth(input, block_size, name=None)`

타입 T의 4-D 텐서에 대한 SpaceToDepth 함수입니다.

공간적인 데이터의 블록을 `depth` 로 재배열합니다. 구체적으로, 입력 텐서의 `height` 와 `width` 차원의 데이터를 `depth` 차원으로 옮깁니다. `block_size` 는 입력 텐서의 블록 사이즈와 데이터가 어떻게 옮겨질지를 지정합니다.

- 크기 `block_size x block_size` 의 블록이 각 위치의 `depth` 로 재배열됩니다.
- 출력 텐서의 `depth` 차원의 크기는 `input_depth * block_size * block_size` 입니다.
- 입력 텐서의 `height` 와 `width` 는 `block_size` 의 배수여야 합니다.

즉, 만약 입력의 구조(shape)가 `[batch, height, width, depth]` 이면, 출력 텐서의 구조는 `[batch, height/block_size, width/block_size, depth*block_size*block_size]` 이 됩니다.

이 함수는 입력 텐서의 랭크(rank)가 4여야 하며, `block_size` 가 1 이상이어야 하고 `height` 와 `width` 의 약수여야 합니다.

이 함수는 합성곱 연산 사이의 활성화에서 데이터를 그대로 유지한 채로 크기를 변경시키는 때 유용합니다(예로, 풀링 대신 사용할 수 있습니다). 합성곱 연산만으로 이루어진 모델의 훈련에도 유용합니다.

예로, 구조 `[1, 2, 2, 1]` 의 입력과 `block_size = 2` 가 주어진 경우,

```
x = [[[[1], [2]],
 [[3], [4]]]]
```

이 함수는 다음과 같은 구조 `[1, 1, 1, 4]` 의 텐서를 반환합니다.

```
[[[[1, 2, 3, 4]]]]
```

여기서, 입력은 크기 1의 배치를 가지며, 각 배치는 구조 `[2, 2, 1]` 로 배열된 원소를 가집니다. 출력에서 각각은 `width` 와 `height` 가 모두 1이고, 4 채널의 `depth` 를 가지도록 배열되며, 따라서 각각은 구조 `[1, 1, 4]` 를 가지게 됩니다.

더 큰 `depth` 를 가지는 입력 텐서의 경우(여기서는 `[1, 2, 2, 3]`)를 봅시다.

```
x = [[[[1, 2, 3], [4, 5, 6]],
 [[7, 8, 9], [10, 11, 12]]]]
```

`block_size` 에 2를 넣어 이 함수를 적용할 경우, 구조 `[1, 1, 1, 12]` 인 텐서

```
[[[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]]]]
```

를 출력합니다. 비슷하게, `block_size` 가 2이고 입력의 구조가 `[1 4 4 1]` 인 경우,

```
x = [[[[1], [2], [5], [6]],
 [[3], [4], [7], [8]],
 [[9], [10], [13], [14]],
 [[11], [12], [15], [16]]]]
```

이 함수는 구조 `[1 2 2 4]` 인 다음 텐서를 반환합니다.

```
x = [[[[1, 2, 3, 4],
 [5, 6, 7, 8]],
 [[9, 10, 11, 12]],
 [[13, 14, 15, 16]]]]
```

인자:

- `input` : `Tensor`.
- `block_size` : `int`. 공간 블록의 크기를 지정합니다.
- `name` : 오퍼레이션의 명칭. (선택사항)

## 반환값:

`input` 과 같은 자료형의 `Tensor`.

## `tf.depth_to_space(input, block_size, name=None)`

타입 T의 4-D 텐서에 대한 DepthToSpace 함수입니다.

`depth` 의 데이터를 공간적인 데이터의 블록으로 재배열합니다. `spaceToDepth` 함수의 역과정과 같습니다. 구체적으로, 이 함수는 `depth` 차원의 값들을 `height` 와 `width` 차원들의 공간적인 블록으로 이동시킵니다. `block_size` 는 입력의 블록 크기와 데이터가 어떻게 이동될지를 지정합니다.

- `depth` 에서 `block_size * block_size` 크기의 데이터가 `block_size x block_size` 의 블록으로 재배열됩니다.
- 출력 텐서의 `width` 의 크기는 `input_depth * block_size` 이며, `height` 의 크기는 `input_height * block_size` 입니다.
- 입력 텐서의 `depth` 는 `block_size * block_size` 의 배수여야 합니다.

즉, 만약 입력의 구조(shape)가 `[batch, height, width, depth]` 라면, 출력 텐서의 구조는 `[batch, height*block_size, width*block_size, depth/(block_size*block_size)]` 와 같습니다.

이 함수는 입력 텐서의 랭크(rank)가 4여야 하며, `block_size` 는 1 이상이고 `block_size * block_size` 가 입력 텐서의 `depth` 의 약수여야 합니다.

이 함수는 합성곱 연산 사이의 활성화에서 데이터를 그대로 유지한 채로 크기를 변경시키는 때 유용합니다(예로, 풀링 대신 사용할 수 있습니다). 합성곱 연산만으로 이루어진 모델의 훈련에도 유용합니다.

예로, 구조 `[1, 1, 1, 4]` 의 입력과 `block_size = 2` 가 주어진 경우,

```
x = [[[1, 2, 3, 4]]]
```

이 함수는 다음과 같은 구조 `[1, 2, 2, 1]` 의 텐서를 반환합니다.

```
[[[1], [2]],
 [[3], [4]]]
```

여기서, 입력은 크기 1의 배치를 가지며, 각 배치는 구조 `[1, 1, 4]` 로 배열된 원소를 가집니다. 출력에서 각각은  $2 \times 2$  원소에 1 채널의 ( $1 = 4 / (\text{block\_size} * \text{block\_size})$ ) `depth` 를 가지도록 배열되며, 따라서 각각은 구조 `[2, 2, 1]` 을 가지게 됩니다.

더 큰 `depth` 를 가지는 입력 텐서의 경우(여기서는 `[1, 1, 1, 12]`)를 봅시다.

```
x = [[[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]]]]
```

`block_size` 에 2를 넣어 이 함수를 적용할 경우, 구조 `[1, 2, 2, 3]` 인 텐서

```
[[[[1, 2, 3], [4, 5, 6]],
 [[7, 8, 9], [10, 11, 12]]]]
```

를 출력합니다. 비슷하게, `block_size` 가 2이고 입력의 구조가 `[1 2 2 4]` 인 경우,

```
x = [[[[1, 2, 3, 4],
 [5, 6, 7, 8]],
 [[9, 10, 11, 12],
 [13, 14, 15, 16]]]]
```

이 함수는 구조 `[1 4 4 1]` 인 다음 텐서를 반환합니다.

```
x = [[[1], [2], [5], [6]],
 [[3], [4], [7], [8]],
 [[9], [10], [13], [14]],
 [[11], [12], [15], [16]]]
```

인자:

- `input` : `Tensor`.
- `block_size` : `int`. 공간 블록의 크기를 지정합니다.
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

`input` 과 같은 자료형의 `Tensor`.

## **`tf.gather(params, indices, validate_indices=None, name=None)`**

`indices` 에 따라 `params` 에서 슬라이스를 모읍니다.

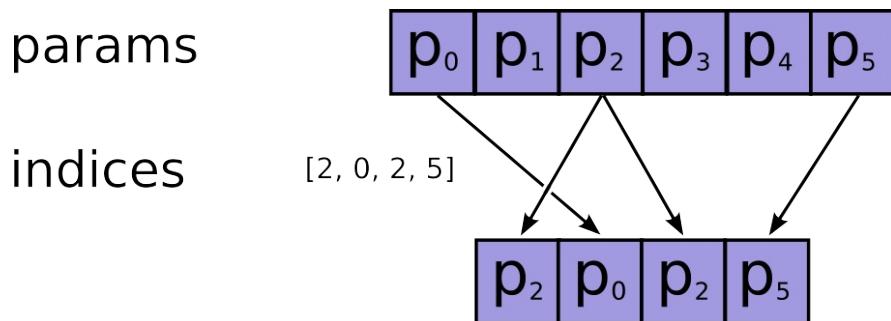
`indices` 는 임의 차원의 정수 텐서여야 합니다(주로 0-D 또는 1-D). 구조(shape)가 `indices.shape + params.shape[1:]` 인 출력 텐서를 생성합니다.

```
Scalar indices
output[:, ..., :] = params[indices, :, ... :]

Vector indices
output[i, :, ..., :] = params[indices[i], :, ... :]

Higher rank indices
output[i, ..., j, :, ... :] = params[indices[i, ..., j], :, ..., :]
```

만약 `indices` 이 순열이고 `len(indices) == params.shape[0]` 이라면, 이 함수는 그에 따라 `params` 를 재배열합니다.



인자:

- `params` : `Tensor` .
- `indices` : `int32` 형 또는 `int64` 형 `Tensor` .
- `validate_indices` : `bool` . 기본값은 `True` . (선택사항)
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

`params` 와 같은 자료형의 `Tensor` .

## `tf.gather_nd(params, indices, name=None)`

`indices` 에 따라 `params` 에서 값들을 모읍니다.

`indices` 는 `params` 의 인덱스들을 포함하는 정수형 텐서입니다. 구조(shape)는  $[d_0, \dots, d_N, R]$  이어야 하고, 여기서 `R` 은 `params` 의 랭크(rank)입니다. `indices` 의 가장 안쪽의 차원들 (길이 `R` 의)는 `params` 의 인덱스들에 해당합니다.

출력 텐서의 구조(shape)는  $[d_0, \dots, d_{n-1}]$  이고, 다음을 만족합니다.

```
output[i, j, k, ...] = params[indices[i, j, k, ...]]
```

예로, `indices` 에 대해서 다음과 같습니다.

```
output[i] = params[indices[i, :]]
```

인자:

- `params` :  $R$ -D Tensor. 값들을 모을 Tensor입니다.
- `indices` : `int32` 형 또는 `int64` 형  $(N+1)$ -D Tensor. 구조(shape)가  $[d_0, \dots, d_N, R]$  이어야 합니다.
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

`params` 와 같은 자료형의  $N$ -D Tensor. `indices`에 따라 주어진 인덱스들로부터 `params`의 값을 모은 텐서입니다.

## **`tf.dynamic_partition(data, partitions, num_partitions, name=None)`**

`partitions` 의 인덱스들을 이용해서 `data` 를 `num_partitions` 개의 텐서로 나눕니다.

각각의 크기 `partitions.ndim` 인 인덱스 튜플 `js`에 대해, 슬라이스 `data[js, ...]` 는 `outputs[partitions[js]]` 의 부분이 됩니다. `partitions[js] = i` 인 슬라이스는 `outputs[i]`에 `js`의 사전 순서대로 배열되고, `outputs[i]`의 첫 번째 차원은 `partitions`의 엔트리 중 `i` 와 같은 것의 수입니다. 구체적으로,

```
outputs[i].shape = [sum(partitions == i)] + data.shape[partitions.ndim:]
outputs[i] = pack([data[js, ...] for js if partitions[js] == i])
```

입니다. `data.shape` 는 `partitions.shape` 로 시작해야 합니다.

예시:

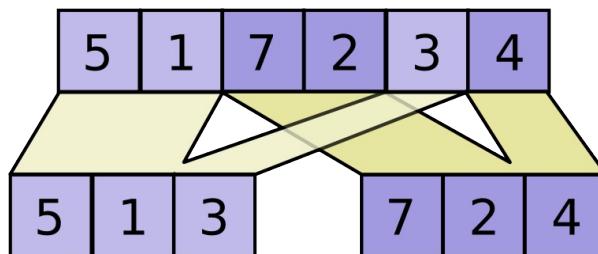
```
스칼라 분할
partitions = 1
num_partitions = 2
data = [10, 20]
outputs[0] = [] # Empty with shape [0, 2]
outputs[1] = [[10, 20]]

벡터 분할
partitions = [0, 0, 1, 1, 0]
num_partitions = 2
data = [10, 20, 30, 40, 50]
outputs[0] = [10, 20, 50]
outputs[1] = [30, 40]
```

partitions

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|

data



인자:

- `data` : `Tensor`.
- `partitions` : `int32` 형 `Tensor`. 임의의 구조(shape)가 가능합니다. 범위 `[0, num_partitions)` 내의 인덱스들을 포함합니다.
- `num_partitions` : `int ( >= 1 )`. 분할의 수.
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

`num_partitions` 개 `Tensor` 의 리스트.

### `tf.dynamic_stitch(indices, data, name=None)`

`data` 안의 텐서들 안의 값을 단일 텐서에 끼웁니다.

```
merged[indices[m][i, ..., j], ...] = data[m][i, ..., j, ...]
```

를 만족하는 `merged` 텐서를 구성합니다. 예로, 만약 각각의 `indices[m]` 이 스칼라이거나 벡터라면, 다음이 성립합니다.

```
Scalar indices
merged[indices[m], ...] = data[m][...]

Vector indices
merged[indices[m][i], ...] = data[m][i, ...]
```

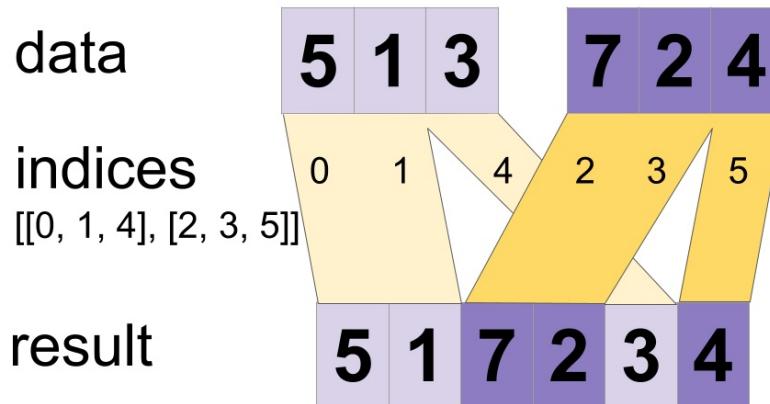
각각의 `data[i].shape` 는 해당하는 `indices[i].shape` 로 시작해야 하며, `data[i].shape` 의 나머지는 `i` 에 대해서 일정해야 합니다. 즉, `data[i].shape = indices[i].shape + constant` 이어야 합니다. 이 `constant`에 대해, 출력 텐서의 구조(shape)는

```
merged.shape = [max(indices)] + constant
```

입니다. 값들은 순서대로 합쳐집니다. 즉, 인덱스가 `indices[m][i]` 와 `indices[n][j]` 모두에서 나타나고  $(m, i) < (n, j)$  인 경우 슬라이스 `data[n][j]` 가 합쳐진 결과에 나타납니다.

예시:

```
indices[0] = 6
indices[1] = [4, 1]
indices[2] = [[5, 2], [0, 3]]
data[0] = [61, 62]
data[1] = [[41, 42], [11, 12]]
data[2] = [[[51, 52], [21, 22]], [[1, 2], [31, 32]]]
merged = [[1, 2], [11, 12], [21, 22], [31, 32], [41, 42],
 [51, 52], [61, 62]]
```



인자:

- `indices` : 2개 이상의 `int32` 형 `Tensor` 의 리스트.
- `data` : 같은 자료형의 `Tensor` 들의 리스트. `indices` 의 텐서 개수와 리스트의 `Tensor` 수가 같아야 합니다.
- `name` : 오퍼레이션의 명칭. (선택사항)

**반환값:**

`data` 와 같은 자료형의 `Tensor`.

```
tf.boolean_mask(tensor, mask,
name='boolean_mask')
```

텐서에 불리언 마스크(boolean mask)를 적용합니다. NumPy의 `tensor[mask]` 와 동일합니다.

```
1-D 예시
tensor = [0, 1, 2, 3]
mask = [True, False, True, False]
boolean_mask(tensor, mask) ==> [0, 2]
```

일반적으로 `0 < dim(mask) = K <= dim(tensor)` 이고, `mask` 의 구조(shape)는 `tensor` 구조의 첫 `K` 차원이 일치해야 합니다. 그렇게 되면 `boolean_mask(tensor, mask)[i, j1, ..., jd] = tensor[i1, ..., iK, j1, ..., jd]` 을 만족하며, `(i1, ..., iK)` 는 `i` 번째 `mask` 의 `i` 번째 `True` 인 원소입니다(행 우선 순서).

**인자:**

- `tensor` : N-D 텐서.
- `mask` : K-D 불리언 텐서,  $K \leq N$ 이고,  $K$ 는 정적으로(statically) 알려져 있어야 합니다.
- `name` : 오퍼레이션의 명칭. (선택사항)

**반환값:**

`mask` 에서 `True` 인 값들에 대한 `tensor` 의 원소들로 이루어진 텐서.

**예외:**

- `ValueError` : 모양이 일치하지 않는 경우.

**예시:**

```
2-D 예시
tensor = [[1, 2], [3, 4], [5, 6]]
mask = [True, False, True]
boolean_mask(tensor, mask) ==> [[1, 2], [5, 6]]
```

```
tf.one_hot(indices, depth, on_value=None,
off_value=None, axis=None, dtype=None,
name=None)
```

One-hot 텐서를 반환합니다.

`indices` 의 인덱스에 있는 위치는 `on_value`, 아닌 위치는 `off_value` 의 값을 가집니다.

`on_value` 와 `off_value` 는 같은 자료형을 가져야 합니다. 만약 `dtype` 이 주어진 경우, 그들은 `dtype` 에 의해 정해진 자료형이어야 합니다.

만약 `on_value` 가 주어지지 않으면, `dtype` 형의 `1` 이 기본값으로 정해집니다.

비슷하게 `off_value` 가 주어지지 않은 경우 `dtype` 형의 `0` 이 기본값입니다.

입력 `indices` 가 랭크 `N` 인 경우, 출력은 랭크 `N+1` 을 가집니다. 새로운 축이 차원 `axis` 에 추가됩니다(기본적으로 새 축은 끝에 추가됩니다).

만약 `indices` 가 스칼라라면 출력의 구조(shape)는 길이 `depth` 의 벡터가 됩니다.

만약 `indices` 가 길이 `features` 의 벡터라면, 출력의 구조는 다음과 같습니다.

```
features x depth if axis == -1
depth x features if axis == 0
```

만약 `indices` 가 구조 `[batch, features]` 의 행렬(또는 배치)이라면, 출력의 구조는 다음과 같습니다.

```
batch x features x depth if axis == -1
batch x depth x features if axis == 1
depth x batch x features if axis == 0
```

`dtype` 가 주어지지 않은 경우, `on_value` 또는 `off_value` 가 주어졌다면 그들로부터 자료형을 추측합니다. 만약 셋 모두 주어지지 않았다면 기본 자료형은 `tf.float32` 형입니다.

참고: 수가 아닌 출력 자료형이 요구되는 경우(`tf.string`, `tf.bool` 등), `on_value` 와 `off_value` 모두 주어져야 합니다.

예시:

다음이 주어진 경우

```
indices = [0, 2, -1, 1]
depth = 3
on_value = 5.0
off_value = 0.0
axis = -1
```

출력은 다음의  $[4 \times 3]$  입니다.

```
output =
[5.0 0.0 0.0] // one_hot(0)
[0.0 0.0 5.0] // one_hot(2)
[0.0 0.0 0.0] // one_hot(-1)
[0.0 5.0 0.0] // one_hot(1)
```

다음이 주어진 경우

```
indices = [[0, 2], [1, -1]]
depth = 3
on_value = 1.0
off_value = 0.0
axis = -1
```

출력은 다음의  $[2 \times 2 \times 3]$  입니다.

```
output =
[
 [
 [1.0, 0.0, 0.0] // one_hot(0)
 [0.0, 0.0, 1.0] // one_hot(2)
]
 [
 [0.0, 1.0, 0.0] // one_hot(1)
 [0.0, 0.0, 0.0] // one_hot(-1)
]
]
```

다음과 같이 `on_value` 와 `off_value` 의 기본값을 이용하는 경우,

```
indices = [0, 1, 2]
depth = 3
```

출력은 다음과 같습니다.

```
output =
[[1., 0., 0.],
 [0., 1., 0.],
 [0., 0., 1.]]
```

**인자:**

- **indices** : 인덱스들의 Tensor .
- **depth** : One-hot 차원의 깊이(depth)를 결정하는 스칼라 값.
- **on\_value** : `indices[j] = i` 인 경우 채울 스칼라 값. (기본값: 1, 선택사항)
- **off\_value** : `indices[j] != i` 인 경우 채울 스칼라 값. (기본값: 0, 선택사항)
- **axis** : 채워질 축 (기본값: -1, 선택사항).
- **dtype** : 출력 텐서의 자료형.

**반환값:**

- **output** : One-hot 텐서.

**예외:**

- **TypeError** : `on_value` 또는 `off_value` 의 자료형이 `dtype` 과 다른 경우
- **TypeError** : `on_value` 와 `off_value` 의 자료형이 서로 다른 경우

## 기타 함수 및 클래스

### **tf.bitcast(input, type, name=None)**

텐서를 다른 자료형으로 데이터 복사 없이 비트캐스트(bitcast)합니다.

`input` 텐서가 주어질 때, 이 함수는 `input` 과 같은 버퍼 데이터를 가진 자료형 `type` 의 텐서를 반환합니다.

만약 입력의 자료형 `T` 가 출력의 자료형 `type` 에 비해 더 큰 경우, 구조(shape)가 [...]에서 [..., `sizeof(T)/sizeof(type)`]으로 변형됩니다.

만약 `T` 가 `type` 에 비해 더 작은 경우, 가장 오른쪽의 차원이 `sizeof(type)/sizeof(T)`와 같아야 합니다. 구조는 [..., `sizeof(type)/sizeof(T)`] to [...]으로 변형됩니다.

**인자:**

- **input** : Tensor . 다음의 자료형이 가능합니다: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`, `int16`, `int8`, `complex64`, `complex128`, `qint8`, `quint8`, `qint32`, `half` .
- **type** : `tf.DType` . 다음 중 하나가 가능합니다: `tf.float32`, `tf.float64`, `tf.int64`, `tf.int32`, `tf.uint8`, `tf.uint16`, `tf.int16`, `tf.int8`, `tf.complex64`, `tf.complex128`, `tf.qint8`, `tf.quint8`, `tf.qint32`, `tf.half` .
- **name** : 오퍼레이션의 명칭. (선택사항)

**반환값:**

```
type 자료형의 Tensor .
```

---

**tf.shape\_n(input, name=None)**

텐서의 구조(shape)를 반환합니다.

이 함수는 `input[i]` 들의 구조를 나타내는 `n` 개의 1-D 정수 텐서를 반환합니다.

**인자:**

- `input` : 같은 자료형의 1개 이상의 `Tensor` 의 리스트.
- `name` : 오퍼레이션의 명칭. (선택사항)

**반환값:**

```
input 의 텐서와 같은 개수의 int32 형 Tensor 의 리스트.
```

---

**tf.unique\_with\_counts(x, name=None)**

1-D 텐서에서 서로 다른 원소를 찾습니다.

이 함수는 텐서 `x` 의 모든 서로 다른 원소를 `x`에서 나타나는 순서대로 나열한 텐서 `y`를 반환합니다. 이 함수는 크기가 `x`와 같고, `x`의 각 원소에 대해 `y`에서의 인덱스를 원소를 가지는 텐서 `idx`도 반환합니다. `y`의 각 원소가 `x`에서 몇 번 나타나는지에 대한 텐서 `count`도 반환합니다. 즉,

```
y[idx[i]] = x[i] for i in [0, 1, ..., rank(x) - 1]
```

입니다.

**예시:**

```
tensor 'x'는 [1, 1, 2, 4, 4, 4, 7, 8, 8]
y, idx, count = unique_with_counts(x)
y ==> [1, 2, 4, 7, 8]
idx ==> [0, 0, 1, 2, 2, 2, 3, 4, 4]
count ==> [2, 1, 3, 1, 2]
```

**인자:**

- `x` : 1-D `Tensor` .
- `name` : 오퍼레이션의 명칭. (선택사항)

반환값:

Tensor 의 튜플 (y, idx, count).

- `y` : `x` 와 자료형이 같은 1-D Tensor .
- `idx` : int32 형 1-D Tensor .
- `count` : int32 형 1-D Tensor .

# Math

Note: Functions taking `Tensor` arguments can also take anything accepted by `tf.convert_to_tensor`.

[TOC]

## Arithmetic Operators

TensorFlow provides several operations that you can use to add basic arithmetic operators to your graph.

### `tf.add(x, y, name=None)`

Returns  $x + y$  element-wise.

*NOTE:* Add supports broadcasting. AddN does not.

#### Args:

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `uint8`, `int8`, `int16`, `int32`, `int64`, `complex64`, `complex128`, `string`.
- `y` : A `Tensor`. Must have the same type as `x`.
- `name` : A name for the operation (optional).

#### Returns:

A `Tensor`. Has the same type as `x`.

### `tf.sub(x, y, name=None)`

Returns  $x - y$  element-wise.

#### Args:

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `int32`, `int64`, `complex64`, `complex128`.
- `y` : A `Tensor`. Must have the same type as `x`.

- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

---

**`tf.mul(x, y, name=None)`**

Returns  $x * y$  element-wise.

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `uint8`, `int8`, `int16`, `int32`, `int64`, `complex64`, `complex128`.
- `y` : A `Tensor`. Must have the same type as `x`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

---

**`tf.div(x, y, name=None)`**

Returns  $x / y$  element-wise.

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `uint8`, `int8`, `int16`, `int32`, `int64`, `complex64`, `complex128`.
- `y` : A `Tensor`. Must have the same type as `x`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

---

**`tf.truediv(x, y, name=None)`**

Divides  $x / y$  elementwise, always producing floating point results.

The same as `tf.div` for floating point arguments, but casts integer arguments to floating point before dividing so that the result is always floating point. This op is generated by normal `x / y` division in Python 3 and in Python 2.7 with `from __future__ import division`. If you want integer division that rounds down, use `x // y` or `tf.floordiv`.

`x` and `y` must have the same numeric type. If the inputs are floating point, the output will have the same type. If the inputs are integral, the inputs are cast to `float32` for `int8` and `int16` and `float64` for `int32` and `int64` (matching the behavior of Numpy).

#### Args:

- `x` : `Tensor` numerator of numeric type.
- `y` : `Tensor` denominator of numeric type.
- `name` : A name for the operation (optional).

#### Returns:

`x / y` evaluated in floating point.

#### Raises:

- `TypeError` : If `x` and `y` have different dtypes.
- 

## `tf.floordiv(x, y, name=None)`

Divides `x / y` elementwise, rounding down for floating point.

The same as `tf.div(x,y)` for integers, but uses `tf.floor(tf.div(x,y))` for floating point arguments so that the result is always an integer (though possibly an integer represented as floating point). This op is generated by `x // y` floor division in Python 3 and in Python 2.7 with `from __future__ import division`.

Note that for efficiency, `floordiv` uses C semantics for negative numbers (unlike Python and Numpy).

`x` and `y` must have the same type, and the result will have the same type as well.

#### Args:

- `x` : `Tensor` numerator of real numeric type.
- `y` : `Tensor` denominator of real numeric type.
- `name` : A name for the operation (optional).

#### Returns:

`x / y` rounded down (except possibly towards zero for negative integers).

**Raises:**

- `TypeError` : If the inputs are complex.

**`tf.mod(x, y, name=None)`**

Returns element-wise remainder of division.

**Args:**

- `x` : A `Tensor` . Must be one of the following types: `int32` , `int64` , `float32` , `float64` .
- `y` : A `Tensor` . Must have the same type as `x` .
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` . Has the same type as `x` .

**`tf.cross(a, b, name=None)`**

Compute the pairwise cross product.

`a` and `b` must be the same shape; they can either be simple 3-element vectors, or any shape where the innermost dimension is 3. In the latter case, each pair of corresponding 3-element vectors is cross-multiplied independently.

**Args:**

- `a` : A `Tensor` . Must be one of the following types: `float32` , `float64` , `int32` , `int64` , `uint8` , `int16` , `int8` , `uint16` , `half` . A tensor containing 3-element vectors.
- `b` : A `Tensor` . Must have the same type as `a` . Another tensor, of same type and shape as `a` .
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` . Has the same type as `a` . Pairwise cross product of the vectors in `a` and `b` .

## Basic Math Functions

TensorFlow provides several operations that you can use to add basic mathematical functions to your graph.

## tf.add\_n(inputs, name=None)

Add all input tensors element wise.

### Args:

- `inputs` : A list of at least 1 `Tensor` objects of the same type in: `float32` , `float64` , `int64` , `int32` , `uint8` , `uint16` , `int16` , `int8` , `complex64` , `complex128` , `qint8` , `quint8` , `qint32` , `half` . Must all be the same size and shape.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor` . Has the same type as `inputs` .

## tf.abs(x, name=None)

Computes the absolute value of a tensor.

Given a tensor of real numbers `x` , this operation returns a tensor containing the absolute value of each element in `x` . For example, if `x` is an input element and `y` is an output element, this operation computes  $(y = |x|)$  .

See [tf.complex\\_abs\(\)](#) to compute the absolute value of a complex number.

### Args:

- `x` : A `Tensor` of type `float` , `double` , `int32` , or `int64` .
- `name` : A name for the operation (optional).

### Returns:

A `Tensor` the same size and type as `x` with absolute values.

## tf.neg(x, name=None)

Computes numerical negative value element-wise.

I.e.,  $\{y = -x\}$ .

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `int32`, `int64`, `complex64`, `complex128`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

---

**`tf.sign(x, name=None)`**

Returns an element-wise indication of the sign of a number.

$y = \text{sign}(x) = -1 \text{ if } x < 0 ; 0 \text{ if } x == 0 ; 1 \text{ if } x > 0 .$

For complex numbers,  $y = \text{sign}(x) = x / |x| \text{ if } x != 0 , \text{ otherwise } y = 0 .$

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `int32`, `int64`, `complex64`, `complex128`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

---

**`tf.inv(x, name=None)`**

Computes the reciprocal of `x` element-wise.

I.e.,  $\{y = 1 / x\}$ .

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `int32`, `int64`, `complex64`, `complex128`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

---

## `tf.square(x, name=None)`

Computes square of `x` element-wise.

I.e.,  $(y = x * x = x^2)$ .

### Args:

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `int32`, `int64`, `complex64`, `complex128`.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor`. Has the same type as `x`.

---

## `tf.round(x, name=None)`

Rounds the values of a tensor to the nearest integer, element-wise.

For example:

```
'a' is [0.9, 2.5, 2.3, -4.4]
tf.round(a) ==> [1.0, 3.0, 2.0, -4.0]
```

### Args:

- `x` : A `Tensor` of type `float` or `double`.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor` of same shape and type as `x`.

---

## `tf.sqrt(x, name=None)`

Computes square root of `x` element-wise.

I.e.,  $(y = \sqrt{x} = x^{1/2})$ .

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `complex64`, `complex128`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

---

**`tf.rsqrt(x, name=None)`**

Computes reciprocal of square root of `x` element-wise.

I.e.,  $y = 1 / \sqrt{x}$ .

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `complex64`, `complex128`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

---

**`tf.pow(x, y, name=None)`**

Computes the power of one value to another.

Given a tensor `x` and a tensor `y`, this operation computes  $(x^y)$  for corresponding elements in `x` and `y`. For example:

```
tensor 'x' is [[2, 2], [3, 3]]
tensor 'y' is [[8, 16], [2, 3]]
tf.pow(x, y) ==> [[256, 65536], [9, 27]]
```

**Args:**

- `x` : A `Tensor` of type `float`, `double`, `int32`, `int64`, `complex64`, or `complex128`.
- `y` : A `Tensor` of type `float`, `double`, `int32`, `int64`, `complex64`, or `complex128`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`.

---

**`tf.exp(x, name=None)`**

Computes exponential of  $x$  element-wise. ( $y = e^x$ ).

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `complex64`, `complex128`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

---

**`tf.log(x, name=None)`**

Computes natural logarithm of  $x$  element-wise.

i.e., ( $y = \log_e x$ ).

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `complex64`, `complex128`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

---

**`tf.ceil(x, name=None)`**

Returns element-wise smallest integer in not less than  $x$ .

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`.

- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

---

**`tf.floor(x, name=None)`**

Returns element-wise largest integer not greater than `x`.

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

---

**`tf.maximum(x, y, name=None)`**

Returns the max of `x` and `y` (i.e.  $x > y ? x : y$ ) element-wise, broadcasts.

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `int32`, `int64`.
- `y` : A `Tensor`. Must have the same type as `x`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

---

**`tf.minimum(x, y, name=None)`**

Returns the min of `x` and `y` (i.e.  $x < y ? x : y$ ) element-wise, broadcasts.

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `int32`,

`int64 .`

- `y` : A `Tensor`. Must have the same type as `x`.
- `name` : A name for the operation (optional).

**Returns:**A `Tensor`. Has the same type as `x`.**`tf.cos(x, name=None)`**Computes cos of `x` element-wise.**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `complex64`, `complex128`.
- `name` : A name for the operation (optional).

**Returns:**A `Tensor`. Has the same type as `x`.**`tf.sin(x, name=None)`**Computes sin of `x` element-wise.**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `complex64`, `complex128`.
- `name` : A name for the operation (optional).

**Returns:**A `Tensor`. Has the same type as `x`.**`tf.lbeta(x, name='lbeta')`**Computes `ln(|Beta(x)|)`, reducing along the last dimension.Given one-dimensional `z = [z_0, ..., z_{K-1}]`, we define

```
Beta(z) = \prod_j Gamma(z_j) / Gamma(\sum_j z_j)
```

And for `n + 1` dimensional `x` with shape `[N1, ..., Nn, K]`, we define `lbeta(x)[i1, ..., in] = Log(|Beta(x[i1, ..., in, :])|)`. In other words, the last dimension is treated as the `z` vector.

Note that if `z = [u, v]`, then `Beta(z) = int_0^1 t^{u-1} (1 - t)^{v-1} dt`, which defines the traditional bivariate beta function.

#### Args:

- `x` : A rank `n + 1` `Tensor` with type `float`, or `double`.
- `name` : A name for the operation (optional).

#### Returns:

The logarithm of `|Beta(x)|` reducing along the last dimension.

#### Raises:

- `ValueError` : If `x` is empty with rank one or less.
- 

## tf.tan(x, name=None)

Computes tan of `x` element-wise.

#### Args:

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `int32`, `int64`, `complex64`, `complex128`.
- `name` : A name for the operation (optional).

#### Returns:

A `Tensor`. Has the same type as `x`.

---

## tf.acos(x, name=None)

Computes acos of `x` element-wise.

#### Args:

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `int32`,

```
int64 , complex64 , complex128 .
```

- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` . Has the same type as `x` .

---

**tf.asin(x, name=None)**

Computes asin of `x` element-wise.

**Args:**

- `x` : A `Tensor` . Must be one of the following types: `half` , `float32` , `float64` , `int32` , `int64` , `complex64` , `complex128` .
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` . Has the same type as `x` .

---

**tf.atan(x, name=None)**

Computes atan of `x` element-wise.

**Args:**

- `x` : A `Tensor` . Must be one of the following types: `half` , `float32` , `float64` , `int32` , `int64` , `complex64` , `complex128` .
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` . Has the same type as `x` .

---

**tf.lgamma(x, name=None)**

Computes the log of the absolute value of `Gamma(x)` element-wise.

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

---

**`tf.digamma(x, name=None)`**

Computes Psi, the derivative of Lgamma (the log of the absolute value of `Gamma(x)`), element-wise.

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

---

**`tf.erf(x, name=None)`**

Computes the Gauss error function of `x` element-wise.

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

---

**`tf.erfc(x, name=None)`**

Computes the complementary error function of `x` element-wise.

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`.

- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

**`tf.squared_difference(x, y, name=None)`**

Returns  $(x - y)(x - y)$  element-wise.

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `int32`, `int64`, `complex64`, `complex128`.
- `y` : A `Tensor`. Must have the same type as `x`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

**`tf.igamma(a, x, name=None)`**

Compute the lower regularized incomplete Gamma function `Q(a, x)`.

The lower regularized incomplete Gamma function is defined as:

$$P(a, x) = \text{gamma}(a, x) / \text{Gamma}(x) = 1 - Q(a, x)$$

where

$$\text{gamma}(a, x) = \int_{0}^x t^{a-1} e^{-t} dt$$

is the lower incomplete Gamma function.

Note, above `Q(a, x)` (`Igammac`) is the upper regularized complete Gamma function.

**Args:**

- `a` : A `Tensor`. Must be one of the following types: `float32`, `float64`.
- `x` : A `Tensor`. Must have the same type as `a`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `a`.

**`tf.igammac(a, x, name=None)`**

Compute the upper regularized incomplete Gamma function  $Q(a, x)$ .

The upper regularized incomplete Gamma function is defined as:

$$Q(a, x) = \text{Gamma}(a, x) / \text{Gamma}(x) = 1 - P(a, x)$$

where

$$\text{Gamma}(a, x) = \int_{x}^{\infty} t^{a-1} \exp(-t) dt$$

is the upper incomplete Gamma function.

Note, above `P(a, x)` (`Igamma`) is the lower regularized complete Gamma function.

**Args:**

- `a` : A `Tensor`. Must be one of the following types: `float32`, `float64`.
- `x` : A `Tensor`. Must have the same type as `a`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `a`.

**`tf.zeta(x, q, name=None)`**

Compute the Hurwitz zeta function  $\zeta(x, q)$ .

The Hurwitz zeta function is defined as:

$$\zeta(x, q) = \sum_{n=0}^{\infty} (q + n)^{-x}$$

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `float32`, `float64`.

- `q` : A `Tensor`. Must have the same type as `x`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `x`.

---

**`tf.polygamma(a, x, name=None)`**

Compute the polygamma function  $\psi^{(n)}(x)$ .

The polygamma function is defined as:

$$\psi^{(n)}(x) = \frac{d^n}{dx^n} \psi(x)$$

where  $\psi(x)$  is the digamma function.

**Args:**

- `a` : A `Tensor`. Must be one of the following types: `float32`, `float64`.
- `x` : A `Tensor`. Must have the same type as `a`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `a`.

## Matrix Math Functions

TensorFlow provides several operations that you can use to add linear algebra functions on matrices to your graph.

---

**`tf.batch_matrix_diag(diagonal, name=None)`**

Returns a batched diagonal tensor with a given batched diagonal values.

Given a `diagonal`, this operation returns a tensor with the `diagonal` and everything else padded with zeros. The diagonal is computed as follows:

Assume `diagonal` has `k` dimensions `[I, J, K, ..., N]`, then the output is a tensor of rank `k+1` with dimensions `[I, J, K, ..., N, N]` where:

```
output[i, j, k, ..., m, n] = 1{m=n} * diagonal[i, j, k, ..., n] .
```

For example:

```
'diagonal' is [[1, 2, 3, 4], [5, 6, 7, 8]]
and diagonal.shape = (2, 4)

tf.batch_matrix_diag(diagonal) ==> [[[1, 0, 0, 0]
[0, 2, 0, 0]
[0, 0, 3, 0]
[0, 0, 0, 4]],
[[5, 0, 0, 0]
[0, 6, 0, 0]
[0, 0, 7, 0]
[0, 0, 0, 8]]]

which has shape (2, 4, 4)
```

### Args:

- `diagonal` : A `Tensor`. Rank `k`, where `k >= 1`.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor`. Has the same type as `diagonal`. Rank `k+1`, with `output.shape = diagonal.shape + [diagonal.shape[-1]]`.

## **tf.batch\_matrix\_diag\_part(input, name=None)**

Returns the batched diagonal part of a batched tensor.

This operation returns a tensor with the `diagonal` part of the batched `input`. The `diagonal` part is computed as follows:

Assume `input` has `k` dimensions `[I, J, K, ..., N, N]`, then the output is a tensor of rank `k - 1` with dimensions `[I, J, K, ..., N]` where:

```
diagonal[i, j, k, ..., n] = input[i, j, k, ..., n, n] .
```

The input must be at least a matrix.

For example:

```
'input' is [[[1, 0, 0, 0]
 [0, 2, 0, 0]
 [0, 0, 3, 0]
 [0, 0, 0, 4]],
 [[5, 0, 0, 0]
 [0, 6, 0, 0]
 [0, 0, 7, 0]
 [0, 0, 0, 8]]]

and input.shape = (2, 4, 4)

tf.batch_matrix_diag_part(input) ==> [[1, 2, 3, 4], [5, 6, 7, 8]]
which has shape (2, 4)
```

**Args:**

- `input` : A `Tensor`. Rank `k` tensor where `k >= 2` and the last two dimensions are equal.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `input`. The extracted diagonal(s) having shape `diagonal.shape = input.shape[:-1]`.

## **`tf.batch_matrix_band_part(input, num_lower, num_upper, name=None)`**

Copy a tensor setting everything outside a central band in each innermost matrix to zero.

The `band` part is computed as follows: Assume `input` has `k` dimensions `[I, J, K, ..., M, N]`, then the output is a tensor with the same shape where

$$\text{band}[i, j, k, \dots, m, n] = \text{in\_band}(m, n) * \text{input}[i, j, k, \dots, m, n].$$

The indicator function '`in_band(m, n)`' is one if `(num_lower < 0 || (m-n) <= num_lower) && (num_upper < 0 || (n-m) <= num_upper)`', and zero otherwise.

For example:

```
if 'input' is [[0, 1, 2, 3]
 [-1, 0, 1, 2]
 [-2, -1, 0, 1]
 [-3, -2, -1, 0]],

tf.batch_matrix_band_part(input, 1, -1) ==> [[0, 1, 2, 3]
 [-1, 0, 1, 2]
 [0, -1, 0, 1]
 [0, 0, -1, 0]],

tf.batch_matrix_band_part(input, 2, 1) ==> [[0, 1, 0, 0]
 [-1, 0, 1, 0]
 [-2, -1, 0, 1]
 [0, -2, -1, 0]]
```

Useful special cases:

```
tf.batch_matrix_band_part(input, 0, -1) ==> Upper triangular part.
tf.batch_matrix_band_part(input, -1, 0) ==> Lower triangular part.
tf.batch_matrix_band_part(input, 0, 0) ==> Diagonal.
```

### Args:

- `input` : A `Tensor`. Rank `k` tensor.
- `num_lower` : A `Tensor` of type `int64`. 0-D tensor. Number of subdiagonals to keep. If negative, keep entire lower triangle.
- `num_upper` : A `Tensor` of type `int64`. 0-D tensor. Number of superdiagonals to keep. If negative, keep entire upper triangle.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor`. Has the same type as `input`. Rank `k` tensor of the same shape as `input`. The extracted banded tensor.

## **tf.diag(diagonal, name=None)**

Returns a diagonal tensor with a given diagonal values.

Given a `diagonal`, this operation returns a tensor with the `diagonal` and everything else padded with zeros. The diagonal is computed as follows:

Assume `diagonal` has dimensions  $[D_1, \dots, D_k]$ , then the output is a tensor of rank  $2k$  with dimensions  $[D_1, \dots, D_k, D_1, \dots, D_k]$  where:

---

`output[i1, ..., ik, i1, ..., ik] = diagonal[i1, ..., ik]` and 0 everywhere else.

For example:

```
'diagonal' is [1, 2, 3, 4]
tf.diag(diagonal) ==> [[1, 0, 0, 0]
 [0, 2, 0, 0]
 [0, 0, 3, 0]
 [0, 0, 0, 4]]
```

### Args:

- `diagonal` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `complex64`. Rank k tensor where k is at most 3.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor`. Has the same type as `diagonal`.

---

## **tf.diag\_part(input, name=None)**

Returns the diagonal part of the tensor.

This operation returns a tensor with the `diagonal` part of the `input`. The `diagonal` part is computed as follows:

Assume `input` has dimensions `[D1, ..., Dk, D1, ..., Dk]`, then the output is a tensor of rank `k` with dimensions `[D1, ..., Dk]` where:

`diagonal[i1, ..., ik] = input[i1, ..., ik, i1, ..., ik]`.

For example:

```
'input' is [[1, 0, 0, 0]
 [0, 2, 0, 0]
 [0, 0, 3, 0]
 [0, 0, 0, 4]]

tf.diag_part(input) ==> [1, 2, 3, 4]
```

### Args:

- `input` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `complex64`. Rank k tensor where k is 2, 4, or 6.

- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `input`. The extracted diagonal.

**`tf.trace(x, name=None)`**

Compute the trace of a tensor `x`.

`trace(x)` returns the sum of along the diagonal.

For example:

```
'x' is [[1, 1],
[1, 1]]
tf.trace(x) ==> 2

'x' is [[1,2,3],
[4,5,6],
[7,8,9]]
tf.trace(x) ==> 15
```

**Args:**

- `x` : 2-D tensor.
- `name` : A name for the operation (optional).

**Returns:**

The trace of input tensor.

**`tf.transpose(a, perm=None, name='transpose')`**

Transpose `a`. Permutes the dimensions according to `perm`.

The returned tensor's dimension `i` will correspond to the input dimension `perm[i]`. If `perm` is not given, it is set to `(n-1...0)`, where `n` is the rank of the input tensor. Hence by default, this operation performs a regular matrix transpose on 2-D input Tensors.

For example:

```

'x' is [[1 2 3]
[4 5 6]]
tf.transpose(x) ==> [[1 4]
 [2 5]
 [3 6]]

Equivalently
tf.transpose(x, perm=[1, 0]) ==> [[1 4]
 [2 5]
 [3 6]]

'perm' is more useful for n-dimensional tensors, for n > 2
'x' is [[[1 2 3]
[4 5 6]]
[[7 8 9]
[10 11 12]]]
Take the transpose of the matrices in dimension-0
tf.transpose(x, perm=[0, 2, 1]) ==> [[[1 4]
 [2 5]
 [3 6]

 [[7 10]
 [8 11]
 [9 12]]]]

```

**Args:**

- `a` : A `Tensor`.
- `perm` : A permutation of the dimensions of `a`.
- `name` : A name for the operation (optional).

**Returns:**

A transposed `Tensor`.

```
tf.matmul(a, b, transpose_a=False,
 transpose_b=False, a_is_sparse=False,
 b_is_sparse=False, name=None)
```

Multiplies matrix `a` by matrix `b`, producing `a * b`.

The inputs must be two-dimensional matrices, with matching inner dimensions, possibly after transposition.

Both matrices must be of the same type. The supported types are: `float`, `double`, `int32`, `complex64`.

Either matrix can be transposed on the fly by setting the corresponding flag to `True`. This is `False` by default.

If one or both of the matrices contain a lot of zeros, a more efficient multiplication algorithm can be used by setting the corresponding `a_is_sparse` or `b_is_sparse` flag to `True`. These are `False` by default.

For example:

```
2-D tensor `a`
a = tf.constant([1, 2, 3, 4, 5, 6], shape=[2, 3]) => [[1. 2. 3.]
 [4. 5. 6.]]
2-D tensor `b`
b = tf.constant([7, 8, 9, 10, 11, 12], shape=[3, 2]) => [[7. 8.]
 [9. 10.]
 [11. 12.]]
c = tf.matmul(a, b) => [[58 64]
 [139 154]]
```

### Args:

- `a` : `Tensor` of type `float`, `double`, `int32` or `complex64`.
- `b` : `Tensor` with same type as `a`.
- `transpose_a` : If `True`, `a` is transposed before multiplication.
- `transpose_b` : If `True`, `b` is transposed before multiplication.
- `a_is_sparse` : If `True`, `a` is treated as a sparse matrix.
- `b_is_sparse` : If `True`, `b` is treated as a sparse matrix.
- `name` : Name for the operation (optional).

### Returns:

A `Tensor` of the same type as `a`.

## `tf.batch_matmul(x, y, adj_x=None, adj_y=None, name=None)`

Multiplies slices of two tensors in batches.

Multiplies all slices of `Tensor` `x` and `y` (each slice can be viewed as an element of a batch), and arranges the individual results in a single output tensor of the same batch size. Each of the individual slices can optionally be adjointed (to adjoint a matrix means to transpose and conjugate it) before multiplication by setting the `adj_x` or `adj_y` flag to `True`, which are by default `False`.

The input tensors `x` and `y` are 3-D or higher with shape `[..., r_x, c_x]` and `[..., r_y, c_y]`.

The output tensor is 3-D or higher with shape `[..., r_o, c_o]`, where:

```
r_o = c_x if adj_x else r_x
c_o = r_y if adj_y else c_y
```

It is computed as:

```
output[..., :, :] = matrix(x[..., :, :]) * matrix(y[..., :, :])
```

### Args:

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `int32`, `complex64`, `complex128`. 3-D or higher with shape `[..., r_x, c_x]`.
- `y` : A `Tensor`. Must have the same type as `x`. 3-D or higher with shape `[..., r_y, c_y]`.
- `adj_x` : An optional `bool`. Defaults to `False`. If `True`, adjoint the slices of `x`. Defaults to `False`.
- `adj_y` : An optional `bool`. Defaults to `False`. If `True`, adjoint the slices of `y`. Defaults to `False`.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor`. Has the same type as `x`. 3-D or higher with shape `[..., r_o, c_o]`

## **tf.matrix\_determinant(input, name=None)**

Calculates the determinant of a square matrix.

### Args:

- `input` : A `Tensor`. Must be one of the following types: `float32`, `float64`. A tensor of shape `[M, M]`.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor`. Has the same type as `input`. A scalar, equal to the determinant of the input.

**tf.batch\_matrix\_determinant(input, name=None)**

Calculates the determinants for a batch of square matrices.

The input is a tensor of shape `[..., M, M]` whose inner-most 2 dimensions form square matrices. The output is a 1-D tensor containing the determinants for all input submatrices `[..., :, :]`.

**Args:**

- `input` : A `Tensor`. Must be one of the following types: `float32`, `float64`. Shape is `[..., M, M]`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `input`. Shape is `[...]`.

**tf.matrix\_inverse(input, adjoint=None, name=None)**

Calculates the inverse of a square invertible matrix or its adjoint (conjugate transpose).

The op uses LU decomposition with partial pivoting to compute the inverse.

If the matrix is not invertible there is no guarantee what the op does. It may detect the condition and raise an exception or it may simply return a garbage result.

**Args:**

- `input` : A `Tensor`. Must be one of the following types: `float64`, `float32`. Shape is `[M, M]`.
- `adjoint` : An optional `bool`. Defaults to `False`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `input`. Shape is `[M, M]`. If `adjoint` is `False` then `output` contains the matrix inverse of `input`. If `adjoint` is `True` then `output` contains the matrix inverse of the adjoint of `input`.

**tf.batch\_matrix\_inverse(input, adjoint=None, name=None)**

Calculates the inverse of square invertible matrices or their adjoints (conjugate transposes).

The input is a tensor of shape `[..., M, M]` whose inner-most 2 dimensions form square matrices. The output is a tensor of the same shape as the input containing the inverse for all input submatrices `[..., :, :]`.

The op uses LU decomposition with partial pivoting to compute the inverses.

If a matrix is not invertible there is no guarantee what the op does. It may detect the condition and raise an exception or it may simply return a garbage result.

**Args:**

- `input` : A `Tensor`. Must be one of the following types: `float64`, `float32`. Shape is `[..., M, M]`.
- `adjoint` : An optional `bool`. Defaults to `False`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `input`. Shape is `[..., M, M]`.

**tf.cholesky(input, name=None)**

Calculates the Cholesky decomposition of a square matrix.

The input has to be symmetric and positive definite. Only the lower-triangular part of the input will be used for this operation. The upper-triangular part will not be read.

The result is the lower-triangular matrix of the Cholesky decomposition of the input, `L`, so that `input = L L^*`.

**Args:**

- `input` : A `Tensor`. Must be one of the following types: `float64`, `float32`. Shape is `[M, M]`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `input`. Shape is `[M, M]`.

## `tf.batch_cholesky(input, name=None)`

Calculates the Cholesky decomposition of a batch of square matrices.

The input is a tensor of shape `[..., M, M]` whose inner-most 2 dimensions form square matrices, with the same constraints as the single matrix Cholesky decomposition above. The output is a tensor of the same shape as the input containing the Cholesky decompositions for all input submatrices `[..., :, :]`.

### Args:

- `input` : A `Tensor`. Must be one of the following types: `float64`, `float32`. Shape is `[..., M, M]`.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor`. Has the same type as `input`. Shape is `[..., M, M]`.

## `tf.cholesky_solve(chol, rhs, name=None)`

Solve linear equations `A x = RHS`, given Cholesky factorization of `A`.

```
Solve one system of linear equations (K = 1).
A = [[3, 1], [1, 3]]
RHS = [[2], [22]] # shape 2 x 1
chol = tf.cholesky(A)
X = tf.cholesky_solve(chol, RHS)
tf.matmul(A, X) ~ RHS
X[:, 0] # Solution to the linear system A x = RHS[:, 0]

Solve five systems of linear equations (K = 5).
A = [[3, 1], [1, 3]]
RHS = [[1, 2, 3, 4, 5], [11, 22, 33, 44, 55]] # shape 2 x 5
...
X[:, 2] # Solution to the linear system A x = RHS[:, 2]
```

### Args:

- `chol` : A `Tensor`. Must be `float32` or `float64`, shape is `[M, M]`. Cholesky factorization of `A`, e.g. `chol = tf.cholesky(A)`. For that reason, only the lower

triangular part (including the diagonal) of `chol` is used. The strictly upper part is assumed to be zero and not accessed.

- `rhs` : A `Tensor`, same type as `chol`, shape is `[M, K]`, designating `K` systems of linear equations.
- `name` : A name to give this `op`. Defaults to `cholesky_solve`.

#### Returns:

Solution to `A X = RHS`, shape `[M, K]`. The solutions to the `K` systems.

## **`tf.batch_cholesky_solve(chol, rhs, name=None)`**

Solve batches of linear eqns `A X = RHS`, given Cholesky factorizations.

```
Solve one linear system (K = 1) for every member of the length 10 batch.
A = ... # shape 10 x 2 x 2
RHS = ... # shape 10 x 2 x 1
chol = tf.batch_cholesky(A) # shape 10 x 2 x 2
X = tf.batch_cholesky_solve(chol, RHS) # shape 10 x 2 x 1
tf.matmul(A, X) ~ RHS
X[3, :, 0] # Solution to the linear system A[3, :, :] x = RHS[3, :, 0]

Solve five linear systems (K = 5) for every member of the length 10 batch.
A = ... # shape 10 x 2 x 2
RHS = ... # shape 10 x 2 x 5
...
X[3, :, 2] # Solution to the linear system A[3, :, :] x = RHS[3, :, 2]
```

#### Args:

- `chol` : A `Tensor`. Must be `float32` or `float64`, shape is `[..., M, M]`. Cholesky factorization of `A`, e.g. `chol = tf.batch_cholesky(A)`. For that reason, only the lower triangular parts (including the diagonal) of the last two dimensions of `chol` are used. The strictly upper part is assumed to be zero and not accessed.
- `rhs` : A `Tensor`, same type as `chol`, shape is `[..., M, K]`.
- `name` : A name to give this `op`. Defaults to `batch_cholesky_solve`.

#### Returns:

Solution to `A x = rhs`, shape `[..., M, K]`.

## **`tf.self_adjoint_eig(input, name=None)`**

Calculates the Eigen Decomposition of a square Self-Adjoint matrix.

Only the lower-triangular part of the input will be used in this case. The upper-triangular part will not be read.

The result is a  $M+1 \times M$  matrix whose first row is the eigenvalues, and subsequent rows are eigenvectors.

**Args:**

- `input` : A `Tensor`. Must be one of the following types: `float64`, `float32`. Shape is  $[M, M]$ .
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `input`. Shape is  $[M+1, M]$ .

---

**`tf.batch_self_adjoint_eig(input, name=None)`**

Calculates the Eigen Decomposition of a batch of square self-adjoint matrices.

The input is a tensor of shape  $[..., M, M]$  whose inner-most 2 dimensions form square matrices, with the same constraints as the single matrix `SelfAdjointEig`.

The result is a  $[..., M+1, M]$  matrix with  $[..., 0, :]$  containing the eigenvalues, and subsequent  $[..., 1:, :]$  containing the eigenvectors.

**Args:**

- `input` : A `Tensor`. Must be one of the following types: `float64`, `float32`. Shape is  $[..., M, M]$ .
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `input`. Shape is  $[..., M+1, M]$ .

---

**`tf.matrix_solve(matrix, rhs, adjoint=None, name=None)`**

Solves a system of linear equations. Checks for invertibility.

**Args:**

- `matrix` : A `Tensor`. Must be one of the following types: `float64`, `float32`. Shape is `[M, M]`.
- `rhs` : A `Tensor`. Must have the same type as `matrix`. Shape is `[M, K]`.
- `adjoint` : An optional `bool`. Defaults to `False`. Boolean indicating whether to solve with `matrix` or its adjoint.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `matrix`. Shape is `[M, K]`. If `adjoint` is `False` then output that solves `matrix * output = rhs`. If `adjoint` is `True` then output that solves `adjoint(matrix) * output = rhs`.

---

```
tf.batch_matrix_solve(matrix, rhs,
adjoint=None, name=None)
```

Solves systems of linear equations. Checks for invertibility.

Matrix is a tensor of shape `[..., M, M]` whose inner-most 2 dimensions form square matrices. Rhs is a tensor of shape `[..., M, K]`. The output is a tensor shape `[..., M, K]`. If `adjoint` is `False` then each output matrix satisfies `matrix[..., :, :] * output[..., :, :] = rhs[..., :, :]`. If `adjoint` is `True` then each output matrix satisfies `adjoint(matrix[..., :, :]) * output[..., :, :] = rhs[..., :, :]`.

**Args:**

- `matrix` : A `Tensor`. Must be one of the following types: `float64`, `float32`. Shape is `[..., M, M]`.
- `rhs` : A `Tensor`. Must have the same type as `matrix`. Shape is `[..., M, K]`.
- `adjoint` : An optional `bool`. Defaults to `False`. Boolean indicating whether to solve with `matrix` or its (block-wise) adjoint.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `matrix`. Shape is `[..., M, K]`.

---

## **tf.matrix\_triangular\_solve(matrix, rhs, lower=None, adjoint=None, name=None)**

Solves a system of linear equations with an upper or lower triangular matrix by backsubstitution.

`matrix` is a matrix of shape `[M, M]`. If `lower` is `True` then the strictly upper triangular part of `matrix` is assumed to be zero and not accessed. If `lower` is `False` then the strictly lower triangular part of `matrix` is assumed to be zero and not accessed. `rhs` is a matrix of shape `[M, K]`.

The output is a matrix of shape `[M, K]`. If `adjoint` is `False` the output satisfies the matrix equation `matrix * output = rhs`. If `adjoint` is `False` then `output` satisfies the matrix equation `matrix * output = rhs`. If `adjoint` is `True` then `output` satisfies the matrix equation `adjoint(matrix) * output = rhs`.

### Args:

- `matrix` : A `Tensor`. Must be one of the following types: `float64`, `float32`. Shape is `[M, M]`.
- `rhs` : A `Tensor`. Must have the same type as `matrix`. Shape is `[M, K]`.
- `lower` : An optional `bool`. Defaults to `True`. Boolean indicating whether `matrix` is lower or upper triangular
- `adjoint` : An optional `bool`. Defaults to `False`. Boolean indicating whether to solve with `matrix` or its adjoint.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor`. Has the same type as `matrix`. Shape is `[M, K]`.

## **tf.batch\_matrix\_triangular\_solve(matrix, rhs, lower=None, adjoint=None, name=None)**

Solves systems of linear equations with upper or lower triangular matrices by backsubstitution.

`matrix` is a tensor of shape `[..., M, M]` whose inner-most 2 dimensions form square matrices. If `lower` is `True` then the strictly upper triangular part of each inner-most matrix is assumed to be zero and not accessed. If `lower` is `False` then the strictly lower triangular

part of each inner-most matrix is assumed to be zero and not accessed. `rhs` is a tensor of shape `[..., M, K]`.

The output is a tensor of shape `[..., M, K]`. If `adjoint` is `True` then the innermost matrices in `output` satisfy matrix equations `matrix[..., :, :] output[..., :, :] = rhs[..., :, :]`. If `adjoint` is `False` then strictly then the innermost matrices in `output` satisfy matrix equations `adjoint(matrix[..., i, k]) output[..., k, j] = rhs[..., i, j]`.

### Args:

- `matrix` : A `Tensor`. Must be one of the following types: `float64`, `float32`. Shape is `[..., M, M]`.
- `rhs` : A `Tensor`. Must have the same type as `matrix`. Shape is `[..., M, K]`.
- `lower` : An optional `bool`. Defaults to `True`. Boolean indicating whether the innermost matrices in `matrix` are lower or upper triangular.
- `adjoint` : An optional `bool`. Defaults to `False`. Boolean indicating whether to solve with `matrix` or its (block-wise) adjoint.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor`. Has the same type as `matrix`. Shape is `[..., M, K]`.

```
tf.matrix_solve_ls(matrix, rhs,
12_regularizer=0.0, fast=True, name=None)
```

Solves a linear least-squares problem.

Below we will use the following notation `matrix` = $\langle A \in \mathbb{R}^{m \times n} \rangle$ , `rhs` = $\langle B \in \mathbb{R}^{m \times k} \rangle$ , `output` = $\langle X \in \mathbb{R}^{n \times k} \rangle$ , `12_regularizer` = $\langle \lambda \rangle$ .

If `fast` is `True`, then the solution is computed by solving the normal equations using Cholesky decomposition. Specifically, if  $m \geq n$  then  $(X = (A^T A + \lambda I)^{-1} A^T B)$ , which solves the regularized least-squares problem  $(X = \operatorname{argmin}_Z \|Z\|_F^2 + \lambda \|Z\|_F^2)$ . If  $m < n$  then `output` is computed as  $(X = A^T (A A^T + \lambda I)^{-1} B)$ , which (for  $\lambda = 0$ ) is the minimum-norm solution to the under-determined linear system, i.e.  $(X = \operatorname{argmin}_Z \|Z\|_F^2)$ , subject to  $(A Z = B)$ . Notice that the fast path is only numerically stable when  $(A)$  is numerically full rank and has a condition number  $(\operatorname{cond}(A))$  less than  $\frac{1}{\sqrt{\epsilon_{\text{mach}}}}$  or  $(\lambda)$  is sufficiently large.

If `fast` is `False` then the solution is computed using the rank revealing QR decomposition with column pivoting. This will always compute a least-squares solution that minimizes the residual norm  $\|A X - B\|_F^2$ , even when  $A$  is rank deficient or ill-conditioned. Notice: The current version does not compute a minimum norm solution. If `fast` is `False` then `l2_regularizer` is ignored.

**Args:**

- `matrix` : 2-D `Tensor` of shape `[M, N]` .
- `rhs` : 2-D `Tensor` of shape is `[M, K]` .
- `l2_regularizer` : 0-D `double Tensor` . Ignored if `fast=False` .
- `fast` : bool. Defaults to `True` .
- `name` : string, optional name of the operation.

**Returns:**

- `output` : Matrix of shape `[N, K]` containing the matrix that solves `matrix * output = rhs` in the least-squares sense.

```
tf.batch_matrix_solve_ls(matrix, rhs,
l2_regularizer=0.0, fast=True, name=None)
```

Solves multiple linear least-squares problems.

`matrix` is a tensor of shape `[..., M, N]` whose inner-most 2 dimensions form `M`-by-`N` matrices. `Rhs` is a tensor of shape `[..., M, K]` whose inner-most 2 dimensions form `M`-by-`K` matrices. The computed output is a `Tensor` of shape `[..., N, K]` whose inner-most 2 dimensions form `M`-by-`K` matrices that solve the equations `matrix[..., :, :] * output[..., :, :] = rhs[..., :, :]` in the least squares sense.

Below we will use the following notation for each pair of matrix and right-hand sides in the batch:

`matrix = \(\mathbf{A} \in \mathbb{R}^{m \times n}\)`, `rhs = \(\mathbf{B} \in \mathbb{R}^{m \times k}\)`, `output = \(\mathbf{X} \in \mathbb{R}^{n \times k}\)`, `l2_regularizer = \(\lambda\)`.

If `fast` is `True`, then the solution is computed by solving the normal equations using Cholesky decomposition. Specifically, if  $m \geq n$  then  $(\mathbf{X} = (\mathbf{A}^\top \mathbf{A} + \lambda \mathbf{I})^{-1} \mathbf{A}^\top \mathbf{B})$ , which solves the least-squares problem  $(\mathbf{X} = \operatorname{argmin}_{\mathbf{Z}} \|\mathbf{Z} \in \mathbb{R}^{n \times k} \| \|\mathbf{A} \mathbf{Z} - \mathbf{B}\|_F^2 + \lambda \|\mathbf{Z}\|_F^2)$ . If  $m < n$  then `output` is computed as  $(\mathbf{X} = \mathbf{A}^\top (\mathbf{A} \mathbf{A}^\top + \lambda \mathbf{I})^{-1} \mathbf{B})$ , which (for  $\lambda = 0$ ) is the minimum-norm solution to the under-determined linear system, i.e.  $(\mathbf{X} = \operatorname{argmin}_{\mathbf{Z}} \|\mathbf{Z} \in \mathbb{R}^{n \times k} \| \|\mathbf{Z}\|_F^2)$ ,

subject to  $\|A\|Z = B$ ). Notice that the fast path is only numerically stable when  $\|A\|$  is numerically full rank and has a condition number  $\mathrm{cond}(A) \ll \frac{1}{\sqrt{\epsilon_{\text{mach}}}}$  or  $\lambda$  is sufficiently large.

If `fast` is `False` an algorithm based on the numerically robust complete orthogonal decomposition is used. This computes the minimum-norm least-squares solution, even when  $\|A\|$  is rank deficient. This path is typically 6-7 times slower than the fast path. If `fast` is `False` then `l2_regularizer` is ignored.

#### Args:

- `matrix` : `Tensor` of shape `[..., M, N]` .
- `rhs` : `Tensor` of shape `[..., M, K]` .
- `l2_regularizer` : 0-D `double Tensor`. Ignored if `fast=False` .
- `fast` : `bool`. Defaults to `True` .
- `name` : `string`, optional name of the operation.

#### Returns:

- `output` : `Tensor` of shape `[..., N, K]` whose inner-most 2 dimensions form `M` - by- `K` matrices that solve the equations `matrix[..., :, :] * output[..., :, :] = rhs[..., :, :]` in the least squares sense.

## Complex Number Functions

TensorFlow provides several operations that you can use to add complex number functions to your graph.

### `tf.complex(real, imag, name=None)`

Converts two real numbers to a complex number.

Given a tensor `real` representing the real part of a complex number, and a tensor `imag` representing the imaginary part of a complex number, this operation returns complex numbers elementwise of the form  $(a + bj)$ , where  $a$  represents the `real` part and  $b$  represents the `imag` part.

The input tensors `real` and `imag` must have the same shape.

For example:

```
tensor 'real' is [2.25, 3.25]
tensor `imag` is [4.75, 5.75]
tf.complex(real, imag) ==> [[2.25 + 4.75j], [3.25 + 5.75j]]
```

**Args:**

- `real` : A `Tensor`. Must be one of the following types: `float32`, `float64`.
- `imag` : A `Tensor`. Must have the same type as `real`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `complex64` or `complex128`.

**`tf.complex_abs(x, name=None)`**

Computes the complex absolute value of a tensor.

Given a tensor `x` of complex numbers, this operation returns a tensor of type `float` or `double` that is the absolute value of each element in `x`. All elements in `x` must be complex numbers of the form  $(a + bj)$ . The absolute value is computed as  $\sqrt{a^2 + b^2}$ .

For example:

```
tensor 'x' is [[-2.25 + 4.75j], [-3.25 + 5.75j]]
tf.complex_abs(x) ==> [5.25594902, 6.60492229]
```

**Args:**

- `x` : A `Tensor` of type `complex64` or `complex128`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `float32` or `float64`.

**`tf.conj(input, name=None)`**

Returns the complex conjugate of a complex number.

Given a tensor `input` of complex numbers, this operation returns a tensor of complex numbers that are the complex conjugate of each element in `input`. The complex numbers in `input` must be of the form  $(a + bj)$ , where  $a$  is the real part and  $b$  is the imaginary part.

The complex conjugate returned by this operation is of the form  $(a - bj)$ .

For example:

```
tensor 'input' is [-2.25 + 4.75j, 3.25 + 5.75j]
tf.conj(input) ==> [-2.25 - 4.75j, 3.25 - 5.75j]
```

### Args:

- `input` : A `Tensor`. Must be one of the following types: `complex64`, `complex128`.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor`. Has the same type as `input`.

## `tf.imag(input, name=None)`

Returns the imaginary part of a complex number.

Given a tensor `input` of complex numbers, this operation returns a tensor of type `float` or `double` that is the imaginary part of each element in `input`. All elements in `input` must be complex numbers of the form  $(a + bj)$ , where  $a$  is the real part and  $b$  is the imaginary part returned by this operation.

For example:

```
tensor 'input' is [-2.25 + 4.75j, 3.25 + 5.75j]
tf.imag(input) ==> [4.75, 5.75]
```

### Args:

- `input` : A `Tensor`. Must be one of the following types: `complex64`, `complex128`.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor` of type `float` or `double`.

## tf.real(input, name=None)

Returns the real part of a complex number.

Given a tensor `input` of complex numbers, this operation returns a tensor of type `float` or `double` that is the real part of each element in `input`. All elements in `input` must be complex numbers of the form  $(a + bj)$ , where  $a$  is the real part returned by this operation and  $b$  is the imaginary part.

For example:

```
tensor 'input' is [-2.25 + 4.75j, 3.25 + 5.75j]
tf.real(input) ==> [-2.25, 3.25]
```

### Args:

- `input` : A `Tensor`. Must be one of the following types: `complex64`,  
`complex128`.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor` of type `float` or `double`.

## tf.fft(input, name=None)

Compute the 1-dimensional discrete Fourier Transform.

### Args:

- `input` : A `Tensor` of type `complex64`. A complex64 vector.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor` of type `complex64`. The 1D Fourier Transform of `input`.

## tf.ifft(input, name=None)

Compute the inverse 1-dimensional discrete Fourier Transform.

**Args:**

- `input` : A `Tensor` of type `complex64`. A complex64 vector.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `complex64`. The inverse 1D Fourier Transform of `input`.

---

**`tf.fft2d(input, name=None)`**

Compute the 2-dimensional discrete Fourier Transform.

**Args:**

- `input` : A `Tensor` of type `complex64`. A complex64 matrix.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `complex64`. The 2D Fourier Transform of `input`.

---

**`tf.ifft2d(input, name=None)`**

Compute the inverse 2-dimensional discrete Fourier Transform.

**Args:**

- `input` : A `Tensor` of type `complex64`. A complex64 matrix.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `complex64`. The inverse 2D Fourier Transform of `input`.

---

**`tf.fft3d(input, name=None)`**

Compute the 3-dimensional discrete Fourier Transform.

**Args:**

- `input` : A `Tensor` of type `complex64`. A complex64 3-D tensor.

- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `complex64`. The 3D Fourier Transform of `input`.

---

**`tf.ifft3d(input, name=None)`**

Compute the inverse 3-dimensional discrete Fourier Transform.

**Args:**

- `input` : A `Tensor` of type `complex64`. A complex64 3-D tensor.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `complex64`. The inverse 3D Fourier Transform of `input`.

---

**`tf.batch_fft(input, name=None)`**

Compute the 1-dimensional discrete Fourier Transform over the inner-most dimension of `input`.

**Args:**

- `input` : A `Tensor` of type `complex64`. A complex64 tensor.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `complex64`. A complex64 tensor of the same shape as `input`. The inner-most dimension of `input` is replaced with its 1D Fourier Transform.

---

**`tf.batch_ifft(input, name=None)`**

Compute the inverse 1-dimensional discrete Fourier Transform over the inner-most dimension of `input`.

**Args:**

- `input` : A `Tensor` of type `complex64`. A complex64 tensor.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `complex64`. A complex64 tensor of the same shape as `input`. The inner-most dimension of `input` is replaced with its inverse 1D Fourier Transform.

**tf.batch\_fft2d(input, name=None)**

Compute the 2-dimensional discrete Fourier Transform over the inner-most 2 dimensions of `input`.

**Args:**

- `input` : A `Tensor` of type `complex64`. A complex64 tensor.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `complex64`. A complex64 tensor of the same shape as `input`. The inner-most 2 dimensions of `input` are replaced with their 2D Fourier Transform.

**tf.batch\_ifft2d(input, name=None)**

Compute the inverse 2-dimensional discrete Fourier Transform over the inner-most 2 dimensions of `input`.

**Args:**

- `input` : A `Tensor` of type `complex64`. A complex64 tensor.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `complex64`. A complex64 tensor of the same shape as `input`. The inner-most 2 dimensions of `input` are replaced with their inverse 2D Fourier Transform.

**tf.batch\_fft3d(input, name=None)**

Compute the 3-dimensional discrete Fourier Transform over the inner-most 3 dimensions of `input`.

**Args:**

- `input` : A `Tensor` of type `complex64`. A complex64 tensor.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `complex64`. A complex64 tensor of the same shape as `input`. The inner-most 3 dimensions of `input` are replaced with their 3D Fourier Transform.

---

**`tf.batch_ifft3d(input, name=None)`**

Compute the inverse 3-dimensional discrete Fourier Transform over the inner-most 3 dimensions of `input`.

**Args:**

- `input` : A `Tensor` of type `complex64`. A complex64 tensor.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `complex64`. A complex64 tensor of the same shape as `input`. The inner-most 3 dimensions of `input` are replaced with their inverse 3D Fourier Transform.

## Reduction

TensorFlow provides several operations that you can use to perform common math computations that reduce various dimensions of a tensor.

---

**`tf.reduce_sum(input_tensor, reduction_indices=None, keep_dims=False, name=None)`**

Computes the sum of elements across dimensions of a tensor.

Reduces `input_tensor` along the dimensions given in `reduction_indices`. Unless `keep_dims` is true, the rank of the tensor is reduced by 1 for each entry in `reduction_indices`. If `keep_dims` is true, the reduced dimensions are retained with length 1.

If `reduction_indices` has no entries, all dimensions are reduced, and a tensor with a single element is returned.

For example:

```
'x' is [[1, 1, 1]
[1, 1, 1]]
tf.reduce_sum(x) ==> 6
tf.reduce_sum(x, 0) ==> [2, 2, 2]
tf.reduce_sum(x, 1) ==> [3, 3]
tf.reduce_sum(x, 1, keep_dims=True) ==> [[3], [3]]
tf.reduce_sum(x, [0, 1]) ==> 6
```

#### Args:

- `input_tensor` : The tensor to reduce. Should have numeric type.
- `reduction_indices` : The dimensions to reduce. If `None` (the default), reduces all dimensions.
- `keep_dims` : If true, retains reduced dimensions with length 1.
- `name` : A name for the operation (optional).

#### Returns:

The reduced tensor.

```
tf.reduce_prod(input_tensor,
reduction_indices=None, keep_dims=False,
name=None)
```

Computes the product of elements across dimensions of a tensor.

Reduces `input_tensor` along the dimensions given in `reduction_indices`. Unless `keep_dims` is true, the rank of the tensor is reduced by 1 for each entry in `reduction_indices`. If `keep_dims` is true, the reduced dimensions are retained with length 1.

If `reduction_indices` has no entries, all dimensions are reduced, and a tensor with a single element is returned.

#### Args:

- `input_tensor` : The tensor to reduce. Should have numeric type.
- `reduction_indices` : The dimensions to reduce. If `None` (the default), reduces all dimensions.
- `keep_dims` : If true, retains reduced dimensions with length 1.
- `name` : A name for the operation (optional).

**Returns:**

The reduced tensor.

---

```
tf.reduce_min(input_tensor,
reduction_indices=None, keep_dims=False,
name=None)
```

Computes the minimum of elements across dimensions of a tensor.

Reduces `input_tensor` along the dimensions given in `reduction_indices`. Unless `keep_dims` is true, the rank of the tensor is reduced by 1 for each entry in `reduction_indices`. If `keep_dims` is true, the reduced dimensions are retained with length 1. If `reduction_indices` has no entries, all dimensions are reduced, and a tensor with a single element is returned.

**Args:**

- `input_tensor` : The tensor to reduce. Should have numeric type.
- `reduction_indices` : The dimensions to reduce. If `None` (the default), reduces all dimensions.
- `keep_dims` : If true, retains reduced dimensions with length 1.
- `name` : A name for the operation (optional).

**Returns:**

The reduced tensor.

---

```
tf.reduce_max(input_tensor,
reduction_indices=None, keep_dims=False,
name=None)
```

Computes the maximum of elements across dimensions of a tensor.

Reduces `input_tensor` along the dimensions given in `reduction_indices`. Unless `keep_dims` is true, the rank of the tensor is reduced by 1 for each entry in `reduction_indices`. If `keep_dims` is true, the reduced dimensions are retained with length 1. If `reduction_indices` has no entries, all dimensions are reduced, and a tensor with a single element is returned.

**Args:**

- `input_tensor` : The tensor to reduce. Should have numeric type.
- `reduction_indices` : The dimensions to reduce. If `None` (the default), reduces all dimensions.
- `keep_dims` : If true, retains reduced dimensions with length 1.
- `name` : A name for the operation (optional).

**Returns:**

The reduced tensor.

```
tf.reduce_mean(input_tensor,
reduction_indices=None, keep_dims=False,
name=None)
```

Computes the mean of elements across dimensions of a tensor.

Reduces `input_tensor` along the dimensions given in `reduction_indices`. Unless `keep_dims` is true, the rank of the tensor is reduced by 1 for each entry in `reduction_indices`. If `keep_dims` is true, the reduced dimensions are retained with length 1. If `reduction_indices` has no entries, all dimensions are reduced, and a tensor with a single element is returned.

If `reduction_indices` has no entries, all dimensions are reduced, and a tensor with a single element is returned.

For example:

```
'x' is [[1., 1.]
[2., 2.]]
tf.reduce_mean(x) ==> 1.5
tf.reduce_mean(x, 0) ==> [1.5, 1.5]
tf.reduce_mean(x, 1) ==> [1., 2.]
```

**Args:**

- `input_tensor` : The tensor to reduce. Should have numeric type.
- `reduction_indices` : The dimensions to reduce. If `None` (the default), reduces all

dimensions.

- `keep_dims` : If true, retains reduced dimensions with length 1.
- `name` : A name for the operation (optional).

#### Returns:

The reduced tensor.

```
tf.reduce_all(input_tensor,
reduction_indices=None, keep_dims=False,
name=None)
```

Computes the "logical and" of elements across dimensions of a tensor.

Reduces `input_tensor` along the dimensions given in `reduction_indices`. Unless `keep_dims` is true, the rank of the tensor is reduced by 1 for each entry in `reduction_indices`. If `keep_dims` is true, the reduced dimensions are retained with length 1. If `reduction_indices` has no entries, all dimensions are reduced, and a tensor with a single element is returned.

For example:

```
'x' is [[True, True]
[False, False]]
tf.reduce_all(x) ==> False
tf.reduce_all(x, 0) ==> [False, False]
tf.reduce_all(x, 1) ==> [True, False]
```

#### Args:

- `input_tensor` : The boolean tensor to reduce.
- `reduction_indices` : The dimensions to reduce. If `None` (the default), reduces all dimensions.
- `keep_dims` : If true, retains reduced dimensions with length 1.
- `name` : A name for the operation (optional).

#### Returns:

The reduced tensor.

```
tf.reduce_any(input_tensor,
reduction_indices=None, keep_dims=False,
name=None)
```

Computes the "logical or" of elements across dimensions of a tensor.

Reduces `input_tensor` along the dimensions given in `reduction_indices`. Unless `keep_dims` is true, the rank of the tensor is reduced by 1 for each entry in `reduction_indices`. If `keep_dims` is true, the reduced dimensions are retained with length 1.

If `reduction_indices` has no entries, all dimensions are reduced, and a tensor with a single element is returned.

For example:

```
'x' is [[True, True]
[False, False]]
tf.reduce_any(x) ==> True
tf.reduce_any(x, 0) ==> [True, True]
tf.reduce_any(x, 1) ==> [True, False]
```

### Args:

- `input_tensor` : The boolean tensor to reduce.
- `reduction_indices` : The dimensions to reduce. If `None` (the default), reduces all dimensions.
- `keep_dims` : If true, retains reduced dimensions with length 1.
- `name` : A name for the operation (optional).

### Returns:

The reduced tensor.

```
tf.accumulate_n(inputs, shape=None,
tensor_dtype=None, name=None)
```

Returns the element-wise sum of a list of tensors.

Optionally, pass `shape` and `tensor_dtype` for shape and type checking, otherwise, these are inferred.

For example:

```
tensor 'a' is [[1, 2], [3, 4]]
tensor `b` is [[5, 0], [0, 6]]
tf.accumulate_n([a, b, a]) ==> [[7, 4], [6, 14]]

Explicitly pass shape and type
tf.accumulate_n([a, b, a], shape=[2, 2], tensor_dtype=tf.int32)
==> [[7, 4], [6, 14]]
```

**Args:**

- `inputs` : A list of `Tensor` objects, each with same shape and type.
- `shape` : Shape of elements of `inputs`.
- `tensor_dtype` : The type of `inputs`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of same shape and type as the elements of `inputs`.

**Raises:**

- `ValueError` : If `inputs` don't all have same shape and dtype or the shape cannot be inferred.

## Segmentation

TensorFlow provides several operations that you can use to perform common math computations on tensor segments. Here a segmentation is a partitioning of a tensor along the first dimension, i.e. it defines a mapping from the first dimension onto `segment_ids`. The `segment_ids` tensor should be the size of the first dimension, `d0`, with consecutive IDs in the range `0` to `k`, where `k < d0`. In particular, a segmentation of a matrix tensor is a mapping of rows to segments.

For example:

```
c = tf.constant([[1,2,3,4], [-1,-2,-3,-4], [5,6,7,8]])
tf.segment_sum(c, tf.constant([0, 0, 1]))
==> [[0 0 0 0]
 [5 6 7 8]]
```

---

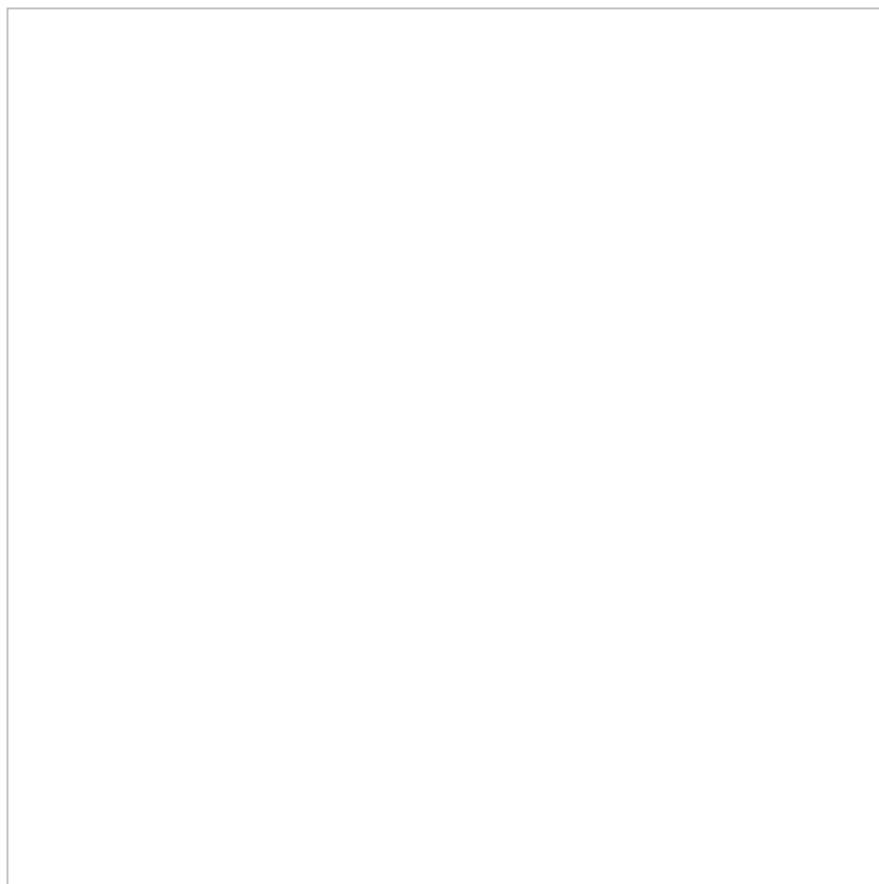
**`tf.segment_sum(data, segment_ids, name=None)`**

---

Computes the sum along segments of a tensor.

Read [the section on Segmentation](#) for an explanation of segments.

Computes a tensor such that  $(\text{output}_i = \sum_j \text{data}_j)$  where sum is over  $j$  such that  $\text{segment\_ids}[j] == i$ .



### Args:

- `data` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`, `int16`, `int8`, `complex64`, `complex128`, `qint8`, `quint8`, `qint32`, `half`.
- `segment_ids` : A `Tensor`. Must be one of the following types: `int32`, `int64`. A 1-D tensor whose rank is equal to the rank of `data`'s first dimension. Values should be sorted and can be repeated.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor`. Has the same type as `data`. Has same shape as `data`, except for dimension 0 which has size `k`, the number of segments.

## tf.segment\_prod(data, segment\_ids, name=None)

Computes the product along segments of a tensor.

Read [the section on Segmentation](#) for an explanation of segments.

Computes a tensor such that  $(\text{output}_i = \prod_j \text{data}_j)$  where the product is over  $j$  such that  $\text{segment\_ids}[j] == i$ .



### Args:

- `data` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`, `int16`, `int8`, `complex64`, `complex128`, `qint8`, `quint8`, `qint32`, `half`.
- `segment_ids` : A `Tensor`. Must be one of the following types: `int32`, `int64`. A 1-D tensor whose rank is equal to the rank of `data`'s first dimension. Values should be sorted and can be repeated.
- `name` : A name for the operation (optional).

### Returns:

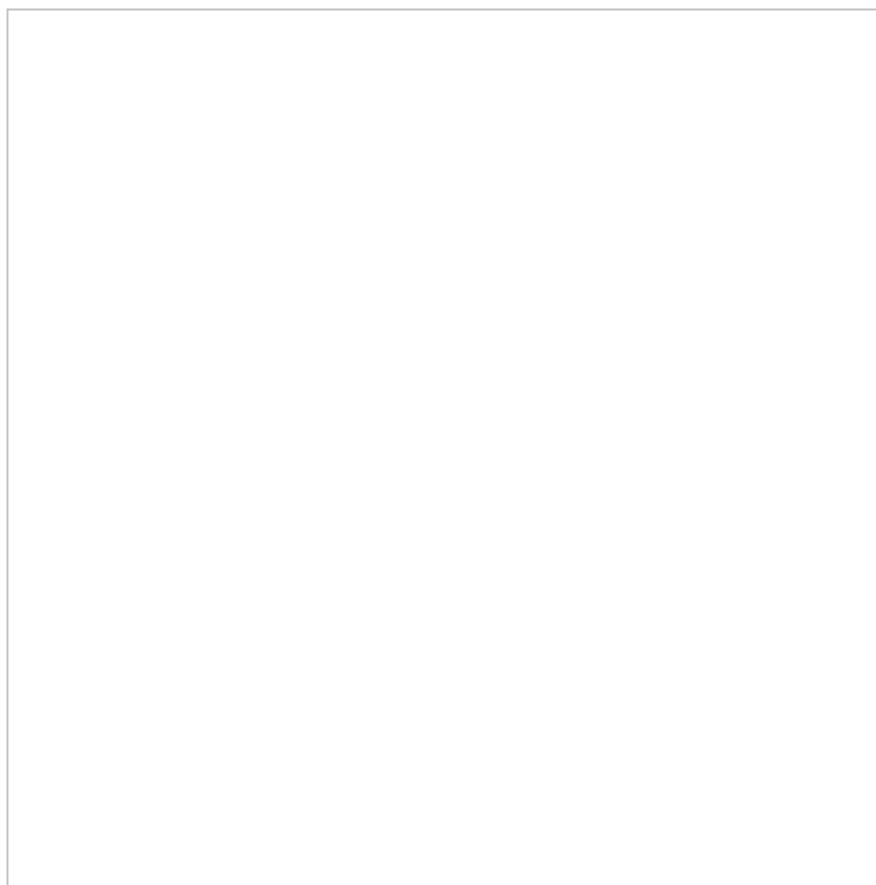
A `Tensor`. Has the same type as `data`. Has same shape as `data`, except for dimension 0 which has size `k`, the number of segments.

## tf.segment\_min(data, segment\_ids, name=None)

Computes the minimum along segments of a tensor.

Read [the section on Segmentation](#) for an explanation of segments.

Computes a tensor such that  $\text{output}_i = \min_j(\text{data}_j)$  where `min` is over `j` such that `segment_ids[j] == i`.



### Args:

- `data` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `segment_ids` : A `Tensor`. Must be one of the following types: `int32`, `int64`. A 1-D tensor whose rank is equal to the rank of `data`'s first dimension. Values should be sorted and can be repeated.
- `name` : A name for the operation (optional).

### Returns:

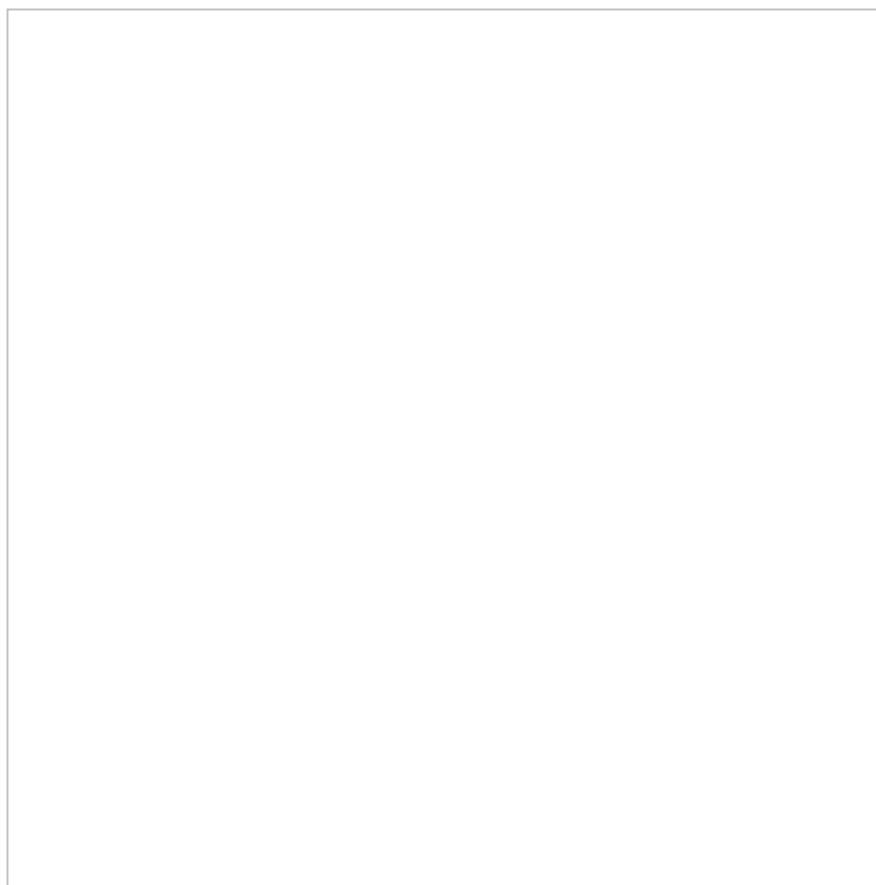
A `Tensor`. Has the same type as `data`. Has same shape as `data`, except for dimension 0 which has size `k`, the number of segments.

## tf.segment\_max(data, segment\_ids, name=None)

Computes the maximum along segments of a tensor.

Read [the section on Segmentation](#) for an explanation of segments.

Computes a tensor such that  $\text{output}_i = \max_j(\text{data}_j)$  where  $\max$  is over  $j$  such that  $\text{segment\_ids}[j] == i$ .



### Args:

- `data` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `segment_ids` : A `Tensor`. Must be one of the following types: `int32`, `int64`. A 1-D tensor whose rank is equal to the rank of `data`'s first dimension. Values should be sorted and can be repeated.
- `name` : A name for the operation (optional).

### Returns:

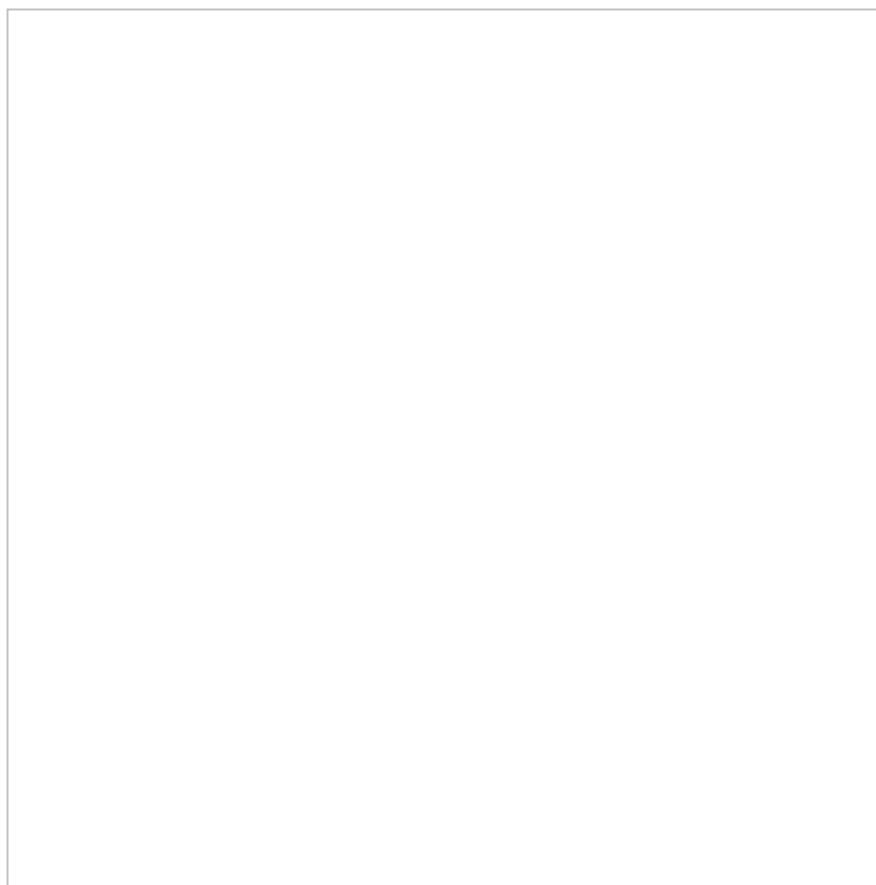
A `Tensor`. Has the same type as `data`. Has same shape as `data`, except for dimension 0 which has size `k`, the number of segments.

## tf.segment\_mean(data, segment\_ids, name=None)

Computes the mean along segments of a tensor.

Read [the section on Segmentation](#) for an explanation of segments.

Computes a tensor such that  $\text{output}_i = \frac{\sum_j \text{data}_j}{N}$  where `mean` is over `j` such that `segment_ids[j] == i` and `N` is the total number of values summed.



### Args:

- `data` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `segment_ids` : A `Tensor`. Must be one of the following types: `int32`, `int64`. A 1-D tensor whose rank is equal to the rank of `data`'s first dimension. Values should be sorted and can be repeated.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor`. Has the same type as `data`. Has same shape as `data`, except for dimension 0 which has size `k`, the number of segments.

## `tf.unsorted_segment_sum(data, segment_ids, num_segments, name=None)`

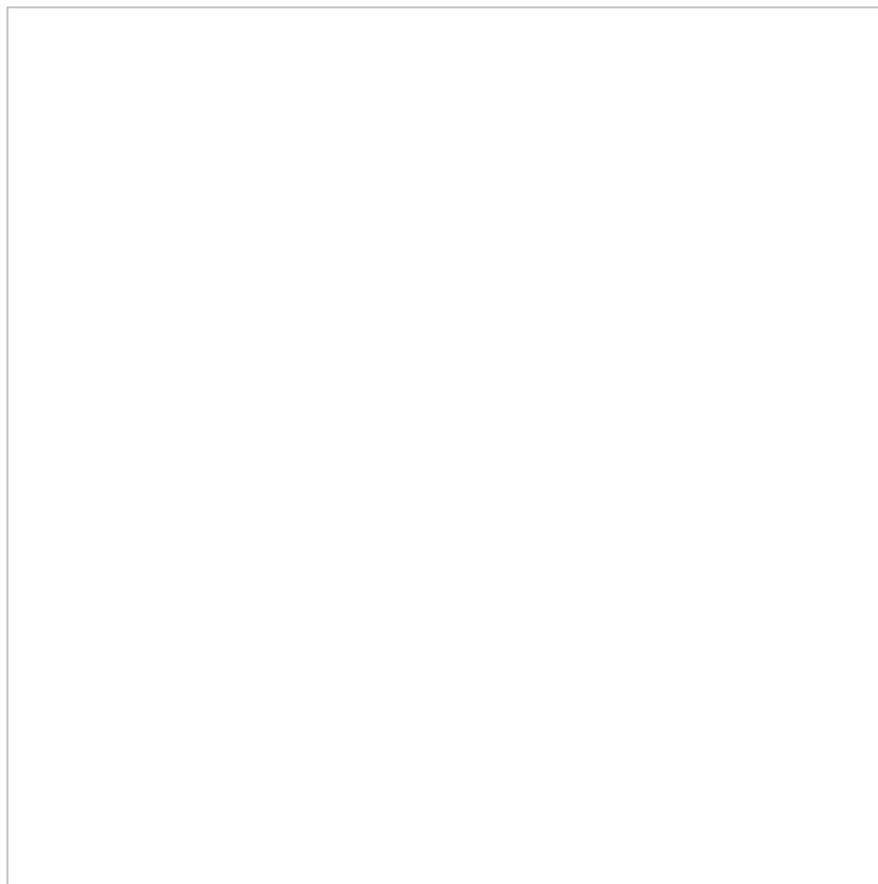
Computes the sum along segments of a tensor.

Read [the section on Segmentation](#) for an explanation of segments.

Computes a tensor such that  $(\text{output}_i = \sum_j \text{data}_j)$  where sum is over  $j$  such that  $\text{segment_ids}[j] == i$ . Unlike `SegmentSum`, `segment_ids` need not be sorted and need not cover all values in the full range of valid values.

If the sum is empty for a given segment ID  $i$ ,  $\text{output}[i] = 0$ .

`num_segments` should equal the number of distinct segment IDs.



### Args:

- `data` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`, `int16`, `int8`, `complex64`, `complex128`, `qint8`, `quint8`, `qint32`, `half`.
- `segment_ids` : A `Tensor`. Must be one of the following types: `int32`, `int64`. A 1-D tensor whose rank is equal to the rank of `data`'s first dimension.

- `num_segments` : A `Tensor` of type `int32`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `data`. Has same shape as `data`, except for dimension 0 which has size `num_segments`.

## `tf.sparse_segment_sum(data, indices, segment_ids, name=None)`

Computes the sum along sparse segments of a tensor.

Read [the section on Segmentation](#) for an explanation of segments.

Like `SegmentSum`, but `segment_ids` can have rank less than `data`'s first dimension, selecting a subset of dimension 0, specified by `indices`.

For example:

```
c = tf.constant([[1,2,3,4], [-1,-2,-3,-4], [5,6,7,8]])

Select two rows, one segment.
tf.sparse_segment_sum(c, tf.constant([0, 1]), tf.constant([0, 0]))
=> [[0 0 0 0]]

Select two rows, two segment.
tf.sparse_segment_sum(c, tf.constant([0, 1]), tf.constant([0, 1]))
=> [[1 2 3 4]
 [-1 -2 -3 -4]]

Select all rows, two segments.
tf.sparse_segment_sum(c, tf.constant([0, 1, 2]), tf.constant([0, 0, 1]))
=> [[0 0 0 0]
 [5 6 7 8]]

Which is equivalent to:
tf.segment_sum(c, tf.constant([0, 0, 1]))
```

**Args:**

- `data` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `indices` : A `Tensor` of type `int32`. A 1-D tensor. Has same rank as `segment_ids`.
- `segment_ids` : A `Tensor` of type `int32`. A 1-D tensor. Values should be sorted and can be repeated.

- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `data`. Has same shape as data, except for dimension 0 which has size `k`, the number of segments.

---

## **`tf.sparse_segment_mean(data, indices, segment_ids, name=None)`**

Computes the mean along sparse segments of a tensor.

Read [the section on Segmentation](#) for an explanation of segments.

Like `SegmentMean`, but `segment_ids` can have rank less than `data`'s first dimension, selecting a subset of dimension 0, specified by `indices`.

**Args:**

- `data` : A `Tensor`. Must be one of the following types: `float32`, `float64`.
- `indices` : A `Tensor` of type `int32`. A 1-D tensor. Has same rank as `segment_ids`.
- `segment_ids` : A `Tensor` of type `int32`. A 1-D tensor. Values should be sorted and can be repeated.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `data`. Has same shape as data, except for dimension 0 which has size `k`, the number of segments.

---

## **`tf.sparse_segment_sqrt_n(data, indices, segment_ids, name=None)`**

Computes the sum along sparse segments of a tensor divided by the sqrt of N.

N is the size of the segment being reduced.

Read [the section on Segmentation](#) for an explanation of segments.

**Args:**

- `data` : A `Tensor`. Must be one of the following types: `float32`, `float64`.
- `indices` : A `Tensor` of type `int32`. A 1-D tensor. Has same rank as `segment_ids`.

- `segment_ids` : A `Tensor` of type `int32`. A 1-D tensor. Values should be sorted and can be repeated.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `data`. Has same shape as `data`, except for dimension 0 which has size `k`, the number of segments.

## Sequence Comparison and Indexing

TensorFlow provides several operations that you can use to add sequence comparison and index extraction to your graph. You can use these operations to determine sequence differences and determine the indexes of specific values in a tensor.

### **`tf.argmax(input, dimension, name=None)`**

Returns the index with the smallest value across dimensions of a tensor.

**Args:**

- `input` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`, `int16`, `int8`, `complex64`, `complex128`, `qint8`, `quint8`, `qint32`, `half`.
- `dimension` : A `Tensor` of type `int32`. `int32`,  $0 \leq \text{dimension} < \text{rank}(\text{input})$ . Describes which dimension of the input Tensor to reduce across. For vectors, use `dimension = 0`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `int64`.

### **`tf.argmax(input, dimension, name=None)`**

Returns the index with the largest value across dimensions of a tensor.

**Args:**

- `input` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`, `int16`, `int8`, `complex64`, `complex128`, `qint8`, `quint8`,

```
qint32 , half .
```

- **dimension** : A `Tensor` of type `int32`. `int32`,  $0 \leq \text{dimension} < \text{rank}(\text{input})$ . Describes which dimension of the input `Tensor` to reduce across. For vectors, use `dimension = 0`.
- **name** : A name for the operation (optional).

**Returns:**

A `Tensor` of type `int64`.

**`tf.listdiff(x, y, name=None)`**

Computes the difference between two lists of numbers or strings.

Given a list `x` and a list `y`, this operation returns a list `out` that represents all values that are in `x` but not in `y`. The returned list `out` is sorted in the same order that the numbers appear in `x` (duplicates are preserved). This operation also returns a list `idx` that represents the position of each `out` element in `x`. In other words:

```
out[i] = x[idx[i]] for i in [0, 1, ..., len(out) - 1]
```

For example, given this input:

```
x = [1, 2, 3, 4, 5, 6]
y = [1, 3, 5]
```

This operation would return:

```
out ==> [2, 4, 6]
idx ==> [1, 3, 5]
```

**Args:**

- `x` : A `Tensor`. 1-D. Values to keep.
- `y` : A `Tensor`. Must have the same type as `x`. 1-D. Values to remove.
- `name` : A name for the operation (optional).

**Returns:**

A tuple of `Tensor` objects (`out, idx`).

- `out` : A `Tensor`. Has the same type as `x`. 1-D. Values present in `x` but not in `y`.
- `idx` : A `Tensor` of type `int32`. 1-D. Positions of `x` values preserved in `out`.

## tf.where(input, name=None)

Returns locations of true values in a boolean tensor.

This operation returns the coordinates of true elements in `input`. The coordinates are returned in a 2-D tensor where the first dimension (rows) represents the number of true elements, and the second dimension (columns) represents the coordinates of the true elements. Keep in mind, the shape of the output tensor can vary depending on how many true values there are in `input`. Indices are output in row-major order.

For example:

```
'input' tensor is [[True, False]
[True, False]]
'input' has two true values, so output has two coordinates.
'input' has rank of 2, so coordinates have two indices.
where(input) ==> [[0, 0],
 [1, 0]]

`input` tensor is [[[True, False]
[True, False]]
[[False, True]
[False, True]]]
[[False, False]
[False, True]]]
'input' has 5 true values, so output has 5 coordinates.
'input' has rank of 3, so coordinates have three indices.
where(input) ==> [[0, 0, 0],
 [0, 1, 0],
 [1, 0, 1],
 [1, 1, 1],
 [2, 1, 1]]]
```

### Args:

- `input` : A `Tensor` of type `bool`.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor` of type `int64`.

## tf.unique(x, name=None)

Finds unique elements in a 1-D tensor.

This operation returns a tensor `y` containing all of the unique elements of `x` sorted in the same order that they occur in `x`. This operation also returns a tensor `idx` the same size as `x` that contains the index of each value of `x` in the unique output `y`. In other words:

```
y[idx[i]] = x[i] for i in [0, 1, ..., rank(x) - 1]
```

For example:

```
tensor 'x' is [1, 1, 2, 4, 4, 4, 7, 8, 8]
y, idx = unique(x)
y ==> [1, 2, 4, 7, 8]
idx ==> [0, 0, 1, 2, 2, 2, 3, 4, 4]
```

---

### Args:

- `x` : A `Tensor` . 1-D.
- `name` : A name for the operation (optional).

### Returns:

A tuple of `Tensor` objects (`y, idx`).

- `y` : A `Tensor` . Has the same type as `x` . 1-D.
  - `idx` : A `Tensor` of type `int32` . 1-D.
- 

```
tf.edit_distance(hypothesis, truth,
normalize=True, name='edit_distance')
```

Computes the Levenshtein distance between sequences.

This operation takes variable-length sequences (`hypothesis` and `truth`), each provided as a `sparseTensor`, and computes the Levenshtein distance. You can normalize the edit distance by length of `truth` by setting `normalize` to true.

For example, given the following input:

```

'hypothesis' is a tensor of shape `[2, 1]` with variable-length values:
(0,0) = ["a"]
(1,0) = ["b"]
hypothesis = tf.SparseTensor(
 [[0, 0, 0],
 [1, 0, 0]],
 ["a", "b"]
 (2, 1))

'truth' is a tensor of shape `[2, 2]` with variable-length values:
(0,0) = []
(0,1) = ["a"]
(1,0) = ["b", "c"]
(1,1) = ["a"]
truth = tf.SparseTensor(
 [[0, 1, 0],
 [1, 0, 0],
 [1, 0, 1],
 [1, 1, 0]],
 ["a", "b", "c", "a"],
 (2, 2))

normalize = True

```

This operation would return the following:

```

'output' is a tensor of shape `[2, 2]` with edit distances normalized
by 'truth' lengths.
output ==> [[inf, 1.0], # (0,0): no truth, (0,1): no hypothesis
 [0.5, 1.0]] # (1,0): addition, (1,1): no hypothesis

```

## Args:

- `hypothesis` : A `SparseTensor` containing hypothesis sequences.
- `truth` : A `SparseTensor` containing truth sequences.
- `normalize` : A `bool`. If `True`, normalizes the Levenshtein distance by length of truth.
- `name` : A name for the operation (optional).

## Returns:

A dense `Tensor` with rank `R - 1`, where R is the rank of the `SparseTensor` inputs `hypothesis` and `truth`.

## Raises:

- `TypeError` : If either `hypothesis` or `truth` are not a `SparseTensor`.

## **tf.invert\_permutation(x, name=None)**

Computes the inverse permutation of a tensor.

This operation computes the inverse of an index permutation. It takes a 1-D integer tensor `x`, which represents the indices of a zero-based array, and swaps each value with its index position. In other words, for an output tensor `y` and an input tensor `x`, this operation computes the following:

```
y[x[i]] = i for i in [0, 1, ..., len(x) - 1]
```

The values must include 0. There can be no duplicate values or negative values.

For example:

```
tensor `x` is [3, 4, 0, 2, 1]
invert_permutation(x) ==> [2, 4, 3, 0, 1]
```

**Args:**

- `x` : A `Tensor` of type `int32`. 1-D.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `int32`. 1-D.

## **Other Functions and Classes**

### **tf.scalar\_mul(scalar, x)**

Multiplies a scalar times a `Tensor` or `IndexedSlices` object.

Intended for use in gradient code which might deal with `IndexedSlices` objects, which are easy to multiply by a scalar but more expensive to multiply with arbitrary tensors.

**Args:**

- `scalar` : A 0-D scalar `Tensor`. Must have known shape.
- `x` : A `Tensor` or `IndexedSlices` to be scaled.

**Returns:**

```
scalar * x of the same type (Tensor or IndexedSlices) as x .
```

**Raises:**

- `ValueError` : if scalar is not a 0-D `scalar` .
- 

```
tf.sparse_segment_sqrt_n_grad(grad, indices,
segment_ids, output_dim0, name=None)
```

Computes gradients for SparseSegmentSqrtN.

Returns tensor "output" with same shape as grad, except for dimension 0 whose value is `output_dim0`.

**Args:**

- `grad` : A `Tensor` . Must be one of the following types: `float32` , `float64` . gradient propagated to the SparseSegmentSqrtN op.
- `indices` : A `Tensor` of type `int32` . indices passed to the corresponding SparseSegmentSqrtN op.
- `segment_ids` : A `Tensor` of type `int32` . segment\_ids passed to the corresponding SparseSegmentSqrtN op.
- `output_dim0` : A `Tensor` of type `int32` . dimension 0 of "data" passed to SparseSegmentSqrtN op.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` . Has the same type as `grad` .

# Control Flow

Note: Functions taking `Tensor` arguments can also take anything accepted by `tf.convert_to_tensor`.

[TOC]

## Control Flow Operations

TensorFlow provides several operations and classes that you can use to control the execution of operations and add conditional dependencies to your graph.

### `tf.identity(input, name=None)`

Return a tensor with the same shape and contents as the input tensor or value.

#### Args:

- `input` : A `Tensor`.
- `name` : A name for the operation (optional).

#### Returns:

A `Tensor`. Has the same type as `input`.

### `tf.tuple(tensors, name=None, control_inputs=None)`

Group tensors together.

This creates a tuple of tensors with the same values as the `tensors` argument, except that the value of each tensor is only returned after the values of all tensors have been computed.

`control_inputs` contains additional ops that have to finish before this op finishes, but whose outputs are not returned.

This can be used as a "join" mechanism for parallel computations: all the argument tensors can be computed in parallel, but the values of any tensor returned by `tuple` are only available after all the parallel computations are done.

See also `group` and `with_dependencies`.

#### Args:

- `tensors` : A list of `Tensor`s or `IndexedSlices`, some entries can be `None`.
- `name` : (optional) A name to use as a `name_scope` for the operation.
- `control_inputs` : List of additional ops to finish before returning.

#### Returns:

Same as `tensors`.

#### Raises:

- `ValueError` : If `tensors` does not contain any `Tensor` or `IndexedSlices`.
- `TypeError` : If `control_inputs` is not a list of `Operation` or `Tensor` objects.

---

## `tf.group(*inputs, **kwargs)`

Create an op that groups multiple operations.

When this op finishes, all ops in `input` have finished. This op has no output.

See also `tuple` and `with_dependencies`.

#### Args:

- `*inputs` : Zero or more tensors to group.
- `**kwargs` : Optional parameters to pass when constructing the `NodeDef`.
- `name` : A name for this operation (optional).

#### Returns:

An Operation that executes all its inputs.

#### Raises:

- `ValueError` : If an unknown keyword argument is provided.

**tf.no\_op(name=None)**

Does nothing. Only useful as a placeholder for control edges.

**Args:**

- `name` : A name for the operation (optional).

**Returns:**

The created Operation.

**tf.count\_up\_to(ref, limit, name=None)**

Increments 'ref' until it reaches 'limit'.

This operation outputs "ref" after the update is done. This makes it easier to chain operations that need to use the updated value.

**Args:**

- `ref` : A mutable `Tensor`. Must be one of the following types: `int32`, `int64`. Should be from a scalar `Variable` node.
- `limit` : An `int`. If incrementing ref would bring it above limit, instead generates an 'OutOfRange' error.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `ref`. A copy of the input before increment. If nothing else modifies the input, the values produced will all be distinct.

**tf.cond(pred, fn1, fn2, name=None)**

Return either `fn1()` or `fn2()` based on the boolean predicate `pred`.

`fn1` and `fn2` both return lists of output tensors. `fn1` and `fn2` must have the same non-zero number and type of outputs.

Note that the conditional execution applies only to the operations defined in `fn1` and `fn2`.

Consider the following simple program:

```
z = tf.mul(a, b)
result = tf.cond(x < y, lambda: tf.add(x, z), lambda: tf.square(y))
```

If  $x < y$ , the `tf.add` operation will be executed and `tf.square` operation will not be executed. Since  $z$  is needed for at least one branch of the `cond`, the `tf.mul` operation is always executed, unconditionally. Although this behavior is consistent with the dataflow model of TensorFlow, it has occasionally surprised some users who expected a lazier semantics.

### Args:

- `pred` : A scalar determining whether to return the result of `fn1` or `fn2` .
- `fn1` : The callable to be performed if `pred` is true.
- `fn2` : The callable to be performed if `pred` is false.
- `name` : Optional name prefix for the returned tensors.

### Returns:

Tensors returned by the call to either `fn1` or `fn2` . If the callables return a singleton list, the element is extracted from the list.

### Raises:

- `TypeError` : if `fn1` or `fn2` is not callable.
- `ValueError` : if `fn1` and `fn2` do not return the same number of tensors, or  
return tensors of different types.
- `Example` :

```
x = tf.constant(2)
y = tf.constant(5)
def f1(): return tf.mul(x, 17)
def f2(): return tf.add(y, 23)
r = cond(tf.less(x, y), f1, f2)
r is set to f1().
Operations in f2 (e.g., tf.add) are not executed.
```

---

**`tf.case(pred_fn_pairs, default, exclusive=False, name='case')`**

Create a case operation.

The `pred_fn_pairs` parameter is a dict or list of pairs of size N. Each pair contains a boolean scalar tensor and a python callable that creates the tensors to be returned if the boolean evaluates to True. `default` is a callable generating a list of tensors. All the callables in `pred_fn_pairs` as well as `default` should return the same number and types of tensors.

If `exclusive==True`, all predicates are evaluated, and a logging operation with an error is returned if more than one of the predicates evaluates to True. If `exclusive==False`, execution stops at the first predicate which evaluates to True, and the tensors generated by the corresponding function are returned immediately. If none of the predicates evaluate to True, this operation returns the tensors generated by `default`.

Example 1: Pseudocode:

```
if (x < y) return 17;
else return 23;
```

Expressions:

```
f1 = lambda: tf.constant(17)
f2 = lambda: tf.constant(23)
r = case([(tf.less(x, y), f1)], default=f2)
```

Example 2: Pseudocode:

```
if (x < y && x > z) raise OpError("Only one predicate may evaluate true");
if (x < y) return 17;
else if (x > z) return 23;
else return -1;
```

Expressions:

```
x = tf.constant(0)
y = tf.constant(1)
z = tf.constant(2)
def f1(): return tf.constant(17)
def f2(): return tf.constant(23)
def f3(): return tf.constant(-1)
r = case({tf.less(x, y): f1, tf.greater(x, z): f2},
 default=f3, exclusive=True)
```

**Args:**

- `pred_fn_pairs` : Dict or list of pairs of a boolean scalar tensor and a

callable which returns a list of tensors.

- `default` : A callable that returns a list of tensors.
- `exclusive` : True iff more than one predicate is allowed to evaluate to True.
- `name` : A name for this operation (optional).

### Returns:

The tensors returned by the first pair whose predicate evaluated to True, or those returned by `default` if none does.

### Raises:

- `TypeError` : If `pred_fn_pairs` is not a list/dictionary.
- `TypeError` : If `pred_fn_pairs` is a list but does not contain 2-tuples.
- `TypeError` : If `fns[i]` is not callable for any i, or `default` is not callable.

```
tf.while_loop(cond, body, loop_vars,
parallel_iterations=10, back_prop=True,
swap_memory=False, name=None)
```

Repeat `body` while the condition `cond` is true.

`cond` is a callable returning a boolean scalar tensor. `body` is a callable returning a list of tensors of the same length and with the same types as `loop_vars`. `loop_vars` is a list of tensors that is passed to both `cond` and `body`. `cond` and `body` both take as many arguments as there are `loop_vars`.

In addition to regular Tensors or IndexedSlices, the body may accept and return TensorArray objects. The flows of the TensorArray objects will be appropriately forwarded between loops and during gradient calculations.

While `cond` evaluates to true, `body` is executed.

`while_loop` implements non-strict semantics, enabling multiple iterations to run in parallel. The maximum number of parallel iterations can be controlled by `parallel_iterations`, which gives users some control over memory consumption and execution order. For correct programs, `while_loop` should return the same result for any `parallel_iterations > 0`.

For training, TensorFlow remembers the tensors that are produced in the forward inference but needed in back propagation. These tensors can be a main source of memory consumption and often cause OOM problems when training on GPUs. When the flag `swap_memory` is true, we swap out these tensors from GPU to CPU. This for example allows us to train RNN models with very long sequences and large batches.

#### Args:

- `cond` : A callable that represents the termination condition of the loop.
- `body` : A callable that represents the loop body.
- `loop_vars` : The list of variable input tensors.
- `parallel_iterations` : The number of iterations allowed to run in parallel.
- `back_prop` : Whether backprop is enabled for this while loop.
- `swap_memory` : Whether GPU-CPU memory swap is enabled for this loop.
- `name` : Optional name prefix for the returned tensors.

#### Returns:

The output tensors for the loop variables after the loop.

#### Raises:

- `TypeError` : if `cond` or `body` is not callable.
- `ValueError` : if `loop_var` is empty.
- `Example` :

```
i = tf.constant(0)
c = lambda i: tf.less(i, 10)
b = lambda i: tf.add(i, 1)
r = tf.while_loop(c, b, [i])
```

## Logical Operators

TensorFlow provides several operations that you can use to add logical operators to your graph.

### `tf.logical_and(x, y, name=None)`

Returns the truth value of x AND y element-wise.

#### Args:

- `x` : A `Tensor` of type `bool`.
- `y` : A `Tensor` of type `bool`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `bool`.

---

**`tf.logical_not(x, name=None)`**

Returns the truth value of NOT `x` element-wise.

**Args:**

- `x` : A `Tensor` of type `bool`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `bool`.

---

**`tf.logical_or(x, y, name=None)`**

Returns the truth value of `x` OR `y` element-wise.

**Args:**

- `x` : A `Tensor` of type `bool`.
- `y` : A `Tensor` of type `bool`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `bool`.

---

**`tf.logical_xor(x, y, name='LogicalXor')`**

$x \wedge y = (x | y) \& \sim(x \& y)$ .

## Comparison Operators

---

TensorFlow provides several operations that you can use to add comparison operators to your graph.

---

## **tf.equal(x, y, name=None)**

Returns the truth value of  $(x == y)$  element-wise.

### **Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `uint8`, `int8`, `int16`, `int32`, `int64`, `complex64`, `quint8`, `qint8`, `qint32`, `string`, `bool`, `complex128`.
- `y` : A `Tensor`. Must have the same type as `x`.
- `name` : A name for the operation (optional).

### **Returns:**

A `Tensor` of type `bool`.

---

## **tf.not\_equal(x, y, name=None)**

Returns the truth value of  $(x != y)$  element-wise.

### **Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`, `uint8`, `int8`, `int16`, `int32`, `int64`, `complex64`, `quint8`, `qint8`, `qint32`, `string`, `bool`, `complex128`.
- `y` : A `Tensor`. Must have the same type as `x`.
- `name` : A name for the operation (optional).

### **Returns:**

A `Tensor` of type `bool`.

---

## **tf.less(x, y, name=None)**

Returns the truth value of  $(x < y)$  element-wise.

### **Args:**

- `x` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `y` : A `Tensor`. Must have the same type as `x`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `bool`.

---

**`tf.less_equal(x, y, name=None)`**

Returns the truth value of ( $x \leq y$ ) element-wise.

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `y` : A `Tensor`. Must have the same type as `x`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `bool`.

---

**`tf.greater(x, y, name=None)`**

Returns the truth value of ( $x > y$ ) element-wise.

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `y` : A `Tensor`. Must have the same type as `x`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `bool`.

---

**`tf.greater_equal(x, y, name=None)`**

Returns the truth value of  $(x \geq y)$  element-wise.

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `y` : A `Tensor`. Must have the same type as `x`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `bool`.

## **tf.select(condition, t, e, name=None)**

Selects elements from `t` or `e`, depending on `condition`.

The `t`, and `e` tensors must all have the same shape, and the output will also have that shape. The `condition` tensor must be a scalar if `t` and `e` are scalars. If `t` and `e` are vectors or higher rank, then `condition` must be either a vector with size matching the first dimension of `t`, or must have the same shape as `t`.

The `condition` tensor acts as a mask that chooses, based on the value at each element, whether the corresponding element / row in the output should be taken from `t` (if true) or `e` (if false).

If `condition` is a vector and `t` and `e` are higher rank matrices, then it chooses which row (outer dimension) to copy from `t` and `e`. If `condition` has the same shape as `t` and `e`, then it chooses which element to copy from `t` and `e`.

For example:

```
'condition' tensor is [[True, False]
[False, True]]
't' is [[1, 2],
[3, 4]]
'e' is [[5, 6],
[7, 8]]
select(condition, t, e) ==> [[1, 6],
 [7, 4]]

'condition' tensor is [True, False]
't' is [[1, 2],
[3, 4]]
'e' is [[5, 6],
[7, 8]]
select(condition, t, e) ==> [[1, 2],
 [7, 8]]
```

**Args:**

- `condition` : A `Tensor` of type `bool`.
- `t` : A `Tensor` which may have the same shape as `condition`. If `condition` is rank 1, `t` may have higher rank, but its first dimension must match the size of `condition`.
- `e` : A `Tensor` with the same type and shape as `t`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` with the same type and shape as `t` and `e`.

**`tf.where(input, name=None)`**

Returns locations of true values in a boolean tensor.

This operation returns the coordinates of true elements in `input`. The coordinates are returned in a 2-D tensor where the first dimension (rows) represents the number of true elements, and the second dimension (columns) represents the coordinates of the true elements. Keep in mind, the shape of the output tensor can vary depending on how many true values there are in `input`. Indices are output in row-major order.

For example:

```

'input' tensor is [[True, False]
[True, False]]
'input' has two true values, so output has two coordinates.
'input' has rank of 2, so coordinates have two indices.
where(input) ==> [[0, 0],
 [1, 0]]

`input` tensor is [[[True, False]
[True, False]]
[[False, True]
[False, True]]
[[False, False]
[False, True]]]
'input' has 5 true values, so output has 5 coordinates.
'input' has rank of 3, so coordinates have three indices.
where(input) ==> [[0, 0, 0],
 [0, 1, 0],
 [1, 0, 1],
 [1, 1, 1],
 [2, 1, 1]]

```

**Args:**

- `input` : A `Tensor` of type `bool`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `int64`.

## Debugging Operations

TensorFlow provides several operations that you can use to validate values and debug your graph.

### `tf.is_finite(x, name=None)`

Returns which elements of `x` are finite.

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `bool`.

---

**`tf.is_inf(x, name=None)`**

Returns which elements of `x` are Inf.

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `bool`.

---

**`tf.is_nan(x, name=None)`**

Returns which elements of `x` are NaN.

**Args:**

- `x` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `bool`.

---

**`tf.verify_tensor_all_finite(t, msg, name=None)`**

Assert that the tensor does not contain any NaN's or Inf's.

**Args:**

- `t` : Tensor to check.
- `msg` : Message to log on failure.
- `name` : A name for this operation (optional).

**Returns:**

Same tensor as `t`.

---

**`tf.check_numerics(tensor, message, name=None)`**

Checks a tensor for NaN and Inf values.

When run, reports an `InvalidArgument` error if `tensor` has any values that are not a number (NaN) or infinity (Inf). Otherwise, passes `tensor` as-is.

**Args:**

- `tensor` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`.
- `message` : A `string`. Prefix of the error message.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `tensor`.

---

**`tf.add_check_numerics_ops()`**

Connect a `check_numerics` to every floating point tensor.

`check_numerics` operations themselves are added for each `float` or `double` tensor in the graph. For all ops in the graph, the `check_numerics` op for all of its (`float` or `double`) inputs is guaranteed to run before the `check_numerics` op on any of its outputs.

**Returns:**

A `group` op depending on all `check_numerics` ops added.

---

**`tf.Assert(condition, data, summarize=None, name=None)`**

Asserts that the given condition is true.

If `condition` evaluates to false, print the list of tensors in `data`. `summarize` determines how many entries of the tensors to print.

NOTE: To ensure that Assert executes, one usually attaches a dependency:

```
Ensure maximum element of x is smaller or equal to 1
assert_op = tf.Assert(tf.less_equal(tf.reduce_max(x), 1.), [x])
x = tf.with_dependencies([assert_op], x)
```

### Args:

- `condition` : The condition to evaluate.
- `data` : The tensors to print out when condition is false.
- `summarize` : Print this many entries of each tensor.
- `name` : A name for this operation (optional).

### Returns:

- `assert_op` : An `Operation` that, when executed, raises a `tf.errors.InvalidArgumentError` if `condition` is not true.

**`tf.Print(input_, data, message=None, first_n=None, summarize=None, name=None)`**

Prints a list of tensors.

This is an identity op with the side effect of printing `data` when evaluating.

### Args:

- `input_` : A tensor passed through this op.
- `data` : A list of tensors to print out when op is evaluated.
- `message` : A string, prefix of the error message.
- `first_n` : Only log `first_n` number of times. Negative numbers log always; this is the default.
- `summarize` : Only print this many entries of each tensor. If None, then a maximum of 3 elements are printed per input tensor.
- `name` : A name for the operation (optional).

### Returns:

Same tensor as `input_`.



# Images

Note: Functions taking `Tensor` arguments can also take anything accepted by `tf.convert_to_tensor`.

[TOC]

## Encoding and Decoding

TensorFlow provides Ops to decode and encode JPEG and PNG formats. Encoded images are represented by scalar string Tensors, decoded images by 3-D uint8 tensors of shape `[height, width, channels]`. (PNG also supports uint16.)

The encode and decode Ops apply to one image at a time. Their input and output are all of variable size. If you need fixed size images, pass the output of the decode Ops to one of the cropping and resizing Ops.

Note: The PNG encode and decode Ops support RGBA, but the conversions Ops presently only support RGB, HSV, and GrayScale. Presently, the alpha channel has to be stripped from the image and re-attached using slicing ops.

---

```
tf.image.decode_jpeg(contents, channels=None,
ratio=None, fancy_upscaling=None,
try_recover_truncated=None,
acceptable_fraction=None, name=None)
```

Decode a JPEG-encoded image to a uint8 tensor.

The attr `channels` indicates the desired number of color channels for the decoded image.

Accepted values are:

- 0: Use the number of channels in the JPEG-encoded image.
- 1: output a grayscale image.
- 3: output an RGB image.

If needed, the JPEG-encoded image is transformed to match the requested number of color channels.

The attr `ratio` allows downscaling the image by an integer factor during decoding. Allowed values are: 1, 2, 4, and 8. This is much faster than downscaling the image later.

### Args:

- `contents` : A `Tensor` of type `string`. 0-D. The JPEG-encoded image.
- `channels` : An optional `int`. Defaults to `0`. Number of color channels for the decoded image.
- `ratio` : An optional `int`. Defaults to `1`. Downscaling ratio.
- `fancy_upscaling` : An optional `bool`. Defaults to `True`. If true use a slower but nicer upscaling of the chroma planes (yuv420/422 only).
- `try_recover_truncated` : An optional `bool`. Defaults to `False`. If true try to recover an image from truncated input.
- `acceptable_fraction` : An optional `float`. Defaults to `1`. The minimum required fraction of lines before a truncated input is accepted.
- `name` : A name for the operation (optional).

### Returns:

A `Tensor` of type `uint8`. 3-D with shape `[height, width, channels]` ..

```
tf.image.encode_jpeg(image, format=None,
quality=None, progressive=None,
optimize_size=None, chroma_downsampling=None,
density_unit=None, x_density=None,
y_density=None, xmp_metadata=None, name=None)
```

JPEG-encode an image.

`image` is a 3-D `uint8` Tensor of shape `[height, width, channels]`.

The attr `format` can be used to override the color format of the encoded output. Values can be:

- `''` : Use a default format based on the number of channels in the image.
- `grayscale` : Output a grayscale JPEG image. The `channels` dimension of `image` must be 1.
- `rgb` : Output an RGB JPEG image. The `channels` dimension of `image` must be 3.

If `format` is not specified or is the empty string, a default format is picked in function of the number of channels in `image`:

- 1: Output a grayscale image.

- 3: Output an RGB image.

**Args:**

- `image` : A `Tensor` of type `uint8`. 3-D with shape `[height, width, channels]`.
- `format` : An optional `string` from: `"", "grayscale", "rgb"`. Defaults to `""`. Per pixel image format.
- `quality` : An optional `int`. Defaults to `95`. Quality of the compression from 0 to 100 (higher is better and slower).
- `progressive` : An optional `bool`. Defaults to `False`. If True, create a JPEG that loads progressively (coarse to fine).
- `optimize_size` : An optional `bool`. Defaults to `False`. If True, spend CPU/RAM to reduce size with no quality change.
- `chroma_downsampling` : An optional `bool`. Defaults to `True`. See [http://en.wikipedia.org/wiki/Chroma\\_subsampling](http://en.wikipedia.org/wiki/Chroma_subsampling).
- `density_unit` : An optional `string` from: `"in", "cm"`. Defaults to `"in"`. Unit used to specify `x_density` and `y_density`: pixels per inch (`'in'`) or centimeter (`'cm'`).
- `x_density` : An optional `int`. Defaults to `300`. Horizontal pixels per density unit.
- `y_density` : An optional `int`. Defaults to `300`. Vertical pixels per density unit.
- `xmp_metadata` : An optional `string`. Defaults to `" "`. If not empty, embed this XMP metadata in the image header.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `string`. 0-D. JPEG-encoded image.

## `tf.image.decode_png(contents, channels=None, dtype=None, name=None)`

Decode a PNG-encoded image to a uint8 or uint16 tensor.

The attr `channels` indicates the desired number of color channels for the decoded image.

Accepted values are:

- 0: Use the number of channels in the PNG-encoded image.
- 1: output a grayscale image.
- 3: output an RGB image.
- 4: output an RGBA image.

If needed, the PNG-encoded image is transformed to match the requested number of color channels.

#### Args:

- `contents` : A `Tensor` of type `string`. 0-D. The PNG-encoded image.
- `channels` : An optional `int`. Defaults to `0`. Number of color channels for the decoded image.
- `dtype` : An optional `tf.DType` from: `tf.uint8, tf.uint16`. Defaults to `tf.uint8`.
- `name` : A name for the operation (optional).

#### Returns:

A `Tensor` of type `dtype`. 3-D with shape `[height, width, channels]`.

## `tf.image.encode_png(image, compression=None, name=None)`

PNG-encode an image.

`image` is a 3-D uint8 or uint16 Tensor of shape `[height, width, channels]` where `channels` is:

- 1: for grayscale.
- 2: for grayscale + alpha.
- 3: for RGB.
- 4: for RGBA.

The ZLIB compression level, `compression`, can be -1 for the PNG-encoder default or a value from 0 to 9. 9 is the highest compression level, generating the smallest output, but is slower.

#### Args:

- `image` : A `Tensor`. Must be one of the following types: `uint8, uint16`. 3-D with shape `[height, width, channels]`.
- `compression` : An optional `int`. Defaults to `-1`. Compression level.
- `name` : A name for the operation (optional).

#### Returns:

A `Tensor` of type `string`. 0-D. PNG-encoded image.

## Resizing

The resizing Ops accept input images as tensors of several types. They always output resized images as float32 tensors.

The convenience function `resize_images()` supports both 4-D and 3-D tensors as input and output. 4-D tensors are for batches of images, 3-D tensors for individual images.

Other resizing Ops only support 4-D batches of images as input: `resize_area`, `resize_bicubic`, `resize_bilinear`, `resize_nearest_neighbor`.

Example:

```
Decode a JPG image and resize it to 299 by 299 using default method.
image = tf.image.decode_jpeg(...)
resized_image = tf.image.resize_images(image, 299, 299)
```

## **tf.image.resize\_images(images, new\_height, new\_width, method=0, align\_corners=False)**

Resize `images` to `new_width`, `new_height` using the specified `method`.

Resized images will be distorted if their original aspect ratio is not the same as `new_width`, `new_height`. To avoid distortions see `resize_image_with_crop_or_pad`.

`method` can be one of:

- `ResizeMethod.BILINEAR` : [Bilinear interpolation.]  
([https://en.wikipedia.org/wiki/Bilinear\\_interpolation](https://en.wikipedia.org/wiki/Bilinear_interpolation))
- `ResizeMethod.NEAREST_NEIGHBOR` : [Nearest neighbor interpolation.]  
([https://en.wikipedia.org/wiki/Nearest-neighbor\\_interpolation](https://en.wikipedia.org/wiki/Nearest-neighbor_interpolation))
- `ResizeMethod.BICUBIC` : [Bicubic interpolation.]  
([https://en.wikipedia.org/wiki/Bicubic\\_interpolation](https://en.wikipedia.org/wiki/Bicubic_interpolation))
- `ResizeMethod.AREA` : Area interpolation.

### Args:

- `images` : 4-D Tensor of shape `[batch, height, width, channels]` or  
3-D Tensor of shape ``[height, width, channels]``.
- `new_height` : integer.
- `new_width` : integer.
- `method` : `ResizeMethod`. Defaults to `ResizeMethod.BILINEAR`.
- `align_corners` : bool. If true, exactly align all 4 corners of the input and

---

output. Defaults to `false`.

---

**Returns:**

- `ValueError` : if the shape of `images` is incompatible with the shape arguments to this function
- `ValueError` : if an unsupported resize method is specified.

**Returns:**

If `images` was 4-D, a 4-D float Tensor of shape `[batch, new_height, new_width, channels]` .  
 If `images` was 3-D, a 3-D float Tensor of shape `[new_height, new_width, channels]` .

---

**`tf.image.resize_area(images, size, align_corners=None, name=None)`**

Resize `images` to `size` using area interpolation.

Input images can be of different types but output images are always float.

**Args:**

- `images` : A `Tensor` . Must be one of the following types: `uint8` , `int8` , `int16` , `int32` , `int64` , `half` , `float32` , `float64` . 4-D with shape `[batch, height, width, channels]` .
- `size` : A 1-D `int32` Tensor of 2 elements: `new_height, new_width` . The new size for the images.
- `align_corners` : An optional `bool` . Defaults to `False` . If true, rescale input by  $(\text{new\_height} - 1) / (\text{height} - 1)$ , which exactly aligns the 4 corners of images and resized images. If false, rescale by `new_height / height`. Treat similarly the width dimension.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `float32` . 4-D with shape `[batch, new_height, new_width, channels]` .

---

**`tf.image.resize_bicubic(images, size, align_corners=None, name=None)`**

Resize `images` to `size` using bicubic interpolation.

Input images can be of different types but output images are always float.

**Args:**

- `images` : A `Tensor`. Must be one of the following types: `uint8`, `int8`, `int16`, `int32`, `int64`, `half`, `float32`, `float64`. 4-D with shape `[batch, height, width, channels]`.
- `size` : A 1-D `int32` Tensor of 2 elements: `new_height, new_width`. The new size for the images.
- `align_corners` : An optional `bool`. Defaults to `False`. If true, rescale input by  $(\text{new\_height} - 1) / (\text{height} - 1)$ , which exactly aligns the 4 corners of images and resized images. If false, rescale by `new_height / height`. Treat similarly the width dimension.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `float32`. 4-D with shape `[batch, new_height, new_width, channels]`.

**`tf.image.resize_bilinear(images, size, align_corners=None, name=None)`**

Resize `images` to `size` using bilinear interpolation.

Input images can be of different types but output images are always float.

**Args:**

- `images` : A `Tensor`. Must be one of the following types: `uint8`, `int8`, `int16`, `int32`, `int64`, `half`, `float32`, `float64`. 4-D with shape `[batch, height, width, channels]`.
- `size` : A 1-D `int32` Tensor of 2 elements: `new_height, new_width`. The new size for the images.
- `align_corners` : An optional `bool`. Defaults to `False`. If true, rescale input by  $(\text{new\_height} - 1) / (\text{height} - 1)$ , which exactly aligns the 4 corners of images and resized images. If false, rescale by `new_height / height`. Treat similarly the width dimension.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `float32`. 4-D with shape `[batch, new_height, new_width, channels]`.

**`tf.image.resize_nearest_neighbor(images, size, align_corners=None, name=None)`**

Resize `images` to `size` using nearest neighbor interpolation.

**Args:**

- `images` : A `Tensor`. Must be one of the following types: `uint8`, `int8`, `int16`, `int32`, `int64`, `half`, `float32`, `float64`. 4-D with shape `[batch, height, width, channels]`.
- `size` : A 1-D `int32` Tensor of 2 elements: `new_height, new_width`. The new size for the images.
- `align_corners` : An optional `bool`. Defaults to `False`. If true, rescale input by  $(\text{new\_height} - 1) / (\text{height} - 1)$ , which exactly aligns the 4 corners of images and resized images. If false, rescale by `new_height / height`. Treat similarly the width dimension.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor`. Has the same type as `images`. 4-D with shape `[batch, new_height, new_width, channels]`.

## Cropping

### `tf.image.resize_image_with_crop_or_pad(image, target_height, target_width)`

Crops and/or pads an image to a target width and height.

Resizes an image to a target width and height by either centrally cropping the image or padding it evenly with zeros.

If `width` or `height` is greater than the specified `target_width` or `target_height` respectively, this op centrally crops along that dimension. If `width` or `height` is smaller than the specified `target_width` or `target_height` respectively, this op centrally pads with 0 along that dimension.

**Args:**

- `image` : 3-D tensor of shape [height, width, channels]
- `target_height` : Target height.
- `target_width` : Target width.

**Raises:**

- `ValueError` : if `target_height` or `target_width` are zero or negative.

**Returns:**

Cropped and/or padded image of shape [target\_height, target\_width, channels]

## `tf.image.central_crop(image, central_fraction)`

Crop the central region of the image.

Remove the outer parts of an image but retain the central region of the image along each dimension. If we specify `central_fraction = 0.5`, this function returns the region marked with "X" in the below diagram.

```

| |
| xxxx |
| xxxx |
| | where "X" is the central 50% of the image.

```

### Args:

- `image` : 3-D float Tensor of shape [height, width, depth]
- `central_fraction` : float (0, 1], fraction of size to crop

### Raises:

- `ValueError` : if `central_crop_fraction` is not within (0, 1].

### Returns:

3-D float Tensor

## `tf.image.pad_to_bounding_box(image, offset_height, offset_width, target_height, target_width)`

Pad `image` with zeros to the specified `height` and `width`.

Adds `offset_height` rows of zeros on top, `offset_width` columns of zeros on the left, and then pads the image on the bottom and right with zeros until it has dimensions `target_height`, `target_width`.

This op does nothing if `offset_*` is zero and the image already has size `target_height` by `target_width`.

#### Args:

- `image` : 3-D tensor with shape `[height, width, channels]`
- `offset_height` : Number of rows of zeros to add on top.
- `offset_width` : Number of columns of zeros to add on the left.
- `target_height` : Height of output image.
- `target_width` : Width of output image.

#### Returns:

3-D tensor of shape `[target_height, target_width, channels]`

#### Raises:

- `ValueError` : If the shape of `image` is incompatible with the `offset_*` or `target_*` arguments

```
tf.image.crop_to_bounding_box(image,
 offset_height, offset_width, target_height,
 target_width)
```

Crops an image to a specified bounding box.

This op cuts a rectangular part out of `image`. The top-left corner of the returned image is at `offset_height, offset_width` in `image`, and its lower-right corner is at `offset_height + target_height, offset_width + target_width`.

#### Args:

- `image` : 3-D tensor with shape `[height, width, channels]`
- `offset_height` : Vertical coordinate of the top-left corner of the result in the input.
- `offset_width` : Horizontal coordinate of the top-left corner of the result in the input.
- `target_height` : Height of the result.
- `target_width` : Width of the result.

**Returns:**

3-D tensor of image with shape [target\_height, target\_width, channels]

**Raises:**

- `ValueError` : If the shape of `image` is incompatible with the `offset_*` or `target_*` arguments

---

```
tf.image.extract_glimpse(input, size, offsets,
centered=None, normalized=None,
uniform_noise=None, name=None)
```

Extracts a glimpse from the input tensor.

Returns a set of windows called glimpses extracted at location `offsets` from the input tensor. If the windows only partially overlaps the inputs, the non overlapping areas will be filled with random noise.

The result is a 4-D tensor of shape [batch\_size, glimpse\_height, glimpse\_width, channels] . The channels and batch dimensions are the same as that of the input tensor. The height and width of the output windows are specified in the `size` parameter.

The argument `normalized` and `centered` controls how the windows are

**Args:**

- `input` : A `Tensor` of type `float32` .
- `size` : A `Tensor` of type `int32` .
- `offsets` : A `Tensor` of type `float32` .
- `centered` : An optional `bool` . Defaults to `True` .
- `normalized` : An optional `bool` . Defaults to `True` .
- `uniform_noise` : An optional `bool` . Defaults to `True` .
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `float32` .

## Flipping and Transposing

---

## tf.image.flip\_up\_down(image)

Flip an image horizontally (upside down).

Outputs the contents of `image` flipped along the first dimension, which is `height`.

See also `reverse()`.

### Args:

- `image` : A 3-D tensor of shape `[height, width, channels]`.

### Returns:

A 3-D tensor of the same type and shape as `image`.

### Raises:

- `ValueError` : if the shape of `image` not supported.

---

## tf.image.random\_flip\_up\_down(image, seed=None)

Randomly flips an image vertically (upside down).

With a 1 in 2 chance, outputs the contents of `image` flipped along the first dimension, which is `height`. Otherwise output the image as-is.

### Args:

- `image` : A 3-D tensor of shape `[height, width, channels]`.
- `seed` : A Python integer. Used to create a random seed. See `set_random_seed` for behavior.

### Returns:

A 3-D tensor of the same type and shape as `image`.

### Raises:

- `ValueError` : if the shape of `image` not supported.

---

## tf.image.flip\_left\_right(image)

Flip an image horizontally (left to right).

Outputs the contents of `image` flipped along the second dimension, which is `width`.

See also `reverse()`.

#### Args:

- `image` : A 3-D tensor of shape `[height, width, channels]`.

#### Returns:

A 3-D tensor of the same type and shape as `image`.

#### Raises:

- `ValueError` : if the shape of `image` not supported.

---

## `tf.image.random_flip_left_right(image, seed=None)`

Randomly flip an image horizontally (left to right).

With a 1 in 2 chance, outputs the contents of `image` flipped along the second dimension, which is `width`. Otherwise output the image as-is.

#### Args:

- `image` : A 3-D tensor of shape `[height, width, channels]`.
- `seed` : A Python integer. Used to create a random seed. See `set_random_seed` for behavior.

#### Returns:

A 3-D tensor of the same type and shape as `image`.

#### Raises:

- `ValueError` : if the shape of `image` not supported.

---

## `tf.image.transpose_image(image)`

Transpose an image by swapping the first and second dimension.

See also `transpose()`.

#### Args:

- `image` : 3-D tensor of shape `[height, width, channels]`

#### Returns:

A 3-D tensor of shape `[width, height, channels]`

#### Raises:

- `ValueError` : if the shape of `image` not supported.

## Converting Between Colorscales.

Image ops work either on individual images or on batches of images, depending on the shape of their input Tensor.

If 3-D, the shape is `[height, width, channels]`, and the Tensor represents one image. If 4-D, the shape is `[batch_size, height, width, channels]`, and the Tensor represents `batch_size` images.

Currently, `channels` can usefully be 1, 2, 3, or 4. Single-channel images are grayscale, images with 3 channels are encoded as either RGB or HSV. Images with 2 or 4 channels include an alpha channel, which has to be stripped from the image before passing the image to most image processing functions (and can be re-attached later).

Internally, images are either stored in as one `float32` per channel per pixel (implicitly, values are assumed to lie in `[0, 1]`) or one `uint8` per channel per pixel (values are assumed to lie in `[0, 255]`).

Tensorflow can convert between images in RGB or HSV. The conversion functions work only on float images, so you need to convert images in other formats using `convert_image_dtype`.

Example:

```
Decode an image and convert it to HSV.
rgb_image = tf.image.decode_png(..., channels=3)
rgb_image_float = tf.image.convert_image_dtype(rgb_image, tf.float32)
hsv_image = tf.image.rgb_to_hsv(rgb_image)
```

**`tf.image.rgb_to_grayscale(images, name=None)`**

Converts one or more images from RGB to Grayscale.

Outputs a tensor of the same `dtype` and rank as `images`. The size of the last dimension of the output is 1, containing the Grayscale value of the pixels.

#### Args:

- `images` : The RGB tensor to convert. Last dimension must have size 3 and should contain RGB values.
- `name` : A name for the operation (optional).

#### Returns:

The converted grayscale image(s).

---

## `tf.image.grayscale_to_rgb(images, name=None)`

Converts one or more images from Grayscale to RGB.

Outputs a tensor of the same `dtype` and rank as `images`. The size of the last dimension of the output is 3, containing the RGB value of the pixels.

#### Args:

- `images` : The Grayscale tensor to convert. Last dimension must be size 1.
- `name` : A name for the operation (optional).

#### Returns:

The converted grayscale image(s).

---

## `tf.image.hsv_to_rgb(images, name=None)`

Convert one or more images from HSV to RGB.

Outputs a tensor of the same shape as the `images` tensor, containing the RGB value of the pixels. The output is only well defined if the value in `images` are in  $[0, 1]$ .

See `rgb_to_hsv` for a description of the HSV encoding.

#### Args:

- `images` : A `Tensor` of type `float32`. 1-D or higher rank. HSV data to convert. Last dimension must be size 3.

- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `float32`. `images` converted to RGB.

## **`tf.image.rgb_to_hsv(images, name=None)`**

Converts one or more images from RGB to HSV.

Outputs a tensor of the same shape as the `images` tensor, containing the HSV value of the pixels. The output is only well defined if the value in `images` are in `[0, 1]`.

`output[..., 0]` contains hue, `output[..., 1]` contains saturation, and `output[..., 2]` contains value. All HSV values are in `[0, 1]`. A hue of 0 corresponds to pure red, hue 1/3 is pure green, and 2/3 is pure blue.

**Args:**

- `images` : A `Tensor` of type `float32`. 1-D or higher rank. RGB data to convert. Last dimension must be size 3.
- `name` : A name for the operation (optional).

**Returns:**

A `Tensor` of type `float32`. `images` converted to HSV.

## **`tf.image.convert_image_dtype(image, dtype, saturate=False, name=None)`**

Convert `image` to `dtype`, scaling its values if needed.

Images that are represented using floating point values are expected to have values in the range [0,1). Image data stored in integer data types are expected to have values in the range `[0, MAX]`, where `MAX` is the largest positive representable number for the data type.

This op converts between data types, scaling the values appropriately before casting.

Note that converting from floating point inputs to integer types may lead to over/underflow problems. Set `saturate` to `True` to avoid such problem in problematic conversions. If enabled, saturation will clip the output into the allowed range before performing a potentially

dangerous cast (and only before performing such a cast, i.e., when casting from a floating point to an integer type, and when casting from a signed to an unsigned type; `saturate` has no effect on casts between floats, or on casts that increase the type's range).

#### Args:

- `image` : An image.
- `dtype` : A `DTType` to convert `image` to.
- `saturate` : If `True`, clip the input before casting (if necessary).
- `name` : A name for this operation (optional).

#### Returns:

`image`, converted to `dtype`.

## Image Adjustments

TensorFlow provides functions to adjust images in various ways: brightness, contrast, hue, and saturation. Each adjustment can be done with predefined parameters or with random parameters picked from predefined intervals. Random adjustments are often useful to expand a training set and reduce overfitting.

If several adjustments are chained it is advisable to minimize the number of redundant conversions by first converting the images to the most natural data type and representation (RGB or HSV).

---

### `tf.image.adjust_brightness(image, delta)`

Adjust the brightness of RGB or Grayscale images.

This is a convenience method that converts an RGB image to float representation, adjusts its brightness, and then converts it back to the original data type. If several adjustments are chained it is advisable to minimize the number of redundant conversions.

The value `delta` is added to all components of the tensor `image`. Both `image` and `delta` are converted to `float` before adding (and `image` is scaled appropriately if it is in fixed-point representation). For regular images, `delta` should be in the range `[0, 1)`, as it is added to the image in floating point representation, where pixel values are in the `[0, 1)` range.

#### Args:

- `image` : A tensor.
- `delta` : A scalar. Amount to add to the pixel values.

**Returns:**

A brightness-adjusted tensor of the same shape and type as `image`.

## **`tf.image.random_brightness(image, max_delta, seed=None)`**

Adjust the brightness of images by a random factor.

Equivalent to `adjust_brightness()` using a `delta` randomly picked in the interval `[-max_delta, max_delta]`.

**Args:**

- `image` : An image.
- `max_delta` : float, must be non-negative.
- `seed` : A Python integer. Used to create a random seed. See [set\\_random\\_seed](#) for behavior.

**Returns:**

The brightness-adjusted image.

**Raises:**

- `ValueError` : if `max_delta` is negative.

## **`tf.image.adjust_contrast(images, contrast_factor)`**

Adjust contrast of RGB or grayscale images.

This is a convenience method that converts an RGB image to float representation, adjusts its contrast, and then converts it back to the original data type. If several adjustments are chained it is advisable to minimize the number of redundant conversions.

`images` is a tensor of at least 3 dimensions. The last 3 dimensions are interpreted as `[height, width, channels]`. The other dimensions only represent a collection of images, such as `[batch, height, width, channels]`.

Contrast is adjusted independently for each channel of each image.

For each channel, this Op computes the mean of the image pixels in the channel and then adjusts each component  $x$  of each pixel to  $(x - \text{mean}) * \text{contrast\_factor} + \text{mean}$ .

#### Args:

- `images` : Images to adjust. At least 3-D.
- `contrast_factor` : A float multiplier for adjusting contrast.

#### Returns:

The contrast-adjusted image or images.

---

## `tf.image.random_contrast(image, lower, upper, seed=None)`

Adjust the contrast of an image by a random factor.

Equivalent to `adjust_contrast()` but uses a `contrast_factor` randomly picked in the interval `[lower, upper]`.

#### Args:

- `image` : An image tensor with 3 or more dimensions.
- `lower` : float. Lower bound for the random contrast factor.
- `upper` : float. Upper bound for the random contrast factor.
- `seed` : A Python integer. Used to create a random seed. See `set_random_seed` for behavior.

#### Returns:

The contrast-adjusted tensor.

#### Raises:

- `ValueError` : if `upper <= lower` or if `lower < 0`.

---

## `tf.image.adjust_hue(image, delta, name=None)`

Adjust hue of an RGB image.

This is a convenience method that converts an RGB image to float representation, converts it to HSV, add an offset to the hue channel, converts back to RGB and then back to the original data type. If several adjustments are chained it is advisable to minimize the number of redundant conversions.

`image` is an RGB image. The image hue is adjusted by converting the image to HSV and rotating the hue channel (H) by `delta`. The image is then converted back to RGB.

`delta` must be in the interval `[-1, 1]`.

#### Args:

- `image` : RGB image or images. Size of the last dimension must be 3.
- `delta` : float. How much to add to the hue channel.
- `name` : A name for this operation (optional).

#### Returns:

Adjusted image(s), same shape and Dtype as `image`.

---

## `tf.image.random_hue(image, max_delta, seed=None)`

Adjust the hue of an RGB image by a random factor.

Equivalent to `adjust_hue()` but uses a `delta` randomly picked in the interval `[-max_delta, max_delta]`.

`max_delta` must be in the interval `[0, 0.5]`.

#### Args:

- `image` : RGB image or images. Size of the last dimension must be 3.
- `max_delta` : float. Maximum value for the random delta.
- `seed` : An operation-specific seed. It will be used in conjunction with the graph-level seed to determine the real seeds that will be used in this operation. Please see the documentation of `set_random_seed` for its interaction with the graph-level random seed.

#### Returns:

3-D float tensor of shape `[height, width, channels]`.

#### Raises:

- `ValueError` : if `max_delta` is invalid.

---

## `tf.image.adjust_saturation(image, saturation_factor, name=None)`

Adjust saturation of an RGB image.

This is a convenience method that converts an RGB image to float representation, converts it to HSV, add an offset to the saturation channel, converts back to RGB and then back to the original data type. If several adjustments are chained it is advisable to minimize the number of redundant conversions.

`image` is an RGB image. The image saturation is adjusted by converting the image to HSV and multiplying the saturation (S) channel by `saturation_factor` and clipping. The image is then converted back to RGB.

### Args:

- `image` : RGB image or images. Size of the last dimension must be 3.
- `saturation_factor` : float. Factor to multiply the saturation by.
- `name` : A name for this operation (optional).

### Returns:

Adjusted image(s), same shape and DType as `image`.

---

## `tf.image.random_saturation(image, lower, upper, seed=None)`

Adjust the saturation of an RGB image by a random factor.

Equivalent to `adjust_saturation()` but uses a `saturation_factor` randomly picked in the interval `[lower, upper]`.

### Args:

- `image` : RGB image or images. Size of the last dimension must be 3.
- `lower` : float. Lower bound for the random saturation factor.
- `upper` : float. Upper bound for the random saturation factor.
- `seed` : An operation-specific seed. It will be used in conjunction with the graph-level seed to determine the real seeds that will be used in this operation. Please see the

documentation of `set_random_seed` for its interaction with the graph-level random seed.

#### Returns:

Adjusted image(s), same shape and DTyoe as `image`.

#### Raises:

- `ValueError` : if `upper <= lower` or if `lower < 0`.

## **`tf.image.per_image_whitening(image)`**

Linearly scales `image` to have zero mean and unit norm.

This op computes `(x - mean) / adjusted_stddev`, where `mean` is the average of all values in `image`, and `adjusted_stddev = max(stddev, 1.0/sqrt(image.NumElements()))`.

`stddev` is the standard deviation of all values in `image`. It is capped away from zero to protect against division by 0 when handling uniform images.

Note that this implementation is limited:

- It only whitens based on the statistics of an individual image.
- It does not take into account the covariance structure.

#### Args:

- `image` : 3-D tensor of shape `[height, width, channels]`.

#### Returns:

The whitened image with same shape as `image`.

#### Raises:

- `ValueError` : if the shape of 'image' is incompatible with this function.

## **Working with Bounding Boxes**

## **`tf.image.draw_bounding_boxes(images, boxes, name=None)`**

Draw bounding boxes on a batch of images.

Outputs a copy of `images` but draws on top of the pixels zero or more bounding boxes specified by the locations in `boxes`. The coordinates of the each bounding box in `boxes` are encoded as `[y_min, x_min, y_max, x_max]`. The bounding box coordinates are floats in `[0.0, 1.0]` relative to the width and height of the underlying image.

For example, if an image is  $100 \times 200$  pixels and the bounding box is `[0.1, 0.5, 0.2, 0.9]`, the bottom-left and upper-right coordinates of the bounding box will be `(10, 40)` to `(50, 180)`.

Parts of the bounding box may fall outside the image.

#### Args:

- `images` : A `Tensor`. Must be one of the following types: `float32`, `half`. 4-D with shape `[batch, height, width, depth]`. A batch of images.
- `boxes` : A `Tensor` of type `float32`. 3-D with shape `[batch, num_bounding_boxes, 4]` containing bounding boxes.
- `name` : A name for the operation (optional).

#### Returns:

A `Tensor`. Has the same type as `images`. 4-D with the same shape as `images`. The batch of input images with bounding boxes drawn on the images.

```
tf.image.sample_distorted_bounding_box(image_size,
 bounding_boxes,
 seed=None,
 seed2=None,
 min_object_covered=None,
 aspect_ratio_range=None,
 area_range=None,
 max_attempts=None,
 use_image_if_no_bounding_boxes=None,
 name=None)
```

Generate a single randomly distorted bounding box for an image.

Bounding box annotations are often supplied in addition to ground-truth labels in image recognition or object localization tasks. A common technique for training such a system is to randomly distort an image while preserving its content, i.e. *data augmentation*. This Op outputs a randomly distorted localization of an object, i.e. bounding box, given an `image_size`, `bounding_boxes` and a series of constraints.

The output of this Op is a single bounding box that may be used to crop the original image. The output is returned as 3 tensors: `begin`, `size` and `bboxes`. The first 2 tensors can be fed directly into `tf.slice` to crop the image. The latter may be supplied to `tf.image.draw_bounding_box` to visualize what the bounding box looks like.

Bounding boxes are supplied and returned as `[y_min, x_min, y_max, x_max]`. The bounding box coordinates are floats in `[0.0, 1.0]` relative to the width and height of the underlying image.

For example,

```
Generate a single distorted bounding box.
begin, size, bbox_for_draw = tf.image.sample_distorted_bounding_box(
 tf.shape(image),
 bounding_boxes=bounding_boxes)

Draw the bounding box in an image summary.
image_with_box = tf.image.draw_bounding_boxes(tf.expand_dims(image, 0),
 bbox_for_draw)
tf.image_summary('images_with_box', image_with_box)

Employ the bounding box to distort the image.
distorted_image = tf.slice(image, begin, size)
```

Note that if no bounding box information is available, setting `use_image_if_no_bounding_boxes = true` will assume there is a single implicit bounding box covering the whole image. If `use_image_if_no_bounding_boxes` is false and no bounding boxes are supplied, an error is raised.

### Args:

- `image_size` : A `Tensor`. Must be one of the following types: `uint8`, `int8`, `int16`, `int32`, `int64`. 1-D, containing `[height, width, channels]`.
- `bounding_boxes` : A `Tensor` of type `float32`. 3-D with shape `[batch, N, 4]` describing the N bounding boxes associated with the image.
- `seed` : An optional `int`. Defaults to `0`. If either `seed` or `seed2` are set to non-zero, the random number generator is seeded by the given `seed`. Otherwise, it is seeded by a random seed.
- `seed2` : An optional `int`. Defaults to `0`. A second seed to avoid seed collision.
- `min_object_covered` : An optional `float`. Defaults to `0.1`. The cropped area of the image must contain at least this fraction of any bounding box supplied.
- `aspect_ratio_range` : An optional list of `floats`. Defaults to `[0.75, 1.33]`. The cropped area of the image must have an aspect ratio = width / height within this range.
- `area_range` : An optional list of `floats`. Defaults to `[0.05, 1]`. The cropped area of

the image must contain a fraction of the supplied image within in this range.

- `max_attempts` : An optional `int`. Defaults to `100`. Number of attempts at generating a cropped region of the image of the specified constraints. After `max_attempts` failures, return the entire image.
- `use_image_if_no_bounding_boxes` : An optional `bool`. Defaults to `False`. Controls behavior if no bounding boxes supplied. If true, assume an implicit bounding box covering the whole input. If false, raise an error.
- `name` : A name for the operation (optional).

### Returns:

A tuple of `Tensor` objects (`begin`, `size`, `bboxes`).

- `begin` : A `Tensor`. Has the same type as `image_size`. 1-D, containing `[offset_height, offset_width, 0]`. Provide as input to `tf.slice`.
- `size` : A `Tensor`. Has the same type as `image_size`. 1-D, containing `[target_height, target_width, -1]`. Provide as input to `tf.slice`.
- `bboxes` : A `Tensor` of type `float32`. 3-D with shape `[1, 1, 4]` containing the distorted bounding box. Provide as input to `tf.image.draw_bounding_boxes`.

# Sparse Tensors

Note: Functions taking `Tensor` arguments can also take anything accepted by `tf.convert_to_tensor`.

[TOC]

## Sparse Tensor Representation

Tensorflow supports a `SparseTensor` representation for data that is sparse in multiple dimensions. Contrast this representation with `IndexedSlices`, which is efficient for representing tensors that are sparse in their first dimension, and dense along all other dimensions.

### `class tf.SparseTensor`

Represents a sparse tensor.

Tensorflow represents a sparse tensor as three separate dense tensors: `indices`, `values`, and `shape`. In Python, the three tensors are collected into a `SparseTensor` class for ease of use. If you have separate `indices`, `values`, and `shape` tensors, wrap them in a `SparseTensor` object before passing to the ops below.

Concretely, the sparse tensor `SparseTensor(indices, values, shape)` is

- `indices` : A 2-D int64 tensor of shape `[N, ndims]` .
- `values` : A 1-D tensor of any type and shape `[N]` .
- `shape` : A 1-D int64 tensor of shape `[ndims]` .

where `N` and `ndims` are the number of values, and number of dimensions in the `SparseTensor` respectively.

The corresponding dense tensor satisfies

```
dense.shape = shape
dense[tuple(indices[i])] = values[i]
```

By convention, `indices` should be sorted in row-major order (or equivalently lexicographic order on the tuples `indices[i]`). This is not enforced when `sparseTensor` objects are constructed, but most ops assume correct ordering. If the ordering of sparse tensor `st` is wrong, a fixed version can be obtained by calling `tf.sparse_reorder(st)`.

Example: The sparse tensor

```
SparseTensor(indices=[[0, 0], [1, 2]], values=[1, 2], shape=[3, 4])
```

represents the dense tensor

```
[[1, 0, 0, 0]
 [0, 0, 2, 0]
 [0, 0, 0, 0]]
```

## `tf.SparseTensor.__init__(indices, values, shape)`

Creates a `sparseTensor`.

**Args:**

- `indices` : A 2-D int64 tensor of shape `[N, ndims]`.
- `values` : A 1-D tensor of any type and shape `[N]`.
- `shape` : A 1-D int64 tensor of shape `[ndims]`.

**Returns:**

A `sparseTensor`

## `tf.SparseTensor.indices`

The indices of non-zero values in the represented dense tensor.

**Returns:**

A 2-D Tensor of int64 with shape `[N, ndims]`, where `N` is the number of non-zero values in the tensor, and `ndims` is the rank.

## `tf.SparseTensor.values`

The non-zero values in the represented dense tensor.

**Returns:**

A 1-D Tensor of any data type.

---

**tf.SparseTensor.shape**

A 1-D Tensor of int64 representing the shape of the dense tensor.

---

**tf.SparseTensor.dtype**

The `dtype` of elements in this tensor.

---

**tf.SparseTensor.op**

The `Operation` that produces `values` as an output.

---

**tf.SparseTensor.graph**

The `Graph` that contains the index, value, and shape tensors.

## Other Methods

---

**tf.SparseTensor.eval(feed\_dict=None, session=None)**

Evaluates this sparse tensor in a `Session`.

Calling this method will execute all preceding operations that produce the inputs needed for the operation that produces this tensor.

*N.B.* Before invoking `SparseTensor.eval()`, its graph must have been launched in a session, and either a default session must be available, or `session` must be specified explicitly.

**Args:**

- `feed_dict` : A dictionary that maps `Tensor` objects to feed values. See `Session.run()` for a description of the valid feed values.

- `session` : (Optional.) The `Session` to be used to evaluate this sparse tensor. If none, the default session will be used.

**Returns:**

A `SparseTensorValue` object.

---

**`tf.SparseTensor.from_value(cls, sparse_tensor_value)`**

---

**`class tf.SparseTensorValue`**

`SparseTensorValue(indices, values, shape)`

---

**`tf.SparseTensorValue.indices`**

Alias for field number 0

---

**`tf.SparseTensorValue.shape`**

Alias for field number 2

---

**`tf.SparseTensorValue.values`**

Alias for field number 1

## Conversion

---

**`tf.sparse_to_dense(sparse_indices,`**  
**`output_shape, sparse_values, default_value=0,`**  
**`validate_indices=True, name=None)`**

Converts a sparse representation into a dense tensor.

Builds an array `dense` with shape `output_shape` such that

```

If sparse_indices is scalar
dense[i] = (i == sparse_indices ? sparse_values : default_value)

If sparse_indices is a vector, then for each i
dense[sparse_indices[i]] = sparse_values[i]

If sparse_indices is an n by d matrix, then for each i in [0, n)
dense[sparse_indices[i][0], ..., sparse_indices[i][d-1]] = sparse_values[i]

```

All other values in `dense` are set to `default_value`. If `sparse_values` is a scalar, all sparse indices are set to this single value.

Indices should be sorted in lexicographic order, and indices must not contain any repeats. If `validate_indices` is True, these properties are checked during execution.

### Args:

- `sparse_indices` : A 0-D, 1-D, or 2-D `Tensor` of type `int32` or `int64`.  
`sparse_indices[i]` contains the complete index where `sparse_values[i]` will be placed.
- `output_shape` : A 1-D `Tensor` of the same type as `sparse_indices`. Shape of the dense output tensor.
- `sparse_values` : A 0-D or 1-D `Tensor`. Values corresponding to each row of `sparse_indices`, or a scalar value to be used for all sparse indices.
- `default_value` : A 0-D `Tensor` of the same type as `sparse_values`. Value to set for indices not specified in `sparse_indices`. Defaults to zero.
- `validate_indices` : A boolean value. If True, indices are checked to make sure they are sorted in lexicographic order and that there are no repeats.
- `name` : A name for the operation (optional).

### Returns:

Dense `Tensor` of shape `output_shape`. Has the same type as `sparse_values`.

---

**`tf.sparse_tensor_to_dense(sp_input, default_value=0, validate_indices=True, name=None)`**

Converts a `SparseTensor` into a dense tensor.

This op is a convenience wrapper around `sparse_to_dense` for `SparseTensor`s.

For example, if `sp_input` has shape `[3, 5]` and non-empty string values:

```
[0, 1]: a
[0, 3]: b
[2, 0]: c
```

and `default_value` is `x`, then the output will be a dense `[3, 5]` string tensor with values:

```
[[x a x b x]
 [x x x x x]
 [c x x x x]]
```

Indices must be without repeats. This is only tested if `validate_indices` is True.

### Args:

- `sp_input` : The input `SparseTensor`.
- `default_value` : Scalar value to set for indices not specified in `sp_input`. Defaults to zero.
- `validate_indices` : A boolean value. If `True`, indices are checked to make sure they are sorted in lexicographic order and that there are no repeats.
- `name` : A name prefix for the returned tensors (optional).

### Returns:

A dense tensor with shape `sp_input.shape` and values specified by the non-empty values in `sp_input`. Indices not in `sp_input` are assigned `default_value`.

### Raises:

- `TypeError` : If `sp_input` is not a `SparseTensor`.

## `tf.sparse_to_indicator(sp_input, vocab_size, name=None)`

Converts a `SparseTensor` of ids into a dense bool indicator tensor.

The last dimension of `sp_input.indices` is discarded and replaced with the values of `sp_input`. If `sp_input.shape = [D0, D1, ..., Dn, K]`, then `output.shape = [D0, D1, ..., Dn, vocab_size]`, where

```
output[d_0, d_1, ..., d_n, sp_input[d_0, d_1, ..., d_n, k]] = True
```

and False elsewhere in `output`.

For example, if `sp_input.shape = [2, 3, 4]` with non-empty values:

```
[0, 0, 0]: 0
[0, 1, 0]: 10
[1, 0, 3]: 103
[1, 1, 2]: 150
[1, 1, 3]: 149
[1, 1, 4]: 150
[1, 2, 1]: 121
```

and `vocab_size = 200`, then the output will be a `[2, 3, 200]` dense bool tensor with False everywhere except at positions

```
(0, 0, 0), (0, 1, 10), (1, 0, 103), (1, 1, 149), (1, 1, 150),
(1, 2, 121).
```

Note that repeats are allowed in the input SparseTensor. This op is useful for converting `SparseTensor`s into dense formats for compatibility with ops that expect dense tensors.

The input `SparseTensor` must be in row-major order.

#### Args:

- `sp_input` : A `SparseTensor` with `values` property of type `int32` or `int64`.
- `vocab_size` : A scalar `int64` Tensor (or Python int) containing the new size of the last dimension, `all(0 <= sp_input.values < vocab_size)`.
- `name` : A name prefix for the returned tensors (optional)

#### Returns:

A dense bool indicator tensor representing the indices with specified value.

#### Raises:

- `TypeError` : If `sp_input` is not a `SparseTensor`.

---

## `tf.sparse_merge(sp_ids, sp_values, vocab_size, name=None)`

Combines a batch of feature ids and values into a single `SparseTensor`.

The most common use case for this function occurs when feature ids and their corresponding values are stored in `Example` protos on disk. `parse_example` will return a batch of ids and a batch of values, and this function joins them into a single logical

`SparseTensor` for use in functions such as `sparse_tensor_dense_matmul`, `sparse_to_dense`, etc.

The `SparseTensor` returned by this function has the following properties:

- `indices` is equivalent to `sp_ids.indices` with the last dimension discarded and replaced with `sp_ids.values`.
- `values` is simply `sp_values.values`.
- If `sp_ids.shape = [D0, D1, ..., Dn, K]`, then `output.shape = [D0, D1, ..., Dn, vocab_size]`.

For example, consider the following feature vectors:

`vector1 = [-3, 0, 0, 0, 0, 0]` `vector2 = [0, 1, 0, 4, 1, 0]` `vector3 = [5, 0, 0, 9, 0, 0]`

These might be stored sparsely in the following Example protos by storing only the feature ids (column number if the vectors are treated as a matrix) of the non-zero elements and the corresponding values:

```
examples = [Example(features={"ids": Feature(int64_list=Int64List(value=[0])), "values": Feature(float_list=FloatList(value=[-3]))}), Example(features={"ids": Feature(int64_list=Int64List(value=[1, 4, 3])), "values": Feature(float_list=FloatList(value=[1, 1, 4]))}), Example(features={"ids": Feature(int64_list=Int64List(value=[0, 3])), "values": Feature(float_list=FloatList(value=[5, 9]))})]
```

The result of calling `parse_example` on these examples will produce a dictionary with entries for "ids" and "values". Passing those two objects to this function along with `vocab_size=6`, will produce a `SparseTensor` that sparsely represents all three instances. Namely, the `indices` property will contain the coordinates of the non-zero entries in the feature matrix (the first dimension is the row number in the matrix, i.e., the index within the batch, and the second dimension is the column number, i.e., the feature id); `values` will contain the actual values. `shape` will be the shape of the original matrix, i.e., `(3, 6)`. For our example above, the output will be equal to:

```
SparseTensor(indices=[[0, 0], [1, 1], [1, 3], [1, 4], [2, 0], [2, 3]], values=[-3, 1, 4, 1, 5, 9], shape=[3, 6])
```

## Args:

- `sp_ids` : A `SparseTensor` with `values` property of type `int32` or `int64`.
- `sp_values` : A `SparseTensor` of any type.
- `vocab_size` : A scalar `int64` Tensor (or Python int) containing the new size of the last dimension, `all(0 <= sp_ids.values < vocab_size)`.
- `name` : A name prefix for the returned tensors (optional)

**Returns:**

A `sparseTensor` compactly representing a batch of feature ids and values, useful for passing to functions that expect such a `SparseTensor`.

**Raises:**

- `TypeError` : If `sp_ids` or `sp_values` are not a `SparseTensor`.

## Manipulation

---

### `tf.sparse.concat(concat_dim, sp_inputs, name=None, expand_nonconcat_dim=False)`

Concatenates a list of `SparseTensor` along the specified dimension.

Concatenation is with respect to the dense versions of each sparse input. It is assumed that each inputs is a `SparseTensor` whose elements are ordered along increasing dimension number.

If `expand_nonconcat_dim` is `False`, all inputs' shapes must match, except for the concat dimension. If `expand_nonconcat_dim` is `True`, then inputs' shapes are allowed to vary among all inputs.

The `indices`, `values`, and `shapes` lists must have the same length.

If `expand_nonconcat_dim` is `False`, then the output shape is identical to the inputs', except along the concat dimension, where it is the sum of the inputs' sizes along that dimension.

If `expand_nonconcat_dim` is `True`, then the output shape along the non-concat dimensions will be expanded to be the largest among all inputs, and it is the sum of the inputs' sizes along the concat dimension.

The output elements will be resorted to preserve the sort order along increasing dimension number.

This op runs in  $O(M \log M)$  time, where  $M$  is the total number of non-empty values across all inputs. This is due to the need for an internal sort in order to concatenate efficiently across an arbitrary dimension.

For example, if `concat_dim = 1` and the inputs are

```
sp_inputs[0]: shape = [2, 3]
[0, 2]: "a"
[1, 0]: "b"
[1, 1]: "c"

sp_inputs[1]: shape = [2, 4]
[0, 1]: "d"
[0, 2]: "e"
```

then the output will be

```
shape = [2, 7]
[0, 2]: "a"
[0, 4]: "d"
[0, 5]: "e"
[1, 0]: "b"
[1, 1]: "c"
```

Graphically this is equivalent to doing

$$\begin{bmatrix} \quad a \\ b \ c \end{bmatrix} \text{ concat } \begin{bmatrix} \quad d \ e \\ \quad \quad \end{bmatrix} = \begin{bmatrix} \quad a \quad d \ e \\ b \ c \quad \quad \end{bmatrix}$$

Another example, if 'concat\_dim = 1' and the inputs are

```
sp_inputs[0]: shape = [3, 3]
[0, 2]: "a"
[1, 0]: "b"
[2, 1]: "c"

sp_inputs[1]: shape = [2, 4]
[0, 1]: "d"
[0, 2]: "e"
```

if `expand_nonconcat_dim = False`, this will result in an error. But if `expand_nonconcat_dim = True`, this will result in:

```
shape = [3, 7]
[0, 2]: "a"
[0, 4]: "d"
[0, 5]: "e"
[1, 0]: "b"
[2, 1]: "c"
```

Graphically this is equivalent to doing

```
[a] concat [d e] = [a d e]
[b] [] [b
[c] [] [c]
```

**Args:**

- `concat_dim` : Dimension to concatenate along.
- `sp_inputs` : List of `SparseTensor` to concatenate.
- `name` : A name prefix for the returned tensors (optional).
- `expand_nonconcat_dim` : Whether to allow the expansion in the non-concat dimensions. Defaulted to False.

**Returns:**

A `SparseTensor` with the concatenated output.

**Raises:**

- `TypeError` : If `sp_inputs` is not a list of `SparseTensor` .

**`tf.sparse_reorder(sp_input, name=None)`**

Reorders a `SparseTensor` into the canonical, row-major ordering.

Note that by convention, all sparse ops preserve the canonical ordering along increasing dimension number. The only time ordering can be violated is during manual manipulation of the indices and values to add entries.

Reordering does not affect the shape of the `SparseTensor` .

For example, if `sp_input` has shape `[4, 5]` and `indices / values` :

```
[0, 3]: b
[0, 1]: a
[3, 1]: d
[2, 0]: c
```

then the output will be a `SparseTensor` of shape `[4, 5]` and `indices / values` :

```
[0, 1]: a
[0, 3]: b
[2, 0]: c
[3, 1]: d
```

**Args:**

- `sp_input` : The input `SparseTensor` .
- `name` : A name prefix for the returned tensors (optional)

**Returns:**

A `SparseTensor` with the same shape and non-empty values, but in canonical ordering.

**Raises:**

- `TypeError` : If `sp_input` is not a `SparseTensor` .

## `tf.sparse_split(split_dim, num_split, sp_input, name=None)`

Split a `SparseTensor` into `num_split` tensors along `split_dim` .

If the `sp_input.shape[split_dim]` is not an integer multiple of `num_split` each slice starting from 0: `shape[split_dim] % num_split` gets extra one dimension. For example, if `split_dim = 1` and `num_split = 2` and the input is:

```
input_tensor = shape = [2, 7]
[a d e]
[b c]
```

Graphically the output tensors are:

```
output_tensor[0] =
[a]
[b c]

output_tensor[1] =
[d e]
[]
```

**Args:**

- `split_dim` : A 0-D `int32` `Tensor` . The dimension along which to split.
- `num_split` : A Python integer. The number of ways to split.
- `sp_input` : The `SparseTensor` to split.
- `name` : A name for the operation (optional).

**Returns:**

---

`num_split` `SparseTensor` objects resulting from splitting `value`.

**Raises:**

- `TypeError` : If `sp_input` is not a `SparseTensor`.
- 

**`tf.sparse_retain(sp_input, to_retain)`**

Retains specified non-empty values within a `SparseTensor`.

For example, if `sp_input` has shape `[4, 5]` and 4 non-empty string values:

```
[0, 1]: a
[0, 3]: b
[2, 0]: c
[3, 1]: d
```

and `to_retain = [True, False, False, True]`, then the output will be a `SparseTensor` of shape `[4, 5]` with 2 non-empty values:

```
[0, 1]: a
[3, 1]: d
```

**Args:**

- `sp_input` : The input `SparseTensor` with `N` non-empty elements.
- `to_retain` : A bool vector of length `N` with `M` true values.

**Returns:**

A `SparseTensor` with the same shape as the input and `M` non-empty elements corresponding to the true positions in `to_retain`.

**Raises:**

- `TypeError` : If `sp_input` is not a `SparseTensor`.
- 

**`tf.sparse_reset_shape(sp_input, new_shape=None)`**

Resets the shape of a `SparseTensor` with indices and values unchanged.

If `new_shape` is `None`, returns a copy of `sp_input` with its shape reset to the tight bounding box of `sp_input`.

If `new_shape` is provided, then it must be larger or equal in all dimensions compared to the shape of `sp_input`. When this condition is met, the returned `SparseTensor` will have its shape reset to `new_shape` and its indices and values unchanged from that of `sp_input`.

For example:

Consider a `sp_input` with shape [2, 3, 5]:

```
[0, 0, 1]: a
[0, 1, 0]: b
[0, 2, 2]: c
[1, 0, 3]: d
```

- It is an error to set `new_shape` as [3, 7] since this represents a rank-2 tensor while `sp_input` is rank-3. This is either a `ValueError` during graph construction (if both shapes are known) or an `OpError` during run time.
- Setting `new_shape` as [2, 3, 6] will be fine as this shape is larger or equal in every dimension compared to the original shape [2, 3, 5].
- On the other hand, setting `new_shape` as [2, 3, 4] is also an error: The third dimension is smaller than the original shape [2, 3, 5] (and an `InvalidArgumentError` will be raised).
- If `new_shape` is `None`, the returned `SparseTensor` will have a shape [2, 3, 4], which is the tight bounding box of `sp_input`.

### Args:

- `sp_input` : The input `SparseTensor`.
- `new_shape` : `None` or a vector representing the new shape for the returned `SpraseTensor`.

### Returns:

A `sparseTensor` indices and values unchanged from `input_sp`. Its shape is `new_shape` if that is set. Otherwise it is the tight bounding box of `input_sp`

### Raises:

- `TypeError` : If `sp_input` is not a `SparseTensor`.
- `ValueError` : If `new_shape` represents a tensor with a different rank from that of `sp_input` (if shapes are known when graph is constructed).
- `OpError` :

- If `new_shape` has dimension sizes that are too small.
- If shapes are not known during graph construction time, and during run time it is found out that the ranks do not match.

## **`tf.sparse_fill_empty_rows(sp_input, default_value, name=None)`**

Fills empty rows in the input 2-D `SparseTensor` with a default value.

This op adds entries with the specified `default_value` at index `[row, 0]` for any row in the input that does not already have a value.

For example, suppose `sp_input` has shape `[5, 6]` and non-empty values:

```
[0, 1]: a
[0, 3]: b
[2, 0]: c
[3, 1]: d
```

Rows 1 and 4 are empty, so the output will be of shape `[5, 6]` with values:

```
[0, 1]: a
[0, 3]: b
[1, 0]: default_value
[2, 0]: c
[3, 1]: d
[4, 0]: default_value
```

Note that the input may have empty columns at the end, with no effect on this op.

The output `SparseTensor` will be in row-major order and will have the same shape as the input.

This op also returns an indicator vector such that

```
empty_row_indicator[i] = True iff row i was an empty row.
```

### Args:

- `sp_input` : A `SparseTensor` with shape `[N, M]`.
- `default_value` : The value to fill for empty rows, with the same type as `sp_input`.
- `name` : A name prefix for the returned tensors (optional)

**Returns:**

- `sp_ordered_output` : A `sparseTensor` with shape `[N, M]` , and with all empty rows filled in with `default_value` .
- `empty_row_indicator` : A bool vector of length `N` indicating whether each input row was empty.

**Raises:**

- `TypeError` : If `sp_input` is not a `SparseTensor` .

## Reduction

---

```
tf.sparse_reduce_sum(sp_input,
reduction_axes=None, keep_dims=False)
```

Computes the sum of elements across dimensions of a `SparseTensor`.

This Op takes a `SparseTensor` and is the sparse counterpart to `tf.reduce_sum()` . In particular, this Op also returns a dense `Tensor` instead of a sparse one.

Reduces `sp_input` along the dimensions given in `reduction_axes` . Unless `keep_dims` is true, the rank of the tensor is reduced by 1 for each entry in `reduction_axes` . If `keep_dims` is true, the reduced dimensions are retained with length 1.

If `reduction_axes` has no entries, all dimensions are reduced, and a tensor with a single element is returned. Additionally, the axes can be negative, similar to the indexing rules in Python.

For example:

```
'x' represents [[1, ?, 1]
[?, 1, ?]]
where ? is implicitly-zero.
tf.sparse_reduce_sum(x) ==> 3
tf.sparse_reduce_sum(x, 0) ==> [1, 1, 1]
tf.sparse_reduce_sum(x, 1) ==> [2, 1] # Can also use -1 as the axis.
tf.sparse_reduce_sum(x, 1, keep_dims=True) ==> [[2], [1]]
tf.sparse_reduce_sum(x, [0, 1]) ==> 3
```

**Args:**

- `sp_input` : The `SparseTensor` to reduce. Should have numeric type.

- `reduction_axes` : The dimensions to reduce; list or scalar. If `None` (the default), reduces all dimensions.
- `keep_dims` : If true, retain reduced dimensions with length 1.

**Returns:**

The reduced Tensor.

## Math Operations

### `tf.sparse_add(a, b, thresh=0)`

Adds two tensors, at least one of each is a `SparseTensor`.

If one `SparseTensor` and one `Tensor` are passed in, returns a `Tensor`. If both arguments are `SparseTensor`s, this returns a `SparseTensor`. The order of arguments does not matter. Use vanilla `tf.add()` for adding two dense `Tensor`s.

The indices of any input `SparseTensor` are assumed ordered in standard lexicographic order. If this is not the case, before this step run `SparseReorder` to restore index ordering.

If both arguments are sparse, we perform "clipping" as follows. By default, if two values sum to zero at some index, the output `SparseTensor` would still include that particular location in its index, storing a zero in the corresponding value slot. To override this, callers can specify `thresh`, indicating that if the sum has a magnitude strictly smaller than `thresh`, its corresponding value and index would then not be included. In particular, `thresh == 0.0` (default) means everything is kept and actual thresholding happens only for a positive value.

For example, suppose the logical sum of two sparse operands is (densified):

```
[2]
[.1 0]
[6 -.2]
```

Then,

- `thresh == 0` (the default): all 5 index/value pairs will be returned.
- `thresh == 0.11`: only `.1` and `0` will vanish, and the remaining three index/value pairs will be returned.
- `thresh == 0.21`: `.1`, `0`, and `-.2` will vanish.

**Args:**

- `a` : The first operand; `SparseTensor` or `Tensor` .
- `b` : The second operand; `SparseTensor` or `Tensor` . At least one operand must be sparse.
- `thresh` : A 0-D `Tensor` . The magnitude threshold that determines if an output value/index pair takes space. Its dtype should match that of the values if they are real; if the latter are complex64/complex128, then the dtype should be float32/float64, correspondingly.

**Returns:**

A `SparseTensor` or a `Tensor` , representing the sum.

**Raises:**

- `TypeError` : If both `a` and `b` are `Tensors`. Use `tf.add()` instead.

## **`tf.sparse_softmax(sp_input, name=None)`**

Applies softmax to a batched N-D `SparseTensor` .

The inputs represent an N-D SparseTensor with logical shape `[..., b, c]` (where `N >= 2` ), and with indices sorted in the canonical lexicographic order.

This op is equivalent to applying the normal `tf.nn.softmax()` to each innermost logical submatrix with shape `[b, c]` , but with the catch that *the implicitly zero elements do not participate*. Specifically, the algorithm is equivalent to:

(1) Applies `tf.nn.softmax()` to a densified view of each innermost submatrix with shape `[b, c]` , along the size-C dimension; (2) Masks out the original implicitly-zero locations; (3) Renormalizes the remaining elements.

Hence, the `SparseTensor` result has exactly the same non-zero indices and shape.

Example:

```

First batch:
[? e.]
[1. ?]
Second batch:
[e ?]
[e e]
shape = [2, 2, 2] # 3-D SparseTensor
values = np.asarray([[[0., np.e], [1., 0.]], [[np.e, 0.], [np.e, np.e]]])
indices = np.vstack(np.where(values)).astype(np.int64).T

result = tf.sparse_softmax(tf.SparseTensor(indices, values, shape))
...returning a 3-D SparseTensor, equivalent to:
[? 1.] [1 ?]
[1. ?] and [.5 .5]
where ? means implicitly zero.

```

**Args:**

- `sp_input` : N-D `SparseTensor`, where `N >= 2`.
- `name` : optional name of the operation.

**Returns:**

- `output` : N-D `sparseTensor` representing the results.

## **`tf.sparse_tensor_dense_matmul(sp_a, b, adjoint_a=False, adjoint_b=False, name=None)`**

Multiply `SparseTensor` (of rank 2) "A" by dense matrix "B".

No validity checking is performed on the indices of A. However, the following input format is recommended for optimal behavior:

if `adjoint_a == false`: A should be sorted in lexicographically increasing order. Use `sparse_reorder` if you're not sure.  
 if `adjoint_a == true`: A should be sorted in order of increasing dimension 1 (i.e., "column major" order instead of "row major" order).

Deciding when to use `sparse_tensor_dense_matmul` vs. `matmul(sp_a=True)`:

There are a number of questions to ask in the decision process, including:

- Will the `SparseTensor` A fit in memory if densified?
- Is the column count of the product large ( $>> 1$ )?
- Is the density of A larger than approximately 15%?

If the answer to several of these questions is yes, consider converting the SparseTensor to a dense one and using `tf.matmul` with `sp_a=True`.

This operation tends to perform well when A is more sparse, if the column size of the product is small (e.g. matrix-vector multiplication), if `sp_a.shape` takes on large values.

Below is a rough speed comparison between `sparse_tensor_dense_matmul`, labelled 'sparse', and `matmul(sp_a=True)`, labelled 'dense'. For purposes of the comparison, the time spent converting from a SparseTensor to a dense Tensor is not included, so it is overly conservative with respect to the time ratio.

Benchmark system: CPU: Intel Ivybridge with HyperThreading (6 cores) dL1:32KB  
dL2:256KB dL3:12MB GPU: NVidia Tesla k40c

Compiled with: `-c opt --config=cuda --copt=-mavx`

```
```tensorflow/python/sparse_tensor_dense_matmul_op_test --benchmarks A sparse [m, k]
with % nonzero values between 1% and 80% B dense [k, n]
```

```
% nnz n gpu m k dt(dense) dt(sparse) dt(sparse)/dt(dense) 0.01 1 True 100 100
0.000221166 0.00010154 0.459112 0.01 1 True 100 1000 0.00033858 0.000109275
0.322745 0.01 1 True 1000 100 0.000310557 9.85661e-05 0.317385 0.01 1 True 1000 1000
0.0008721 0.000100875 0.115669 0.01 1 False 100 100 0.000208085 0.000107603 0.51711
0.01 1 False 100 1000 0.000327112 9.51118e-05 0.290762 0.01 1 False 1000 100
0.000308222 0.00010345 0.335635 0.01 1 False 1000 1000 0.000865721 0.000101397
0.117124 0.01 10 True 100 100 0.000218522 0.000105537 0.482958 0.01 10 True 100 1000
0.000340882 0.000111641 0.327506 0.01 10 True 1000 100 0.000315472 0.000117376
0.372064 0.01 10 True 1000 1000 0.000905493 0.000123263 0.136128 0.01 10 False 100
100 0.000221529 9.82571e-05 0.44354 0.01 10 False 100 1000 0.000330552 0.000112615
0.340687 0.01 10 False 1000 100 0.000341277 0.000114097 0.334324 0.01 10 False 1000
1000 0.000819944 0.000120982 0.147549 0.01 25 True 100 100 0.000207806 0.000105977
0.509981 0.01 25 True 100 1000 0.000322879 0.00012921 0.400181 0.01 25 True 1000
100 0.00038262 0.000141583 0.370035 0.01 25 True 1000 1000 0.000865438 0.000202083
0.233504 0.01 25 False 100 100 0.000209401 0.000104696 0.499979 0.01 25 False 100
1000 0.000321161 0.000130737 0.407076 0.01 25 False 1000 100 0.000377012
0.000136801 0.362856 0.01 25 False 1000 1000 0.000861125 0.00020272 0.235413 0.2 1
True 100 100 0.000206952 9.69219e-05 0.46833 0.2 1 True 100 1000 0.000348674
0.000147475 0.422959 0.2 1 True 1000 100 0.000336908 0.00010122 0.300439 0.2 1 True
1000 1000 0.001022 0.000203274 0.198898 0.2 1 False 100 100 0.000207532 9.5412e-05
0.459746 0.2 1 False 100 1000 0.000356127 0.000146824 0.41228 0.2 1 False 1000 100
0.000322664 0.000100918 0.312764 0.2 1 False 1000 1000 0.000998987 0.000203442
0.203648 0.2 10 True 100 100 0.000211692 0.000109903 0.519165 0.2 10 True 100 1000
0.000372819 0.000164321 0.440753 0.2 10 True 1000 100 0.000338651 0.000144806
```

0.427596 0.2 10 True 1000 1000 0.00108312 0.000758876 0.70064 0.2 10 False 100 100
0.000215727 0.000110502 0.512231 0.2 10 False 100 1000 0.000375419 0.0001613
0.429653 0.2 10 False 1000 100 0.000336999 0.000145628 0.432132 0.2 10 False 1000
1000 0.00110502 0.000762043 0.689618 0.2 25 True 100 100 0.000218705 0.000129913
0.594009 0.2 25 True 100 1000 0.000394794 0.00029428 0.745402 0.2 25 True 1000 100
0.000404483 0.0002693 0.665788 0.2 25 True 1000 1000 0.0012002 0.00194494 1.62052
0.2 25 False 100 100 0.000221494 0.0001306 0.589632 0.2 25 False 100 1000
0.000396436 0.000297204 0.74969 0.2 25 False 1000 100 0.000409346 0.000270068
0.659754 0.2 25 False 1000 1000 0.00121051 0.00193737 1.60046 0.5 1 True 100 100
0.000214981 9.82111e-05 0.456836 0.5 1 True 100 1000 0.000415328 0.000223073
0.537101 0.5 1 True 1000 100 0.000358324 0.00011269 0.314492 0.5 1 True 1000 1000
0.00137612 0.000437401 0.317851 0.5 1 False 100 100 0.000224196 0.000101423
0.452386 0.5 1 False 100 1000 0.000400987 0.000223286 0.556841 0.5 1 False 1000 100
0.000368825 0.00011224 0.304318 0.5 1 False 1000 1000 0.00136036 0.000429369
0.31563 0.5 10 True 100 100 0.000222125 0.000112308 0.505608 0.5 10 True 100 1000
0.000461088 0.00032357 0.701753 0.5 10 True 1000 100 0.000394624 0.000225497
0.571422 0.5 10 True 1000 1000 0.00158027 0.00190898 1.20801 0.5 10 False 100 100
0.000232083 0.000114978 0.495418 0.5 10 False 100 1000 0.000454574 0.000324632
0.714146 0.5 10 False 1000 100 0.000379097 0.000227768 0.600817 0.5 10 False 1000
1000 0.00160292 0.00190168 1.18638 0.5 25 True 100 100 0.00023429 0.000151703
0.647501 0.5 25 True 100 1000 0.000497462 0.000598873 1.20386 0.5 25 True 1000 100
0.000460778 0.000557038 1.20891 0.5 25 True 1000 1000 0.00170036 0.00467336
2.74845 0.5 25 False 100 100 0.000228981 0.000155334 0.678371 0.5 25 False 100 1000
0.000496139 0.000620789 1.25124 0.5 25 False 1000 100 0.00045473 0.000551528
1.21287 0.5 25 False 1000 1000 0.00171793 0.00467152 2.71927 0.8 1 True 100 100
0.000222037 0.000105301 0.47425 0.8 1 True 100 1000 0.000410804 0.000329327
0.801664 0.8 1 True 1000 100 0.000349735 0.000131225 0.375212 0.8 1 True 1000 1000
0.00139219 0.000677065 0.48633 0.8 1 False 100 100 0.000214079 0.000107486
0.502085 0.8 1 False 100 1000 0.000413746 0.000323244 0.781261 0.8 1 False 1000 100
0.000348983 0.000131983 0.378193 0.8 1 False 1000 1000 0.00136296 0.000685325
0.50282 0.8 10 True 100 100 0.000229159 0.00011825 0.516017 0.8 10 True 100 1000
0.000498845 0.000532618 1.0677 0.8 10 True 1000 100 0.000383126 0.00029935
0.781336 0.8 10 True 1000 1000 0.00162866 0.00307312 1.88689 0.8 10 False 100 100
0.000230783 0.000124958 0.541452 0.8 10 False 100 1000 0.000493393 0.000550654
1.11606 0.8 10 False 1000 100 0.000377167 0.000298581 0.791642 0.8 10 False 1000
1000 0.00165795 0.00305103 1.84024 0.8 25 True 100 100 0.000233496 0.000175241
0.75051 0.8 25 True 100 1000 0.00055654 0.00102658 1.84458 0.8 25 True 1000 100
0.000463814 0.000783267 1.68875 0.8 25 True 1000 1000 0.00186905 0.00755344

```
4.04132 0.8 25 False 100 100 0.000240243 0.000175047 0.728625 0.8 25 False 100 1000  
0.000578102 0.00104499 1.80763 0.8 25 False 1000 100 0.000485113 0.000776849  
1.60138 0.8 25 False 1000 1000 0.00211448 0.00752736 3.55992 ````
```

Args:

- `sp_a` : SparseTensor A, of rank 2.
- `b` : A dense Matrix with the same dtype as `sp_a`.
- `adjoint_a` : Use the adjoint of A in the matrix multiply. If A is complex, this is transpose(conj(A)). Otherwise it's transpose(A).
- `adjoint_b` : Use the adjoint of B in the matrix multiply. If B is complex, this is transpose(conj(B)). Otherwise it's transpose(B).
- `name` : A name prefix for the returned tensors (optional)

Returns:

A dense matrix (pseudo-code in dense np.matrix notation): $A = A.H$ if `adjoint_a` else A $B = B.H$ if `adjoint_b` else B return A^*B

Inputs and Readers

Note: Functions taking `Tensor` arguments can also take anything accepted by `tf.convert_to_tensor`.

[TOC]

Placeholders

TensorFlow provides a placeholder operation that must be fed with data on execution. For more info, see the section on [Feeding data](#).

`tf.placeholder(dtype, shape=None, name=None)`

Inserts a placeholder for a tensor that will be always fed.

Important: This tensor will produce an error if evaluated. Its value must be fed using the `feed_dict` optional argument to `Session.run()`, `Tensor.eval()`, or `Operation.run()`.

For example:

```
x = tf.placeholder(tf.float32, shape=(1024, 1024))
y = tf.matmul(x, x)

with tf.Session() as sess:
    print(sess.run(y)) # ERROR: will fail because x was not fed.

rand_array = np.random.rand(1024, 1024)
print(sess.run(y, feed_dict={x: rand_array})) # Will succeed.
```

Args:

- `dtype` : The type of elements in the tensor to be fed.
- `shape` : The shape of the tensor to be fed (optional). If the shape is not specified, you can feed a tensor of any shape.
- `name` : A name for the operation (optional).

Returns:

A `Tensor` that may be used as a handle for feeding a value, but not evaluated directly.

tf.placeholder_with_default(input, shape, name=None)

A placeholder op that passes through `input` when its output is not fed.

Args:

- `input` : A `Tensor`. The default value to produce when `output` is not fed.
- `shape` : A `tf.TensorShape` or list of `ints`. The (possibly partial) shape of the tensor.
- `name` : A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`. A placeholder tensor that defaults to `input` if it is not fed.

For feeding `SparseTensor`s which are composite type, there is a convenience function:

tf.sparse_placeholder(dtype, shape=None, name=None)

Inserts a placeholder for a sparse tensor that will be always fed.

Important: This sparse tensor will produce an error if evaluated. Its value must be fed using the `feed_dict` optional argument to `Session.run()`, `Tensor.eval()`, or `Operation.run()`.

For example:

```

x = tf.sparse_placeholder(tf.float32)
y = tf.sparse_reduce_sum(x)

with tf.Session() as sess:
    print(sess.run(y)) # ERROR: will fail because x was not fed.

indices = np.array([[3, 2, 0], [4, 5, 1]], dtype=np.int64)
values = np.array([1.0, 2.0], dtype=np.float32)
shape = np.array([7, 9, 2], dtype=np.int64)
print(sess.run(y, feed_dict={
    x: tf.SparseTensorValue(indices, values, shape)))) # Will succeed.
print(sess.run(y, feed_dict={
    x: (indices, values, shape)))) # Will succeed.

sp = tf.SparseTensor(indices=indices, values=values, shape=shape)
sp_value = sp.eval(session)
print(sess.run(y, feed_dict={x: sp_value})) # Will succeed.

```

Args:

- `dtype` : The type of `values` elements in the tensor to be fed.
- `shape` : The shape of the tensor to be fed (optional). If the shape is not specified, you can feed a sparse tensor of any shape.
- `name` : A name for prefixing the operations (optional).

Returns:

A `sparseTensor` that may be used as a handle for feeding a value, but not evaluated directly.

Readers

TensorFlow provides a set of Reader classes for reading data formats. For more information on inputs and readers, see [Reading data](#).

`class tf.ReaderBase`

Base class for different Reader types, that produce a record every step.

Conceptually, Readers convert string 'work units' into records (key, value pairs). Typically the 'work units' are filenames and the records are extracted from the contents of those files. We want a single record produced per step, but a work unit can correspond to many records.

Therefore we introduce some decoupling using a queue. The queue contains the work units and the Reader dequeues from the queue when it is asked to produce a record (via Read()) but it has finished the last work unit.

`tf.ReaderBase.__init__(reader_ref, supports_serialize=False)`

Creates a new ReaderBase.

Args:

- `reader_ref` : The operation that implements the reader.
 - `supports_serialize` : True if the reader implementation can serialize its state.
-

`tf.ReaderBase.num_records_produced(name=None)`

Returns the number of records this reader has produced.

This is the same as the number of Read executions that have succeeded.

Args:

- `name` : A name for the operation (optional).

Returns:

An int64 Tensor.

`tf.ReaderBase.num_work_units_completed(name=None)`

Returns the number of work units this reader has finished processing.

Args:

- `name` : A name for the operation (optional).

Returns:

An int64 Tensor.

`tf.ReaderBase.read(queue, name=None)`

Returns the next record (key, value pair) produced by a reader.

Will dequeue a work unit from queue if necessary (e.g. when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

- `queue` : A Queue or a mutable string Tensor representing a handle to a Queue, with string work items.
- `name` : A name for the operation (optional).

Returns:

A tuple of Tensors (key, value).

- `key` : A string scalar Tensor.
- `value` : A string scalar Tensor.

`tf.ReaderBase.read_up_to(queue, num_records, name=None)`

Returns up to `num_records` (key, value pairs) produced by a reader.

Will dequeue a work unit from queue if necessary (e.g., when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

- `queue` : A Queue or a mutable string Tensor representing a handle to a Queue, with string work items.
- `num_records` : Number of records to read.
- `name` : A name for the operation (optional).

Returns:

A tuple of Tensors (keys, values).

- `keys` : A 1-D string Tensor.
- `values` : A 1-D string Tensor.

`tf.ReaderBase.reader_ref`

Op that implements the reader.

tf.ReaderBase.reset(name=None)

Restore a reader to its initial clean state.

Args:

- `name` : A name for the operation (optional).

Returns:

The created Operation.

tf.ReaderBase.restore_state(state, name=None)

Restore a reader to a previously saved state.

Not all Readers support being restored, so this can produce an Unimplemented error.

Args:

- `state` : A string Tensor. Result of a SerializeState of a Reader with matching type.
- `name` : A name for the operation (optional).

Returns:

The created Operation.

tf.ReaderBase.serialize_state(name=None)

Produce a string tensor that encodes the state of a reader.

Not all Readers support being serialized, so this can produce an Unimplemented error.

Args:

- `name` : A name for the operation (optional).

Returns:

A string Tensor.

tf.ReaderBase.supports_serialize

Whether the Reader implementation can serialize its state.

class tf.TextLineReader

A Reader that outputs the lines of a file delimited by newlines.

Newlines are stripped from the output. See ReaderBase for supported methods.

```
tf.TextLineReader.__init__(skip_header_lines=None,  
name=None)
```

Create a TextLineReader.

Args:

- `skip_header_lines` : An optional int. Defaults to 0. Number of lines to skip from the beginning of every file.
 - `name` : A name for the operation (optional).
-

```
tf.TextLineReader.num_records_produced(name=None)
```

Returns the number of records this reader has produced.

This is the same as the number of Read executions that have succeeded.

Args:

- `name` : A name for the operation (optional).

Returns:

An int64 Tensor.

```
tf.TextLineReader.num_work_units_completed(name=None)
```

Returns the number of work units this reader has finished processing.

Args:

- `name` : A name for the operation (optional).
-

Returns:

An int64 Tensor.

tf.TextLineReader.read(queue, name=None)

Returns the next record (key, value pair) produced by a reader.

Will dequeue a work unit from queue if necessary (e.g. when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

- `queue` : A Queue or a mutable string Tensor representing a handle to a Queue, with string work items.
- `name` : A name for the operation (optional).

Returns:

A tuple of Tensors (key, value).

- `key` : A string scalar Tensor.
- `value` : A string scalar Tensor.

tf.TextLineReader.read_up_to(queue, num_records, name=None)

Returns up to `num_records` (key, value pairs) produced by a reader.

Will dequeue a work unit from queue if necessary (e.g., when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

- `queue` : A Queue or a mutable string Tensor representing a handle to a Queue, with string work items.
- `num_records` : Number of records to read.
- `name` : A name for the operation (optional).

Returns:

A tuple of Tensors (keys, values).

- `keys` : A 1-D string Tensor.

- `values` : A 1-D string Tensor.
-

`tf.TextLineReader.reader_ref`

Op that implements the reader.

`tf.TextLineReader.reset(name=None)`

Restore a reader to its initial clean state.

Args:

- `name` : A name for the operation (optional).

Returns:

The created Operation.

`tf.TextLineReader.restore_state(state, name=None)`

Restore a reader to a previously saved state.

Not all Readers support being restored, so this can produce an Unimplemented error.

Args:

- `state` : A string Tensor. Result of a SerializeState of a Reader with matching type.
- `name` : A name for the operation (optional).

Returns:

The created Operation.

`tf.TextLineReader.serialize_state(name=None)`

Produce a string tensor that encodes the state of a reader.

Not all Readers support being serialized, so this can produce an Unimplemented error.

Args:

- `name` : A name for the operation (optional).

Returns:

A string Tensor.

`tf.TextLineReader.supports_serialize`

Whether the Reader implementation can serialize its state.

`class tf.WholeFileReader`

A Reader that outputs the entire contents of a file as a value.

To use, enqueue filenames in a Queue. The output of Read will be a filename (key) and the contents of that file (value).

See ReaderBase for supported methods.

`tf.WholeFileReader.__init__(name=None)`

Create a WholeFileReader.

Args:

- `name` : A name for the operation (optional).
-

`tf.WholeFileReader.num_records_produced(name=None)`

Returns the number of records this reader has produced.

This is the same as the number of Read executions that have succeeded.

Args:

- `name` : A name for the operation (optional).

Returns:

An int64 Tensor.

tf.WholeFileReader.num_work_units_completed(name=None)

Returns the number of work units this reader has finished processing.

Args:

- `name` : A name for the operation (optional).

Returns:

An int64 Tensor.

tf.WholeFileReader.read(queue, name=None)

Returns the next record (key, value pair) produced by a reader.

Will dequeue a work unit from queue if necessary (e.g. when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

- `queue` : A Queue or a mutable string Tensor representing a handle to a Queue, with string work items.
- `name` : A name for the operation (optional).

Returns:

A tuple of Tensors (key, value).

- `key` : A string scalar Tensor.
- `value` : A string scalar Tensor.

tf.WholeFileReader.read_up_to(queue, num_records, name=None)

Returns up to num_records (key, value pairs) produced by a reader.

Will dequeue a work unit from queue if necessary (e.g., when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

- `queue` : A Queue or a mutable string Tensor representing a handle to a Queue, with string work items.

- `num_records` : Number of records to read.
- `name` : A name for the operation (optional).

Returns:

A tuple of Tensors (keys, values).

- `keys` : A 1-D string Tensor.
- `values` : A 1-D string Tensor.

`tf.WholeFileReader.reader_ref`

Op that implements the reader.

`tf.WholeFileReader.reset(name=None)`

Restore a reader to its initial clean state.

Args:

- `name` : A name for the operation (optional).

Returns:

The created Operation.

`tf.WholeFileReader.restore_state(state, name=None)`

Restore a reader to a previously saved state.

Not all Readers support being restored, so this can produce an Unimplemented error.

Args:

- `state` : A string Tensor. Result of a SerializeState of a Reader with matching type.
- `name` : A name for the operation (optional).

Returns:

The created Operation.

tf.WholeFileReader.serialize_state(name=None)

Produce a string tensor that encodes the state of a reader.

Not all Readers support being serialized, so this can produce an Unimplemented error.

Args:

- `name` : A name for the operation (optional).

Returns:

A string Tensor.

tf.WholeFileReader.supports_serialize

Whether the Reader implementation can serialize its state.

class tf.IdentityReader

A Reader that outputs the queued work as both the key and value.

To use, enqueue strings in a Queue. Read will take the front work string and output (work, work).

See ReaderBase for supported methods.

tf.IdentityReader.__init__(name=None)

Create a IdentityReader.

Args:

- `name` : A name for the operation (optional).

tf.IdentityReader.num_records_produced(name=None)

Returns the number of records this reader has produced.

This is the same as the number of Read executions that have succeeded.

Args:

- `name` : A name for the operation (optional).

Returns:

An int64 Tensor.

`tf.IdentityReader.num_work_units_completed(name=None)`

Returns the number of work units this reader has finished processing.

Args:

- `name` : A name for the operation (optional).

Returns:

An int64 Tensor.

`tf.IdentityReader.read(queue, name=None)`

Returns the next record (key, value pair) produced by a reader.

Will dequeue a work unit from queue if necessary (e.g. when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

- `queue` : A Queue or a mutable string Tensor representing a handle to a Queue, with string work items.
- `name` : A name for the operation (optional).

Returns:

A tuple of Tensors (key, value).

- `key` : A string scalar Tensor.
- `value` : A string scalar Tensor.

`tf.IdentityReader.read_up_to(queue, num_records, name=None)`

Returns up to num_records (key, value pairs) produced by a reader.

Will dequeue a work unit from queue if necessary (e.g., when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

- `queue` : A Queue or a mutable string Tensor representing a handle to a Queue, with string work items.
- `num_records` : Number of records to read.
- `name` : A name for the operation (optional).

Returns:

A tuple of Tensors (keys, values).

- `keys` : A 1-D string Tensor.
 - `values` : A 1-D string Tensor.
-

tf.IdentityReader.reader_ref

Op that implements the reader.

tf.IdentityReader.reset(name=None)

Restore a reader to its initial clean state.

Args:

- `name` : A name for the operation (optional).

Returns:

The created Operation.

tf.IdentityReader.restore_state(state, name=None)

Restore a reader to a previously saved state.

Not all Readers support being restored, so this can produce an Unimplemented error.

Args:

- `state` : A string Tensor. Result of a SerializeState of a Reader with matching type.
- `name` : A name for the operation (optional).

Returns:

The created Operation.

`tf.IdentityReader.serialize_state(name=None)`

Produce a string tensor that encodes the state of a reader.

Not all Readers support being serialized, so this can produce an Unimplemented error.

Args:

- `name` : A name for the operation (optional).

Returns:

A string Tensor.

`tf.IdentityReader.supports_serialize`

Whether the Reader implementation can serialize its state.

`class tf.TFRecordReader`

A Reader that outputs the records from a TFRecords file.

See ReaderBase for supported methods.

`tf.TFRecordReader.__init__(name=None)`

Create a TFRecordReader.

Args:

- `name` : A name for the operation (optional).

tf.TFRecordReader.num_records_produced(name=None)

Returns the number of records this reader has produced.

This is the same as the number of Read executions that have succeeded.

Args:

- `name` : A name for the operation (optional).

Returns:

An int64 Tensor.

tf.TFRecordReader.num_work_units_completed(name=None)

Returns the number of work units this reader has finished processing.

Args:

- `name` : A name for the operation (optional).

Returns:

An int64 Tensor.

tf.TFRecordReader.read(queue, name=None)

Returns the next record (key, value pair) produced by a reader.

Will dequeue a work unit from queue if necessary (e.g. when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

- `queue` : A Queue or a mutable string Tensor representing a handle to a Queue, with string work items.
- `name` : A name for the operation (optional).

Returns:

A tuple of Tensors (key, value).

- `key` : A string scalar Tensor.
- `value` : A string scalar Tensor.

tf.TFRecordReader.read_up_to(queue, num_records, name=None)

Returns up to num_records (key, value pairs) produced by a reader.

Will dequeue a work unit from queue if necessary (e.g., when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

- `queue` : A Queue or a mutable string Tensor representing a handle to a Queue, with string work items.
- `num_records` : Number of records to read.
- `name` : A name for the operation (optional).

Returns:

A tuple of Tensors (keys, values).

- `keys` : A 1-D string Tensor.
 - `values` : A 1-D string Tensor.
-

tf.TFRecordReader.reader_ref

Op that implements the reader.

tf.TFRecordReader.reset(name=None)

Restore a reader to its initial clean state.

Args:

- `name` : A name for the operation (optional).

Returns:

The created Operation.

tf.TFRecordReader.restore_state(state, name=None)

Restore a reader to a previously saved state.

Not all Readers support being restored, so this can produce an Unimplemented error.

Args:

- `state` : A string Tensor. Result of a SerializeState of a Reader with matching type.
- `name` : A name for the operation (optional).

Returns:

The created Operation.

`tf.TFRecordReader.serialize_state(name=None)`

Produce a string tensor that encodes the state of a reader.

Not all Readers support being serialized, so this can produce an Unimplemented error.

Args:

- `name` : A name for the operation (optional).

Returns:

A string Tensor.

`tf.TFRecordReader.supports_serialize`

Whether the Reader implementation can serialize its state.

`class tf.FixedLengthRecordReader`

A Reader that outputs fixed-length records from a file.

See ReaderBase for supported methods.

`tf.FixedLengthRecordReader.__init__(record_bytes, header_bytes=None, footer_bytes=None, name=None)`

Create a FixedLengthRecordReader.

Args:

- `record_bytes` : An int.
 - `header_bytes` : An optional int. Defaults to 0.
 - `footer_bytes` : An optional int. Defaults to 0.
 - `name` : A name for the operation (optional).
-

tf.FixedLengthRecordReader.num_records_produced(name=None)

Returns the number of records this reader has produced.

This is the same as the number of Read executions that have succeeded.

Args:

- `name` : A name for the operation (optional).

Returns:

An int64 Tensor.

tf.FixedLengthRecordReader.num_work_units_completed(name=None)

Returns the number of work units this reader has finished processing.

Args:

- `name` : A name for the operation (optional).

Returns:

An int64 Tensor.

tf.FixedLengthRecordReader.read(queue, name=None)

Returns the next record (key, value pair) produced by a reader.

Will dequeue a work unit from queue if necessary (e.g. when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

- `queue` : A Queue or a mutable string Tensor representing a handle to a Queue, with
-

string work items.

- `name` : A name for the operation (optional).

Returns:

A tuple of Tensors (key, value).

- `key` : A string scalar Tensor.
- `value` : A string scalar Tensor.

`tf.FixedLengthRecordReader.read_up_to(queue, num_records, name=None)`

Returns up to `num_records` (key, value pairs) produced by a reader.

Will dequeue a work unit from `queue` if necessary (e.g., when the Reader needs to start reading from a new file since it has finished with the previous file).

Args:

- `queue` : A Queue or a mutable string Tensor representing a handle to a Queue, with string work items.
- `num_records` : Number of records to read.
- `name` : A name for the operation (optional).

Returns:

A tuple of Tensors (keys, values).

- `keys` : A 1-D string Tensor.
- `values` : A 1-D string Tensor.

`tf.FixedLengthRecordReader.reader_ref`

Op that implements the reader.

`tf.FixedLengthRecordReader.reset(name=None)`

Restore a reader to its initial clean state.

Args:

- `name` : A name for the operation (optional).

Returns:

The created Operation.

`tf.FixedLengthRecordReader.restore_state(state, name=None)`

Restore a reader to a previously saved state.

Not all Readers support being restored, so this can produce an Unimplemented error.

Args:

- `state` : A string Tensor. Result of a SerializeState of a Reader with matching type.
- `name` : A name for the operation (optional).

Returns:

The created Operation.

`tf.FixedLengthRecordReader.serialize_state(name=None)`

Produce a string tensor that encodes the state of a reader.

Not all Readers support being serialized, so this can produce an Unimplemented error.

Args:

- `name` : A name for the operation (optional).

Returns:

A string Tensor.

`tf.FixedLengthRecordReader.supports_serialize`

Whether the Reader implementation can serialize its state.

Converting

TensorFlow provides several operations that you can use to convert various data formats into tensors.

`tf.decode_csv(records, record_defaults, field_delim=None, name=None)`

Convert CSV records to tensors. Each column maps to one tensor.

RFC 4180 format is expected for the CSV records. (<https://tools.ietf.org/html/rfc4180>) Note that we allow leading and trailing spaces with int or float field.

Args:

- `records` : A `Tensor` of type `string`. Each string is a record/row in the csv and all records should have the same format.
- `record_defaults` : A list of `Tensor` objects with types from: `float32`, `int32`, `int64`, `string`. One tensor per column of the input record, with either a scalar default value for that column or empty if the column is required.
- `field_delim` : An optional `string`. Defaults to `","`. delimiter to separate fields in a record.
- `name` : A name for the operation (optional).

Returns:

A list of `Tensor` objects. Has the same type as `record_defaults`. Each tensor will have the same shape as records.

`tf.decode_raw(bytess, out_type, little_endian=None, name=None)`

Reinterpret the bytes of a string as a vector of numbers.

Args:

- `bytes` : A `Tensor` of type `string`. All the elements must have the same length.
- `out_type` : A `tf.DType` from: `tf.float32`, `tf.float64`, `tf.int32`, `tf.uint8`, `tf.int16`, `tf.int8`, `tf.int64`.
- `little_endian` : An optional `bool`. Defaults to `True`. Whether the input `bytes` are in little-endian order. Ignored for `out_type` values that are stored in a single byte like `uint8`.

- `name` : A name for the operation (optional).

Returns:

A `Tensor` of type `out_type`. A Tensor with one more dimension than the input `bytes`. The added dimension will have size equal to the length of the elements of `bytes` divided by the number of bytes to represent `out_type`.

Example protocol buffer

TensorFlow's [recommended format for training examples](#) is serialized `Example` protocol buffers, [described here](#). They contain `Features`, [described here](#).

`class tf.VarLenFeature`

Configuration for parsing a variable-length input feature.

Fields: `dtype`: Data type of input.

`tf.VarLenFeature.dtype`

Alias for field number 0

`class tf.FixedLenFeature`

Configuration for parsing a fixed-length input feature.

To treat sparse input as dense, provide a `default_value`; otherwise, the parse functions will fail on any examples missing this feature.

Fields: `shape`: Shape of input data. `dtype`: Data type of input. `default_value`: Value to be used if an example is missing this feature. It must be compatible with `dtype`.

`tf.FixedLenFeature.default_value`

Alias for field number 2

tf.FixedLenFeature.dtype

Alias for field number 1

tf.FixedLenFeature.shape

Alias for field number 0

class tf.FixedLenSequenceFeature

Configuration for a dense input feature in a sequence item.

To treat a sparse input as dense, provide `allow_missing=True` ; otherwise, the parse functions will fail on any examples missing this feature.

Fields: `shape`: Shape of input data. `dtype`: Data type of input. `allow_missing`: Whether to allow this feature to be missing from a feature list item.

tf.FixedLenSequenceFeature.allow_missing

Alias for field number 2

tf.FixedLenSequenceFeature.dtype

Alias for field number 1

tf.FixedLenSequenceFeature.shape

Alias for field number 0

tf.parse_example(serialized, features, name=None, example_names=None)

Parses `Example` protos into a `dict` of tensors.

Parses a number of serialized [Example]

(<https://www.tensorflow.org/code/tensorflow/core/example/example.proto>) protos given in `serialized`.

`example_names` may contain descriptive names for the corresponding serialized protos. These may be useful for debugging purposes, but they have no effect on the output. If not `None`, `example_names` must be the same length as `serialized`.

This op parses serialized examples into a dictionary mapping keys to `Tensor` and `SparseTensor` objects. `features` is a dict from keys to `VarLenFeature` and `FixedLenFeature` objects. Each `VarLenFeature` is mapped to a `SparseTensor`, and each `FixedLenFeature` is mapped to a `Tensor`.

Each `VarLenFeature` maps to a `SparseTensor` of the specified type representing a ragged matrix. Its indices are `[batch, index]` where `batch` is the batch entry the value is from in `serialized`, and `index` is the value's index in the list of values associated with that feature and example.

Each `FixedLenFeature` `df` maps to a `Tensor` of the specified type (or `tf.float32` if not specified) and shape `(serialized.size(),) + df.shape`.

`FixedLenFeature` entries with a `default_value` are optional. With no default value, we will fail if that `Feature` is missing from any example in `serialized`.

Examples:

For example, if one expects a `tf.float32` sparse feature `ft` and three serialized `Example`s are provided:

```
serialized = [
    features
    { feature { key: "ft" value { float_list { value: [1.0, 2.0] } } },
    features
    { feature []},
    features
    { feature { key: "ft" value { float_list { value: [3.0] } } }
]
```

then the output will look like:

```
{"ft": SparseTensor(indices=[[0, 0], [0, 1], [2, 0]],
                     values=[1.0, 2.0, 3.0],
                     shape=(3, 2)) }
```

Given two `Example` input protos in `serialized`:

```
[  
  features {  
    feature { key: "kw" value { bytes_list { value: [ "knit", "big" ] } } }  
    feature { key: "gps" value { float_list { value: [] } } }  
  },  
  features {  
    feature { key: "kw" value { bytes_list { value: [ "emmy" ] } } }  
    feature { key: "dank" value { int64_list { value: [ 42 ] } } }  
    feature { key: "gps" value { } }  
  }  
]
```

And arguments

```
example_names: ["input0", "input1"],  
features: {  
  "kw": VarLenFeature(tf.string),  
  "dank": VarLenFeature(tf.int64),  
  "gps": VarLenFeature(tf.float32),  
}
```

Then the output is a dictionary:

```
{  
  "kw": SparseTensor(  
    indices=[[0, 0], [0, 1], [1, 0]],  
    values=["knit", "big", "emmy"]  
    shape=[2, 2]),  
  "dank": SparseTensor(  
    indices=[[1, 0]],  
    values=[42],  
    shape=[2, 1]),  
  "gps": SparseTensor(  
    indices=[],  
    values=[],  
    shape=[2, 0]),  
}
```

For dense results in two serialized Example S:

```
[  
    features {  
        feature { key: "age" value { int64_list { value: [ 0 ] } } }  
        feature { key: "gender" value { bytes_list { value: [ "f" ] } } }  
    },  
    features {  
        feature { key: "age" value { int64_list { value: [] } } }  
        feature { key: "gender" value { bytes_list { value: [ "f" ] } } }  
    }  
]
```

We can use arguments:

```
example_names: ["input0", "input1"],  
features: {  
    "age": FixedLenFeature([], dtype=tf.int64, default_value=-1),  
    "gender": FixedLenFeature([], dtype=tf.string),  
}
```

And the expected output is:

```
{  
    "age": [[0], [-1]],  
    "gender": [["f"], ["f"]],  
}
```

Args:

- `serialized` : A vector (1-D Tensor) of strings, a batch of binary serialized `Example` protos.
- `features` : A `dict` mapping feature keys to `FixedLenFeature` or `VarLenFeature` values.
- `name` : A name for this operation (optional).
- `example_names` : A vector (1-D Tensor) of strings (optional), the names of the serialized protos in the batch.

Returns:

A `dict` mapping feature keys to `Tensor` and `SparseTensor` values.

Raises:

- `ValueError` : if any feature is invalid.

`tf.parse_single_example(serialized, features, name=None, example_names=None)`

Parses a single `Example` proto.

Similar to `parse_example`, except:

For dense tensors, the returned `Tensor` is identical to the output of `parse_example`, except there is no batch dimension, the output shape is the same as the shape given in `dense_shape`.

For `SparseTensor`s, the first (batch) column of the indices matrix is removed (the indices matrix is a column vector), the values vector is unchanged, and the first (`batch_size`) entry of the shape vector is removed (it is now a single element vector).

Args:

- `serialized` : A scalar string Tensor, a single serialized Example. See `_parse_single_example_raw` documentation for more details.
- `features` : A `dict` mapping feature keys to `FixedLenFeature` or `VarLenFeature` values.
- `name` : A name for this operation (optional).
- `example_names` : (Optional) A scalar string Tensor, the associated name. See `_parse_single_example_raw` documentation for more details.

Returns:

A `dict` mapping feature keys to `Tensor` and `SparseTensor` values.

Raises:

- `ValueError` : if any feature is invalid.

`tf.decode_json_example(json_examples, name=None)`

Convert JSON-encoded Example records to binary protocol buffer strings.

This op translates a tensor containing Example records, encoded using the [standard JSON mapping](#), into a tensor containing the same records encoded as binary protocol buffers. The resulting tensor can then be fed to any of the other Example-parsing ops.

Args:

- `json_examples` : A `Tensor` of type `string`. Each string is a JSON object serialized according to the JSON mapping of the Example proto.
- `name` : A name for the operation (optional).

Returns:

A `Tensor` of type `string`. Each string is a binary Example protocol buffer corresponding to the respective element of `json_examples`.

Queues

TensorFlow provides several implementations of 'Queues', which are structures within the TensorFlow computation graph to stage pipelines of tensors together. The following describe the basic Queue interface and some implementations. To see an example use, see [Threading and Queues](#).

`class tf.QueueBase`

Base class for queue implementations.

A queue is a TensorFlow data structure that stores tensors across multiple steps, and exposes operations that enqueue and dequeue tensors.

Each queue element is a tuple of one or more tensors, where each tuple component has a static `dtype`, and may have a static shape. The queue implementations support versions of enqueue and dequeue that handle single elements, versions that support enqueueing and dequeuing a batch of elements at once.

See [`tf.FIFOQueue`](#) and [`tf.RandomShuffleQueue`](#) for concrete implementations of this class, and instructions on how to create them.

`tf.QueueBase.enqueue(vals, name=None)`

Enqueues one element to this queue.

If the queue is full when this operation executes, it will block until the element has been enqueued.

At runtime, this operation may raise an error if the queue is [closed](#) before or during its execution. If the queue is closed before this operation runs, `tf.errors.AbortedError` will be raised. If this operation is blocked, and either (i) the queue is closed by a close operation

with `cancel_pending_enqueues=True`, or (ii) the session is `closed`, `tf.errors.CancelledError` will be raised.

Args:

- `vals` : A tensor, a list or tuple of tensors, or a dictionary containing the values to enqueue.
- `name` : A name for the operation (optional).

Returns:

The operation that enqueues a new tuple of tensors to the queue.

`tf.QueueBase.enqueue_many(vals, name=None)`

Enqueues zero or more elements to this queue.

This operation slices each component tensor along the 0th dimension to make multiple queue elements. All of the tensors in `vals` must have the same size in the 0th dimension.

If the queue is full when this operation executes, it will block until all of the elements have been enqueued.

At runtime, this operation may raise an error if the queue is `closed` before or during its execution. If the queue is closed before this operation runs, `tf.errors.AbortedError` will be raised. If this operation is blocked, and either (i) the queue is closed by a close operation with `cancel_pending_enqueues=True`, or (ii) the session is `closed`, `tf.errors.CancelledError` will be raised.

Args:

- `vals` : A tensor, a list or tuple of tensors, or a dictionary from which the queue elements are taken.
- `name` : A name for the operation (optional).

Returns:

The operation that enqueues a batch of tuples of tensors to the queue.

`tf.QueueBase.dequeue(name=None)`

Dequeues one element from this queue.

If the queue is empty when this operation executes, it will block until there is an element to dequeue.

At runtime, this operation may raise an error if the queue is `closed` before or during its execution. If the queue is closed, the queue is empty, and there are no pending enqueue operations that can fulfil this request, `tf.errorsOutOfRange` will be raised. If the session is `closed`, `tf.errors.CancelledError` will be raised.

Args:

- `name` : A name for the operation (optional).

Returns:

The tuple of tensors that was dequeued.

`tf.QueueBase.dequeue_many(n, name=None)`

Dequeues and concatenates `n` elements from this queue.

This operation concatenates queue-element component tensors along the 0th dimension to make a single component tensor. All of the components in the dequeued tuple will have size `n` in the 0th dimension.

If the queue is closed and there are less than `n` elements left, then an `OutOfRange` exception is raised.

At runtime, this operation may raise an error if the queue is `closed` before or during its execution. If the queue is closed, the queue contains fewer than `n` elements, and there are no pending enqueue operations that can fulfil this request, `tf.errorsOutOfRange` will be raised. If the session is `closed`, `tf.errors.CancelledError` will be raised.

Args:

- `n` : A scalar `Tensor` containing the number of elements to dequeue.
- `name` : A name for the operation (optional).

Returns:

The tuple of concatenated tensors that was dequeued.

`tf.QueueBase.size(name=None)`

Compute the number of elements in this queue.

Args:

- `name` : A name for the operation (optional).

Returns:

A scalar tensor containing the number of elements in this queue.

```
tf.QueueBase.close(cancel_pending_enqueues=False,  
name=None)
```

Closes this queue.

This operation signals that no more elements will be enqueued in the given queue.

Subsequent `enqueue` and `enqueue_many` operations will fail. Subsequent `dequeue` and `dequeue_many` operations will continue to succeed if sufficient elements remain in the queue. Subsequent `dequeue` and `dequeue_many` operations that would block will fail immediately.

If `cancel_pending_enqueues` is `True`, all pending requests will also be cancelled.

Args:

- `cancel_pending_enqueues` : (Optional.) A boolean, defaulting to `False` (described above).
- `name` : A name for the operation (optional).

Returns:

The operation that closes the queue.

Other Methods

```
tf.QueueBase.__init__(dtypes, shapes, names, queue_ref)
```

Constructs a queue object from a queue reference.

The two optional lists, `shapes` and `names`, must be of the same length as `dtypes` if provided. The values at a given index `i` indicate the shape and name to use for the corresponding queue component in `dtypes`.

Args:

- `dtypes` : A list of types. The length of dtypes must equal the number of tensors in each element.
- `shapes` : Constraints on the shapes of tensors in an element: A list of shape tuples or None. This list is the same length as dtypes. If the shape of any tensors in the element are constrained, all must be; shapes can be None if the shapes should not be constrained.
- `names` : Optional list of names. If provided, the `enqueue()` and `dequeue()` methods will use dictionaries with these names as keys. Must be None or a list or tuple of the same length as `dtypes`.
- `queue_ref` : The queue reference, i.e. the output of the queue op.

Raises:

- `ValueError` : If one of the arguments is invalid.

`tf.QueueBase.dequeue_up_to(n, name=None)`

Dequeues and concatenates `n` elements from this queue.

Note This operation is not supported by all queues. If a queue does not support `DequeueUpTo`, then a `tf.errors.UnimplementedError` is raised.

This operation concatenates queue-element component tensors along the 0th dimension to make a single component tensor. If the queue has not been closed, all of the components in the dequeued tuple will have size `n` in the 0th dimension.

If the queue is closed and there are more than `0` but fewer than `n` elements remaining, then instead of raising a `tf.errorsOutOfRangeError` like `dequeue_many`, the remaining elements are returned immediately. If the queue is closed and there are `0` elements left in the queue, then a `tf.errorsOutOfRangeError` is raised just like in `dequeue_many`. Otherwise the behavior is identical to `dequeue_many`.

Args:

- `n` : A scalar `Tensor` containing the number of elements to dequeue.
- `name` : A name for the operation (optional).

Returns:

The tuple of concatenated tensors that was dequeued.

`tf.QueueBase.dtypes`

The list of dtypes for each component of a queue element.

tf.QueueBase.from_list(index, queues)

Create a queue using the queue reference from `queues[index]`.

Args:

- `index` : An integer scalar tensor that determines the input that gets selected.
- `queues` : A list of `QueueBase` objects.

Returns:

A `queueBase` object.

Raises:

- `TypeError` : When `queues` is not a list of `QueueBase` objects, or when the data types of `queues` are not all the same.
-

tf.QueueBase.name

The name of the underlying queue.

tf.QueueBase.names

The list of names for each component of a queue element.

tf.QueueBase.queue_ref

The underlying queue reference.

class tf.FIFOQueue

A queue implementation that dequeues elements in first-in-first out order.

See [tf.QueueBase](#) for a description of the methods on this class.

```
tf.FIFOQueue.__init__(capacity, dtypes, shapes=None,
names=None, shared_name=None, name='fifo_queue')
```

Creates a queue that dequeues elements in a first-in first-out order.

A `FIFOQueue` has bounded capacity; supports multiple concurrent producers and consumers; and provides exactly-once delivery.

A `FIFOQueue` holds a list of up to `capacity` elements. Each element is a fixed-length tuple of tensors whose `dtypes` are described by `dtypes`, and whose `shapes` are optionally described by the `shapes` argument.

If the `shapes` argument is specified, each component of a queue element must have the respective fixed shape. If it is unspecified, different queue elements may have different shapes, but the use of `dequeue_many` is disallowed.

Args:

- `capacity` : An integer. The upper bound on the number of elements that may be stored in this queue.
 - `dtypes` : A list of `DTType` objects. The length of `dtypes` must equal the number of tensors in each queue element.
 - `shapes` : (Optional.) A list of fully-defined `TensorShape` objects with the same length as `dtypes`, or `None`.
 - `names` : (Optional.) A list of string naming the components in the queue with the same length as `dtypes`, or `None`. If specified the dequeue methods return a dictionary with the names as keys.
 - `shared_name` : (Optional.) If non-empty, this queue will be shared under the given name across multiple sessions.
 - `name` : Optional name for the queue operation.
-

class `tf.PaddingFIFOQueue`

A FIFOQueue that supports batching variable-sized tensors by padding.

A `PaddingFIFOQueue` may contain components with dynamic shape, while also supporting `dequeue_many`. See the constructor for more details.

See `tf.QueueBase` for a description of the methods on this class.

```
tf.PaddingFIFOQueue.__init__(capacity, dtypes, shapes,
names=None, shared_name=None, name='padding_fifo_queue')
```

Creates a queue that dequeues elements in a first-in first-out order.

A `PaddingFIFOQueue` has bounded capacity; supports multiple concurrent producers and consumers; and provides exactly-once delivery.

A `PaddingFIFOQueue` holds a list of up to `capacity` elements. Each element is a fixed-length tuple of tensors whose dtypes are described by `dtypes`, and whose shapes are described by the `shapes` argument.

The `shapes` argument must be specified; each component of a queue element must have the respective shape. Shapes of fixed rank but variable size are allowed by setting any shape dimension to `None`. In this case, the inputs' shape may vary along the given dimension, and `dequeue_many` will pad the given dimension with zeros up to the maximum shape of all elements in the given batch.

Args:

- `capacity` : An integer. The upper bound on the number of elements that may be stored in this queue.
- `dtypes` : A list of `DTType` objects. The length of `dtypes` must equal the number of tensors in each queue element.
- `shapes` : A list of `TensorShape` objects, with the same length as `dtypes`. Any dimension in the `TensorShape` containing value `None` is dynamic and allows values to be enqueued with variable size in that dimension.
- `names` : (Optional.) A list of string naming the components in the queue with the same length as `dtypes`, or `None`. If specified the dequeue methods return a dictionary with the names as keys.
- `shared_name` : (Optional.) If non-empty, this queue will be shared under the given name across multiple sessions.
- `name` : Optional name for the queue operation.

Raises:

- `ValueError` : If `shapes` is not a list of shapes, or the lengths of `dtypes` and `shapes` do not match, or if `names` is specified and the lengths of `dtypes` and `names` do not match.

class `tf.RandomShuffleQueue`

A queue implementation that dequeues elements in a random order.

See `tf.QueueBase` for a description of the methods on this class.

```
tf.RandomShuffleQueue.__init__(capacity, min_after_dequeue,
                               dtypes, shapes=None, names=None, seed=None,
                               shared_name=None, name='random_shuffle_queue')
```

Create a queue that dequeues elements in a random order.

A `RandomShuffleQueue` has bounded capacity; supports multiple concurrent producers and consumers; and provides exactly-once delivery.

A `RandomShuffleQueue` holds a list of up to `capacity` elements. Each element is a fixed-length tuple of tensors whose `dtypes` are described by `dtypes`, and whose `shapes` are optionally described by the `shapes` argument.

If the `shapes` argument is specified, each component of a queue element must have the respective fixed shape. If it is unspecified, different queue elements may have different shapes, but the use of `dequeue_many` is disallowed.

The `min_after_dequeue` argument allows the caller to specify a minimum number of elements that will remain in the queue after a `dequeue` or `dequeue_many` operation completes, to ensure a minimum level of mixing of elements. This invariant is maintained by blocking those operations until sufficient elements have been enqueued. The `min_after_dequeue` argument is ignored after the queue has been closed.

Args:

- `capacity` : An integer. The upper bound on the number of elements that may be stored in this queue.
- `min_after_dequeue` : An integer (described above).
- `dtypes` : A list of `DTType` objects. The length of `dtypes` must equal the number of tensors in each queue element.
- `shapes` : (Optional.) A list of fully-defined `TensorShape` objects with the same length as `dtypes`, or `None`.
- `names` : (Optional.) A list of string naming the components in the queue with the same length as `dtypes`, or `None`. If specified the dequeue methods return a dictionary with the names as keys.
- `seed` : A Python integer. Used to create a random seed. See [set_random_seed](#) for behavior.
- `shared_name` : (Optional.) If non-empty, this queue will be shared under the given name across multiple sessions.
- `name` : Optional name for the queue operation.

Dealing with the filesystem

tf.matching_files(pattern, name=None)

Returns the set of files matching a pattern.

Note that this routine only supports wildcard characters in the basename portion of the pattern, not in the directory portion.

Args:

- `pattern` : A `Tensor` of type `string`. A (scalar) shell wildcard pattern.
- `name` : A name for the operation (optional).

Returns:

A `Tensor` of type `string`. A vector of matching filenames.

tf.read_file(filename, name=None)

Reads and outputs the entire contents of the input filename.

Args:

- `filename` : A `Tensor` of type `string`.
- `name` : A name for the operation (optional).

Returns:

A `Tensor` of type `string`.

Input pipeline

TensorFlow functions for setting up an input-prefetching pipeline. Please see the [reading data how-to](#) for context.

Beginning of an input pipeline

The "producer" functions add a queue to the graph and a corresponding `QueueRunner` for running the subgraph that fills that queue.

tf.train.match_filenames_once(pattern, name=None)

Save the list of files matching pattern, so it is only computed once.

Args:

- `pattern` : A file pattern (glob).
- `name` : A name for the operations (optional).

Returns:

A variable that is initialized to the list of files matching pattern.

tf.train.limit_epochs(tensor, num_epochs=None, name=None)

Returns tensor `num_epochs` times and then raises an `OutOfRange` error.

Args:

- `tensor` : Any `Tensor`.
- `num_epochs` : A positive integer (optional). If specified, limits the number of steps the output tensor may be evaluated.
- `name` : A name for the operations (optional).

Returns:

tensor or `OutOfRange`.

Raises:

- `ValueError` : if `num_epochs` is invalid.
-

tf.train.input_producer(input_tensor, element_shape=None, num_epochs=None, shuffle=True, seed=None, capacity=32, shared_name=None, summary_name=None, name=None)

Output the rows of `input_tensor` to a queue for an input pipeline.

Args:

- `input_tensor` : A tensor with the rows to produce. Must be at one-dimensional. Must either have a fully-defined shape, or `element_shape` must be defined.
- `element_shape` : (Optional.) A `TensorShape` representing the shape of a row of `input_tensor`, if it cannot be inferred.
- `num_epochs` : (Optional.) An integer. If specified `input_producer` produces each row of `input_tensor` `num_epochs` times before generating an `OutOfRange` error. If not specified, `input_producer` can cycle through the rows of `input_tensor` an unlimited number of times.
- `shuffle` : (Optional.) A boolean. If true, the rows are randomly shuffled within each epoch.
- `seed` : (Optional.) An integer. The seed to use if `shuffle` is true.
- `capacity` : (Optional.) The capacity of the queue to be used for buffering the input.
- `shared_name` : (Optional.) If set, this queue will be shared under the given name across multiple sessions.
- `summary_name` : (Optional.) If set, a scalar summary for the current queue size will be generated, using this name as part of the tag.
- `name` : (Optional.) A name for queue.

Returns:

A queue with the output rows. A `QueueRunner` for the queue is added to the current `QUEUE_RUNNER` collection of the current graph.

Raises:

- `ValueError` : If the shape of the input cannot be inferred from the arguments.

```
tf.train.range_input_producer(limit,
                               num_epochs=None, shuffle=True, seed=None,
                               capacity=32, shared_name=None, name=None)
```

Produces the integers from 0 to limit-1 in a queue.

Args:

- `limit` : An int32 scalar tensor.
- `num_epochs` : An integer (optional). If specified, `range_input_producer` produces each integer `num_epochs` times before generating an `OutOfRange` error. If not specified, `range_input_producer` can cycle through the integers an unlimited number of times.
- `shuffle` : Boolean. If true, the integers are randomly shuffled within each epoch.

- `seed` : An integer (optional). Seed used if shuffle == True.
- `capacity` : An integer. Sets the queue capacity.
- `shared_name` : (optional). If set, this queue will be shared under the given name across multiple sessions.
- `name` : A name for the operations (optional).

Returns:

A Queue with the output integers. A `QueueRunner` for the Queue is added to the current `Graph`'s `QUEUE_RUNNER` collection.

```
tf.train.slice_input_producer(tensor_list,
num_epochs=None, shuffle=True, seed=None,
capacity=32, shared_name=None, name=None)
```

Produces a slice of each `Tensor` in `tensor_list`.

Implemented using a Queue -- a `QueueRunner` for the Queue is added to the current `Graph`'s `QUEUE_RUNNER` collection.

Args:

- `tensor_list` : A list of `Tensor` objects. Every `Tensor` in `tensor_list` must have the same size in the first dimension.
- `num_epochs` : An integer (optional). If specified, `slice_input_producer` produces each slice `num_epochs` times before generating an `OutOfRange` error. If not specified, `slice_input_producer` can cycle through the slices an unlimited number of times.
- `shuffle` : Boolean. If true, the integers are randomly shuffled within each epoch.
- `seed` : An integer (optional). Seed used if shuffle == True.
- `capacity` : An integer. Sets the queue capacity.
- `shared_name` : (optional). If set, this queue will be shared under the given name across multiple sessions.
- `name` : A name for the operations (optional).

Returns:

A list of tensors, one for each element of `tensor_list`. If the tensor in `tensor_list` has shape `[N, a, b, ..., z]`, then the corresponding output tensor will have shape `[a, b, ..., z]`.

Raises:

- `ValueError` : if `slice_input_producer` produces nothing from `tensor_list`.

```
tf.train.string_input_producer(string_tensor,
num_epochs=None, shuffle=True, seed=None,
capacity=32, shared_name=None, name=None)
```

Output strings (e.g. filenames) to a queue for an input pipeline.

Args:

- `string_tensor` : A 1-D string tensor with the strings to produce.
- `num_epochs` : An integer (optional). If specified, `string_input_producer` produces each string from `string_tensor` `num_epochs` times before generating an `OutOfRange` error. If not specified, `string_input_producer` can cycle through the strings in `string_tensor` an unlimited number of times.
- `shuffle` : Boolean. If true, the strings are randomly shuffled within each epoch.
- `seed` : An integer (optional). Seed used if `shuffle == True`.
- `capacity` : An integer. Sets the queue capacity.
- `shared_name` : (optional). If set, this queue will be shared under the given name across multiple sessions.
- `name` : A name for the operations (optional).

Returns:

A queue with the output strings. A `QueueRunner` for the Queue is added to the current `Graph`'s `QUEUE_RUNNER` collection.

Raises:

- `ValueError` : If the `string_tensor` is a null Python list. At runtime, will fail with an assertion if `string_tensor` becomes a null tensor.

Batching at the end of an input pipeline

These functions add a queue to the graph to assemble a batch of examples, with possible shuffling. They also add a `QueueRunner` for running the subgraph that fills that queue.

Use `batch` or `batch_join` for batching examples that have already been well shuffled. Use `shuffle_batch` or `shuffle_batch_join` for examples that would benefit from additional shuffling.

Use `batch` or `shuffle_batch` if you want a single thread producing examples to batch, or if you have a single subgraph producing examples but you want to run it in N threads (where you increase N until it can keep the queue full). Use `batch_join` or `shuffle_batch_join` if you have N different subgraphs producing examples to batch and you want them run by N threads.

```
tf.train.batch(tensors, batch_size,
num_threads=1, capacity=32, enqueue_many=False,
shapes=None, dynamic_pad=False,
shared_name=None, name=None)
```

Creates batches of tensors in `tensors`.

The argument `tensors` can be a list or a dictionary of tensors. The value returned by the function will be of the same type as `tensors`.

This function is implemented using a queue. A `QueueRunner` for the queue is added to the current `Graph`'s `QUEUE_RUNNER` collection.

If `enqueue_many` is `False`, `tensors` is assumed to represent a single example. An input tensor with shape `[x, y, z]` will be output as a tensor with shape `[batch_size, x, y, z]`.

If `enqueue_many` is `True`, `tensors` is assumed to represent a batch of examples, where the first dimension is indexed by example, and all members of `tensor_list` should have the same size in the first dimension. If an input tensor has shape `[*, x, y, z]`, the output will have shape `[batch_size, x, y, z]`. The `capacity` argument controls the how long the prefetching is allowed to grow the queues.

The returned operation is a `dequeue` operation and will throw `tf.errors.OutOfRangeError` if the input queue is exhausted. If this operation is feeding another input queue, its queue runner will catch this exception, however, if this operation is used in your main thread you are responsible for catching this yourself.

N.B.: If `dynamic_pad` is `False`, you must ensure that either (i) the `shapes` argument is passed, or (ii) all of the tensors in `tensors` must have fully-defined shapes. `ValueError` will be raised if neither of these conditions holds.

If `dynamic_pad` is `True`, it is sufficient that the `rank` of the tensors is known, but individual dimensions may have shape `None`. In this case, for each enqueue the dimensions with value `None` may have a variable length; upon dequeue, the output tensors will be padded

on the right to the maximum shape of the tensors in the current minibatch. For numbers, this padding takes value 0. For strings, this padding is the empty string. See [PaddingFIFOQueue](#) for more info.

Args:

- `tensors` : The list or dictionary of tensors to enqueue.
- `batch_size` : The new batch size pulled from the queue.
- `num_threads` : The number of threads enqueueing `tensor_list`.
- `capacity` : An integer. The maximum number of elements in the queue.
- `enqueue_many` : Whether each tensor in `tensor_list` is a single example.
- `shapes` : (Optional) The shapes for each example. Defaults to the inferred shapes for `tensor_list`.
- `dynamic_pad` : Boolean. Allow variable dimensions in input shapes. The given dimensions are padded upon dequeue so that tensors within a batch have the same shapes.
- `shared_name` : (optional). If set, this queue will be shared under the given name across multiple sessions.
- `name` : (Optional) A name for the operations.

Returns:

A list or dictionary of tensors with the same types as `tensors`.

Raises:

- `ValueError` : If the `shapes` are not specified, and cannot be inferred from the elements of `tensors`.

```
tf.train.batch_join(tensors_list, batch_size,
capacity=32, enqueue_many=False, shapes=None,
dynamic_pad=False, shared_name=None,
name=None)
```

Runs a list of tensors to fill a queue to create batches of examples.

The `tensors_list` argument is a list of tuples of tensors, or a list of dictionaries of tensors. Each element in the list is treated similarly to the `tensors` argument of `tf.train.batch()`.

Enqueues a different list of tensors in different threads. Implemented using a queue -- a `QueueRunner` for the queue is added to the current `Graph`'s `QUEUE_RUNNER` collection.

`len(tensor_list)` threads will be started, with thread `i` enqueueing the tensors from `tensor_list[i]`. `tensor_list[i1][j]` must match `tensor_list[i2][j]` in type and shape, except in the first dimension if `enqueue_many` is true.

If `enqueue_many` is `False`, each `tensor_list[i]` is assumed to represent a single example. An input tensor `x` will be output as a tensor with shape `[batch_size] + x.shape`.

If `enqueue_many` is `True`, `tensor_list[i]` is assumed to represent a batch of examples, where the first dimension is indexed by example, and all members of `tensor_list[i]` should have the same size in the first dimension. The slices of any input tensor `x` are treated as examples, and the output tensors will have shape `[batch_size] + x.shape[1:]`.

The `capacity` argument controls the how long the prefetching is allowed to grow the queues.

The returned operation is a dequeue operation and will throw `tf.errorsOutOfRange` if the input queue is exhausted. If this operation is feeding another input queue, its queue runner will catch this exception, however, if this operation is used in your main thread you are responsible for catching this yourself.

N.B.: If `dynamic_pad` is `False`, you must ensure that either (i) the `shapes` argument is passed, or (ii) all of the tensors in `tensor_list` must have fully-defined shapes.

`ValueError` will be raised if neither of these conditions holds.

If `dynamic_pad` is `True`, it is sufficient that the `rank` of the tensors is known, but individual dimensions may have value `None`. In this case, for each enqueue the dimensions with value `None` may have a variable length; upon dequeue, the output tensors will be padded on the right to the maximum shape of the tensors in the current minibatch. For numbers, this padding takes value 0. For strings, this padding is the empty string. See `PaddingFIFOQueue` for more info.

Args:

- `tensor_list` : A list of tuples or dictionaries of tensors to enqueue.
- `batch_size` : An integer. The new batch size pulled from the queue.
- `capacity` : An integer. The maximum number of elements in the queue.
- `enqueue_many` : Whether each tensor in `tensor_list` is a single example.
- `shapes` : (Optional) The shapes for each example. Defaults to the inferred shapes for `tensor_list[i]`.
- `dynamic_pad` : Boolean. Allow variable dimensions in input shapes. The given dimensions are padded upon dequeue so that tensors within a batch have the same shapes.
- `shared_name` : (Optional) If set, this queue will be shared under the given name across multiple sessions.

- `name` : (Optional) A name for the operations.

Returns:

A list or dictionary of tensors with the same number and types as `tensors_list[i]`.

Raises:

- `ValueError` : If the `shapes` are not specified, and cannot be inferred from the elements of `tensor_list_list`.

```
tf.train.shuffle_batch(tensors, batch_size,
capacity, min_after_dequeue, num_threads=1,
seed=None, enqueue_many=False, shapes=None,
shared_name=None, name=None)
```

Creates batches by randomly shuffling tensors.

This function adds the following to the current `Graph`:

- A shuffling queue into which tensors from `tensors` are enqueued.
- A `dequeue_many` operation to create batches from the queue.
- A `QueueRunner` to `QUEUE_RUNNER` collection, to enqueue the tensors from `tensors`.

If `enqueue_many` is `False`, `tensors` is assumed to represent a single example. An input tensor with shape `[x, y, z]` will be output as a tensor with shape `[batch_size, x, y, z]`.

If `enqueue_many` is `True`, `tensors` is assumed to represent a batch of examples, where the first dimension is indexed by example, and all members of `tensors` should have the same size in the first dimension. If an input tensor has shape `[*, x, y, z]`, the output will have shape `[batch_size, x, y, z]`.

The `capacity` argument controls the how long the prefetching is allowed to grow the queues.

The returned operation is a `dequeue` operation and will throw `tf.errorsOutOfRangeError` if the input queue is exhausted. If this operation is feeding another input queue, its queue runner will catch this exception, however, if this operation is used in your main thread you are responsible for catching this yourself.

For example:

```
# Creates batches of 32 images and 32 labels.
image_batch, label_batch = tf.train.shuffle_batch(
    [single_image, single_label],
    batch_size=32,
    num_threads=4,
    capacity=50000,
    min_after_dequeue=10000)
```

N.B.: You must ensure that either (i) the `shapes` argument is passed, or (ii) all of the tensors in `tensors` must have fully-defined shapes. `ValueError` will be raised if neither of these conditions holds.

Args:

- `tensors` : The list or dictionary of tensors to enqueue.
- `batch_size` : The new batch size pulled from the queue.
- `capacity` : An integer. The maximum number of elements in the queue.
- `min_after_dequeue` : Minimum number elements in the queue after a dequeue, used to ensure a level of mixing of elements.
- `num_threads` : The number of threads enqueueing `tensor_list`.
- `seed` : Seed for the random shuffling within the queue.
- `enqueue_many` : Whether each tensor in `tensor_list` is a single example.
- `shapes` : (Optional) The shapes for each example. Defaults to the inferred shapes for `tensor_list`.
- `shared_name` : (Optional) If set, this queue will be shared under the given name across multiple sessions.
- `name` : (Optional) A name for the operations.

Returns:

A list or dictionary of tensors with the types as `tensors`.

Raises:

- `ValueError` : If the `shapes` are not specified, and cannot be inferred from the elements of `tensors`.

```
tf.train.shuffle_batch_join(tensor_list,
batch_size, capacity, min_after_dequeue,
seed=None, enqueue_many=False, shapes=None,
shared_name=None, name=None)
```

Create batches by randomly shuffling tensors.

The `tensors_list` argument is a list of tuples of tensors, or a list of dictionaries of tensors. Each element in the list is treated similarly to the `tensors` argument of `tf.train.shuffle_batch()`.

This version enqueues a different list of tensors in different threads. It adds the following to the current `Graph`:

- A shuffling queue into which tensors from `tensors_list` are enqueued.
- A `dequeue_many` operation to create batches from the queue.
- A `QueueRunner` to `QUEUE_RUNNER` collection, to enqueue the tensors from `tensors_list`.

`len(tensors_list)` threads will be started, with thread `i` enqueueing the tensors from `tensors_list[i]`. `tensors_list[i1][j]` must match `tensors_list[i2][j]` in type and shape, except in the first dimension if `enqueue_many` is true.

If `enqueue_many` is `False`, each `tensors_list[i]` is assumed to represent a single example. An input tensor with shape `[x, y, z]` will be output as a tensor with shape `[batch_size, x, y, z]`.

If `enqueue_many` is `True`, `tensors_list[i]` is assumed to represent a batch of examples, where the first dimension is indexed by example, and all members of `tensors_list[i]` should have the same size in the first dimension. If an input tensor has shape `[*, x, y, z]`, the output will have shape `[batch_size, x, y, z]`.

The `capacity` argument controls the how long the prefetching is allowed to grow the queues.

The returned operation is a `dequeue` operation and will throw `tf.errorsOutOfRangeError` if the input queue is exhausted. If this operation is feeding another input queue, its queue runner will catch this exception, however, if this operation is used in your main thread you are responsible for catching this yourself.

Args:

- `tensors_list` : A list of tuples or dictionaries of tensors to enqueue.
- `batch_size` : An integer. The new batch size pulled from the queue.
- `capacity` : An integer. The maximum number of elements in the queue.
- `min_after_dequeue` : Minimum number elements in the queue after a `dequeue`, used to ensure a level of mixing of elements.
- `seed` : Seed for the random shuffling within the queue.
- `enqueue_many` : Whether each tensor in `tensor_list_list` is a single example.
- `shapes` : (Optional) The shapes for each example. Defaults to the inferred shapes for `tensors_list[i]`.

- `shared_name` : (optional). If set, this queue will be shared under the given name across multiple sessions.
- `name` : (Optional) A name for the operations.

Returns:

A list or dictionary of tensors with the same number and types as `tensors_list[i]`.

Raises:

- `ValueError` : If the `shapes` are not specified, and cannot be inferred from the elements of `tensors_list`.

Data IO (Python functions)

[TOC]

Data IO (Python Functions)

A TFRecords file represents a sequence of (binary) strings. The format is not random access, so it is suitable for streaming large amounts of data but not suitable if fast sharding or other non-sequential access is desired.

`class tf.python_io.TFRecordWriter`

A class to write records to a TFRecords file.

This class implements `__enter__` and `__exit__`, and can be used in `with` blocks like a normal file.

`tf.python_io.TFRecordWriter.__init__(path)`

Opens file `path` and creates a `TFRecordWriter` writing to it.

Args:

- `path` : The path to the TFRecords file.

Raises:

- `IOError` : If `path` cannot be opened for writing.
-

`tf.python_io.TFRecordWriter.write(record)`

Write a string record to the file.

Args:

- `record` : str
-

tf.python_io.TFRecordWriter.close()

Close the file.

tf.python_io.tf_record_iterator(path)

An iterator that read the records from a TFRecords file.

Args:

- `path` : The path to the TFRecords file.

Yields:

Strings.

Raises:

- `IOError` : If `path` cannot be opened for reading.
-

TFRecords Format Details

A TFRecords file contains a sequence of strings with CRC hashes. Each record has the format

```
uint64 length
uint32 masked_crc32_of_length
byte   data[length]
uint32 masked_crc32_of_data
```

and the records are concatenated together to produce the file. The CRC32s are [described here](#), and the mask of a CRC is

```
masked_crc = ((crc >> 15) | (crc << 17)) + 0xa282ead8ul
```

Neural Network

Note: Functions taking `Tensor` arguments can also take anything accepted by `tf.convert_to_tensor`.

[TOC]

Activation Functions

The activation ops provide different types of nonlinearities for use in neural networks. These include smooth nonlinearities (`sigmoid`, `tanh`, `elu`, `softplus`, and `softsign`), continuous but not everywhere differentiable functions (`relu`, `relu6`, and `relu_x`), and random regularization (`dropout`).

All activation ops apply componentwise, and produce a tensor of the same shape as the input tensor.

`tf.nn.relu(features, name=None)`

Computes rectified linear: `max(features, 0)`.

Args:

- `features` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `name` : A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `features`.

`tf.nn.relu6(features, name=None)`

Computes Rectified Linear 6: `min(max(features, 0), 6)`.

Args:

- `features` : A `Tensor` with type `float`, `double`, `int32`, `int64`, `uint8`, `int16`, or

```
int8 .
```

- `name` : A name for the operation (optional).

Returns:

A `Tensor` with the same type as `features`.

tf.nn.elu(features, name=None)

Computes exponential linear: $\exp(\text{features}) - 1$ if < 0 , `features` otherwise.

See [Fast and Accurate Deep Network Learning by Exponential Linear Units \(ELUs\)](#)

Args:

- `features` : A `Tensor`. Must be one of the following types: `float32`, `float64`.
- `name` : A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `features`.

tf.nn.softplus(features, name=None)

Computes softplus: $\log(\exp(\text{features}) + 1)$.

Args:

- `features` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `name` : A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `features`.

tf.nn.softsign(features, name=None)

Computes softsign: $\text{features} / (\text{abs}(\text{features}) + 1)$.

Args:

- `features` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `name` : A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `features`.

```
tf.nn.dropout(x, keep_prob, noise_shape=None, seed=None, name=None)
```

Computes dropout.

With probability `keep_prob`, outputs the input element scaled up by `1 / keep_prob`, otherwise outputs `0`. The scaling is so that the expected sum is unchanged.

By default, each element is kept or dropped independently. If `noise_shape` is specified, it must be `broadcastable` to the shape of `x`, and only dimensions with `noise_shape[i] == shape(x)[i]` will make independent decisions. For example, if `shape(x) = [k, 1, m, n]` and `noise_shape = [k, 1, 1, n]`, each batch and channel component will be kept independently and each row and column will be kept or not kept together.

Args:

- `x` : A tensor.
- `keep_prob` : A scalar `Tensor` with the same type as `x`. The probability that each element is kept.
- `noise_shape` : A 1-D `Tensor` of type `int32`, representing the shape for randomly generated keep/drop flags.
- `seed` : A Python integer. Used to create random seeds. See `set_random_seed` for behavior.
- `name` : A name for this operation (optional).

Returns:

A Tensor of the same shape of `x`.

Raises:

- `ValueError` : If `keep_prob` is not in `(0, 1]`.

tf.nn.bias_add(value, bias, data_format=None, name=None)

Adds `bias` to `value`.

This is (mostly) a special case of `tf.add` where `bias` is restricted to 1-D. Broadcasting is supported, so `value` may have any number of dimensions. Unlike `tf.add`, the type of `bias` is allowed to differ from `value` in the case where both types are quantized.

Args:

- `value` : A `Tensor` with type `float`, `double`, `int64`, `int32`, `uint8`, `int16`, `int8`, `complex64`, or `complex128`.
- `bias` : A 1-D `Tensor` with size matching the last dimension of `value`. Must be the same type as `value` unless `value` is a quantized type, in which case a different quantized type may be used.
- `data_format` : A string. 'NHWC' and 'NCHW' are supported.
- `name` : A name for the operation (optional).

Returns:

A `Tensor` with the same type as `value`.

tf.sigmoid(x, name=None)

Computes sigmoid of `x` element-wise.

Specifically, $y = 1 / (1 + \exp(-x))$.

Args:

- `x` : A `Tensor` with type `float`, `double`, `int32`, `complex64`, `int64`, or `qint32`.
- `name` : A name for the operation (optional).

Returns:

A `Tensor` with the same type as `x` if `x.dtype != qint32` otherwise the return type is `quint8`.

tf.tanh(x, name=None)

Computes hyperbolic tangent of `x` element-wise.

Args:

- `x` : A Tensor with type `float`, `double`, `int32`, `complex64`, `int64`, or `qint32`.
- `name` : A name for the operation (optional).

Returns:

A Tensor with the same type as `x` if `x.dtype != qint32` otherwise the return type is `quint8`.

Convolution

The convolution ops sweep a 2-D filter over a batch of images, applying the filter to each window of each image of the appropriate size. The different ops trade off between generic vs. specific filters:

- `conv2d` : Arbitrary filters that can mix channels together.
- `depthwise_conv2d` : Filters that operate on each channel independently.
- `separable_conv2d` : A depthwise spatial filter followed by a pointwise filter.

Note that although these ops are called "convolution", they are strictly speaking "cross-correlation" since the filter is combined with an input window without reversing the filter. For details, see [the properties of cross-correlation](#).

The filter is applied to image patches of the same size as the filter and strided according to the `strides` argument. `strides = [1, 1, 1, 1]` applies the filter to a patch at every offset, `strides = [1, 2, 2, 1]` applies the filter to every other image patch in each dimension, etc.

Ignoring channels for the moment, and assume that the 4-D `input` has shape `[batch, in_height, in_width, ...]` and the 4-D `filter` has shape `[filter_height, filter_width, ...]`, then the spatial semantics of the convolution ops are as follows: first, according to the padding scheme chosen as `'SAME'` or `'VALID'`, the output size and the padding pixels are computed. For the `'SAME'` padding, the output height and width are computed as:

```
out_height = ceil(float(in_height) / float(strides[1]))
out_width = ceil(float(in_width) / float(strides[2]))
```

and the padding on the top and left are computed as:

```

pad_along_height = ((out_height - 1) * strides[1] +
                     filter_height - in_height)
pad_along_width = ((out_width - 1) * strides[2] +
                     filter_width - in_width)
pad_top = pad_along_height / 2
pad_left = pad_along_width / 2

```

Note that the division by 2 means that there might be cases when the padding on both sides (top vs bottom, right vs left) are off by one. In this case, the bottom and right sides always get the one additional padded pixel. For example, when `pad_along_height` is 5, we pad 2 pixels at the top and 3 pixels at the bottom. Note that this is different from existing libraries such as cuDNN and Caffe, which explicitly specify the number of padded pixels and always pad the same number of pixels on both sides.

For the '`VALID`' padding, the output height and width are computed as:

```

out_height = ceil(float(in_height - filter_height + 1) / float(strides[1]))
out_width  = ceil(float(in_width - filter_width + 1) / float(strides[2]))

```

and the padding values are always zero. The output is then computed as

```

output[b, i, j, :] =
    sum_{di, dj} input[b, strides[1] * i + di - pad_top,
                      strides[2] * j + dj - pad_left, ...] *
    filter[di, dj, ...]

```

where any value outside the original input image region are considered zero (i.e. we pad zero values around the border of the image).

Since `input` is 4-D, each `input[b, i, j, :]` is a vector. For `conv2d`, these vectors are multiplied by the `filter[di, dj, :, :]` matrices to produce new vectors. For `depthwise_conv_2d`, each scalar component `input[b, i, j, k]` is multiplied by a vector `filter[di, dj, k]`, and all the vectors are concatenated.

```
tf.nn.conv2d(input, filter, strides, padding,
use_cudnn_on_gpu=None, data_format=None,
name=None)
```

Computes a 2-D convolution given 4-D `input` and `filter` tensors.

Given an input tensor of shape `[batch, in_height, in_width, in_channels]` and a filter / kernel tensor of shape `[filter_height, filter_width, in_channels, out_channels]`, this op performs the following:

1. Flattens the filter to a 2-D matrix with shape `[filter_height * filter_width * in_channels, output_channels]`.
2. Extracts image patches from the input tensor to form a *virtual* tensor of shape `[batch, out_height, out_width, filter_height * filter_width * in_channels]`.
3. For each patch, right-multiplies the filter matrix and the image patch vector.

In detail, with the default NHWC format,

```
output[b, i, j, k] =
    sum_{di, dj, q} input[b, strides[1] * i + di, strides[2] * j + dj, q] *
                    filter[di, dj, q, k]
```

Must have `strides[0] = strides[3] = 1`. For the most common case of the same horizontal and vertical strides, `strides = [1, stride, stride, 1]`.

Args:

- `input` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`.
- `filter` : A `Tensor`. Must have the same type as `input`.
- `strides` : A list of `ints`. 1-D of length 4. The stride of the sliding window for each dimension of `input`. Must be in the same order as the dimension specified with `format`.
- `padding` : A `string` from: "SAME", "VALID". The type of padding algorithm to use.
- `use_cudnn_on_gpu` : An optional `bool`. Defaults to `True`.
- `data_format` : An optional `string` from: "NHWC", "NCHW". Defaults to "NHWC". Specify the data format of the input and output data. With the default format "NHWC", the data is stored in the order of:

```
[batch, in_height, in_width, in_channels].
```

Alternatively, the format could be "NCHW", the data storage order of:

```
[batch, in_channels, in_height, in_width].
```

- `name` : A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`.

tf.nn.depthwise_conv2d(input, filter, strides, padding, name=None)

Depthwise 2-D convolution.

Given an input tensor of shape `[batch, in_height, in_width, in_channels]` and a filter tensor of shape `[filter_height, filter_width, in_channels, channel_multiplier]` containing `in_channels` convolutional filters of depth 1, `depthwise_conv2d` applies a different filter to each input channel (expanding from 1 channel to `channel_multiplier` channels for each), then concatenates the results together. The output has `in_channels * channel_multiplier` channels.

In detail,

```
output[b, i, j, k * channel_multiplier + q] =
    sum_{di, dj} input[b, strides[1] * i + di, strides[2] * j + dj, k] *
        filter[di, dj, k, q]
```

Must have `strides[0] = strides[3] = 1`. For the most common case of the same horizontal and vertical strides, `strides = [1, stride, stride, 1]`.

Args:

- `input` : 4-D with shape `[batch, in_height, in_width, in_channels]` .
- `filter` : 4-D with shape `[filter_height, filter_width, in_channels, channel_multiplier]` .
- `strides` : 1-D of size 4. The stride of the sliding window for each dimension of `input` .
- `padding` : A string, either `'VALID'` or `'SAME'` . The padding algorithm. See the [comment here](#)
- `name` : A name for this operation (optional).

Returns:

A 4-D `Tensor` of shape `[batch, out_height, out_width, in_channels * channel_multiplier]`.

tf.nn.separable_conv2d(input, depthwise_filter, pointwise_filter, strides, padding, name=None)

2-D convolution with separable filters.

Performs a depthwise convolution that acts separately on channels followed by a pointwise convolution that mixes channels. Note that this is separability between dimensions [1, 2] and 3, not spatial separability between dimensions 1 and 2.

In detail,

```
output[b, i, j, k] = sum_{di, dj, q, r}
    input[b, strides[1] * i + di, strides[2] * j + dj, q] *
    depthwise_filter[di, dj, q, r] *
    pointwise_filter[0, 0, q * channel_multiplier + r, k]
```

`strides` controls the strides for the depthwise convolution only, since the pointwise convolution has implicit strides of [1, 1, 1, 1]. Must have `strides[0] = strides[3] = 1`. For the most common case of the same horizontal and vertical strides, `strides = [1, stride, stride, 1]`.

Args:

- `input` : 4-D `Tensor` with shape [batch, in_height, in_width, in_channels].
- `depthwise_filter` : 4-D `Tensor` with shape [filter_height, filter_width, in_channels, channel_multiplier]. Contains `in_channels` convolutional filters of depth 1.
- `pointwise_filter` : 4-D `Tensor` with shape [1, 1, channel_multiplier * in_channels, out_channels]. Pointwise filter to mix channels after `depthwise_filter` has convolved spatially.
- `strides` : 1-D of size 4. The strides for the depthwise convolution for each dimension of `input`.
- `padding` : A string, either '`VALID`' or '`SAME`'. The padding algorithm. See the [comment here](#)
- `name` : A name for this operation (optional).

Returns:

A 4-D `Tensor` of shape [batch, out_height, out_width, out_channels].

Raises:

- `ValueError` : If `channel_multiplier * in_channels > out_channels`, which means that the separable convolution is overparameterized.

`tf.nn.atrous_conv2d(value, filters, rate, padding, name=None)`

Atrous convolution (a.k.a. convolution with holes or dilated convolution).

Computes a 2-D atrous convolution, also known as convolution with holes or dilated convolution, given 4-D `value` and `filters` tensors. If the `rate` parameter is equal to one, it performs regular 2-D convolution. If the `rate` parameter is greater than one, it performs convolution with holes, sampling the input values every `rate` pixels in the `height` and `width` dimensions. This is equivalent to convolving the input with a set of upsampled filters, produced by inserting `rate - 1` zeros between two consecutive values of the filters along the `height` and `width` dimensions, hence the name atrous convolution or convolution with holes (the French word *trous* means holes in English).

More specifically:

```
output[b, i, j, k] = sum_{di, dj, q} filters[di, dj, q, k] *
    value[b, i + rate * di, j + rate * dj, q]
```

Atrous convolution allows us to explicitly control how densely to compute feature responses in fully convolutional networks. Used in conjunction with bilinear interpolation, it offers an alternative to `conv2d_transpose` in dense prediction tasks such as semantic image segmentation, optical flow computation, or depth estimation. It also allows us to effectively enlarge the field of view of filters without increasing the number of parameters or the amount of computation.

For a description of atrous convolution and how it can be used for dense feature extraction, please see: [Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs](#). The same operation is investigated further in [Multi-Scale Context Aggregation by Dilated Convolutions](#). Previous works that effectively use atrous convolution in different ways are, among others, [OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks](#) and [Fast Image Scanning with Deep Max-Pooling Convolutional Neural Networks] (<http://arxiv.org/abs/1302.1700>). Atrous convolution is also closely related to the so-called noble identities in multi-rate signal processing.

There are many different ways to implement atrous convolution (see the refs above). The implementation here reduces

```
atrous_conv2d(value, filters, rate, padding=padding)
```

to the following three operations:

```

paddings = ...
net = space_to_batch(value, paddings, block_size=rate)
net = conv2d(net, filters, strides=[1, 1, 1, 1], padding="VALID")
crops = ...
net = batch_to_space(net, crops, block_size=rate)

```

Advanced usage. Note the following optimization: A sequence of `atrous_conv2d` operations with identical `rate` parameters, 'SAME' `padding`, and filters with odd heights/ widths:

```

net = atrous_conv2d(net, filters1, rate, padding="SAME")
net = atrous_conv2d(net, filters2, rate, padding="SAME")
...
net = atrous_conv2d(net, filtersK, rate, padding="SAME")

```

can be equivalently performed cheaper in terms of computation and memory as:

```

pad = ... # padding so that the input dims are multiples of rate
net = space_to_batch(net, paddings=pad, block_size=rate)
net = conv2d(net, filters1, strides=[1, 1, 1, 1], padding="SAME")
net = conv2d(net, filters2, strides=[1, 1, 1, 1], padding="SAME")
...
net = conv2d(net, filtersK, strides=[1, 1, 1, 1], padding="SAME")
net = batch_to_space(net, crops=pad, block_size=rate)

```

because a pair of consecutive `space_to_batch` and `batch_to_space` ops with the same `block_size` cancel out when their respective `paddings` and `crops` inputs are identical.

Args:

- `value` : A 4-D `Tensor` of type `float`. It needs to be in the default "NHWC" format. Its `shape` is `[batch, in_height, in_width, in_channels]`.
- `filters` : A 4-D `Tensor` with the same type as `value` and shape `[filter_height, filter_width, in_channels, out_channels]`. `filters`' `in_channels` dimension must match that of `value`. Atrous convolution is equivalent to standard convolution with upsampled filters with effective height `filter_height + (filter_height - 1) * (rate - 1)` and effective width `filter_width + (filter_width - 1) * (rate - 1)`, produced by inserting `rate - 1` zeros along consecutive elements across the `filters`' spatial dimensions.
- `rate` : A positive `int32`. The stride with which we sample input values across the `height` and `width` dimensions. Equivalently, the rate by which we upsample the filter values by inserting zeros across the `height` and `width` dimensions. In the literature, the same parameter is sometimes called `input stride` or `dilation`.
- `padding` : A string, either `'VALID'` or `'SAME'`. The padding algorithm.

- `name` : Optional name for the returned tensor.

Returns:

A `Tensor` with the same type as `value`.

Raises:

- `ValueError` : If input/output depth does not match `filters`' shape, or if padding is other than `'VALID'` or `'SAME'`.

```
tf.nn.conv2d_transpose(value, filter,
output_shape, strides, padding='SAME',
name=None)
```

The transpose of `conv2d`.

This operation is sometimes called "deconvolution" after [Deconvolutional Networks](#), but is actually the transpose (gradient) of `conv2d` rather than an actual deconvolution.

Args:

- `value` : A 4-D `Tensor` of type `float` and shape `[batch, height, width, in_channels]`.
- `filter` : A 4-D `Tensor` with the same type as `value` and shape `[height, width, output_channels, in_channels]`. `filter`'s `in_channels` dimension must match that of `value`.
- `output_shape` : A 1-D `Tensor` representing the output shape of the deconvolution op.
- `strides` : A list of ints. The stride of the sliding window for each dimension of the input tensor.
- `padding` : A string, either `'VALID'` or `'SAME'`. The padding algorithm. See the [comment here](#)
- `name` : Optional name for the returned tensor.

Returns:

A `Tensor` with the same type as `value`.

Raises:

- `ValueError` : If input/output depth does not match `filter`'s shape, or if padding is other than `'VALID'` or `'SAME'`.

tf.nn.conv3d(input, filter, strides, padding, name=None)

Computes a 3-D convolution given 5-D `input` and `filter` tensors.

In signal processing, cross-correlation is a measure of similarity of two waveforms as a function of a time-lag applied to one of them. This is also known as a sliding dot product or sliding inner-product.

Our Conv3D implements a form of cross-correlation.

Args:

- `input` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`, `int16`, `int8`, `complex64`, `complex128`, `qint8`, `quint8`, `qint32`, `half`. Shape `[batch, in_depth, in_height, in_width, in_channels]`.
- `filter` : A `Tensor`. Must have the same type as `input`. Shape `[filter_depth, filter_height, filter_width, in_channels, out_channels]`. `in_channels` must match between `input` and `filter`.
- `strides` : A list of `ints` that has length `>= 5`. 1-D tensor of length 5. The stride of the sliding window for each dimension of `input`. Must have `strides[0] = strides[4] = 1`.
- `padding` : A `string` from: `"SAME"`, `"VALID"`. The type of padding algorithm to use.
- `name` : A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`.

Pooling

The pooling ops sweep a rectangular window over the input tensor, computing a reduction operation for each window (average, max, or max with argmax). Each pooling op uses rectangular windows of size `ksize` separated by offset `strides`. For example, if `strides` is all ones every window is used, if `strides` is all twos every other window is used in each dimension, etc.

In detail, the output is

```
output[i] = reduce(value[strides * i:strides * i + ksize])
```

where the indices also take into consideration the padding values. Please refer to the `Convolution` section for details about the padding calculation.

```
tf.nn.avg_pool(value, ksize, strides, padding,
data_format='NHWC', name=None)
```

Performs the average pooling on the input.

Each entry in `output` is the mean of the corresponding size `ksize` window in `value`.

Args:

- `value` : A 4-D `Tensor` of shape `[batch, height, width, channels]` and type `float32`, `float64`, `qint8`, `quint8`, or `qint32`.
- `ksize` : A list of ints that has length ≥ 4 . The size of the window for each dimension of the input tensor.
- `strides` : A list of ints that has length ≥ 4 . The stride of the sliding window for each dimension of the input tensor.
- `padding` : A string, either `'VALID'` or `'SAME'`. The padding algorithm. See the [comment here](#)
- `data_format` : A string. `'NHWC'` and `'NCHW'` are supported.
- `name` : Optional name for the operation.

Returns:

A `Tensor` with the same type as `value`. The average pooled output tensor.

```
tf.nn.max_pool(value, ksize, strides, padding,
data_format='NHWC', name=None)
```

Performs the max pooling on the input.

Args:

- `value` : A 4-D `Tensor` with shape `[batch, height, width, channels]` and type `tf.float32`.
- `ksize` : A list of ints that has length ≥ 4 . The size of the window for each dimension of the input tensor.
- `strides` : A list of ints that has length ≥ 4 . The stride of the sliding window for each dimension of the input tensor.
- `padding` : A string, either `'VALID'` or `'SAME'`. The padding algorithm. See the [comment here](#)
- `data_format` : A string. `'NHWC'` and `'NCHW'` are supported.

- `name` : Optional name for the operation.

Returns:

A `Tensor` with type `tf.float32`. The max pooled output tensor.

`tf.nn.max_pool_with_argmax(input, ksize, strides, padding, Targmax=None, name=None)`

Performs max pooling on the input and outputs both max values and indices.

The indices in `argmax` are flattened, so that a maximum value at position `[b, y, x, c]` becomes flattened index `((b * height + y) * width + x) * channels + c`.

Args:

- `input` : A `Tensor` of type `float32`. 4-D with shape `[batch, height, width, channels]`. Input to pool over.
- `ksize` : A list of `ints` that has length `>= 4`. The size of the window for each dimension of the input tensor.
- `strides` : A list of `ints` that has length `>= 4`. The stride of the sliding window for each dimension of the input tensor.
- `padding` : A `string` from: `"SAME", "VALID"`. The type of padding algorithm to use.
- `Targmax` : An optional `tf.DType` from: `tf.int32, tf.int64`. Defaults to `tf.int64`.
- `name` : A name for the operation (optional).

Returns:

A tuple of `Tensor` objects (`output, argmax`).

- `output` : A `Tensor` of type `float32`. The max pooled output tensor.
- `argmax` : A `Tensor` of type `Targmax`. 4-D. The flattened indices of the max values chosen for each output.

`tf.nn.avg_pool3d(input, ksize, strides, padding, name=None)`

Performs 3D average pooling on the input.

Args:

- `input` : A `Tensor`. Must be one of the following types: `float32, float64, int64,`

- int32 , uint8 , uint16 , int16 , int8 , complex64 , complex128 , qint8 , quint8 , qint32 , half . Shape [batch, depth, rows, cols, channels] tensor to pool over.
- **ksize** : A list of ints that has length ≥ 5 . 1-D tensor of length 5. The size of the window for each dimension of the input tensor. Must have `ksize[0] = ksize[1] = 1` .
- **strides** : A list of ints that has length ≥ 5 . 1-D tensor of length 5. The stride of the sliding window for each dimension of `input` . Must have `strides[0] = strides[4] = 1` .
- **padding** : A string from: "SAME", "VALID" . The type of padding algorithm to use.
- **name** : A name for the operation (optional).

Returns:

A Tensor . Has the same type as `input` . The average pooled output tensor.

tf.nn.max_pool3d(input, ksize, strides, padding, name=None)

Performs 3D max pooling on the input.

Args:

- **input** : A Tensor . Must be one of the following types: float32 , float64 , int64 , int32 , uint8 , uint16 , int16 , int8 , complex64 , complex128 , qint8 , quint8 , qint32 , half . Shape [batch, depth, rows, cols, channels] tensor to pool over.
- **ksize** : A list of ints that has length ≥ 5 . 1-D tensor of length 5. The size of the window for each dimension of the input tensor. Must have `ksize[0] = ksize[1] = 1` .
- **strides** : A list of ints that has length ≥ 5 . 1-D tensor of length 5. The stride of the sliding window for each dimension of `input` . Must have `strides[0] = strides[4] = 1` .
- **padding** : A string from: "SAME", "VALID" . The type of padding algorithm to use.
- **name** : A name for the operation (optional).

Returns:

A Tensor . Has the same type as `input` . The max pooled output tensor.

Morphological filtering

Morphological operators are non-linear filters used in image processing.

[Greyscale morphological dilation] ([https://en.wikipedia.org/wiki/Dilation_\(morphology\)](https://en.wikipedia.org/wiki/Dilation_(morphology))) is the max-sum counterpart of standard sum-product convolution:

```

output[b, y, x, c] =
    max_{dy, dx} input[b,
                          strides[1] * y + rates[1] * dy,
                          strides[2] * x + rates[2] * dx,
                          c] +
    filter[dy, dx, c]

```

The `filter` is usually called structuring function. Max-pooling is a special case of greyscale morphological dilation when the filter assumes all-zero values (a.k.a. flat structuring function).

[Greyscale morphological erosion] ([https://en.wikipedia.org/wiki/Erosion_\(morphology\)](https://en.wikipedia.org/wiki/Erosion_(morphology))) is the min-sum counterpart of standard sum-product convolution:

```

output[b, y, x, c] =
    min_{dy, dx} input[b,
                          strides[1] * y - rates[1] * dy,
                          strides[2] * x - rates[2] * dx,
                          c] -
    filter[dy, dx, c]

```

Dilation and erosion are dual to each other. The dilation of the input signal `f` by the structuring signal `g` is equal to the negation of the erosion of `-f` by the reflected `g`, and vice versa.

Striding and padding is carried out in exactly the same way as in standard convolution. Please refer to the `Convolution` section for details.

`tf.nn.dilation2d(input, filter, strides, rates, padding, name=None)`

Computes the grayscale dilation of 4-D `input` and 3-D `filter` tensors.

The `input` tensor has shape `[batch, in_height, in_width, depth]` and the `filter` tensor has shape `[filter_height, filter_width, depth]`, i.e., each input channel is processed independently of the others with its own structuring function. The `output` tensor has shape `[batch, out_height, out_width, depth]`. The spatial dimensions of the output tensor depend on the `padding` algorithm. We currently only support the default "NHWC" `data_format`.

In detail, the grayscale morphological 2-D dilation is the max-sum correlation (for consistency with `conv2d`, we use unmirrored filters):

```

output[b, y, x, c] =
    max_{dy, dx} input[b,
        strides[1] * y + rates[1] * dy,
        strides[2] * x + rates[2] * dx,
        c] +
    filter[dy, dx, c]

```

Max-pooling is a special case when the filter has size equal to the pooling kernel size and contains all zeros.

Args:

- `input` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, `half`.
- `filter` : A `Tensor`. Must have the same type as `input`.
- `strides` : A list of `ints` that has length `>= 4`.
- `rates` : A list of `ints` that has length `>= 4`.
- `padding` : A `string` from: `"SAME"`, `"VALID"`.
- `name` : A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`.

`tf.nn.erosion2d(value, kernel, strides, rates, padding, name=None)`

Computes the grayscale erosion of 4-D `value` and 3-D `kernel` tensors.

The `value` tensor has shape `[batch, in_height, in_width, depth]` and the `kernel` tensor has shape `[kernel_height, kernel_width, depth]`, i.e., each input channel is processed independently of the others with its own structuring function. The `output` tensor has shape `[batch, out_height, out_width, depth]`. The spatial dimensions of the output tensor depend on the `padding` algorithm. We currently only support the default "NHWC" `data_format`.

In detail, the grayscale morphological 2-D erosion is given by:

```

output[b, y, x, c] =
    min_{dy, dx} value[b,
        strides[1] * y - rates[1] * dy,
        strides[2] * x - rates[2] * dx,
        c] -
    kernel[dy, dx, c]

```

Duality: The erosion of `value` by the `kernel` is equal to the negation of the dilation of `-value` by the reflected `kernel`.

Args:

- `value` : A `Tensor`. 4-D with shape `[batch, in_height, in_width, depth]`.
- `kernel` : A `Tensor`. Must have the same type as `value`. 3-D with shape `[kernel_height, kernel_width, depth]`.
- `strides` : A list of `ints` that has length `>= 4`. 1-D of length 4. The stride of the sliding window for each dimension of the input tensor. Must be: `[1, stride_height, stride_width, 1]`.
- `rates` : A list of `ints` that has length `>= 4`. 1-D of length 4. The input stride for atrous morphological dilation. Must be: `[1, rate_height, rate_width, 1]`.
- `padding` : A `string` from: `"SAME", "VALID"`. The type of padding algorithm to use.
- `name` : A name for the operation (optional). If not specified "erosion2d" is used.

Returns:

A `Tensor`. Has the same type as `value`. 4-D with shape `[batch, out_height, out_width, depth]`.

Raises:

- `ValueError` : If the `value` `depth` does not match `kernel` ' shape, or if padding is other than `'VALID'` or `'SAME'`.

Normalization

Normalization is useful to prevent neurons from saturating when inputs may have varying scale, and to aid generalization.

```
tf.nn.l2_normalize(x, dim, epsilon=1e-12,
name=None)
```

Normalizes along dimension `dim` using an L2 norm.

For a 1-D tensor with `dim = 0`, computes

```
output = x / sqrt(max(sum(x**2), epsilon))
```

For `x` with more dimensions, independently normalizes each 1-D slice along dimension `dim`.

Args:

- `x` : A `Tensor`.
- `dim` : Dimension along which to normalize.
- `epsilon` : A lower bound value for the norm. Will use `sqrt(epsilon)` as the divisor if `norm < sqrt(epsilon)`.
- `name` : A name for this operation (optional).

Returns:

A `Tensor` with the same shape as `x`.

```
tf.nn.local_response_normalization(input,
depth_radius=None, bias=None, alpha=None,
beta=None, name=None)
```

Local Response Normalization.

The 4-D `input` tensor is treated as a 3-D array of 1-D vectors (along the last dimension), and each vector is normalized independently. Within a given vector, each component is divided by the weighted, squared sum of inputs within `depth_radius`. In detail,

```
sqr_sum[a, b, c, d] =
    sum(input[a, b, c, d - depth_radius : d + depth_radius + 1] ** 2)
output = input / (bias + alpha * sqr_sum) ** beta
```

For details, see [Krizhevsky et al., ImageNet classification with deep convolutional neural networks (NIPS 2012)] (<http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks>).

Args:

- `input` : A `Tensor` of type `float32`. 4-D.
- `depth_radius` : An optional `int`. Defaults to `5`. 0-D. Half-width of the 1-D normalization window.
- `bias` : An optional `float`. Defaults to `1`. An offset (usually positive to avoid dividing by 0).
- `alpha` : An optional `float`. Defaults to `1`. A scale factor, usually positive.
- `beta` : An optional `float`. Defaults to `0.5`. An exponent.

- `name` : A name for the operation (optional).

Returns:

A `Tensor` of type `float32`.

`tf.nn.sufficient_statistics(x, axes, shift=None, keep_dims=False, name=None)`

Calculate the sufficient statistics for the mean and variance of `x`.

These sufficient statistics are computed using the one pass algorithm on an input that's optionally shifted. See:

https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Computing_shifted_data

Args:

- `x` : A `Tensor`.
- `axes` : Array of ints. Axes along which to compute mean and variance.
- `shift` : A `Tensor` containing the value by which to shift the data for numerical stability, or `None` if no shift is to be performed. A shift close to the true mean provides the most numerically stable results.
- `keep_dims` : produce statistics with the same dimensionality as the input.
- `name` : Name used to scope the operations that compute the sufficient stats.

Returns:

Four `Tensor` objects of the same type as `x`:

- the count (number of elements to average over).
- the (possibly shifted) sum of the elements in the array.
- the (possibly shifted) sum of squares of the elements in the array.
- the shift by which the mean must be corrected or `None` if `shift` is `None`.

`tf.nn.normalize_moments(counts, mean_ss, variance_ss, shift, name=None)`

Calculate the mean and variance of based on the sufficient statistics.

Args:

- `counts` : A `Tensor` containing a the total count of the data (one value).

- `mean_ss` : A `Tensor` containing the mean sufficient statistics: the (possibly shifted) sum of the elements to average over.
- `variance_ss` : A `Tensor` containing the variance sufficient statistics: the (possibly shifted) squared sum of the data to compute the variance over.
- `shift` : A `Tensor` containing the value by which the data is shifted for numerical stability, or `None` if no shift was performed.
- `name` : Name used to scope the operations that compute the moments.

Returns:

Two `Tensor` objects: `mean` and `variance`.

`tf.nn.moments(x, axes, shift=None, name=None, keep_dims=False)`

Calculate the mean and variance of `x`.

The mean and variance are calculated by aggregating the contents of `x` across `axes`. If `x` is 1-D and `axes = [0]` this is just the mean and variance of a vector.

When using these moments for batch normalization (see `tf.nn.batch_normalization`):

- for so-called "global normalization", used with convolutional filters with shape `[batch, height, width, depth]`, pass `axes=[0, 1, 2]`.
- for simple batch normalization pass `axes=[0]` (batch only).

Args:

- `x` : A `Tensor`.
- `axes` : array of ints. Axes along which to compute mean and variance.
- `shift` : A `Tensor` containing the value by which to shift the data for numerical stability, or `None` if no shift is to be performed. A shift close to the true mean provides the most numerically stable results.
- `keep_dims` : produce moments with the same dimensionality as the input.
- `name` : Name used to scope the operations that compute the moments.

Returns:

Two `Tensor` objects: `mean` and `variance`.

Losses

The loss ops measure error between two tensors, or between a tensor and zero. These can be used for measuring accuracy of a network in a regression task or for regularization purposes (weight decay).

tf.nn.l2_loss(t, name=None)

L2 Loss.

Computes half the L2 norm of a tensor without the `sqrt` :

```
output = sum(t ** 2) / 2
```

Args:

- `t` : A `Tensor`. Must be one of the following types: `float32`, `float64`, `int64`, `int32`, `uint8`, `uint16`, `int16`, `int8`, `complex64`, `complex128`, `qint8`, `quint8`, `qint32`, `half`. Typically 2-D, but may have any dimensions.
- `name` : A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `t`. 0-D.

Classification

TensorFlow provides several operations that help you perform classification.

tf.nn.sigmoid_cross_entropy_with_logits(logits, targets, name=None)

Computes sigmoid cross entropy given `logits`.

Measures the probability error in discrete classification tasks in which each class is independent and not mutually exclusive. For instance, one could perform multilabel classification where a picture can contain both an elephant and a dog at the same time.

For brevity, let `x = logits`, `z = targets`. The logistic loss is

```

z * -log(sigmoid(x)) + (1 - z) * -log(1 - sigmoid(x))
= z * -log(1 / (1 + exp(-x))) + (1 - z) * -log(exp(-x) / (1 + exp(-x)))
= z * log(1 + exp(-x)) + (1 - z) * (-log(exp(-x)) + log(1 + exp(-x)))
= z * log(1 + exp(-x)) + (1 - z) * (x + log(1 + exp(-x)))
= (1 - z) * x + log(1 + exp(-x))
= x - x * z + log(1 + exp(-x))

```

For $x < 0$, to avoid overflow in $\exp(-x)$, we reformulate the above

```

x - x * z + log(1 + exp(-x))
= log(exp(x)) - x * z + log(1 + exp(-x))
= -x * z + log(1 + exp(x))

```

Hence, to ensure stability and avoid overflow, the implementation uses this equivalent formulation

```
max(x, 0) - x * z + log(1 + exp(-abs(x)))
```

`logits` and `targets` must have the same type and shape.

Args:

- `logits` : A `Tensor` of type `float32` or `float64`.
- `targets` : A `Tensor` of the same type and shape as `logits`.
- `name` : A name for the operation (optional).

Returns:

A `Tensor` of the same shape as `logits` with the componentwise logistic losses.

Raises:

- `ValueError` : If `logits` and `targets` do not have the same shape.

tf.nn.softmax(logits, name=None)

Computes softmax activations.

For each batch `i` and class `j` we have

```
softmax[i, j] = exp(logits[i, j]) / sum(exp(logits[i]))
```

Args:

- `logits` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`. 2-D with shape `[batch_size, num_classes]`.
- `name` : A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `logits`. Same shape as `logits`.

`tf.nn.log_softmax(logits, name=None)`

Computes log softmax activations.

For each batch `i` and class `j` we have

```
logsoftmax[i, j] = logits[i, j] - log(sum(exp(logits[i])))
```

Args:

- `logits` : A `Tensor`. Must be one of the following types: `half`, `float32`, `float64`. 2-D with shape `[batch_size, num_classes]`.
- `name` : A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `logits`. Same shape as `logits`.

`tf.nn.softmax_cross_entropy_with_logits(logits, labels, name=None)`

Computes softmax cross entropy between `logits` and `labels`.

Measures the probability error in discrete classification tasks in which the classes are mutually exclusive (each entry is in exactly one class). For example, each CIFAR-10 image is labeled with one and only one label: an image can be a dog or a truck, but not both.

NOTE: While the classes are mutually exclusive, their probabilities need not be. All that is required is that each row of `labels` is a valid probability distribution. If they are not, the computation of the gradient will be incorrect.

If using exclusive `labels` (wherein one and only one class is true at a time), see `sparse_softmax_cross_entropy_with_logits`.

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.

`logits` and `labels` must have the same shape `[batch_size, num_classes]` and the same `dtype` (either `float32` or `float64`).

Args:

- `logits` : Unscaled log probabilities.
- `labels` : Each row `labels[i]` must be a valid probability distribution.
- `name` : A name for the operation (optional).

Returns:

A 1-D `Tensor` of length `batch_size` of the same type as `logits` with the softmax cross entropy loss.

`tf.nn.sparse_softmax_cross_entropy_with_logits`(`logits`, `labels`, `name=None`)

Computes sparse softmax cross entropy between `logits` and `labels`.

Measures the probability error in discrete classification tasks in which the classes are mutually exclusive (each entry is in exactly one class). For example, each CIFAR-10 image is labeled with one and only one label: an image can be a dog or a truck, but not both.

NOTE: For this operation, the probability of a given label is considered exclusive. That is, soft classes are not allowed, and the `labels` vector must provide a single specific index for the true class for each row of `logits` (each minibatch entry). For soft softmax classification with a probability distribution for each entry, see `softmax_cross_entropy_with_logits`.

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.

`logits` must have the shape `[batch_size, num_classes]` and `dtype float32` or `float64`.

`labels` must have the shape `[batch_size]` and `dtype int32` or `int64`.

Args:

- `logits` : Unscaled log probabilities.
- `labels` : Each entry `labels[i]` must be an index in `[0, num_classes)`. Other values will result in a loss of 0, but incorrect gradient computations.
- `name` : A name for the operation (optional).

Returns:

A 1-D `Tensor` of length `batch_size` of the same type as `logits` with the softmax cross entropy loss.

`tf.nn.weighted_cross_entropy_with_logits(logits, targets, pos_weight, name=None)`

Computes a weighted cross entropy.

This is like `sigmoid_cross_entropy_with_logits()` except that `pos_weight`, allows one to trade off recall and precision by up- or down-weighting the cost of a positive error relative to a negative error.

The usual cross-entropy cost is defined as:

$$\text{targets} \cdot -\log(\text{sigmoid}(\text{logits})) + (1 - \text{targets}) \cdot -\log(1 - \text{sigmoid}(\text{logits}))$$

The argument `pos_weight` is used as a multiplier for the positive targets:

$$\text{targets} \cdot -\log(\text{sigmoid}(\text{logits})) \cdot \text{pos_weight} + (1 - \text{targets}) \cdot -\log(1 - \text{sigmoid}(\text{logits}))$$

For brevity, let `x = logits`, `z = targets`, `q = pos_weight`. The loss is:

$$\begin{aligned} & qz \cdot -\log(\text{sigmoid}(x)) + (1 - z) \cdot -\log(1 - \text{sigmoid}(x)) \\ &= qz \cdot -\log(1 / (1 + \exp(-x))) + (1 - z) \cdot -\log(\exp(-x) / (1 + \exp(-x))) \\ &= qz \cdot \log(1 + \exp(-x)) + (1 - z) \cdot (-\log(\exp(-x)) + \log(1 + \exp(-x))) \\ &= qz \cdot \log(1 + \exp(-x)) + (1 - z) \cdot (x + \log(1 + \exp(-x))) \\ &= (1 - z) \cdot x + (qz + 1 - z) \cdot \log(1 + \exp(-x)) \\ &= (1 - z) \cdot x + (1 + (q - 1) \cdot z) \cdot \log(1 + \exp(-x)) \end{aligned}$$

Setting `l = (1 + (q - 1) * z)`, to ensure stability and avoid overflow, the implementation uses

$$(1 - z) \cdot x + l \cdot (\log(1 + \exp(-\text{abs}(x))) + \max(-x, 0))$$

`logits` and `targets` must have the same type and shape.

Args:

- `logits` : A `Tensor` of type `float32` or `float64`.
- `targets` : A `Tensor` of the same type and shape as `logits`.
- `pos_weight` : A coefficient to use on the positive examples.
- `name` : A name for the operation (optional).

Returns:

A `Tensor` of the same shape as `logits` with the componentwise weighted logistic losses.

Raises:

- `ValueError` : If `logits` and `targets` do not have the same shape.

Embeddings

TensorFlow provides library support for looking up values in embedding tensors.

```
tf.nn.embedding_lookup(params, ids,
partition_strategy='mod', name=None,
validate_indices=True)
```

Looks up `ids` in a list of embedding tensors.

This function is used to perform parallel lookups on the list of tensors in `params`. It is a generalization of `tf.gather()`, where `params` is interpreted as a partition of a larger embedding tensor.

If `len(params) > 1`, each element `id` of `ids` is partitioned between the elements of `params` according to the `partition_strategy`. In all strategies, if the id space does not evenly divide the number of partitions, each of the first `(max_id + 1) % len(params)` partitions will be assigned one more id.

If `partition_strategy` is "mod", we assign each id to partition `p = id % len(params)`. For instance, 13 ids are split across 5 partitions as: `[[0, 5, 10], [1, 6, 11], [2, 7, 12], [3, 8], [4, 9]]`

If `partition_strategy` is "div", we assign ids to partitions in a contiguous manner. In this case, 13 ids are split across 5 partitions as: `[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10], [11, 12]]`

The results of the lookup are concatenated into a dense tensor. The returned tensor has shape `shape(ids) + shape(params)[1:]`.

Args:

- `params` : A list of tensors with the same type and which can be concatenated along dimension 0. Each `Tensor` must be appropriately sized for the given `partition_strategy`.
- `ids` : A `Tensor` with type `int32` or `int64` containing the ids to be looked up in `params`.
- `partition_strategy` : A string specifying the partitioning strategy, relevant if `len(params) > 1`. Currently `"div"` and `"mod"` are supported. Default is `"mod"`.
- `name` : A name for the operation (optional).
- `validate_indices` : Whether or not to validate gather indices.

Returns:

A `Tensor` with the same type as the tensors in `params`.

Raises:

- `ValueError` : If `params` is empty.

```
tf.nn.embedding_lookup_sparse(params, sp_ids,
sp_weights, partition_strategy='mod',
name=None, combiner='mean')
```

Computes embeddings for the given ids and weights.

This op assumes that there is at least one id for each row in the dense tensor represented by `sp_ids` (i.e. there are no rows with empty features), and that all the indices of `sp_ids` are in canonical row-major order.

It also assumes that all id values lie in the range $[0, p_0]$, where p_0 is the sum of the size of `params` along dimension 0.

Args:

- `params` : A single tensor representing the complete embedding tensor, or a list of P tensors all of same shape except for the first dimension, representing sharded embedding tensors.
- `sp_ids` : N x M SparseTensor of int64 ids (typically from `FeatureValueToId`), where N is typically batch size and M is arbitrary.
- `sp_weights` : either a SparseTensor of float / double weights, or None to indicate all weights should be taken to be 1. If specified, `sp_weights` must have exactly the same

shape and indices as sp_ids.

- `partition_strategy` : A string specifying the partitioning strategy, relevant if `len(params) > 1`. Currently "div" and "mod" are supported. Default is "mod". See `tf.nn.embedding_lookup` for more details.
- `name` : Optional name for the op.
- `combiner` : A string specifying the reduction op. Currently "mean", "sqrt" and "sum" are supported. "sum" computes the weighted sum of the embedding results for each row. "mean" is the weighted sum divided by the total weight. "sqrt" is the weighted sum divided by the square root of the sum of the squares of the weights.

Returns:

A dense tensor representing the combined embeddings for the sparse ids. For each row in the dense tensor represented by sp_ids, the op looks up the embeddings for all ids in that row, multiplies them by the corresponding weight, and combines these embeddings as specified.

In other words, if `shape(combined_params) = [p0, p1, ..., pm]` and `shape(sp_ids) = shape(sp_weights) = [d0, d1, ..., dn]` then `shape(output) = [d0, d1, ..., dn-1, p1, ..., pm]`.

For instance, if params is a 10x20 matrix, and sp_ids / sp_weights are

```
[0, 0]: id 1, weight 2.0
[0, 1]: id 3, weight 0.5
[1, 0]: id 0, weight 1.0
[2, 3]: id 1, weight 3.0
```

with `combiner="mean"`, then the output will be a 3x20 matrix where `output[0, :] = (params[1, :] 2.0 + params[3, :] 0.5) / (2.0 + 0.5)` `output[1, :] = params[0, :] 1.0` `output[2, :] = params[1, :] 3.0`

Raises:

- `TypeError` : If sp_ids is not a SparseTensor, or if sp_weights is neither None nor SparseTensor.
- `ValueError` : If combiner is not one of {"mean", "sqrt", "sum"}.

Recurrent Neural Networks

TensorFlow provides a number of methods for constructing Recurrent Neural Networks. Most accept an `RNNCell`-subclassed object (see the documentation for `tf.nn.rnn_cell`).

```
tf.nn.dynamic_rnn(cell, inputs,
sequence_length=None, initial_state=None,
dtype=None, parallel_iterations=None,
swap_memory=False, time_major=False,
scope=None)
```

Creates a recurrent neural network specified by RNNCell `cell`.

This function is functionally identical to the function `rnn` above, but performs fully dynamic unrolling of `inputs`.

Unlike `rnn`, the input `inputs` is not a Python list of `Tensors`. Instead, it is a single `Tensor` where the maximum time is either the first or second dimension (see the parameter `time_major`). The corresponding output is a single `Tensor` having the same number of time steps and batch size.

The parameter `sequence_length` is required and dynamic calculation is automatically performed.

Args:

- `cell` : An instance of RNNCell.
- `inputs` : The RNN inputs. If `time_major == False` (default), this must be a tensor of shape: `[batch_size, max_time, input_size]`. If `time_major == True`, this must be a tensor of shape: `[max_time, batch_size, input_size]`.
- `sequence_length` : (optional) An int32/int64 vector sized `[batch_size]`.
- `initial_state` : (optional) An initial state for the RNN. If `cell.state_size` is an integer, this must be a tensor of appropriate type and shape `[batch_size x cell.state_size]`. If `cell.state_size` is a tuple, this should be a tuple of tensors having shapes `[batch_size, s]` for `s` in `cell.state_size`.
- `dtype` : (optional) The data type for the initial state. Required if `initial_state` is not provided.
- `parallel_iterations` : (Default: 32). The number of iterations to run in parallel. Those operations which do not have any temporal dependency and can be run in parallel, will be. This parameter trades off time for space. Values $>> 1$ use more memory but take less time, while smaller values use less memory but computations take longer.
- `swap_memory` : Transparently swap the tensors produced in forward inference but needed for back prop from GPU to CPU. This allows training RNNs which would typically not fit on a single GPU, with very minimal (or no) performance penalty.
- `time_major` : The shape format of the `inputs` and `outputs` Tensors. If true, these Tensors must be shaped `[max_time, batch_size, depth]`. If false, these Tensors must be shaped `[batch_size, max_time, depth]`. Using `time_major = True` is a bit more

efficient because it avoids transposes at the beginning and end of the RNN calculation. However, most TensorFlow data is batch-major, so by default this function accepts input and emits output in batch-major form.

- `scope` : VariableScope for the created subgraph; defaults to "RNN".

Returns:

A pair (`outputs`, `state`) where:

- `outputs` : The RNN output `Tensor`. If `time_major == False` (default), this will be a `Tensor` shaped:

```
`[batch_size, max_time, cell.output_size]`.
```

If `time_major == True`, this will be a `Tensor` shaped:

```
`[max_time, batch_size, cell.output_size]`.
```

- `state` : The final state. If `cell.state_size` is a `Tensor`, this will be shaped `[batch_size, cell.state_size]`. If it is a tuple, this be a tuple with shapes `[batch_size, s]` for `s` in `cell.state_size`.

Raises:

- `TypeError` : If `cell` is not an instance of `RNNCell`.
- `ValueError` : If `inputs` is None or an empty list.

```
tf.nn.rnn(cell, inputs, initial_state=None, dtype=None, sequence_length=None, scope=None)
```

Creates a recurrent neural network specified by `RNNCell cell`.

The simplest form of RNN network generated is:

```
state = cell.zero_state(...) outputs = [] for input in inputs: output, state = cell(input_, state)
outputs.append(output) return (outputs, state)
```

However, a few other options are available:

An initial state can be provided. If the `sequence_length` vector is provided, dynamic calculation is performed. This method of calculation does not compute the RNN steps past the maximum sequence length of the minibatch (thus saving computational time), and properly propagates the state at an example's sequence length to the final state output.

The dynamic calculation performed is, at time t for batch row b , $(\text{output}, \text{state})(b, t) = (\text{t} \geq \text{sequence_length}(b)) ? (\text{zeros}(\text{cell.output_size}), \text{states}(b, \text{sequence_length}(b) - 1)) : \text{cell}(\text{input}(b, t), \text{state}(b, t - 1))$

Args:

- `cell` : An instance of RNNCell.
- `inputs` : A length T list of inputs, each a tensor of shape [batch_size, input_size].
- `initial_state` : (optional) An initial state for the RNN. If `cell.state_size` is an integer, this must be a tensor of appropriate type and shape [batch_size x cell.state_size] . If `cell.state_size` is a tuple, this should be a tuple of tensors having shapes [batch_size, s] for s in `cell.state_size` .
- `dtype` : (optional) The data type for the initial state. Required if `initial_state` is not provided.
- `sequence_length` : Specifies the length of each sequence in inputs. An int32 or int64 vector (tensor) size [batch_size] , values in [0, T] .
- `scope` : VariableScope for the created subgraph; defaults to "RNN".

Returns:

A pair (outputs, state) where:

- outputs is a length T list of outputs (one for each input)
- state is the final state

Raises:

- `TypeError` : If `cell` is not an instance of RNNCell.
- `ValueError` : If `inputs` is `None` or an empty list, or if the input depth (column size) cannot be inferred from inputs via shape inference.

```
tf.nn.state_saving_rnn(cell, inputs,
state_saver, state_name, sequence_length=None,
scope=None)
```

RNN that accepts a state saver for time-truncated RNN calculation.

Args:

- `cell` : An instance of `RNNCell` .
- `inputs` : A length T list of inputs, each a tensor of shape [batch_size, input_size] .
- `state_saver` : A state saver object with methods `state` and `save_state` .

- `state_name` : Python string or tuple of strings. The name to use with the state_saver. If the cell returns tuples of states (i.e., `cell.state_size` is a tuple) then `state_name` should be a tuple of strings having the same length as `cell.state_size`. Otherwise it should be a single string.
- `sequence_length` : (optional) An int32/int64 vector size [batch_size]. See the documentation for `rnn()` for more details about `sequence_length`.
- `scope` : VariableScope for the created subgraph; defaults to "RNN".

Returns:

A pair (`outputs`, `state`) where: `outputs` is a length `T` list of outputs (one for each input) `states` is the final state

Raises:

- `TypeError` : If `cell` is not an instance of `RNNCell`.
- `ValueError` : If `inputs` is `None` or an empty list, or if the arity and type of `state_name` does not match that of `cell.state_size`.

```
tf.nn.bidirectional_rnn(cell_fw, cell_bw,
inputs, initial_state_fw=None,
initial_state_bw=None, dtype=None,
sequence_length=None, scope=None)
```

Creates a bidirectional recurrent neural network.

Similar to the unidirectional case above (`rnn`) but takes input and builds independent forward and backward RNNs with the final forward and backward outputs depth-concatenated, such that the output will have the format [time][batch][`cell_fw.output_size + cell_bw.output_size`]. The `input_size` of forward and backward cell must match. The initial state for both directions is zero by default (but can be set optionally) and no intermediate states are ever returned -- the network is fully unrolled for the given (passed in) length(s) of the sequence(s) or completely unrolled if length(s) is not given.

Args:

- `cell_fw` : An instance of `RNNCell`, to be used for forward direction.
- `cell_bw` : An instance of `RNNCell`, to be used for backward direction.
- `inputs` : A length `T` list of inputs, each a tensor of shape [batch_size, `input_size`].
- `initial_state_fw` : (optional) An initial state for the forward RNN. This must be a tensor of appropriate type and shape `[batch_size x cell_fw.state_size]`. If

`cell_fw.state_size` is a tuple, this should be a tuple of tensors having shapes `[batch_size, s]` for `s` in `cell_fw.state_size`.

- `initial_state_bw` : (optional) Same as for `initial_state_fw`, but using the corresponding properties of `cell_bw`.
- `dtype` : (optional) The data type for the initial state. Required if either of the initial states are not provided.
- `sequence_length` : (optional) An int32/int64 vector, size `[batch_size]`, containing the actual lengths for each of the sequences.
- `scope` : VariableScope for the created subgraph; defaults to "BiRNN"

Returns:

A tuple (`outputs`, `output_state_fw`, `output_state_bw`) where: `outputs` is a length T list of outputs (one for each input), which are depth-concatenated forward and backward outputs. `output_state_fw` is the final state of the forward rnn. `output_state_bw` is the final state of the backward rnn.

Raises:

- `TypeError` : If `cell_fw` or `cell_bw` is not an instance of `RNNCell`.
- `ValueError` : If `inputs` is None or an empty list.

Evaluation

The evaluation ops are useful for measuring the performance of a network. Since they are nondifferentiable, they are typically used at evaluation time.

`tf.nn.top_k(input, k=1, sorted=True, name=None)`

Finds values and indices of the `k` largest entries for the last dimension.

If the input is a vector (rank-1), finds the `k` largest entries in the vector and outputs their values and indices as vectors. Thus `values[j]` is the `j`-th largest entry in `input`, and its index is `indices[j]`.

For matrices (resp. higher rank input), computes the top `k` entries in each row (resp. vector along the last dimension). Thus,

```
values.shape = indices.shape = input.shape[:-1] + [k]
```

If two elements are equal, the lower-index element appears first.

Args:

- `input` : 1-D or higher `Tensor` with last dimension at least `k`.
- `k` : 0-D `int32 Tensor`. Number of top elements to look for along the last dimension (along each row for matrices).
- `sorted` : If true the resulting `k` elements will be sorted by the values in descending order.
- `name` : Optional name for the operation.

Returns:

- `values` : The `k` largest elements along each last dimensional slice.
- `indices` : The indices of `values` within the last dimension of `input`.

`tf.nn.in_top_k(predictions, targets, k, name=None)`

Says whether the targets are in the top `k` predictions.

This outputs a `batch_size` bool array, an entry `out[i]` is `true` if the prediction for the target class is among the top `k` predictions among all predictions for example `i`. Note that the behavior of `InTopK` differs from the `TopK` op in its handling of ties; if multiple classes have the same prediction value and straddle the top-`k` boundary, all of those classes are considered to be in the top `k`.

More formally, let

$\{predictions_i\}$ be the predictions for all classes for example `i`, $\{targets_i\}$ be the target class for example `i`, $\{out_i\}$ be the output for example `i`,

$$out_i = predictions_{i,targets_i} \in TopKIncludingTies(predictions_i)$$

Args:

- `predictions` : A `Tensor` of type `float32`. A `batch_size` \times `classes` `tensor`.
- `targets` : A `Tensor`. Must be one of the following types: `int32`, `int64`. A `batch_size` vector of class ids.
- `k` : An `int`. Number of top elements to look at for computing precision.
- `name` : A name for the operation (optional).

Returns:

A `Tensor` of type `bool`. Computed Precision at `k` as a `bool Tensor`.

Candidate Sampling

Do you want to train a multiclass or multilabel model with thousands or millions of output classes (for example, a language model with a large vocabulary)? Training with a full Softmax is slow in this case, since all of the classes are evaluated for every training example. Candidate Sampling training algorithms can speed up your step times by only considering a small randomly-chosen subset of contrastive classes (called candidates) for each batch of training examples.

See our [Candidate Sampling Algorithms Reference] ([..../extras/candidate_sampling.pdf](#))

Sampled Loss Functions

TensorFlow provides the following sampled loss functions for faster training.

```
tf.nn.nce_loss(weights, biases, inputs,
labels, num_sampled, num_classes, num_true=1,
sampled_values=None,
remove_accidental_hits=False,
partition_strategy='mod', name='nce_loss')
```

Computes and returns the noise-contrastive estimation training loss.

See [Noise-contrastive estimation: A new estimation principle for unnormalized statistical models] (<http://www.jmlr.org/proceedings/papers/v9/gutmann10a/gutmann10a.pdf>). Also see our [Candidate Sampling Algorithms Reference] ([..../extras/candidate_sampling.pdf](#))

Note: In the case where `num_true > 1`, we assign to each target class the target probability $1 / \text{num_true}$ so that the target probabilities sum to 1 per-example.

Note: It would be useful to allow a variable number of target classes per example. We hope to provide this functionality in a future release. For now, if you have a variable number of target classes, you can pad them out to a constant number by either repeating them or by padding with an otherwise unused class.

Args:

- `weights` : A `Tensor` of shape `[num_classes, dim]`, or a list of `Tensor` objects whose concatenation along dimension 0 has shape `[num_classes, dim]`. The (possibly-

partitioned) class embeddings.

- `biases` : A `Tensor` of shape `[num_classes]`. The class biases.
- `inputs` : A `Tensor` of shape `[batch_size, dim]`. The forward activations of the input network.
- `labels` : A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- `num_sampled` : An `int`. The number of classes to randomly sample per batch.
- `num_classes` : An `int`. The number of possible classes.
- `num_true` : An `int`. The number of target classes per training example.
- `sampled_values` : a tuple of (`sampled_candidates`, `true_expected_count`, `sampled_expected_count`) returned by a `*_candidate_sampler` function. (if `None`, we default to `log_uniform_candidate_sampler`)
- `remove_accidental_hits` : A `bool`. Whether to remove "accidental hits" where a sampled class equals one of the target classes. If set to `True`, this is a "Sampled Logistic" loss instead of NCE, and we are learning to generate log-odds instead of log probabilities. See our [Candidate Sampling Algorithms Reference] (../../extras/candidate_sampling.pdf). Default is `False`.
- `partition_strategy` : A string specifying the partitioning strategy, relevant if `len(weights) > 1`. Currently `"div"` and `"mod"` are supported. Default is `"mod"`. See `tf.nn.embedding_lookup` for more details.
- `name` : A name for the operation (optional).

Returns:

A `batch_size` 1-D tensor of per-example NCE losses.

```
tf.nn.sampled_softmax_loss(weights, biases,
inputs, labels, num_sampled, num_classes,
num_true=1, sampled_values=None,
remove_accidental_hits=True,
partition_strategy='mod',
name='sampled_softmax_loss')
```

Computes and returns the sampled softmax training loss.

This is a faster way to train a softmax classifier over a huge number of classes.

This operation is for training only. It is generally an underestimate of the full softmax loss.

At inference time, you can compute full softmax probabilities with the expression

```
tf.nn.softmax(tf.matmul(inputs, tf.transpose(weights)) + biases).
```

See our [Candidate Sampling Algorithms Reference] (../../extras/candidate_sampling.pdf)

Also see Section 3 of [Jean et al., 2014 \(pdf\)](#) for the math.

Args:

- `weights` : A `Tensor` of shape `[num_classes, dim]` , or a list of `Tensor` objects whose concatenation along dimension 0 has shape `[num_classes, dim]`. The (possibly-sharded) class embeddings.
- `biases` : A `Tensor` of shape `[num_classes]` . The class biases.
- `inputs` : A `Tensor` of shape `[batch_size, dim]` . The forward activations of the input network.
- `labels` : A `Tensor` of type `int64` and shape `[batch_size, num_true]` . The target classes. Note that this format differs from the `labels` argument of `nn.softmax_cross_entropy_with_logits` .
- `num_sampled` : An `int` . The number of classes to randomly sample per batch.
- `num_classes` : An `int` . The number of possible classes.
- `num_true` : An `int` . The number of target classes per training example.
- `sampled_values` : a tuple of (`sampled_candidates` , `true_expected_count` , `sampled_expected_count`) returned by a `*_candidate_sampler` function. (if `None`, we default to `log_uniform_candidate_sampler`)
- `remove_accidental_hits` : A `bool` . whether to remove "accidental hits" where a sampled class equals one of the target classes. Default is `True`.
- `partition_strategy` : A string specifying the partitioning strategy, relevant if `len(weights) > 1` . Currently `"div"` and `"mod"` are supported. Default is `"mod"` . See `tf.nn.embedding_lookup` for more details.
- `name` : A name for the operation (optional).

Returns:

A `batch_size` 1-D tensor of per-example sampled softmax losses.

Candidate Samplers

TensorFlow provides the following samplers for randomly sampling candidate classes when using one of the sampled loss functions above.

```
tf.nn.uniform_candidate_sampler(true_classes,
                                num_true, num_sampled, unique,
                                range_max, seed=None, name=None)
```

Samples a set of classes using a uniform base distribution.

This operation randomly samples a tensor of sampled classes (`sampled_candidates`) from the range of integers `[0, range_max]`.

The elements of `sampled_candidates` are drawn without replacement (if `unique=True`) or with replacement (if `unique=False`) from the base distribution.

The base distribution for this operation is the uniform distribution over the range of integers `[0, range_max]`.

In addition, this operation returns tensors `true_expected_count` and `sampled_expected_count` representing the number of times each of the target classes (`true_classes`) and the sampled classes (`sampled_candidates`) is expected to occur in an average tensor of sampled classes. These values correspond to $q(y|x)$ defined in [this document](#). If `unique=True`, then these are post-rejection probabilities and we compute them approximately.

Args:

- `true_classes` : A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- `num_true` : An `int`. The number of target classes per training example.
- `num_sampled` : An `int`. The number of classes to randomly sample per batch.
- `unique` : A `bool`. Determines whether all sampled classes in a batch are unique.
- `range_max` : An `int`. The number of possible classes.
- `seed` : An `int`. An operation-specific seed. Default is 0.
- `name` : A name for the operation (optional).

Returns:

- `sampled_candidates` : A tensor of type `int64` and shape `[num_sampled]`. The sampled classes.
- `true_expected_count` : A tensor of type `float`. Same shape as `true_classes`. The expected counts under the sampling distribution of each of `true_classes`.
- `sampled_expected_count` : A tensor of type `float`. Same shape as `sampled_candidates`. The expected counts under the sampling distribution of each of `sampled_candidates`.

```
tf.nn.log_uniform_candidate_sampler(true_classes, num_true, num_sampled, unique, range_max,
seed=None, name=None)
```

Samples a set of classes using a log-uniform (Zipfian) base distribution.

This operation randomly samples a tensor of sampled classes (`sampled_candidates`) from the range of integers `[0, range_max]`.

The elements of `sampled_candidates` are drawn without replacement (if `unique=True`) or with replacement (if `unique=False`) from the base distribution.

The base distribution for this operation is an approximately log-uniform or Zipfian distribution:

$$P(\text{class}) = (\log(\text{class} + 2) - \log(\text{class} + 1)) / \log(\text{range_max} + 1)$$

This sampler is useful when the target classes approximately follow such a distribution - for example, if the classes represent words in a lexicon sorted in decreasing order of frequency. If your classes are not ordered by decreasing frequency, do not use this op.

In addition, this operation returns tensors `true_expected_count` and `sampled_expected_count` representing the number of times each of the target classes (`true_classes`) and the sampled classes (`sampled_candidates`) is expected to occur in an average tensor of sampled classes. These values correspond to $Q(y|x)$ defined in [this document](#). If `unique=True`, then these are post-rejection probabilities and we compute them approximately.

Args:

- `true_classes` : A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- `num_true` : An `int`. The number of target classes per training example.
- `num_sampled` : An `int`. The number of classes to randomly sample per batch.
- `unique` : A `bool`. Determines whether all sampled classes in a batch are unique.
- `range_max` : An `int`. The number of possible classes.
- `seed` : An `int`. An operation-specific seed. Default is 0.
- `name` : A name for the operation (optional).

Returns:

- `sampled_candidates` : A tensor of type `int64` and shape `[num_sampled]`. The sampled classes.
- `true_expected_count` : A tensor of type `float`. Same shape as `true_classes`. The expected counts under the sampling distribution of each of `true_classes`.
- `sampled_expected_count` : A tensor of type `float`. Same shape as `sampled_candidates`. The expected counts under the sampling distribution of each of `sampled_candidates`.

```
tf.nn.learned_unigram_candidate_sampler(true_classes, num_true, num_sampled, unique, range_max, seed=None, name=None)
```

Samples a set of classes from a distribution learned during training.

This operation randomly samples a tensor of sampled classes (`sampled_candidates`) from the range of integers `[0, range_max]`.

The elements of `sampled_candidates` are drawn without replacement (if `unique=True`) or with replacement (if `unique=False`) from the base distribution.

The base distribution for this operation is constructed on the fly during training. It is a unigram distribution over the target classes seen so far during training. Every integer in `[0, range_max]` begins with a weight of 1, and is incremented by 1 each time it is seen as a target class. The base distribution is not saved to checkpoints, so it is reset when the model is reloaded.

In addition, this operation returns tensors `true_expected_count` and `sampled_expected_count` representing the number of times each of the target classes (`true_classes`) and the sampled classes (`sampled_candidates`) is expected to occur in an average tensor of sampled classes. These values correspond to $Q(y|x)$ defined in [this document](#). If `unique=True`, then these are post-rejection probabilities and we compute them approximately.

Args:

- `true_classes` : A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- `num_true` : An `int`. The number of target classes per training example.
- `num_sampled` : An `int`. The number of classes to randomly sample per batch.
- `unique` : A `bool`. Determines whether all sampled classes in a batch are unique.
- `range_max` : An `int`. The number of possible classes.
- `seed` : An `int`. An operation-specific seed. Default is 0.
- `name` : A name for the operation (optional).

Returns:

- `sampled_candidates` : A tensor of type `int64` and shape `[num_sampled]`. The sampled classes.
- `true_expected_count` : A tensor of type `float`. Same shape as `true_classes`. The expected counts under the sampling distribution of each of `true_classes`.
- `sampled_expected_count` : A tensor of type `float`. Same shape as `sampled_candidates`.

The expected counts under the sampling distribution of each of `sampled_candidates`.

```
tf.nn.fixed_unigram_candidate_sampler(true_classes, num_true, num_sampled, unique, range_max,
vocab_file='', distortion=1.0,
num_reserved_ids=0, num_shards=1, shard=0,
unigrams=(), seed=None, name=None)
```

Samples a set of classes using the provided (fixed) base distribution.

This operation randomly samples a tensor of sampled classes (`sampled_candidates`) from the range of integers `[0, range_max]`.

The elements of `sampled_candidates` are drawn without replacement (if `unique=True`) or with replacement (if `unique=False`) from the base distribution.

The base distribution is read from a file or passed in as an in-memory array. There is also an option to skew the distribution by applying a distortion power to the weights.

In addition, this operation returns tensors `true_expected_count` and `sampled_expected_count` representing the number of times each of the target classes (`true_classes`) and the sampled classes (`sampled_candidates`) is expected to occur in an average tensor of sampled classes. These values correspond to $q(y|x)$ defined in [this document](#). If `unique=True`, then these are post-rejection probabilities and we compute them approximately.

Args:

- `true_classes` : A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- `num_true` : An `int`. The number of target classes per training example.
- `num_sampled` : An `int`. The number of classes to randomly sample per batch.
- `unique` : A `bool`. Determines whether all sampled classes in a batch are unique.
- `range_max` : An `int`. The number of possible classes.
- `vocab_file` : Each valid line in this file (which should have a CSV-like format) corresponds to a valid word ID. IDs are in sequential order, starting from `num_reserved_ids`. The last entry in each line is expected to be a value corresponding to the count or relative probability. Exactly one of `vocab_file` and `unigrams` needs to be passed to this operation.
- `distortion` : The distortion is used to skew the unigram probability distribution. Each weight is first raised to the distortion's power before adding to the internal unigram

distribution. As a result, `distortion = 1.0` gives regular unigram sampling (as defined by the vocab file), and `distortion = 0.0` gives a uniform distribution.

- `num_reserved_ids` : Optionally some reserved IDs can be added in the range `[0, num_reserved_ids]` by the users. One use case is that a special unknown word token is used as ID 0. These IDs will have a sampling probability of 0.
- `num_shards` : A sampler can be used to sample from a subset of the original range in order to speed up the whole computation through parallelism. This parameter (together with `shard`) indicates the number of partitions that are being used in the overall computation.
- `shard` : A sampler can be used to sample from a subset of the original range in order to speed up the whole computation through parallelism. This parameter (together with `num_shards`) indicates the particular partition number of the operation, when partitioning is being used.
- `unigrams` : A list of unigram counts or probabilities, one per ID in sequential order. Exactly one of `vocab_file` and `unigrams` should be passed to this operation.
- `seed` : An `int`. An operation-specific seed. Default is 0.
- `name` : A name for the operation (optional).

Returns:

- `sampled_candidates` : A tensor of type `int64` and shape `[num_sampled]`. The sampled classes.
- `true_expected_count` : A tensor of type `float`. Same shape as `true_classes`. The expected counts under the sampling distribution of each of `true_classes`.
- `sampled_expected_count` : A tensor of type `float`. Same shape as `sampled_candidates`. The expected counts under the sampling distribution of each of `sampled_candidates`.

Miscellaneous candidate sampling utilities

```
tf.nn.compute_accidental_hits(true_classes,
sampled_candidates, num_true, seed=None,
name=None)
```

Compute the position ids in `sampled_candidates` matching `true_classes`.

In Candidate Sampling, this operation facilitates virtually removing sampled classes which happen to match target classes. This is done in Sampled Softmax and Sampled Logistic.

See our [Candidate Sampling Algorithms Reference](#).

We presuppose that the `sampled_candidates` are unique.

We call it an 'accidental hit' when one of the target classes matches one of the sampled classes. This operation reports accidental hits as triples `(index, id, weight)`, where `index` represents the row number in `true_classes`, `id` represents the position in `sampled_candidates`, and `weight` is `-FLOAT_MAX`.

The result of this op should be passed through a `sparse_to_dense` operation, then added to the logits of the sampled classes. This removes the contradictory effect of accidentally sampling the true target classes as noise classes for the same example.

Args:

- `true_classes` : A `Tensor` of type `int64` and shape `[batch_size, num_true]`. The target classes.
- `sampled_candidates` : A tensor of type `int64` and shape `[num_sampled]`. The `sampled_candidates` output of CandidateSampler.
- `num_true` : An `int`. The number of target classes per training example.
- `seed` : An `int`. An operation-specific seed. Default is 0.
- `name` : A name for the operation (optional).

Returns:

- `indices` : A `Tensor` of type `int32` and shape `[num_accidental_hits]`. Values indicate rows in `true_classes`.
- `ids` : A `Tensor` of type `int64` and shape `[num_accidental_hits]`. Values indicate positions in `sampled_candidates`.
- `weights` : A `Tensor` of type `float` and shape `[num_accidental_hits]`. Each value is `-FLOAT_MAX`.

Other Functions and Classes

`tf.nn.batch_normalization(x, mean, variance, offset, scale, variance_epsilon, name=None)`

Batch normalization.

As described in <http://arxiv.org/abs/1502.03167>. Normalizes a tensor by `mean` and `variance`, and applies (optionally) a `scale` $\backslash(\gamma)$ to it, as well as an `offset` $\backslash(\beta)$:

$$\frac{\gamma}{\sqrt{\sigma^2 + \epsilon}}(x - \mu) + \beta$$

`mean`, `variance`, `offset` and `scale` are all expected to be of one of two shapes:

- In all generality, they can have the same number of dimensions as the input `x`, with identical sizes as `x` for the dimensions that are not normalized over (the 'depth' dimension(s)), and dimension 1 for the others which are being normalized over. `mean` and `variance` in this case would typically be the outputs of `tf.nn.moments(..., keep_dims=True)` during training, or running averages thereof during inference.
- In the common case where the 'depth' dimension is the last dimension in the input tensor `x`, they may be one dimensional tensors of the same size as the 'depth' dimension. This is the case for example for the common `[batch, depth]` layout of fully-connected layers, and `[batch, height, width, depth]` for convolutions. `mean` and `variance` in this case would typically be the outputs of `tf.nn.moments(..., keep_dims=False)` during training, or running averages thereof during inference.

Args:

- `x` : Input `Tensor` of arbitrary dimensionality.
- `mean` : A mean `Tensor`.
- `variance` : A variance `Tensor`.
- `offset` : An offset `Tensor`, often denoted β in equations, or `None`. If present, will be added to the normalized tensor.
- `scale` : A scale `Tensor`, often denoted γ in equations, or `None`. If present, the scale is applied to the normalized tensor.
- `variance_epsilon` : A small float number to avoid dividing by 0.
- `name` : A name for this operation (optional).

Returns:

the normalized, scaled, offset tensor.

`tf.nn.depthwise_conv2d_native(input, filter, strides, padding, name=None)`

Computes a 2-D depthwise convolution given 4-D `input` and `filter` tensors.

Given an input tensor of shape `[batch, in_height, in_width, in_channels]` and a filter / kernel tensor of shape `[filter_height, filter_width, in_channels, channel_multiplier]`, containing `in_channels` convolutional filters of depth 1, `depthwise_conv2d` applies a different filter to each input channel (expanding from 1 channel to `channel_multiplier` channels for each), then concatenates the results together. Thus, the output has `in_channels * channel_multiplier` channels.

```
for k in 0..inchannels-1 for q in 0..channel_multiplier-1 output[b, i, j, k * channel_multiplier +  
q] = sum{di, dj} input[b, strides[1] i + di, strides[2] j + dj, k] * filter[di, dj, k, q]
```

Must have `strides[0] = strides[3] = 1`. For the most common case of the same horizontal and vertices strides, `strides = [1, stride, stride, 1]`.

Args:

- `input` : A `Tensor`. Must be one of the following types: `float32`, `float64`.
- `filter` : A `Tensor`. Must have the same type as `input`.
- `strides` : A list of `ints`. 1-D of length 4. The stride of the sliding window for each dimension of `input`.
- `padding` : A `string` from: `"SAME", "VALID"`. The type of padding algorithm to use.
- `name` : A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`.

그래프 실행하기

[TOC]

이 라이브러리는 그래프를 시작하고 연산을 실행하기 위한 클래스를 포함하고 있습니다.

[basic usage](#) 가이드에는 `tf.Session`에서 그래프가 어떻게 시작되는지에 대한 예시들이 있습니다.

세션 관리

class `tf.Session`

TensorFlow 연산들을 실행하기 위한 클래스입니다.

`Session` 객체는 `Operation` 객체가 실행되고 `Tensor` 객체가 계산되는 환경을 캡슐화합니다. 예를 들면

```
# 그래프를 만듭니다.
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b

# 세션에서 그래프를 시작합니다.
sess = tf.Session()

# 텐서 `c`를 계산합니다.
print(sess.run(c))
```

세션은 [변수](#), [큐](#) 그리고 [리더](#) 같은 자체 리소스를 가질 것입니다. 이 리소스들이 더 이상 필요하지 않을 때 이를 해제시키는건 중요합니다. 이를 위해선, 세션에서 `close()` 메서드를 실행하거나 컨텍스트 매니저로써 세션을 사용해야합니다. 다음의 두 예시는 동일합니다.

```
# `close()` 메서드를 사용합니다.
sess = tf.Session()
sess.run(...)
sess.close()

# 컨텍스트 매니저를 사용합니다.
with tf.Session() as sess:
    sess.run(...)
```

[`ConfigProto`] (<https://www.tensorflow.org/code/tensorflow/core/protobuf/config.proto>) 프로토콜 버퍼는 세션을 위한 여러가지 설정 옵션을 제공합니다. 예를 들면, 디바이스 위치에 대해 유연한 제약 조건을 사용하고 위치 결정 결과를 로깅하기 위해 다음과 같이 세션을 생성합니다:

```
# 유연한 디바이스 위치와 위치 결정 로깅을 사용하는 세션에서 그래프를 시작합니다.  
sess = tf.Session(config=tf.ConfigProto(allow_soft_placement=True,  
                                         log_device_placement=True))
```

`tf.Session.__init__(target='', graph=None, config=None)`

새로운 TensorFlow 세션을 생성합니다.

세션을 생성할 때 `graph` 인자가 지정되지 않으면, 세션에선 기본 그래프가 시작됩니다. 만약 같은 프로세스에서 `tf.Graph()`로 생성되는 그래프를 하나 이상 사용한다면, 각 그래프에 대해서 서로 다른 세션을 사용해야 할 것입니다. 그러나 각 그래프는 여러 세션에서 사용될 수 있습니다. 이 경우에는 때때로 세션 생성자에 그래프가 시작된다는 걸 명시적으로 전달하는게 깔끔합니다.

인자:

- `target` : (선택) 접속을 위한 실행 엔진. 기본값으로 프로세스 내부 엔진을 사용합니다. 지금은, 빈 문자열 이외의 값은 지원되지 않습니다.
- `graph` : (선택) 시작되는 `Graph` (위에서 설명됨).
- `config` : (선택) 세션을 위한 설정 옵션을 가진 `ConfigProto` 프로토콜 버퍼.

`tf.Session.run(fetches, feed_dict=None, options=None, run_metadata=None)`

`fetches`에서 연산과 텐서를 실행합니다.

이 메서드는 모든 `Operation`들과 `fetches`에 있는 모든 `Tensor`들을 실행하기 위해 꼭 필요한 그래프 단편을 실행하면서 TensorFlow 계산을 한 스텝씩 실행하고, 입력값들에 대한 `feed_dict`의 값들을 교체합니다.

`fetches` 인자는 단일 그래프 요소, 그래프 요소들의 리스트 또는 위의 값들의 딕셔너리가 될 수 있습니다. `fetches`의 타입은 이 메서드의 반환값들 결정합니다. 그래프 요소는 다음 타입 중 하나가 될 수 있습니다.

- `fetches`의 요소가 `Operation` 일 때, 해당 패치값은 `None`이 될 것입니다.
- `fetches`의 요소가 `Tensor` 일 때, 해당 패치값은 텐서값을 포함하는 `numpy ndarray`가 될 것입니다.
- `fetches`의 요소가 `SparseTensor` 일 때, 해당 패치값은 희소 텐서의 값을 포함하는

`SparseTensorValue` 가 될 것입니다.

- `fetches` 의 요소가 `get_tensor_handle` 연산에 의해 생성되었을 경우, 해당 패치값은 텐서의 처리를 포함하는 numpy ndarray가 될 것 입니다.

선택적인 `feed_dict` 인자는 그래프의 텐서의 값들을 덮어씌울 수 있도록 해줍니다.

`feed_dict` 의 각 키들은 다음 타입중 하나가 될 수 있습니다.

- 키가 `Tensor` 라면, 값은 텐서와 같은 `dtype` 으로 변환될 수 있는 Python 스칼라, 문자열, 리스트 또는 numpy ndarray가 될 것입니다. 추가적으로, 키가 `placeholder`라면, 값의 구조 (`shape`)가 플레이스홀더(`placeholder`)와 호환되는지 확인될 것입니다.
- 키가 `SparseTensor` 라면, 값은 `SparseTensorValue` 이어야합니다.

`feed_dict` 의 각 값들은 해당하는 키의 `dtype`의 numpy 배열로 변환이 가능해야합니다.

선택적인 `options` 인자는 [`RunOptions`] 프로토콜 버퍼를 받습니다. 옵션은 이 특정한 단계의 행동을 컨트롤할 수 있도록합니다. (예로, 추적을 가능하게함)

선택적인 `run_metadata` 인자는 [`RunMetadata`] 프로토콜 버퍼를 인자로 받습니다. 적절한 때에, 이 단계의 텐서가 아닌 출력값은 수집될 것입니다. 예를 들면, 사용자가 `options` 에서 추적을 활성화할 때, 프로파일 정보는 이 인자로 수집될 것이며 역으로 전달될 것입니다.

인자:

- `fetches` : 단일 그래프 요소, 그래프 요소의 리스트 또는 이들의 값을 갖는 딕셔너리. (위에서 설명함.)
- `feed_dict` : 값들에 매핑되는 그래프 요소들의 딕셔너리.
- `options` : [`RunOptions`] 프로토콜 버퍼.
- `run_metadata` : [`RunMetadata`] 프로토콜 버퍼.

반환값:

`fetches` 가 단일 그래프 요소일 때에는 단일값, `fetches` 가 리스트일 경우엔 값 리스트, 또는 딕셔너리일 경우엔 `fetches` 와 같은 키값들을 가진 딕셔너리. (위에서 설명함)

예외:

- `RuntimeError` : `Session` 이 유효하지 않은 상태일 경우 발생. (예로, 닫혀진 경우)
- `TypeError` : `fetches` 또는 `feed_dict` 키들이 적절하지 않은 타입일 경우 발생.
- `ValueError` : `fetches` 또는 `feed_dict` 키들이 잘못되거나 존재하지 않는 `Tensor` 를 참조 할 경우 발생.

`tf.Session.close()`

세션을 닫습니다.

이 메서드를 실행하면 세션과 관련된 모든 리소스를 해제합니다.

예외:

`tf.errors.OpError`: TensorFlow 세션을 닫는 도중에 에러가 발생할 경우 이 예외 또는 이 예외의 서브클래스중 하나가 발생합니다.

`tf.Session.graph`

세션에서 시작된 그래프.

`tf.Session.as_default()`

이 객체를 기본 세션으로 만드는 컨텍스트 매니저를 반환합니다.

`Operation.run()` 또는 `Tensor.run()` 가 이 세션에서 실행되도록하는 호출을 지정하기위해 `with` 키워드와 함께 사용합니다.

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

with sess.as_default():
    assert tf.get_default_session() is sess
    print(c.eval())
```

현재 기본 세션을 얻기위해 `tf.get_default_session()` 을 사용합니다.

주의 `as_default` 컨텍스트 매니저는 컨텍스트를 빠져나왔을 때 세션을 닫지 않으며, 명시적으로 세션을 닫아줘야합니다.

```
c = tf.constant(..)
sess = tf.Session()
with sess.as_default():
    print(c.eval())
# ...
with sess.as_default():
    print(c.eval())

sess.close()
```

대안으로는, 잡을 수 없는 예외가 발생하는 경우를 포함해서, 컨텍스트를 빠져나갈 때 자동으로 닫히는 세션을 생성하기 위해서는 `with tf.Session()` 을 사용할 수 있습니다.

주의 기본 그래프는 현재 스레드의 프로퍼티입니다. 새로운 스레드를 생성하고, 스레드 안에서 기본 스레드를 사용하고 싶을 경우엔 반드시 그 스레드의 함수에 `with sess.as_default()` 를 명시적으로 추가해 주어야합니다.

반환값:

이 세션을 기본 세션으로 사용하는 컨텍스트 매니저.

class tf.InteractiveSession

쉘과 같은 인터랙티브 컨텍스트에서 사용하기 위한 TensorFlow `Session`

일반 `Session` 과의 유일한 차이점은 `InteractiveSession` 은 생성시 자기 자신을 기본 세션으로 설치한다는 것입니다. `Tensor.eval()` 메서드와 `Operation.run()` 메서드는 연산을 실행하기 위해 그 세션을 사용할 것입니다.

이는 인터랙티브 쉘과 [IPython notebooks](#)에서 편리하며, 연산을 실행하기 위한 `Session` 객체를 명시적으로 전달하지 않아도됩니다.

예시:

```
sess = tf.InteractiveSession()
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b
# 'sess'의 전달없이도 'c.eval()'을 실행할 수 있습니다.
print(c.eval())
sess.close()
```

일반 세션은 `with` 문 안에서 생성될 경우 자기 자신을 기본 세션으로 설치합니다. 인터랙티브 프로그램이 아닌 경우의 일반적인 사용법은 다음 패턴을 따르는 것입니다.

```
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b
with tf.Session():
    # 'c.eval()'을 여기에서도 사용할 수 있습니다.
    print(c.eval())
```

`tf.InteractiveSession.__init__(target='', graph=None, config=None)`

새로운 인터랙티브 TensorFlow 세션을 생성합니다.

세션을 생성할 때 `graph` 인자가 지정되지 않으면, 세션에선 디폴트 그래프가 시작됩니다. 만약 같은 프로세스에서 `tf.Graph()`로 생성되는 그래프를 하나 이상 사용한다면, 각 그래프에 대해서 서로 다른 세션을 사용해야 할 것입니다. 그러나 각 그래프는 여러 세션에서 사용될 수 있습니다. 이 경우에는 때때로 세션 생성자에 그래프가 시작된다는 걸 명시적으로 전달하는게 깔끔합니다.

인자:

- `target` : (선택) 접속을 위한 실행 엔진. 기본값으로 프로세스 내부 엔진을 사용합니다. 지금은, 빈 문자열 이외의 값은 지원되지 않습니다.
 - `graph` : (선택) 시작되는 `Graph`. (위에서 설명됨.)
 - `config` : (선택) 세션을 위한 설정 옵션을 가진 `ConfigProto` 프로토콜 버퍼.
-

`tf.InteractiveSession.close()`

`InteractiveSession` 을 닫습니다.

`tf.get_default_session()`

현재 스레드에 대한 기본 세션을 반환합니다.

반환된 `Session` 은 입력된 `Session` 또는 `Session.as_default()` 컨텍스트에서 가장 안 쪽의 세션이 될 것입니다.

참고 : 기본 그래프는 현재 스레드의 프로퍼티입니다. 새로운 스레드를 생성하고, 스레드 안에서 기본 스레드를 사용하고 싶을 경우엔 반드시 그 스레드의 함수에 `with sess.as_default():` 를 명시적으로 추가해 주어야합니다.

반환값:

기본 `Session` 은 현재 스레드에서 사용됩니다.

에러 클래스

`class tf.OpError`

TensorFlow 실행이 실패할 때 발생하는 일반적인 에러.

세션은 언제든지 `tf.errors` 모듈에 있는 더 많은 `OpError` 의 특정한 서브클래스 예외를 발생시킬 수 있습니다.

tf.OpError.op

알려진 실패한 연산.

주의 실패한 연산이 런타임때 합쳐진 경우엔, 예를 들면, `Send` 또는 `Recv` 연산, 해당하는 `Operation` 객체가 없을 것입니다. 이 경우, 이는 `None` 을 반환할 것이며, 연산에 대한 정보를 찾으려면 `OpError.node_def` 를 대신 사용해야합니다.

반환값:

실패한 `operation` 또는 `None`.

tf.OpError.node_def

실패한 연산을 나타내는 `NodeDef` 프로토콜 버퍼.

그 외 메서드

tf.OpError.__init__(node_def, op, message, error_code)

특정한 실패한 연산을 가리키는 새로운 `OpError` 를 생성합니다.

인자:

- `node_def` : 알려진 연산의 경우 실패한 연산을 나타내는 `graph_pb2.NodeDef` 프로토콜 버퍼, 이 외에는 `None`.
 - `op` : 알려진 연산의 경우 실패한 `ops.Operation`, 이 외에는 `None`.
 - `message` : 실패를 설명하는 메시지 문자열.
 - `error_code` : 에러를 나타내는 `error_codes_pb2.Code`.
-

tf.OpError.error_code

에러를 나타내는 정수 에러 코드.

tf.OpError.message

에러를 설명하는 에러 메시지.

class tf.errors.CancelledError

연산이나 단계가 취소되었을 때 발생합니다.

예를 들면, 장시간 실행하는 연산 (예로, `queue.enqueue()`)는 또 다른 연산 (예로, `queue.close(cancel_pending_enqueues=True)`) 또는 `closing the session`을 실행함으로써 취소될 수 있습니다. 장기 실행 연산을 실행하는 단계는 `CancelledError` 를 발생시키며 실패할 것입니다.

`tf.errors.CancelledError.__init__(node_def, op, message)`

`CancelledError` 를 생성합니다.

class tf.errors.UnknownError

알려지지 않은 에러.

이 에러가 반환될 수 있는 한 예는 상태값을 현재 주소 공간에서는 알려지지 않은 에러 공간에 속한 다른 주소 공간으로부터 받는 경우입니다. 또한 충분한 에러 정보를 반환하지 않는 API에 의해 발생하는 에러도 이 에러로 변환될 수 있습니다.

`tf.errors.UnknownError.__init__(node_def, op, message, error_code=2)`

`UnknownError` 를 생성합니다.

class tf.errors.InvalidArgumentError

연산이 잘못된 인자를 받는 경우 발생합니다.

이 에러는, 예를 들면, 만약 연산이 잘못된 값이나 구조(shape)를 가진 입력 텐서를 받을 경우에 발생할 수 있습니다. `tf.matmul()` 연산은 행렬이 아닌 입력을 받을 경우에 이 에러를 발생시킬 것이며, `tf.reshape()` 연산은 새로운 구조(shape)가 입력 텐서의 요소들의 갯수와 매칭이 안될 경우 이 에러를 발생시킬 것입니다.

`tf.errors.InvalidArgumentError.__init__(node_def, op, message)`

`InvalidArgumentError` 를 생성합니다.

class tf.errors.DeadlineExceededError

연산이 완료되기 전에 기한이 만료될 경우 발생합니다.

이 예외는 현재 사용되지 않습니다.

`tf.errors.DeadlineExceededError.__init__(node_def, op, message)`

`DeadlineExceededError` 를 생성합니다.

class tf.errors.NotFoundError

요청된 엔티티 (예로, 파일이나 디렉토리)를 찾을 수 없는 경우 발생합니다.

예를 들면, `tf.WholeFileReader.read()` 연산을 실행하는데 존재하지 않는 파일의 이름을 받게되면 `NotFoundError` 을 발생시킬 수 있습니다.

`tf.errors.NotFoundError.__init__(node_def, op, message)`

`NotFoundError` 를 생성합니다.

class tf.errors.AlreadyExistsError

이미 존재하는 엔티티를 생성하려고 할 경우 발생합니다.

예를 들면, 파일을 저장하는 연산 (예로, `tf.train.Saver.save()`)을 실행할 때 존재하는 파일의 파일명을 명시적으로 전달할 경우 잠재적으로 이 에러를 발생시킬 수 있습니다.

`tf.errors.AlreadyExistsError.__init__(node_def, op, message)`

`AlreadyExistsError` 를 생성합니다.

class tf.errors.PermissionDeniedError

호출자가 연산 실행에 대한 권한이 없을 경우 발생합니다.

예를 들면, `tf.WholeFileReader.read()` 연산을 실행할 때 사용자가 읽기 권한을 갖고 있지 않는 파일명을 받을 경우 `PermissionDeniedError` 를 발생시킬 수 있습니다.

`tf.errors.PermissionDeniedError.__init__(node_def, op, message)`

`PermissionDeniedError` 를 생성합니다.

class tf.errors.UnauthenticatedError

요청이 유효한 인증 자격 증명을 가지지 않은 경우.

이 예외는 현재 사용되지 않습니다.

`tf.errors.UnauthenticatedError.__init__(node_def, op, message)`

`UnauthenticatedError` 를 생성합니다.

class tf.errors.ResourceExhaustedError

리소스가 소진된 경우.

예를 들면, 사용자별 할당량(quota)이 소진되거나 전 파일 시스템에 공간이 부족할 경우 이 에러가 발생할 수 있습니다.

`tf.errors.ResourceExhaustedError.__init__(node_def, op, message)`

`ResourceExhaustedError` 를 생성합니다.

class tf.errors.FailedPreconditionError

시스템이 연산을 실행시킬 수 있는 상태가 아니라 연산이 거부됨.

이 예외는 `tf.Variable` 를 읽는 연산을 초기화 전에 실행할 경우에 발생하는 가장 빈번한 예외입니다.

`tf.errors.FailedPreconditionError.__init__(node_def, op, message)`

`FailedPreconditionError` 를 생성합니다.

`class tf.errors.AbortedError`

보통 동시 작업때문에 연산이 중단됨.

예를 들면, `queue.close()` 이 이전에 실행된 상태에서 `queue.enqueue()` 연산을 실행하면 `AbortedError` 가 발생할 수 있습니다.

`tf.errors.AbortedError.__init__(node_def, op, message)`

`AbortedError` 를 생성합니다.

`class tf.errors.OutOfRangeError`

연산이 유효한 입력 범위를 지나쳐 순회할 경우 발생합니다.

이 예외는 `queue.dequeue()` 연산이 빈 큐에서 블로킹 되고 `queue.close()` 연산이 실행되는 경우와 같은 "파일의 끝(end-of-file)"이라는 조건에서 발생합니다.

`tf.errors.OutOfRangeError.__init__(node_def, op, message)`

`OutOfRangeError` 를 생성합니다.

`class tf.errors.UnimplementedError`

연산이 구현되지 않은 경우 발생합니다.

몇가지 연산은 유효하지만 현재 지원되지 않는 인자들을 전달할 경우 이 에러를 발생시킬 수 있습니다. `tf.nn.max_pool()` 연산을 실행할 때 배치 차원에 폴링이 요청된 경우 이는 아직 지원되지 않기 때문에 이 에러를 발생시킬 것입니다.

`tf.errors.UnimplementedError.__init__(node_def, op, message)`

`UnimplementedError` 를 생성합니다.

`class tf.errors.InternalError`

시스템 내부 에러가 생길 경우 발생.

망가진 런타임에 의해 몇가지 불변이 예상될 경우 발생하는 예외입니다. 이 예외를 잡는 것은 추천하지 않습니다.

`tf.errors.InternalError.__init__(node_def, op, message)`

`InternalError` 를 생성합니다.

`class tf.errors.UnavailableError`

런타임이 현재 이용불가능할 때 발생합니다.

이 예외는 현재 사용되지 않습니다.

`tf.errors.UnavailableError.__init__(node_def, op, message)`

`UnavailableError` 를 생성합니다.

`class tf.errors.DataLossError`

복구불가능한 데이터를 읽거나 손상이 생겼을 때 발생합니다.

예를 들면, 이는 `tf.WholeFileReader.read()` 연산을 실행하는데, 읽는 도중에 파일이 잘릴 경우 발생할 수 있습니다.

```
tf.errors.DataLossError.__init__(node_def, op, message)
```

DataLossError 를 생성합니다.

Training

[TOC]

This library provides a set of classes and functions that helps train models.

Optimizers

The Optimizer base class provides methods to compute gradients for a loss and apply gradients to variables. A collection of subclasses implement classic optimization algorithms such as GradientDescent and Adagrad.

You never instantiate the Optimizer class itself, but instead instantiate one of the subclasses.

`class tf.train.Optimizer`

Base class for optimizers.

This class defines the API to add Ops to train a model. You never use this class directly, but instead instantiate one of its subclasses such as `GradientDescentOptimizer` , `AdagradOptimizer` , or `MomentumOptimizer` .

Usage

```
# Create an optimizer with the desired parameters.  
opt = GradientDescentOptimizer(learning_rate=0.1)  
# Add Ops to the graph to minimize a cost by updating a list of variables.  
# "cost" is a Tensor, and the list of variables contains tf.Variable  
# objects.  
opt_op = opt.minimize(cost, var_list=<list of variables>)
```

In the training program you will just have to run the returned Op.

```
# Execute opt_op to do one step of training:  
opt_op.run()
```

Processing gradients before applying them.

Calling `minimize()` takes care of both computing the gradients and applying them to the variables. If you want to process the gradients before applying them you can instead use the optimizer in three steps:

1. Compute the gradients with `compute_gradients()`.
2. Process the gradients as you wish.
3. Apply the processed gradients with `apply_gradients()`.

Example:

```
# Create an optimizer.
opt = GradientDescentOptimizer(learning_rate=0.1)

# Compute the gradients for a list of variables.
grads_and_vars = opt.compute_gradients(loss, <list of variables>

# grads_and_vars is a list of tuples (gradient, variable). Do whatever you
# need to the 'gradient' part, for example cap them, etc.
capped_grads_and_vars = [(MyCapper(gv[0]), gv[1]) for gv in grads_and_vars]

# Ask the optimizer to apply the capped gradients.
opt.apply_gradients(capped_grads_and_vars)
```

`tf.train.Optimizer.__init__(use_locking, name)`

Create a new Optimizer.

This must be called by the constructors of subclasses.

Args:

- `use_locking` : Bool. If True apply locks to prevent concurrent updates to variables.
- `name` : A non-empty string. The name to use for accumulators created for the optimizer.

Raises:

- `ValueError` : If name is malformed.

`tf.train.Optimizer.minimize(loss, global_step=None, var_list=None, gate_gradients=1, aggregation_method=None, colocate_gradients_with_ops=False, name=None, grad_loss=None)`

Add operations to minimize `loss` by updating `var_list`.

This method simply combines calls `compute_gradients()` and `apply_gradients()`. If you want to process the gradient before applying them call `compute_gradients()` and `apply_gradients()` explicitly instead of using this function.

Args:

- `loss` : A `Tensor` containing the value to minimize.
- `global_step` : Optional `Variable` to increment by one after the variables have been updated.
- `var_list` : Optional list of `Variable` objects to update to minimize `loss`. Defaults to the list of variables collected in the graph under the key `GraphKeys.TRAINABLE_VARIABLES`.
- `gate_gradients` : How to gate the computation of gradients. Can be `GATE_NONE`, `GATE_OP`, or `GATE_GRAPH`.
- `aggregation_method` : Specifies the method used to combine gradient terms. Valid values are defined in the class `AggregationMethod`.
- `colocate_gradients_with_ops` : If True, try colocating gradients with the corresponding op.
- `name` : Optional name for the returned operation.
- `grad_loss` : Optional. A `Tensor` holding the gradient computed for `loss`.

Returns:

An Operation that updates the variables in `var_list`. If `global_step` was not `None`, that operation also increments `global_step`.

Raises:

- `ValueError` : If some of the variables are not `Variable` objects.

```
tf.train.Optimizer.compute_gradients(loss, var_list=None,
                                     gate_gradients=1, aggregation_method=None,
                                     colocate_gradients_with_ops=False, grad_loss=None)
```

Compute gradients of `loss` for the variables in `var_list`.

This is the first part of `minimize()`. It returns a list of (gradient, variable) pairs where "gradient" is the gradient for "variable". Note that "gradient" can be a `Tensor`, an `IndexedSlices`, or `None` if there is no gradient for the given variable.

Args:

- `loss` : A Tensor containing the value to minimize.
- `var_list` : Optional list of `tf.Variable` to update to minimize `loss`. Defaults to the list of

variables collected in the graph under the key `GraphKey.TRAINABLE_VARIABLES` .

- `gate_gradients` : How to gate the computation of gradients. Can be `GATE_NONE` , `GATE_OP` , or `GATE_GRAPH` .
- `aggregation_method` : Specifies the method used to combine gradient terms. Valid values are defined in the class `AggregationMethod` .
- `colocate_gradients_with_ops` : If True, try colocating gradients with the corresponding op.
- `grad_loss` : Optional. A `Tensor` holding the gradient computed for `loss` .

Returns:

A list of (gradient, variable) pairs.

Raises:

- `TypeError` : If `var_list` contains anything else than `Variable` objects.
- `ValueError` : If some arguments are invalid.

```
tf.train.Optimizer.apply_gradients(grads_and_vars,
global_step=None, name=None)
```

Apply gradients to variables.

This is the second part of `minimize()` . It returns an `Operation` that applies gradients.

Args:

- `grads_and_vars` : List of (gradient, variable) pairs as returned by `compute_gradients()` .
- `global_step` : Optional `Variable` to increment by one after the variables have been updated.
- `name` : Optional name for the returned operation. Default to the name passed to the `Optimizer` constructor.

Returns:

An `Operation` that applies the specified gradients. If `global_step` was not None, that operation also increments `global_step` .

Raises:

- `TypeError` : If `grads_and_vars` is malformed.
- `ValueError` : If none of the variables have gradients.

Gating Gradients

Both `minimize()` and `compute_gradients()` accept a `gate_gradient` argument that controls the degree of parallelism during the application of the gradients.

The possible values are: `GATE_NONE`, `GATE_OP`, and `GATE_GRAPH`.

`GATE_NONE`: Compute and apply gradients in parallel. This provides the maximum parallelism in execution, at the cost of some non-reproducibility in the results. For example the two gradients of `matmul` depend on the input values: With `GATE_NONE` one of the gradients could be applied to one of the inputs *before* the other gradient is computed resulting in non-reproducible results.

`GATE_OP`: For each Op, make sure all gradients are computed before they are used. This prevents race conditions for Ops that generate gradients for multiple inputs where the gradients depend on the inputs.

`GATE_GRAPH`: Make sure all gradients for all variables are computed before any one of them is used. This provides the least parallelism but can be useful if you want to process all gradients before applying any of them.

Slots

Some optimizer subclasses, such as `MomentumOptimizer` and `AdagradOptimizer` allocate and manage additional variables associated with the variables to train. These are called *Slots*. Slots have names and you can ask the optimizer for the names of the slots that it uses. Once you have a slot name you can ask the optimizer for the variable it created to hold the slot value.

This can be useful if you want to log debug a training algorithm, report stats about the slots, etc.

`tf.train.Optimizer.get_slot_names()`

Return a list of the names of slots created by the `optimizer`.

See `get_slot()`.

Returns:

A list of strings.

tf.train.Optimizer.get_slot(var, name)

Return a slot named `name` created for `var` by the Optimizer.

Some `Optimizer` subclasses use additional variables. For example `Momentum` and `Adagrad` use variables to accumulate updates. This method gives access to these `Variable` objects if for some reason you need them.

Use `get_slot_names()` to get the list of slot names created by the `Optimizer`.

Args:

- `var` : A variable passed to `minimize()` or `apply_gradients()`.
- `name` : A string.

Returns:

The `Variable` for the slot if it was created, `None` otherwise.

class tf.train.GradientDescentOptimizer

Optimizer that implements the gradient descent algorithm.

tf.train.GradientDescentOptimizer.__init__(learning_rate, use_locking=False, name='GradientDescent')

Construct a new gradient descent optimizer.

Args:

- `learning_rate` : A Tensor or a floating point value. The learning rate to use.
 - `use_locking` : If True use locks for update operations.
 - `name` : Optional name prefix for the operations created when applying gradients.
Defaults to "GradientDescent".
-

class tf.train.AdadeltaOptimizer

Optimizer that implements the Adadelta algorithm.

See [M. D. Zeiler \(pdf\)](#)

```
tf.train.AdadeltaOptimizer.__init__(learning_rate=0.001,
rho=0.95, epsilon=1e-08, use_locking=False,
name='Adadelta')
```

Construct a new Adadelta optimizer.

Args:

- `learning_rate` : A `Tensor` or a floating point value. The learning rate.
 - `rho` : A `Tensor` or a floating point value. The decay rate.
 - `epsilon` : A `Tensor` or a floating point value. A constant epsilon used
to better conditioning the grad update.
 - `use_locking` : If `True` use locks for update operations.
 - `name` : Optional name prefix for the operations created when applying gradients.
Defaults to "Adadelta".
-

class `tf.train.AdagradOptimizer`

Optimizer that implements the Adagrad algorithm.

See this [paper](#).

```
tf.train.AdagradOptimizer.__init__(learning_rate,
initial_accumulator_value=0.1, use_locking=False,
name='Adagrad')
```

Construct a new Adagrad optimizer.

Args:

- `learning_rate` : A `Tensor` or a floating point value. The learning rate.
- `initial_accumulator_value` : A floating point value. Starting value for the accumulators,
must be positive.
- `use_locking` : If `True` use locks for update operations.
- `name` : Optional name prefix for the operations created when applying gradients.
Defaults to "Adagrad".

Raises:

- `ValueError` : If the `initial_accumulator_value` is invalid.

class tf.train.MomentumOptimizer

Optimizer that implements the Momentum algorithm.

```
tf.train.MomentumOptimizer.__init__(learning_rate,  
momentum, use_locking=False, name='Momentum')
```

Construct a new Momentum optimizer.

Args:

- `learning_rate` : A `Tensor` or a floating point value. The learning rate.
- `momentum` : A `Tensor` or a floating point value. The momentum.
- `use_locking` : If `True` use locks for update operations.
- `name` : Optional name prefix for the operations created when applying gradients.
Defaults to "Momentum".

class tf.train.AdamOptimizer

Optimizer that implements the Adam algorithm.

See Kingma et. al., 2014 ([pdf](#)).

```
tf.train.AdamOptimizer.__init__(learning_rate=0.001,  
beta1=0.9, beta2=0.999, epsilon=1e-08, use_locking=False,  
name='Adam' )
```

Construct a new Adam optimizer.

Initialization:

```
m_0 <- 0 (Initialize initial 1st moment vector)  
v_0 <- 0 (Initialize initial 2nd moment vector)  
t <- 0 (Initialize timestep)
```

The update rule for `variable` with gradient `g` uses an optimization described at the end of section2 of the paper:

```

t <- t + 1
lr_t <- learning_rate * sqrt(1 - beta2^t) / (1 - beta1^t)

m_t <- beta1 * m_{t-1} + (1 - beta1) * g
v_t <- beta2 * v_{t-1} + (1 - beta2) * g * g
variable <- variable - lr_t * m_t / (sqrt(v_t) + epsilon)

```

The default value of 1e-8 for epsilon might not be a good default in general. For example, when training an Inception network on ImageNet a current good choice is 1.0 or 0.1.

Args:

- `learning_rate` : A Tensor or a floating point value. The learning rate.
- `beta1` : A float value or a constant float tensor. The exponential decay rate for the 1st moment estimates.
- `beta2` : A float value or a constant float tensor. The exponential decay rate for the 2nd moment estimates.
- `epsilon` : A small constant for numerical stability.
- `use_locking` : If True use locks for update operations.
- `name` : Optional name for the operations created when applying gradients. Defaults to "Adam".

class tf.train.FtrlOptimizer

Optimizer that implements the FTRL algorithm.

See this [paper](#).

```

tf.train.FtrlOptimizer.__init__(learning_rate,
learning_rate_power=-0.5, initial_accumulator_value=0.1,
l1_regularization_strength=0.0,
l2_regularization_strength=0.0, use_locking=False,
name='Ftrl')

```

Construct a new FTRL optimizer.

Args:

- `learning_rate` : A float value or a constant float `Tensor`.
- `learning_rate_power` : A float value, must be less or equal to zero.
- `initial_accumulator_value` : The starting value for accumulators. Only positive values

are allowed.

- `l1_regularization_strength` : A float value, must be greater than or equal to zero.
- `l2_regularization_strength` : A float value, must be greater than or equal to zero.
- `use_locking` : If `True` use locks for update operations.
- `name` : Optional name prefix for the operations created when applying gradients.
Defaults to "Ftrl".

Raises:

- `ValueError` : If one of the arguments is invalid.

class `tf.train.RMSPropOptimizer`

Optimizer that implements the RMSProp algorithm.

See the [paper] (http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf).

```
tf.train.RMSPropOptimizer.__init__(learning_rate,
decay=0.9, momentum=0.0, epsilon=1e-10, use_locking=False,
name='RMSProp')
```

Construct a new RMSProp optimizer.

Args:

- `learning_rate` : A Tensor or a floating point value. The learning rate.
- `decay` : Discounting factor for the history/coming gradient
- `momentum` : A scalar tensor.
- `epsilon` : Small value to avoid zero denominator.
- `use_locking` : If `True` use locks for update operation.
- `name` : Optional name prefix for the operations created when applying gradients.
Defaults to "RMSProp".

Gradient Computation

TensorFlow provides functions to compute the derivatives for a given TensorFlow computation graph, adding operations to the graph. The optimizer classes automatically compute derivatives on your graph, but creators of new Optimizers or expert users can call the lower-level functions below.

```
tf.gradients(ys, xs, grad_ys=None,
name='gradients',
colocate_gradients_with_ops=False,
gate_gradients=False, aggregation_method=None)
```

Constructs symbolic partial derivatives of sum of `ys` w.r.t. `x` in `xs`.

`ys` and `xs` are each a `Tensor` or a list of tensors. `grad_ys` is a list of `Tensor`, holding the gradients received by the `ys`. The list must be the same length as `ys`.

`gradients()` adds ops to the graph to output the partial derivatives of `ys` with respect to `xs`. It returns a list of `Tensor` of length `len(xs)` where each tensor is the `sum(dy/dx)` for `y` in `ys`.

`grad_ys` is a list of tensors of the same length as `ys` that holds the initial gradients for each `y` in `ys`. When `grad_ys` is `None`, we fill in a tensor of '1's of the shape of `y` for each `y` in `ys`. A user can provide their own initial `grad_ys` to compute the derivatives using a different initial gradient for each `y` (e.g., if one wanted to weight the gradient differently for each value in each `y`).

Args:

- `ys` : A `Tensor` or list of tensors to be differentiated.
- `xs` : A `Tensor` or list of tensors to be used for differentiation.
- `grad_ys` : Optional. A `Tensor` or list of tensors the same size as `ys` and holding the gradients computed for each `y` in `ys`.
- `name` : Optional name to use for grouping all the gradient ops together. defaults to 'gradients'.
- `colocate_gradients_with_ops` : If True, try colocating gradients with the corresponding op.
- `gate_gradients` : If True, add a tuple around the gradients returned for an operations. This avoids some race conditions.
- `aggregation_method` : Specifies the method used to combine gradient terms. Accepted values are constants defined in the class `AggregationMethod`.

Returns:

A list of `sum(dy/dx)` for each `x` in `xs`.

Raises:

- `LookupError` : if one of the operations between `x` and `y` does not have a registered gradient function.
- `ValueError` : if the arguments are invalid.

class tf.AggregationMethod

A class listing aggregation methods used to combine gradients.

Computing partial derivatives can require aggregating gradient contributions. This class lists the various methods that can be used to combine gradients in the graph:

- `ADD_N` : All of the gradient terms are summed as part of one operation using the "AddN" op. It has the property that all gradients must be ready before any aggregation is performed.
- `DEFAULT` : The system-chosen default aggregation method.

tf.stop_gradient(input, name=None)

Stops gradient computation.

When executed in a graph, this op outputs its input tensor as-is.

When building ops to compute gradients, this op prevents the contribution of its inputs to be taken into account. Normally, the gradient generator adds ops to a graph to compute the derivatives of a specified 'loss' by recursively finding out inputs that contributed to its computation. If you insert this op in the graph its inputs are masked from the gradient generator. They are not taken into account for computing gradients.

This is useful any time you want to compute a value with TensorFlow but need to pretend that the value was a constant. Some examples include:

- The *EM* algorithm where the *M-step* should not involve backpropagation through the output of the *E-step*.
- Contrastive divergence training of Boltzmann machines where, when differentiating the energy function, the training must not backpropagate through the graph that generated the samples from the model.
- Adversarial training, where no backprop should happen through the adversarial example generation process.

Args:

- `input` : A `Tensor`.
- `name` : A name for the operation (optional).

Returns:

A `Tensor`. Has the same type as `input`.

Gradient Clipping

TensorFlow provides several operations that you can use to add clipping functions to your graph. You can use these functions to perform general data clipping, but they're particularly useful for handling exploding or vanishing gradients.

```
tf.clip_by_value(t, clip_value_min,
clip_value_max, name=None)
```

Clips tensor values to a specified min and max.

Given a tensor `t`, this operation returns a tensor of the same type and shape as `t` with its values clipped to `clip_value_min` and `clip_value_max`. Any values less than `clip_value_min` are set to `clip_value_min`. Any values greater than `clip_value_max` are set to `clip_value_max`.

Args:

- `t` : A `Tensor`.
- `clip_value_min` : A 0-D (scalar) `Tensor`. The minimum value to clip by.
- `clip_value_max` : A 0-D (scalar) `Tensor`. The maximum value to clip by.
- `name` : A name for the operation (optional).

Returns:

A clipped `Tensor`.

```
tf.clip_by_norm(t, clip_norm, name=None)
```

Clips tensor values to a maximum L2-norm.

Given a tensor `t`, and a maximum clip value `clip_norm`, this operation normalizes `t` so that its L2-norm is less than or equal to `clip_norm`. Specifically, if the L2-norm is already less than or equal to `clip_norm`, then `t` is not modified. If the L2-norm is greater than `clip_norm`, then this operation returns a tensor of the same type and shape as `t` with its values set to:

```
t * clip_norm / l2norm(t)
```

In this case, the L2-norm of the output tensor is `clip_norm`.

This operation is typically used to clip gradients before applying them with an optimizer.

Args:

- `t` : A `Tensor`.
- `clip_norm` : A 0-D (scalar) `Tensor` > 0 . A maximum clipping value.
- `name` : A name for the operation (optional).

Returns:

A clipped `Tensor`.

```
tf.clip_by_average_norm(t, clip_norm,
name=None)
```

Clips tensor values to a maximum average L2-norm.

Given a tensor `t`, and a maximum clip value `clip_norm`, this operation normalizes `t` so that its average L2-norm is less than or equal to `clip_norm`. Specifically, if the average L2-norm is already less than or equal to `clip_norm`, then `t` is not modified. If the average L2-norm is greater than `clip_norm`, then this operation returns a tensor of the same type and shape as `t` with its values set to:

```
t * clip_norm / l2norm_avg(t)
```

In this case, the average L2-norm of the output tensor is `clip_norm`.

This operation is typically used to clip gradients before applying them with an optimizer.

Args:

- `t` : A `Tensor`.
- `clip_norm` : A 0-D (scalar) `Tensor` > 0 . A maximum clipping value.
- `name` : A name for the operation (optional).

Returns:

A clipped `Tensor`.

```
tf.clip_by_global_norm(t_list, clip_norm,
use_norm=None, name=None)
```

Clips values of multiple tensors by the ratio of the sum of their norms.

Given a tuple or list of tensors `t_list`, and a clipping ratio `clip_norm`, this operation returns a list of clipped tensors `list_clipped` and the global norm (`global_norm`) of all tensors in `t_list`. Optionally, if you've already computed the global norm for `t_list`, you can specify the global norm with `use_norm`.

To perform the clipping, the values `t_list[i]` are set to:

```
t_list[i] * clip_norm / max(global_norm, clip_norm)
```

where:

```
global_norm = sqrt(sum([l2norm(t)**2 for t in t_list]))
```

If `clip_norm > global_norm` then the entries in `t_list` remain as they are, otherwise they're all shrunk by the global ratio.

Any of the entries of `t_list` that are of type `None` are ignored.

This is the correct way to perform gradient clipping (for example, see [Pascanu et al., 2012 \(pdf\)](#)).

However, it is slower than `clip_by_norm()` because all the parameters must be ready before the clipping operation can be performed.

Args:

- `t_list` : A tuple or list of mixed `Tensors`, `IndexedSlices`, or `None`.
- `clip_norm` : A 0-D (scalar) `Tensor` > 0 . The clipping ratio.
- `use_norm` : A 0-D (scalar) `Tensor` of type `float` (optional). The global norm to use. If not provided, `global_norm()` is used to compute the norm.
- `name` : A name for the operation (optional).

Returns:

- `list_clipped` : A list of `Tensors` of the same type as `list_t`.
- `global_norm` : A 0-D (scalar) `Tensor` representing the global norm.

Raises:

- `TypeError` : If `t_list` is not a sequence.

tf.global_norm(t_list, name=None)

Computes the global norm of multiple tensors.

Given a tuple or list of tensors `t_list`, this operation returns the global norm of the elements in all tensors in `t_list`. The global norm is computed as:

```
global_norm = sqrt(sum([l2norm(t)**2 for t in t_list]))
```

Any entries in `t_list` that are of type None are ignored.

Args:

- `t_list` : A tuple or list of mixed `Tensors`, `IndexedSlices`, or `None`.
- `name` : A name for the operation (optional).

Returns:

A 0-D (scalar) `Tensor` of type `float`.

Raises:

- `TypeError` : If `t_list` is not a sequence.

Decaying the learning rate

tf.train.exponential_decay(learning_rate, global_step, decay_steps, decay_rate, staircase=False, name=None)

Applies exponential decay to the learning rate.

When training a model, it is often recommended to lower the learning rate as the training progresses. This function applies an exponential decay function to a provided initial learning rate. It requires a `global_step` value to compute the decayed learning rate. You can just pass a TensorFlow variable that you increment at each training step.

The function returns the decayed learning rate. It is computed as:

```
decayed_learning_rate = learning_rate *  
                      decay_rate ^ (global_step / decay_steps)
```

If the argument `staircase` is `True`, then `global_step /decay_steps` is an integer division and the decayed learning rate follows a staircase function.

Example: decay every 100000 steps with a base of 0.96:

```
...
global_step = tf.Variable(0, trainable=False)
starter_learning_rate = 0.1
learning_rate = tf.train.exponential_decay(starter_learning_rate, global_step,
                                           100000, 0.96, staircase=True)
# Passing global_step to minimize() will increment it at each step.
learning_step = (
    tf.GradientDescentOptimizer(learning_rate)
    .minimize(...my loss..., global_step=global_step)
)
```

Args:

- `learning_rate` : A scalar `float32` or `float64` `Tensor` or a Python number. The initial learning rate.
- `global_step` : A scalar `int32` or `int64` `Tensor` or a Python number. Global step to use for the decay computation. Must not be negative.
- `decay_steps` : A scalar `int32` or `int64` `Tensor` or a Python number. Must be positive. See the decay computation above.
- `decay_rate` : A scalar `float32` or `float64` `Tensor` or a Python number. The decay rate.
- `staircase` : Boolean. If `True` decay the learning rate at discrete intervals.
- `name` : String. Optional name of the operation. Defaults to 'ExponentialDecay'

Returns:

A scalar `Tensor` of the same type as `learning_rate`. The decayed learning rate.

Moving Averages

Some training algorithms, such as GradientDescent and Momentum often benefit from maintaining a moving average of variables during optimization. Using the moving averages for evaluations often improve results significantly.

class `tf.train.ExponentialMovingAverage`

Maintains moving averages of variables by employing an exponential decay.

When training a model, it is often beneficial to maintain moving averages of the trained parameters. Evaluations that use averaged parameters sometimes produce significantly better results than the final trained values.

The `apply()` method adds shadow copies of trained variables and add ops that maintain a moving average of the trained variables in their shadow copies. It is used when building the training model. The ops that maintain moving averages are typically run after each training step. The `average()` and `average_name()` methods give access to the shadow variables and their names. They are useful when building an evaluation model, or when restoring a model from a checkpoint file. They help use the moving averages in place of the last trained values for evaluations.

The moving averages are computed using exponential decay. You specify the decay value when creating the `ExponentialMovingAverage` object. The shadow variables are initialized with the same initial values as the trained variables. When you run the ops to maintain the moving averages, each shadow variable is updated with the formula:

```
shadow_variable -= (1 - decay) * (shadow_variable - variable)
```

This is mathematically equivalent to the classic formula below, but the use of an `assign_sub` op (the `"-="` in the formula) allows concurrent lockless updates to the variables:

```
shadow_variable = decay * shadow_variable + (1 - decay) * variable
```

Reasonable values for `decay` are close to 1.0, typically in the multiple-nines range: 0.999, 0.9999, etc.

Example usage when creating a training model:

```

# Create variables.
var0 = tf.Variable(...)
var1 = tf.Variable(...)

# ... use the variables to build a training model...
...

# Create an op that applies the optimizer. This is what we usually
# would use as a training op.
opt_op = opt.minimize(my_loss, [var0, var1])

# Create an ExponentialMovingAverage object
ema = tf.train.ExponentialMovingAverage(decay=0.9999)

# Create the shadow variables, and add ops to maintain moving averages
# of var0 and var1.
maintain_averages_op = ema.apply([var0, var1])

# Create an op that will update the moving averages after each training
# step. This is what we will use in place of the usual training op.
with tf.control_dependencies([opt_op]):
    training_op = tf.group(maintain_averages_op)

...train the model by running training_op...

```

There are two ways to use the moving averages for evaluations:

- Build a model that uses the shadow variables instead of the variables. For this, use the `average()` method which returns the shadow variable for a given variable.
- Build a model normally but load the checkpoint files to evaluate by using the shadow variable names. For this use the `average_name()` method. See the [Saver class](#) for more information on restoring saved variables.

Example of restoring the shadow variable values:

```

# Create a Saver that loads variables from their saved shadow values.
shadow_var0_name = ema.average_name(var0)
shadow_var1_name = ema.average_name(var1)
saver = tf.train.Saver({shadow_var0_name: var0, shadow_var1_name: var1})
saver.restore(...checkpoint filename...)
# var0 and var1 now hold the moving average values

```

`tf.train.ExponentialMovingAverage.__init__(decay, num_updates=None, name='ExponentialMovingAverage')`

Creates a new ExponentialMovingAverage object.

The `apply()` method has to be called to create shadow variables and add ops to maintain moving averages.

The optional `num_updates` parameter allows one to tweak the decay rate dynamically. It is typical to pass the count of training steps, usually kept in a variable that is incremented at each step, in which case the decay rate is lower at the start of training. This makes moving averages move faster. If passed, the actual decay rate used is:

```
min(decay, (1 + num_updates) / (10 + num_updates))
```

Args:

- `decay` : Float. The decay to use.
- `num_updates` : Optional count of number of updates applied to variables.
- `name` : String. Optional prefix name to use for the name of ops added in `apply()`.

`tf.train.ExponentialMovingAverage.apply(var_list=None)`

Maintains moving averages of variables.

`var_list` must be a list of `Variable` or `Tensor` objects. This method creates shadow variables for all elements of `var_list`. Shadow variables for `Variable` objects are initialized to the variable's initial value. They will be added to the `GraphKeys.MOVING_AVERAGE_VARIABLES` collection. For `Tensor` objects, the shadow variables are initialized to 0.

shadow variables are created with `trainable=False` and added to the `GraphKeys.ALL_VARIABLES` collection. They will be returned by calls to `tf.all_variables()`.

Returns an op that updates all shadow variables as described above.

Note that `apply()` can be called multiple times with different lists of variables.

Args:

- `var_list` : A list of Variable or Tensor objects. The variables and Tensors must be of types float32 or float64.

Returns:

An Operation that updates the moving averages.

Raises:

- `TypeError` : If the arguments are not all float32 or float64.

- `ValueError` : If the moving average of one of the variables is already being computed.
-

`tf.train.ExponentialMovingAverage.average_name(var)`

Returns the name of the `variable` holding the average for `var`.

The typical scenario for `ExponentialMovingAverage` is to compute moving averages of variables during training, and restore the variables from the computed moving averages during evaluations.

To restore variables, you have to know the name of the shadow variables. That name and the original variable can then be passed to a `saver()` object to restore the variable from the moving average value with: `saver = tf.train.Saver({ema.average_name(var): var})`

`average_name()` can be called whether or not `apply()` has been called.

Args:

- `var` : A `Variable` object.

Returns:

A string: The name of the variable that will be used or was used by the `ExponentialMovingAverage` class to hold the moving average of `var`.

`tf.train.ExponentialMovingAverage.average(var)`

Returns the `variable` holding the average of `var`.

Args:

- `var` : A `Variable` object.

Returns:

A `variable` object or `None` if the moving average of `var` is not maintained..

`tf.train.ExponentialMovingAverage.variables_to_restore(moving_avg_variables=None)`

Returns a map of names to `Variables` to restore.

If a variable has a moving average, use the moving average variable name as the restore name; otherwise, use the variable name.

For example,

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

Below is an example of such mapping:

```
conv/batchnorm/gamma/ExponentialMovingAverage: conv/batchnorm/gamma,  
conv_4/conv2d_params/ExponentialMovingAverage: conv_4/conv2d_params,  
global_step: global_step
```

Args:

- `moving_avg_variables` : a list of variables that require to use of the moving variable name to be restored. If None, it will default to `variables.moving_average_variables() + variables.trainable_variables()`

Returns:

A map from `restore_names` to variables. The `restore_name` can be the `moving_average` version of the variable name if it exist, or the original variable name.

Coordinator and QueueRunner

See [Threading and Queues](#) for how to use threads and queues. For documentation on the Queue API, see [Queues](#).

class `tf.train.Coordinator`

A coordinator for threads.

This class implements a simple mechanism to coordinate the termination of a set of threads.

Usage:

```
# Create a coordinator.
coord = Coordinator()
# Start a number of threads, passing the coordinator to each of them.
...start thread 1...(coord, ...)
...start thread N...(coord, ...)
# Wait for all the threads to terminate.
coord.join(threads)
```

Any of the threads can call `coord.request_stop()` to ask for all the threads to stop. To cooperate with the requests, each thread must check for `coord.should_stop()` on a regular basis. `coord.should_stop()` returns `True` as soon as `coord.request_stop()` has been called.

A typical thread running with a coordinator will do something like:

```
while not coord.should_stop():
    ...do some work...
```

Exception handling:

A thread can report an exception to the coordinator as part of the `should_stop()` call. The exception will be re-raised from the `coord.join()` call.

Thread code:

```
try:
    while not coord.should_stop():
        ...do some work...
except Exception as e:
    coord.request_stop(e)
```

Main code:

```
try:
    ...
coord = Coordinator()
# Start a number of threads, passing the coordinator to each of them.
...start thread 1...(coord, ...)
...start thread N...(coord, ...)
# Wait for all the threads to terminate.
coord.join(threads)
except Exception as e:
    ...exception that was passed to coord.request_stop()
```

To simplify the thread implementation, the Coordinator provides a context handler `stop_on_exception()` that automatically requests a stop if an exception is raised. Using the context handler the thread code above can be written as:

```
with coord.stop_on_exception():
    while not coord.should_stop():
        ...do some work...
```

Grace period for stopping:

After a thread has called `coord.request_stop()` the other threads have a fixed time to stop, this is called the 'stop grace period' and defaults to 2 minutes. If any of the threads is still alive after the grace period expires `coord.join()` raises a `RuntimeException` reporting the laggards.

```
try:
    ...
    coord = Coordinator()
    # Start a number of threads, passing the coordinator to each of them.
    ...start thread 1...(coord, ...)
    ...start thread N...(coord, ...)
    # Wait for all the threads to terminate, give them 10s grace period
    coord.join(threads, stop_grace_period_secs=10)
except RuntimeException:
    ...one of the threads took more than 10s to stop after request_stop()
    ...was called.
except Exception:
    ...exception that was passed to coord.request_stop()
```

tf.train.Coordinator.__init__()

Create a new Coordinator.

tf.train.Coordinator.clear_stop()

Clears the stop flag.

After this is called, calls to `should_stop()` will return `False`.

tf.train.Coordinator.join(threads, stop_grace_period_secs=120)

Wait for threads to terminate.

Blocks until all `threads` have terminated or `request_stop()` is called.

After the threads stop, if an `exc_info` was passed to `request_stop`, that exception is re-raised.

Grace period handling: When `request_stop()` is called, threads are given 'stop_grace_period_secs' seconds to terminate. If any of them is still alive after that period expires, a `RuntimeError` is raised. Note that if an `exc_info` was passed to `request_stop()` then it is raised instead of that `RuntimeError`.

Args:

- `threads` : List of `threading.Thread`. The started threads to join.
- `stop_grace_period_secs` : Number of seconds given to threads to stop after `request_stop()` has been called.

Raises:

- `RuntimeError` : If any thread is still alive after `request_stop()` is called and the grace period expires.

tf.train.Coordinator.request_stop(ex=None)

Request that the threads stop.

After this is called, calls to `should_stop()` will return `True`.

Note: If an exception is being passed in, it must be in the context of handling the exception (i.e. `try: ... except Exception as ex: ...`) and not a newly created one.

Args:

- `ex` : Optional `Exception`, or Python `exc_info` tuple as returned by `sys.exc_info()`. If this is the first call to `request_stop()` the corresponding exception is recorded and re-raised from `join()`.

tf.train.Coordinator.should_stop()

Check if stop was requested.

Returns:

True if a stop was requested.

tf.train.Coordinator.stop_on_exception()

Context manager to request stop when an Exception is raised.

Code that uses a coordinator must catch exceptions and pass them to the `request_stop()` method to stop the other threads managed by the coordinator.

This context handler simplifies the exception handling. Use it as follows:

```
with coord.stop_on_exception():
    # Any exception raised in the body of the with
    # clause is reported to the coordinator before terminating
    # the execution of the body.
    ...body...
```

This is completely equivalent to the slightly longer code:

```
try:
    ...body...
except Exception as ex:
    coord.request_stop(ex)
```

Yields:

nothing.

tf.train.Coordinator.wait_for_stop(timeout=None)

Wait till the Coordinator is told to stop.

Args:

- `timeout` : Float. Sleep for up to that many seconds waiting for `should_stop()` to become True.

Returns:

True if the Coordinator is told stop, False if the timeout expired.

class tf.train.QueueRunner

Holds a list of enqueue operations for a queue, each to be run in a thread.

Queues are a convenient TensorFlow mechanism to compute tensors asynchronously using multiple threads. For example in the canonical 'Input Reader' setup one set of threads generates filenames in a queue; a second set of threads read records from the files, processes them, and enqueues tensors on a second queue; a third set of threads dequeues these input records to construct batches and runs them through training operations.

There are several delicate issues when running multiple threads that way: closing the queues in sequence as the input is exhausted, correctly catching and reporting exceptions, etc.

The `QueueRunner`, combined with the `Coordinator`, helps handle these issues.

`tf.train.QueueRunner.__init__(queue=None, enqueue_ops=None, close_op=None, cancel_op=None, queue_runner_def=None)`

Create a QueueRunner.

On construction the `QueueRunner` adds an op to close the queue. That op will be run if the enqueue ops raise exceptions.

When you later call the `create_threads()` method, the `QueueRunner` will create one thread for each op in `enqueue_ops`. Each thread will run its enqueue op in parallel with the other threads. The enqueue ops do not have to all be the same op, but it is expected that they all enqueue tensors in `queue`.

Args:

- `queue` : A `Queue`.
- `enqueue_ops` : List of enqueue ops to run in threads later.
- `close_op` : Op to close the queue. Pending enqueue ops are preserved.
- `cancel_op` : Op to close the queue and cancel pending enqueue ops.
- `queue_runner_def` : Optional `QueueRunnerDef` protocol buffer. If specified, recreates the QueueRunner from its contents. `queue_runner_def` and the other arguments are mutually exclusive.

Raises:

- `ValueError` : If both `queue_runner_def` and `queue` are both specified.
- `ValueError` : If `queue` or `enqueue_ops` are not provided when not restoring from

```
queue_runner_def .
```

tf.train.QueueRunner.cancel_op

tf.train.QueueRunner.close_op

```
tf.train.QueueRunner.create_threads(sess, coord=None,  
daemon=False, start=False)
```

Create threads to run the enqueue ops.

This method requires a session in which the graph was launched. It creates a list of threads, optionally starting them. There is one thread for each op passed in `enqueue_ops`.

The `coord` argument is an optional coordinator, that the threads will use to terminate together and report exceptions. If a coordinator is given, this method starts an additional thread to close the queue when the coordinator requests a stop.

This method may be called again as long as all threads from a previous call have stopped.

Args:

- `sess` : A `Session`.
- `coord` : Optional `Coordinator` object for reporting errors and checking stop conditions.
- `daemon` : Boolean. If `True` make the threads daemon threads.
- `start` : Boolean. If `True` starts the threads. If `False` the caller must call the `start()` method of the returned threads.

Returns:

A list of threads.

Raises:

- `RuntimeError` : If threads from a previous call to `create_threads()` are still running.

tf.train.QueueRunner.enqueue_ops

tf.train.QueueRunner.exceptions_raised

Exceptions raised but not handled by the `QueueRunner` threads.

Exceptions raised in queue runner threads are handled in one of two ways depending on whether or not a `Coordinator` was passed to `create_threads()`:

- With a `Coordinator`, exceptions are reported to the coordinator and forgotten by the `QueueRunner`.
- Without a `Coordinator`, exceptions are captured by the `QueueRunner` and made available in this `exceptions_raised` property.

Returns:

A list of Python `Exception` objects. The list is empty if no exception was captured. (No exceptions are captured when using a Coordinator.)

tf.train.QueueRunner.from_proto(queue_runner_def)

Returns a `QueueRunner` object created from `queue_runner_def`.

tf.train.QueueRunner.name

The string name of the underlying Queue.

tf.train.QueueRunner.queue

tf.train.QueueRunner.to_proto()

Converts this `QueueRunner` to a `QueueRunnerDef` protocol buffer.

Returns:

A `QueueRunnerDef` protocol buffer.

tf.train.add_queue_runner(qr, collection='queue_runners')

Adds a `QueueRunner` to a collection in the graph.

When building a complex model that uses many queues it is often difficult to gather all the queue runners that need to be run. This convenience function allows you to add a queue runner to a well known collection in the graph.

The companion method `start_queue_runners()` can be used to start threads for all the collected queue runners.

Args:

- `qr` : A `QueueRunner`.
- `collection` : A `GraphKey` specifying the graph collection to add the queue runner to.
Defaults to `GraphKeys.QUEUE_RUNNERS`.

```
tf.train.start_queue_runners(sess=None,
coord=None, daemon=True, start=True,
collection='queue_runners')
```

Starts all queue runners collected in the graph.

This is a companion method to `add_queue_runner()`. It just starts threads for all queue runners collected in the graph. It returns the list of all threads.

Args:

- `sess` : `Session` used to run the queue ops. Defaults to the default session.
- `coord` : Optional `Coordinator` for coordinating the started threads.
- `daemon` : Whether the threads should be marked as `daemons`, meaning they don't block program exit.
- `start` : Set to `False` to only create the threads, not start them.
- `collection` : A `GraphKey` specifying the graph collection to get the queue runners from.
Defaults to `GraphKeys.QUEUE_RUNNERS`.

Returns:

A list of threads.

Distributed execution

See [Distributed TensorFlow](#) for more information about how to configure a distributed TensorFlow program.

class tf.train.Server

An in-process TensorFlow server, for use in distributed training.

A `tf.train.Server` instance encapsulates a set of devices and a `tf.Session` target that can participate in distributed training. A server belongs to a cluster (specified by a `tf.train.ClusterSpec`), and corresponds to a particular task in a named job. The server can communicate with any other server in the same cluster.

```
tf.train.Server.__init__(server_or_cluster_def,  
job_name=None, task_index=None, protocol=None, start=True)
```

Creates a new server with the given definition.

The `job_name`, `task_index`, and `protocol` arguments are optional, and override any information provided in `server_or_cluster_def`.

Args:

- `server_or_cluster_def` : A `tf.train.ServerDef` or `tf.train.ClusterDef` protocol buffer, or a `tf.train.ClusterSpec` object, describing the server to be created and/or the cluster of which it is a member.
- `job_name` : (Optional.) Specifies the name of the job of which the server is a member. Defaults to the value in `server_or_cluster_def`, if specified.
- `task_index` : (Optional.) Specifies the task index of the server in its job. Defaults to the value in `server_or_cluster_def`, if specified. Otherwise defaults to 0 if the server's job has only one task.
- `protocol` : (Optional.) Specifies the protocol to be used by the server. Acceptable values include `"grpc"`. Defaults to the value in `server_or_cluster_def`, if specified. Otherwise defaults to `"grpc"`.
- `start` : (Optional.) Boolean, indicating whether to start the server after creating it. Defaults to `True`.

Raises:

`tf.errors.OpError`: Or one of its subclasses if an error occurs while creating the TensorFlow server.

```
tf.train.Server.create_local_server(start=True)
```

Creates a new single-process cluster running on the local host.

This method is a convenience wrapper for creating a `tf.train.Server` with a `tf.train.ServerDef` that specifies a single-process cluster containing a single task in a job called `"local"`.

Args:

- `start` : (Optional.) Boolean, indicating whether to start the server after creating it. Defaults to `True`.

Returns:

A local `tf.train.Server`.

`tf.train.Server.target`

Returns the target for a `tf.Session` to connect to this server.

To create a `tf.Session` that connects to this server, use the following snippet:

```
server = tf.train.Server(...)  
with tf.Session(server.target):  
    # ...
```

Returns:

A string containing a session target for this server.

`tf.train.Server.start()`

Starts this server.

Raises:

`tf.errors.OpError`: Or one of its subclasses if an error occurs while starting the TensorFlow server.

`tf.train.Server.join()`

Blocks until the server has shut down.

This method currently blocks forever.

Raises:

`tf.errors.OpError`: Or one of its subclasses if an error occurs while joining the TensorFlow server.

class `tf.train.Supervisor`

A training helper that checkpoints models and computes summaries.

The Supervisor is a small wrapper around a `Coordinator`, a `Saver`, and a `SessionManager` that takes care of common needs of Tensorflow training programs.

Use for a single program

```
with tf.Graph().as_default():
    ...add operations to the graph...
    # Create a Supervisor that will checkpoint the model in '/tmp/mydir'.
    sv = Supervisor(logdir='/tmp/mydir')
    # Get a Tensorflow session managed by the supervisor.
    with sv.managed_session(FLAGS.master) as sess:
        # Use the session to train the graph.
        while not sv.should_stop():
            sess.run(<my_train_op>)
```

Within the `with sv.managed_session()` block all variables in the graph have been initialized. In addition, a few services have been started to checkpoint the model and add summaries to the event log.

If the program crashes and is restarted, the managed session automatically reinitialize variables from the most recent checkpoint.

The supervisor is notified of any exception raised by one of the services. After an exception is raised, `should_stop()` returns `True`. In that case the training loop should also stop. This is why the training loop has to check for `sv.should_stop()`.

Exceptions that indicate that the training inputs have been exhausted, `tf.errorsOutOfRange`, also cause `sv.should_stop()` to return `True` but are not re-raised from the `with` block: they indicate a normal termination.

Use for multiple replicas

To train with replicas you deploy the same program in a `cluster`. One of the tasks must be identified as the *chief*: the task that handles initialization, checkpoints, summaries, and recovery. The other tasks depend on the *chief* for these services.

The only change you have to do to the single program code is to indicate if the program is running as the *chief*.

```
# Choose a task as the chief. This could be based on server_def.task_index,
# or job_def.name, or job_def.tasks. It's entirely up to the end user.
# But there can be only one *chief*.
is_chief = (server_def.task_index == 0)
server = tf.train.Server(server_def)

with tf.Graph().as_default():
    ...add operations to the graph...
    # Create a Supervisor that uses log directory on a shared file system.
    # Indicate if you are the 'chief'
    sv = Supervisor(logdir='/shared_directory/...', is_chief=is_chief)
    # Get a Session in a TensorFlow server on the cluster.
    with sv.managed_session(server.target) as sess:
        # Use the session to train the graph.
        while not sv.should_stop():
            sess.run(<my_train_op>)
```

In the *chief* task, the `Supervisor` works exactly as in the first example above. In the other tasks `sv.managed_session()` waits for the Model to have been initialized before returning a session to the training code. The non-chief tasks depend on the chief tasks for initializing the model.

If one of the tasks crashes and restarts, `managed_session()` checks if the Model is initialized. If yes, it just creates a session and returns it to the training code that proceeds normally. If the model needs to be initialized, the chief task takes care of reinitializing it; the other tasks just wait for the model to have been initialized.

NOTE: This modified program still works fine as a single program. The single program marks itself as the chief.

What `master` string to use

Whether you are running on your machine or in the cluster you can use the following values for the `--master` flag:

- Specifying `''` requests an in-process session that does not use RPC.
- Specifying `'local'` requests a session that uses the RPC-based "Master interface" to run TensorFlow programs. See `tf.train.Server.create_local_server()` for details.

- Specifying `'grpc://hostname:port'` requests a session that uses the RPC interface to a specific , and also allows the in-process master to access remote tensorflow workers. Often, it is appropriate to pass `server.target` (for some `tf.train.Server` named `'server'`).

Advanced use

Launching additional services

`managed_session()` launches the Checkpoint and Summary services (threads). If you need more services to run you can simply launch them in the block controlled by `managed_session()` .

Example: Start a thread to print losses. We want this thread to run every 60 seconds, so we launch it with `sv.loop()` .

```
...
sv = Supervisor(logdir='/tmp/mydir')
with sv.managed_session(FLAGS.master) as sess:
    sv.loop(60, print_loss, (sess))
    while not sv.should_stop():
        sess.run(my_train_op)
```

Launching fewer services

`managed_session()` launches the "summary" and "checkpoint" threads which use either the optionally `summary_op` and `saver` passed to the constructor, or default ones created automatically by the supervisor. If you want to run your own summary and checkpointing logic, disable these services by passing `None` to the `summary_op` and `saver` parameters.

Example: Create summaries manually every 100 steps in the chief.

```
# Create a Supervisor with no automatic summaries.
sv = Supervisor(logdir='/tmp/mydir', is_chief=is_chief, summary_op=None)
# As summary_op was None, managed_session() does not start the
# summary thread.
with sv.managed_session(FLAGS.master) as sess:
    for step in xrange(1000000):
        if sv.should_stop():
            break
        if is_chief and step % 100 == 0:
            # Create the summary every 100 chief steps.
            sv.summary_computed(sess, sess.run(my_summary_op))
        else:
            # Train normally
            sess.run(my_train_op)
```

Custom model initialization

`managed_session()` only supports initializing the model by running an `init_op` or restoring from the latest checkpoint. If you have special initialization needs, see how to specify a `local_init_op` when creating the supervisor. You can also use the `SessionManager` directly to create a session and check if it could be initialized automatically.

```
tf.train.Supervisor.__init__(graph=None, ready_op=0,
is_chief=True, init_op=0, init_feed_dict=None,
local_init_op=0, logdir=None, summary_op=0, saver=0,
global_step=0, save_summaries_secs=120, save_model_secs=600,
recovery_wait_secs=30, stop_grace_secs=120,
checkpoint_basename='model.ckpt', session_manager=None,
summary_writer=0, init_fn=None)
```

Create a `Supervisor`.

Args:

- `graph` : A `Graph`. The graph that the model will use. Defaults to the default `Graph`. The supervisor may add operations to the graph before creating a session, but the graph should not be modified by the caller after passing it to the supervisor.
- `ready_op` : 1-D string `Tensor`. This tensor is evaluated by supervisors in `prepare_or_wait_for_session()` to check if the model is ready to use. The model is considered ready if it returns an empty array. Defaults to the tensor returned from `tf.report_uninitialized_variables()`. If `None`, the model is not checked for readiness.
- `is_chief` : If `True`, create a chief supervisor in charge of initializing and restoring the model. If `False`, create a supervisor that relies on a chief supervisor for inits and restore.
- `init_op` : `Operation`. Used by chief supervisors to initialize the model when it can not be recovered. Defaults to an `Operation` that initializes all variables. If `None`, no initialization is done automatically unless you pass a value for `init_fn`, see below.
- `init_feed_dict` : A dictionary that maps `Tensor` objects to feed values. This feed dictionary will be used when `init_op` is evaluated.
- `local_init_op` : `Operation`. Used by all supervisors to run initializations that should run for every new supervisor instance. By default these are table initializers and initializers for local variables. If `None`, no further per supervisor-instance initialization is done automatically.
- `logdir` : A string. Optional path to a directory where to checkpoint the model and log events for the visualizer. Used by chief supervisors. The directory will be created if it does not exist.
- `summary_op` : An `Operation` that returns a `Summary` for the event logs. Used by chief

supervisors if a `logdir` was specified. Defaults to the operation returned from `merge_all_summaries()`. If `None`, summaries are not computed automatically.

- `saver` : A Saver object. Used by chief supervisors if a `logdir` was specified. Defaults to the saved returned by `Saver()`. If `None`, the model is not saved automatically.
- `global_step` : An integer Tensor of size 1 that counts steps. The value from '`global_step`' is used in summaries and checkpoint filenames. Default to the op named '`global_step`' in the graph if it exists, is of rank 1, size 1, and of type `tf.int32` or `tf.int64`. If `None` the global step is not recorded in summaries and checkpoint files. Used by chief supervisors if a `logdir` was specified.
- `save_summaries_secs` : Number of seconds between the computation of summaries for the event log. Defaults to 120 seconds. Pass 0 to disable summaries.
- `save_model_secs` : Number of seconds between the creation of model checkpoints. Defaults to 600 seconds. Pass 0 to disable checkpoints.
- `recovery_wait_secs` : Number of seconds between checks that the model is ready. Used by supervisors when waiting for a chief supervisor to initialize or restore the model. Defaults to 30 seconds.
- `stop_grace_secs` : Grace period, in seconds, given to running threads to stop when `stop()` is called. Defaults to 120 seconds.
- `checkpoint_basename` : The basename for checkpoint saving.
- `session_manager` : `SessionManager`, which manages Session creation and recovery. If it is `None`, a default `SessionManager` will be created with the set of arguments passed in for backwards compatibility.
- `summary_writer` : `SummaryWriter` to use or `USE_DEFAULT`. Can be `None` to indicate that no summaries should be written.
- `init_fn` : Optional callable used to initialize the model. Called after the optional `init_op` is called. The callable must accept one argument, the session being initialized.

Returns:

A `Supervisor`.

```
tf.train.Supervisor.managed_session(master='', config=None,
start_standard_services=True, close_summary_writer=True)
```

Returns a context manager for a managed session.

This context manager creates and automatically recovers a session. It optionally starts the standard services that handle checkpoints and summaries. It monitors exceptions raised from the `with` block or from the services and stops the supervisor as needed.

The context manager is typically used as follows:

```
def train():
    sv = tf.train.Supervisor(...)
    with sv.managed_session(<master>) as sess:
        for step in xrange(..):
            if sv.should_stop():
                break
            sess.run(<my training op>
            ...do other things needed at each training step...
```

An exception raised from the `with` block or one of the service threads is raised again when the block exits. This is done after stopping all threads and closing the session. For example, an `AbortedError` exception, raised in case of preemption of one of the workers in a distributed model, is raised again when the block exits.

If you want to retry the training loop in case of preemption you can do it as follows:

```
def main(...):
    while True
        try:
            train()
        except tf.errors.Aborted:
            pass
```

As a special case, exceptions used for control flow, such as `OutOfRangeError` which reports that input queues are exhausted, are not raised again from the `with` block: they indicate a clean termination of the training loop and are considered normal termination.

Args:

- `master` : name of the TensorFlow master to use. See the `tf.Session` constructor for how this is interpreted.
- `config` : Optional `ConfigProto` proto used to configure the session. Passed as-is to create the session.
- `start_standard_services` : Whether to start the standard services, such as checkpoint, summary and step counter.
- `close_summary_writer` : Whether to close the summary writer when closing the session. Defaults to True.

Returns:

A context manager that yields a `Session` restored from the latest checkpoint or initialized from scratch if no checkpoint exists. The session is closed when the `with` block exits.

```
tf.train.Supervisor.prepare_or_wait_for_session(master='',
config=None, wait_for_checkpoint=False, max_wait_secs=7200,
start_standard_services=True)
```

Make sure the model is ready to be used.

Create a session on 'master', recovering or initializing the model as needed, or wait for a session to be ready. If running as the chief and `start_standard_service` is set to True, also call the session manager to start the standard services.

Args:

- `master` : name of the TensorFlow master to use. See the `tf.Session` constructor for how this is interpreted.
- `config` : Optional ConfigProto proto used to configure the session, which is passed as-is to create the session.
- `wait_for_checkpoint` : Whether we should wait for the availability of a checkpoint before creating Session. Defaults to False.
- `max_wait_secs` : Maximum time to wait for the session to become available.
- `start_standard_services` : Whether to start the standard services and the queue runners.

Returns:

A Session object that can be used to drive the model.

```
tf.train.Supervisor.start_standard_services(sess)
```

Start the standard services for 'sess'.

This starts services in the background. The services started depend on the parameters to the constructor and may include:

- A Summary thread computing summaries every `save_summaries_secs`.
- A Checkpoint thread saving the model every `save_model_secs`.
- A StepCounter thread measure step time.

Args:

- `sess` : A Session.

Returns:

A list of threads that are running the standard services. You can use the Supervisor's Coordinator to join these threads with: `sv.coord.Join()`

Raises:

- `RuntimeError` : If called with a non-chief Supervisor.
- `ValueError` : If not `logdir` was passed to the constructor as the services need a log directory.

```
tf.train.Supervisor.start_queue_runners(sess,  
queue_runners=None)
```

Start threads for `QueueRunners`.

Note that the queue runners collected in the graph key `QUEUE_RUNNERS` are already started automatically when you create a session with the supervisor, so unless you have non-collected queue runners to start you do not need to call this explicitly.

Args:

- `sess` : A `Session`.
- `queue_runners` : A list of `QueueRunners`. If not specified, we'll use the list of queue runners gathered in the graph under the key `GraphKeys.QUEUE_RUNNERS`.

Returns:

The list of threads started for the `QueueRunners`.

```
tf.train.Supervisor.summary_computed(sess, summary,  
global_step=None)
```

Indicate that a summary was computed.

Args:

- `sess` : A `Session` object.
- `summary` : A `Summary` proto, or a string holding a serialized `summary` proto.
- `global_step` : Int. global step this summary is associated with. If `None`, it will try to fetch the current step.

Raises:

- `TypeError` : if 'summary' is not a `Summary` proto or a string.

- `RuntimeError` : if the Supervisor was created without a `logdir` .
-

`tf.train.Supervisor.stop(threads=None, close_summary_writer=True)`

Stop the services and the coordinator.

This does not close the session.

Args:

- `threads` : Optional list of threads to join with the coordinator. If `None` , defaults to the threads running the standard services, the threads started for `QueueRunners` , and the threads started by the `loop()` method. To wait on additional threads, pass the list in this parameter.
 - `close_summary_writer` : Whether to close the `summary_writer` . Defaults to `True` if the summary writer was created by the supervisor, `False` otherwise.
-

`tf.train.Supervisor.request_stop(ex=None)`

Request that the coordinator stop the threads.

See `Coordinator.request_stop()` .

Args:

- `ex` : Optional `Exception` , or Python `exc_info` tuple as returned by `sys.exc_info()` . If this is the first call to `request_stop()` the corresponding exception is recorded and re-raised from `join()` .
-

`tf.train.Supervisor.should_stop()`

Check if the coordinator was told to stop.

See `Coordinator.should_stop()` .

Returns:

True if the coordinator was told to stop, False otherwise.

`tf.train.Supervisor.stop_on_exception()`

Context handler to stop the supervisor when an exception is raised.

See `Coordinator.stop_on_exception()`.

Returns:

A context handler.

`tf.train.Supervisor.wait_for_stop()`

Block waiting for the coordinator to stop.

Other Methods

`tf.train.Supervisor.Loop(timer_interval_secs, target, args=None, kwargs=None)`

Start a LooperThread that calls a function periodically.

If `timer_interval_secs` is `None` the thread calls `target(*args, **kwargs)` repeatedly.

Otherwise it calls it every `timer_interval_secs` seconds. The thread terminates when a stop is requested.

The started thread is added to the list of threads managed by the supervisor so it does not need to be passed to the `stop()` method.

Args:

- `timer_interval_secs` : Number. Time boundaries at which to call `target` .
- `target` : A callable object.
- `args` : Optional arguments to pass to `target` when calling it.
- `kwargs` : Optional keyword arguments to pass to `target` when calling it.

Returns:

The started thread.

```
tf.train.Supervisor.PrepareSession(master='', config=None,  
wait_for_checkpoint=False, max_wait_secs=7200,  
start_standard_services=True)
```

Make sure the model is ready to be used.

Create a session on 'master', recovering or initializing the model as needed, or wait for a session to be ready. If running as the chief and `start_standard_service` is set to True, also call the session manager to start the standard services.

Args:

- `master` : name of the TensorFlow master to use. See the `tf.Session` constructor for how this is interpreted.
- `config` : Optional ConfigProto proto used to configure the session, which is passed as-is to create the session.
- `wait_for_checkpoint` : Whether we should wait for the availability of a checkpoint before creating Session. Defaults to False.
- `max_wait_secs` : Maximum time to wait for the session to become available.
- `start_standard_services` : Whether to start the standard services and the queue runners.

Returns:

A Session object that can be used to drive the model.

```
tf.train.Supervisor.RequestStop(ex=None)
```

Request that the coordinator stop the threads.

See `Coordinator.request_stop()`.

Args:

- `ex` : Optional `Exception`, or Python `exc_info` tuple as returned by `sys.exc_info()`. If this is the first call to `request_stop()` the corresponding exception is recorded and re-raised from `join()`.

```
tf.train.Supervisor.ShouldStop()
```

Check if the coordinator was told to stop.

See `Coordinator.should_stop()`.

Returns:

True if the coordinator was told to stop, False otherwise.

**`tf.train.Supervisor.StartQueueRunners(sess,
queue_runners=None)`**

Start threads for `QueueRunners`.

Note that the queue runners collected in the graph key `QUEUE_RUNNERS` are already started automatically when you create a session with the supervisor, so unless you have non-collected queue runners to start you do not need to call this explicitly.

Args:

- `sess` : A `Session`.
- `queue_runners` : A list of `QueueRunners`. If not specified, we'll use the list of queue runners gathered in the graph under the key `GraphKeys.QUEUE_RUNNERS`.

Returns:

The list of threads started for the `QueueRunners`.

`tf.train.Supervisor.StartStandardServices(sess)`

Start the standard services for 'sess'.

This starts services in the background. The services started depend on the parameters to the constructor and may include:

- A Summary thread computing summaries every `save_summaries_secs`.
- A Checkpoint thread saving the model every `save_model_secs`.
- A StepCounter thread measure step time.

Args:

- `sess` : A Session.

Returns:

A list of threads that are running the standard services. You can use the Supervisor's Coordinator to join these threads with: `sv.coord.Join()`

Raises:

- `RuntimeError` : If called with a non-chief Supervisor.
 - `ValueError` : If not `logdir` was passed to the constructor as the services need a log directory.
-

`tf.train.Supervisor.Stop(threads=None, close_summary_writer=True)`

Stop the services and the coordinator.

This does not close the session.

Args:

- `threads` : Optional list of threads to join with the coordinator. If `None`, defaults to the threads running the standard services, the threads started for `QueueRunners`, and the threads started by the `loop()` method. To wait on additional threads, pass the list in this parameter.
 - `close_summary_writer` : Whether to close the `summary_writer`. Defaults to `True` if the summary writer was created by the supervisor, `False` otherwise.
-

`tf.train.Supervisor.StopOnException()`

Context handler to stop the supervisor when an exception is raised.

See `Coordinator.stop_on_exception()`.

Returns:

A context handler.

`tf.train.Supervisor.SummaryComputed(sess, summary, global_step=None)`

Indicate that a summary was computed.

Args:

- `sess` : A `Session` object.
- `summary` : A `Summary` proto, or a string holding a serialized summary proto.
- `global_step` : Int. global step this summary is associated with. If `None`, it will try to fetch the current step.

Raises:

- `TypeError` : if 'summary' is not a Summary proto or a string.
- `RuntimeError` : if the Supervisor was created without a `logdir`.

tf.train.Supervisor.WaitForStop()

Block waiting for the coordinator to stop.

tf.train.Supervisor.coord

Return the Coordinator used by the Supervisor.

The Coordinator can be useful if you want to run multiple threads during your training.

Returns:

A Coordinator object.

tf.train.Supervisor.global_step

Return the global_step Tensor used by the supervisor.

Returns:

An integer Tensor for the global_step.

tf.train.Supervisor.init_feed_dict

Return the feed dictionary used when evaluating the `init_op`.

Returns:

A feed dictionary or `None`.

tf.train.Supervisor.init_op

Return the Init Op used by the supervisor.

Returns:

An Op or `None`.

`tf.train.Supervisor.is_chief`

Return True if this is a chief supervisor.

Returns:

A bool.

`tf.train.Supervisor.loop(timer_interval_secs, target, args=None, kwargs=None)`

Start a LooperThread that calls a function periodically.

If `timer_interval_secs` is `None` the thread calls `target(*args, **kwargs)` repeatedly.

Otherwise it calls it every `timer_interval_secs` seconds. The thread terminates when a stop is requested.

The started thread is added to the list of threads managed by the supervisor so it does not need to be passed to the `stop()` method.

Args:

- `timer_interval_secs` : Number. Time boundaries at which to call `target`.
- `target` : A callable object.
- `args` : Optional arguments to pass to `target` when calling it.
- `kwargs` : Optional keyword arguments to pass to `target` when calling it.

Returns:

The started thread.

`tf.train.Supervisor.ready_op`

Return the Ready Op used by the supervisor.

Returns:

An Op or `None`.

`tf.train.Supervisor.save_model_secs`

Return the delay between checkpoints.

Returns:

A timestamp.

`tf.train.Supervisor.save_path`

Return the save path used by the supervisor.

Returns:

A string.

`tf.train.Supervisor.save_summaries_secs`

Return the delay between summary computations.

Returns:

A timestamp.

`tf.train.Supervisor.saver`

Return the Saver used by the supervisor.

Returns:

A Saver object.

`tf.train.Supervisor.session_manager`

Return the SessionManager used by the Supervisor.

Returns:

A SessionManager object.

tf.train.Supervisor.summary_op

Return the Summary Tensor used by the chief supervisor.

Returns:

A string Tensor for the summary or `None`.

tf.train.Supervisor.summary_writer

Return the SummaryWriter used by the chief supervisor.

Returns:

A SummaryWriter.

class tf.train.SessionManager

Training helper that restores from checkpoint and creates session.

This class is a small wrapper that takes care of session creation and checkpoint recovery. It also provides functions that facilitate coordination among multiple training threads or processes.

- Checkpointing trained variables as the training progresses.
- Initializing variables on startup, restoring them from the most recent checkpoint after a crash, or wait for checkpoints to become available.

Usage:

```
with tf.Graph().as_default():
    ...add operations to the graph...
    # Create a SessionManager that will checkpoint the model in '/tmp/mydir'.
    sm = SessionManager()
    sess = sm.prepare_session(master, init_op, saver, checkpoint_dir)
    # Use the session to train the graph.
    while True:
        sess.run(<my_train_op>)
```

`prepare_session()` initializes or restores a model. It requires `init_op` and `saver` as arguments.

A second process could wait for the model to be ready by doing the following:

```
with tf.Graph().as_default():
    ...add operations to the graph...
    # Create a SessionManager that will wait for the model to become ready.
    sm = SessionManager()
    sess = sm.wait_for_session(master)
    # Use the session to train the graph.
    while True:
        sess.run(<my_train_op>)
```

`wait_for_session()` waits for a model to be initialized by other processes.

`tf.train.SessionManager.__init__(local_init_op=None, ready_op=None, graph=None, recovery_wait_secs=30)`

Creates a SessionManager.

The `local_init_op` is an `Operation` that is run always after a new session was created. If `None`, this step is skipped.

The `ready_op` is an `Operation` used to check if the model is ready. The model is considered ready if that operation returns an empty string tensor. If the operation returns non empty string tensor, the elements are concatenated and used to indicate to the user why the model is not ready.

If `ready_op` is `None`, the model is not checked for readiness.

`recovery_wait_secs` is the number of seconds between checks that the model is ready. It is used by processes to wait for a model to be initialized or restored. Defaults to 30 seconds.

Args:

- `local_init_op` : An `Operation` run immediately after session creation. Usually used to initialize tables and local variables.
- `ready_op` : An `Operation` to check if the model is initialized.
- `graph` : The `Graph` that the model will use.
- `recovery_wait_secs` : Seconds between checks for the model to be ready.

`tf.train.SessionManager.prepare_session(master, init_op=None, saver=None, checkpoint_dir=None, wait_for_checkpoint=False, max_wait_secs=7200, config=None, init_feed_dict=None, init_fn=None)`

Creates a `Session`. Makes sure the model is ready to be used.

Creates a `Session` on 'master'. If a `saver` object is passed in, and `checkpoint_dir` points to a directory containing valid checkpoint files, then it will try to recover the model from checkpoint. If no checkpoint files are available, and `wait_for_checkpoint` is `True`, then the process would check every `recovery_wait_secs`, up to `max_wait_secs`, for recovery to succeed.

If the model cannot be recovered successfully then it is initialized by either running the provided `init_op`, or calling the provided `init_fn`. It is an error if the model cannot be recovered and neither an `init_op` or an `init_fn` are passed.

This is a convenient function for the following, with a few error checks added:

```
sess, initialized = self.recover_session(master)
if not initialized:
    if init_op:
        sess.run(init_op, feed_dict=init_feed_dict)
    if init_fn:
        init_fn(sess)
return sess
```

Args:

- `master` : String representation of the TensorFlow master to use.
- `init_op` : Optional `Operation` used to initialize the model.
- `saver` : A `Saver` object used to restore a model.
- `checkpoint_dir` : Path to the checkpoint files.
- `wait_for_checkpoint` : Whether to wait for checkpoint to become available.
- `max_wait_secs` : Maximum time to wait for checkpoints to become available.
- `config` : Optional `ConfigProto` proto used to configure the session.
- `init_feed_dict` : Optional dictionary that maps `Tensor` objects to feed values. This feed dictionary is passed to the session `run()` call when running the init op.
- `init_fn` : Optional callable used to initialize the model. Called after the optional `init_op` is called. The callable must accept one argument, the session being initialized.

Returns:

A `Session` object that can be used to drive the model.

Raises:

- `RuntimeError` : If the model cannot be initialized or recovered.

```
tf.train.SessionManager.recover_session(master, saver=None,
checkpoint_dir=None, wait_for_checkpoint=False,
max_wait_secs=7200, config=None)
```

Creates a `Session`, recovering if possible.

Creates a new session on 'master'. If the session is not initialized and can be recovered from a checkpoint, recover it.

Args:

- `master` : `String` representation of the TensorFlow master to use.
- `saver` : A `Saver` object used to restore a model.
- `checkpoint_dir` : Path to the checkpoint files.
- `wait_for_checkpoint` : Whether to wait for checkpoint to become available.
- `max_wait_secs` : Maximum time to wait for checkpoints to become available.
- `config` : Optional `ConfigProto` proto used to configure the session.

Returns:

A pair (`sess, initialized`) where 'initialized' is `True` if the session could be recovered, `False` otherwise.

```
tf.train.SessionManager.wait_for_session(master,
config=None, max_wait_secs=inf)
```

Creates a new `Session` and waits for model to be ready.

Creates a new `Session` on 'master'. Waits for the model to be initialized or recovered from a checkpoint. It's expected that another thread or process will make the model ready, and that this is intended to be used by threads/processes that participate in a distributed training configuration where a different thread/process is responsible for initializing or recovering the model being trained.

NB: The amount of time this method waits for the session is bounded by `max_wait_secs`. By default, this function will wait indefinitely.

Args:

- `master` : `String` representation of the TensorFlow master to use.
- `config` : Optional `ConfigProto` proto used to configure the session.
- `max_wait_secs` : Maximum time to wait for the session to become available.

Returns:

A `Session`. May be `None` if the operation exceeds the timeout specified by `config.operation_timeout_in_ms`.

Raises:

`tf.DeadlineExceededError`: if the session is not available after `max_wait_secs`.

class `tf.train.ClusterSpec`

Represents a cluster as a set of "tasks", organized into "jobs".

A `tf.train.ClusterSpec` represents the set of processes that participate in a distributed TensorFlow computation. Every `tf.train.Server` is constructed in a particular cluster.

To create a cluster with two jobs and five tasks, you specify the mapping from job names to lists of network addresses (typically hostname-port pairs).

```
cluster = tf.train.ClusterSpec({"worker": ["worker0.example.com:2222",
                                             "worker1.example.com:2222",
                                             "worker2.example.com:2222"],
                                 "ps": ["ps0.example.com:2222",
                                        "ps1.example.com:2222"]})
```

`tf.train.ClusterSpec.as_cluster_def()`

Returns a `tf.train.ClusterDef` protocol buffer based on this cluster.

`tf.train.ClusterSpec.as_dict()`

Returns a dictionary from job names to lists of network addresses.

Other Methods

`tf.train.ClusterSpec.__init__(cluster)`

Creates a `clusterSpec`.

Args:

- `cluster` : A dictionary mapping one or more job names to lists of network addresses, or a `tf.train.ClusterDef` protocol buffer.

Raises:

- `TypeError` : If `cluster` is not a dictionary mapping strings to lists of strings, and not a `tf.train.ClusterDef` protobuf.

`tf.train.ClusterSpec.job_tasks(job_name)`

Returns a list of tasks in the given job.

Args:

- `job_name` : The string name of a job in this cluster.

Returns:

A list of strings, corresponding to the network addresses of tasks in the given job, ordered by task index.

Raises:

- `ValueError` : If `job_name` does not name a job in this cluster.

`tf.train.ClusterSpec.jobs`

Returns a list of job names in this cluster.

Returns:

A list of strings, corresponding to the names of jobs in this cluster.

**`tf.train.replica_device_setter(ps_tasks=0,
ps_device='/job:ps',
worker_device='/job:worker',
merge_devices=True, cluster=None, ps_ops=None)`**

Return a `device` function to use when building a Graph for replicas.

Device Functions are used in `with tf.device(device_function):` statement to automatically assign devices to `Operation` objects as they are constructed. Device constraints are added from the inner-most context first, working outwards. The merging behavior adds constraints to fields that are yet unset by a more inner context. Currently the fields are (job, task, cpu/gpu).

If `cluster` is `None`, and `ps_tasks` is 0, the returned function is a no-op.

For example,

```
# To build a cluster with two ps jobs on hosts ps0 and ps1, and 3 worker
# jobs on hosts worker0, worker1 and worker2.
cluster_spec = {
    "ps": ["ps0:2222", "ps1:2222"],
    "worker": ["worker0:2222", "worker1:2222", "worker2:2222"]}
with tf.device(tf.replica_device_setter(cluster=cluster_spec)):
    # Build your graph
    v1 = tf.Variable(...) # assigned to /job:ps/task:0
    v2 = tf.Variable(...) # assigned to /job:ps/task:1
    v3 = tf.Variable(...) # assigned to /job:ps/task:0
# Run compute
```

Args:

- `ps_tasks` : Number of tasks in the `ps` job.
- `ps_device` : String. Device of the `ps` job. If empty no `ps` job is used. Defaults to `ps`.
- `worker_device` : String. Device of the `worker` job. If empty no `worker` job is used.
- `merge_devices` : Boolean . If `True`, merges or only sets a device if the device constraint is completely unset. merges device specification rather than overriding them.
- `cluster` : `ClusterDef` proto or `ClusterSpec` .
- `ps_ops` : List of `Operation` objects that need to be placed on `ps` devices.

Returns:

A function to pass to `tf.device()`.

Raises:

`TypeError` if `cluster` is not a dictionary or `clusterDef` protocol buffer.

Summary Operations

The following ops output `Summary` protocol buffers as serialized string tensors.

You can fetch the output of a summary op in a session, and pass it to a [SummaryWriter](#) to append it to an event file. Event files contain `Event` protos that can contain `summary` protos along with the timestamp and step. You can then use TensorBoard to visualize the contents of the event files. See [TensorBoard and Summaries](#) for more details.

`tf.scalar_summary(tags, values, collections=None, name=None)`

Outputs a `summary` protocol buffer with scalar values.

The input `tags` and `values` must have the same shape. The generated summary has a summary value for each tag-value pair in `tags` and `values`.

Args:

- `tags` : A `string` `Tensor`. Tags for the summaries.
- `values` : A real numeric `Tensor`. Values for the summaries.
- `collections` : Optional list of graph collections keys. The new summary op is added to these collections. Defaults to `[GraphKeys.SUMMARIES]`.
- `name` : A name for the operation (optional).

Returns:

A scalar `Tensor` of type `string`. The serialized `Summary` protocol buffer.

`tf.image_summary(tag, tensor, max_images=3, collections=None, name=None)`

Outputs a `summary` protocol buffer with images.

The summary has up to `max_images` summary values containing images. The images are built from `tensor` which must be 4-D with shape `[batch_size, height, width, channels]` and where `channels` can be:

- 1: `tensor` is interpreted as Grayscale.
- 3: `tensor` is interpreted as RGB.
- 4: `tensor` is interpreted as RGBA.

The images have the same number of channels as the input tensor. For float input, the values are normalized one image at a time to fit in the range `[0, 255]`. `uint8` values are unchanged. The op uses two different normalization algorithms:

- If the input values are all positive, they are rescaled so the largest one is 255.
- If any input value is negative, the values are shifted so input value 0.0 is at 127. They are then rescaled so that either the smallest value is 0, or the largest one is 255.

The `tag` argument is a scalar `Tensor` of type `string`. It is used to build the `tag` of the summary values:

- If `max_images` is 1, the summary value tag is '`tag/image`'.
- If `max_images` is greater than 1, the summary value tags are generated sequentially as '`tag/image/0`', '`tag/image/1`', etc.

Args:

- `tag` : A scalar `Tensor` of type `string`. Used to build the `tag` of the summary values.
- `tensor` : A 4-D `uint8` or `float32` `Tensor` of shape `[batch_size, height, width, channels]` where `channels` is 1, 3, or 4.
- `max_images` : Max number of batch elements to generate images for.
- `collections` : Optional list of `ops.GraphKeys`. The collections to add the summary to. Defaults to `[ops.GraphKeys.SUMMARIES]`
- `name` : A name for the operation (optional).

Returns:

A scalar `Tensor` of type `string`. The serialized `Summary` protocol buffer.

```
tf.audio_summary(tag, tensor, sample_rate,
max_outputs=3, collections=None, name=None)
```

Outputs a `Summary` protocol buffer with audio.

The summary has up to `max_outputs` summary values containing audio. The audio is built from `tensor` which must be 3-D with shape `[batch_size, frames, channels]` or 2-D with shape `[batch_size, frames]`. The values are assumed to be in the range of `[-1.0, 1.0]` with a sample rate of `sample_rate`.

The `tag` argument is a scalar `Tensor` of type `string`. It is used to build the `tag` of the summary values:

- If `max_outputs` is 1, the summary value tag is '`tag/audio`'.
- If `max_outputs` is greater than 1, the summary value tags are generated sequentially as '`tag/audio/0`', '`tag/audio/1`', etc.

Args:

- `tag` : A scalar `Tensor` of type `string`. Used to build the `tag` of the summary values.
- `tensor` : A 3-D `float32` `Tensor` of shape `[batch_size, frames, channels]` or a 2-D `float32` `Tensor` of shape `[batch_size, frames]`.
- `sample_rate` : The sample rate of the signal in hertz.
- `max_outputs` : Max number of batch elements to generate audio for.
- `collections` : Optional list of ops.GraphKeys. The collections to add the summary to. Defaults to `[ops.GraphKeys.SUMMARIES]`
- `name` : A name for the operation (optional).

Returns:

A scalar `Tensor` of type `string`. The serialized `Summary` protocol buffer.

`tf.histogram_summary(tag, values, collections=None, name=None)`

Outputs a `Summary` protocol buffer with a histogram.

The generated `Summary` has one summary value containing a histogram for `values`.

This op reports an `InvalidArgument` error if any value is not finite.

Args:

- `tag` : A `string` `Tensor`. 0-D. Tag to use for the summary value.
- `values` : A real numeric `Tensor`. Any shape. Values to use to build the histogram.
- `collections` : Optional list of graph collections keys. The new summary op is added to these collections. Defaults to `[GraphKeys.SUMMARIES]`.
- `name` : A name for the operation (optional).

Returns:

A scalar `Tensor` of type `string`. The serialized `Summary` protocol buffer.

`tf.nn.zero_fraction(value, name=None)`

Returns the fraction of zeros in `value`.

If `value` is empty, the result is `nan`.

This is useful in summaries to measure and report sparsity. For example,

```
z = tf.Relu(...)  
summ = tf.scalar_summary('sparsity', tf.nn.zero_fraction(z))
```

Args:

- `value` : A tensor of numeric type.
- `name` : A name for the operation (optional).

Returns:

The fraction of zeros in `value`, with type `float32`.

tf.merge_summary(inputs, collections=None, name=None)

Merges summaries.

This op creates a `Summary` protocol buffer that contains the union of all the values in the input summaries.

When the Op is run, it reports an `InvalidArgumentException` error if multiple values in the summaries to merge use the same tag.

Args:

- `inputs` : A list of `string Tensor` objects containing serialized `summary` protocol buffers.
- `collections` : Optional list of graph collections keys. The new summary op is added to these collections. Defaults to `[GraphKeys.SUMMARIES]`.
- `name` : A name for the operation (optional).

Returns:

A scalar `Tensor` of type `string`. The serialized `Summary` protocol buffer resulting from the merging.

tf.merge_all_summaries(key='summaries')

Merges all summaries collected in the default graph.

Args:

- `key` : `GraphKey` used to collect the summaries. Defaults to `GraphKeys.SUMMARIES` .

Returns:

If no summaries were collected, returns `None`. Otherwise returns a scalar `Tensor` of type `string` containing the serialized `Summary` protocol buffer resulting from the merging.

Adding Summaries to Event Files

See [Summaries and TensorBoard](#) for an overview of summaries, event files, and visualization in TensorBoard.

`class tf.train.SummaryWriter`

Writes `Summary` protocol buffers to event files.

The `SummaryWriter` class provides a mechanism to create an event file in a given directory and add summaries and events to it. The class updates the file contents asynchronously. This allows a training program to call methods to add data to the file directly from the training loop, without slowing down training.

```
tf.train.SummaryWriter.__init__(logdir, graph=None,
max_queue=10, flush_secs=120, graph_def=None)
```

Creates a `SummaryWriter` and an event file.

On construction the summary writer creates a new event file in `logdir` . This event file will contain `Event` protocol buffers constructed when you call one of the following functions:

```
add_summary(), add_session_log(), add_event(), or add_graph().
```

If you pass a `Graph` to the constructor it is added to the event file. (This is equivalent to calling `add_graph()` later).

TensorBoard will pick the graph from the file and display it graphically so you can interactively explore the graph you built. You will usually pass the graph from the session in which you launched it:

```

...create a graph...
# Launch the graph in a session.
sess = tf.Session()
# Create a summary writer, add the 'graph' to the event file.
writer = tf.train.SummaryWriter(<some-directory>, sess.graph)

```

The other arguments to the constructor control the asynchronous writes to the event file:

- `flush_secs` : How often, in seconds, to flush the added summaries and events to disk.
- `max_queue` : Maximum number of summaries or events pending to be written to disk before one of the 'add' calls block.

Args:

- `logdir` : A string. Directory where event file will be written.
- `graph` : A `Graph` object, such as `sess.graph`.
- `max_queue` : Integer. Size of the queue for pending events and summaries.
- `flush_secs` : Number. How often, in seconds, to flush the pending events and summaries to disk.
- `graph_def` : DEPRECATED: Use the `graph` argument instead.

`tf.train.SummaryWriter.add_summary(summary, global_step=None)`

Adds a `Summary` protocol buffer to the event file.

This method wraps the provided summary in an `Event` protocol buffer and adds it to the event file.

You can pass the result of evaluating any summary op, using `Session.run()` or `Tensor.eval()`, to this function. Alternatively, you can pass a `tf.Summary` protocol buffer that you populate with your own data. The latter is commonly done to report evaluation results in event files.

Args:

- `summary` : A `Summary` protocol buffer, optionally serialized as a string.
- `global_step` : Number. Optional global step value to record with the summary.

`tf.train.SummaryWriter.add_session_log(session_log, global_step=None)`

Adds a `SessionLog` protocol buffer to the event file.

This method wraps the provided session in an `Event` protocol buffer and adds it to the event file.

Args:

- `session_log` : A `SessionLog` protocol buffer.
- `global_step` : Number. Optional global step value to record with the summary.

`tf.train.SummaryWriter.add_event(event)`

Adds an event to the event file.

Args:

- `event` : An `Event` protocol buffer.

`tf.train.SummaryWriter.add_graph(graph, global_step=None, graph_def=None)`

Adds a `Graph` to the event file.

The graph described by the protocol buffer will be displayed by TensorBoard. Most users pass a graph in the constructor instead.

Args:

- `graph` : A `Graph` object, such as `sess.graph`.
- `global_step` : Number. Optional global step counter to record with the graph.
- `graph_def` : DEPRECATED. Use the `graph` parameter instead.

Raises:

- `ValueError` : If both `graph` and `graph_def` are passed to the method.

`tf.train.SummaryWriter.add_run_metadata(run_metadata, tag, global_step=None)`

Adds a metadata information for a single `session.run()` call.

Args:

- `run_metadata` : A `RunMetadata` protobuf object.
- `tag` : The tag name for this metadata.
- `global_step` : Number. Optional global step counter to record with the StepStats.

Raises:

- `ValueError` : If the provided tag was already used for this type of event.

`tf.train.SummaryWriter.flush()`

Flushes the event file to disk.

Call this method to make sure that all pending events have been written to disk.

`tf.train.SummaryWriter.close()`

Flushes the event file to disk and close the file.

Call this method when you do not need the summary writer anymore.

`tf.train.summary_iterator(path)`

An iterator for reading `Event` protocol buffers from an event file.

You can use this function to read events written to an event file. It returns a Python iterator that yields `Event` protocol buffers.

Example: Print the contents of an events file.

```
for e in tf.train.summary_iterator(path to events file):
    print(e)
```

Example: Print selected summary values.

```
# This example supposes that the events file contains summaries with a
# summary value tag 'loss'. These could have been added by calling
# `add_summary()`, passing the output of a scalar summary op created with
# with: `tf.scalar_summary(['loss'], loss_tensor)`.

for e in tf.train.summary_iterator(path to events file):
    for v in e.summary.value:
        if v.tag == 'loss':
            print(v.simple_value)
```

See the protocol buffer definitions of [Event](#) and [Summary](#) for more information about their attributes.

Args:

- `path` : The path to an event file created by a `SummaryWriter`.

Yields:

`Event` protocol buffers.

Training utilities

`tf.train.global_step(sess, global_step_tensor)`

Small helper to get the global step.

```
# Creates a variable to hold the global_step.
global_step_tensor = tf.Variable(10, trainable=False, name='global_step')
# Creates a session.
sess = tf.Session()
# Initializes the variable.
sess.run(global_step_tensor.initializer)
print('global_step: %s' % tf.train.global_step(sess, global_step_tensor))

global_step: 10
```

Args:

- `sess` : A TensorFlow `Session` object.
- `global_step_tensor` : `Tensor` or the `name` of the operation that contains the global step.

Returns:

The global step value.

```
tf.train.write_graph(graph_def, logdir, name,  
as_text=True)
```

Writes a graph proto to a file.

The graph is written as a binary proto unless `as_text` is `True`.

```
v = tf.Variable(0, name='my_variable')  
sess = tf.Session()  
tf.train.write_graph(sess.graph_def, '/tmp/my-model', 'train.pbtxt')
```

Args:

- `graph_def` : A `GraphDef` protocol buffer.
- `logdir` : Directory where to write the graph. This can refer to remote filesystems, such as Google Cloud Storage (GCS).
- `name` : Filename for the graph.
- `as_text` : If `True`, writes the graph as an ASCII proto.

Other Functions and Classes

```
class tf.trainLooperThread
```

A thread that runs code repeatedly, optionally on a timer.

This thread class is intended to be used with a `Coordinator`. It repeatedly runs code specified either as `target` and `args` or by the `run_loop()` method.

Before each run the thread checks if the coordinator has requested stop. In that case the looper thread terminates immediately.

If the code being run raises an exception, that exception is reported to the coordinator and the thread terminates. The coordinator will then request all the other threads it coordinates to stop.

You typically pass looper threads to the supervisor `Join()` method.

```
tf.trainLooperThread.__init__(coord, timer_interval_secs,
target=None, args=None, kwargs=None)
```

Create a LooperThread.

Args:

- `coord` : A Coordinator.
- `timer_interval_secs` : Time boundaries at which to call Run(), or None if it should be called back to back.
- `target` : Optional callable object that will be executed in the thread.
- `args` : Optional arguments to pass to `target` when calling it.
- `kwargs` : Optional keyword arguments to pass to `target` when calling it.

Raises:

- `ValueError` : If one of the arguments is invalid.
-

```
tf.trainLooperThread.daemon
```

A boolean value indicating whether this thread is a daemon thread (True) or not (False).

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

```
tf.trainLooperThread.getName()
```

```
tf.trainLooperThread.ident
```

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the `thread.get_ident()` function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

```
tf.trainLooperThread.isAlive()
```

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

tf.trainLooperThread.isDaemon()

tf.trainLooperThread.is_alive()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

tf.trainLooperThread.join(timeout=None)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates -- either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call isAlive() after join() to decide whether a timeout happened -- if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

tf.trainLooperThread.loop(coord, timer_interval_secs, target, args=None, kwargs=None)

Start a LooperThread that calls a function periodically.

If `timer_interval_secs` is `None` the thread calls `target(args)` repeatedly. Otherwise `target(args)` is called every `timer_interval_secs` seconds. The thread terminates when a stop of the coordinator is requested.

Args:

- `coord` : A Coordinator.
- `timer_interval_secs` : Number. Time boundaries at which to call `target`.
- `target` : A callable object.
- `args` : Optional arguments to pass to `target` when calling it.
- `kwargs` : Optional keyword arguments to pass to `target` when calling it.

Returns:

The started thread.

`tf.train.LooperThread.name`

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

`tf.train.LooperThread.run()`

`tf.train.LooperThread.run_loop()`

Called at 'timer_interval_secs' boundaries.

`tf.train.LooperThread.setDaemon(daemonic)`

`tf.train.LooperThread.setName(name)`

`tf.train.LooperThread.start()`

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

tf.trainLooperThread.start_loop()

Called when the thread starts.

tf.trainLooperThread.stop_loop()

Called when the thread stops.

tf.train.generate_checkpoint_state_proto(save_dir, model_checkpoint_path, all_model_checkpoint_paths=None)

Generates a checkpoint state proto.

Args:

- `save_dir` : Directory where the model was saved.
- `model_checkpoint_path` : The checkpoint file.
- `all_model_checkpoint_paths` : List of strings. Paths to all not-yet-deleted checkpoints, sorted from oldest to newest. If this is a non-empty list, the last element must be equal to `model_checkpoint_path`. These paths are also saved in the CheckpointState proto.

Returns:

CheckpointState proto with `model_checkpoint_path` and `all_model_checkpoint_paths` updated to either absolute paths or relative paths to the current `save_dir`.

파이썬 함수 래핑하기

참고 : `Tensor` 를 인자로 받는 함수들은 `tf.convert_to_tensor` 의 인자로 들어갈 수 있는 값들 또한 받을 수 있습니다.

[TOC]

스크립트 언어 연산자

TensorFlow는 python/numpy 함수들을 TensorFlow의 연산자로써 래핑할 수 있도록 해줍니다.

다른 함수와 클래스들

`tf.py_func(func, inp, Tout, name=None)`

python 함수를 래핑하고 이를 tensorflow의 연산자로써 사용합니다.

`func` 로 주어지는 python 함수는 numpy 배열을 입력으로 받고 numpy 배열을 출력합니다. 예를 들면,

```
def my_func(x):
    # x는 아래의 placeholder의 값을 가지는 numpy 배열이 될 것입니다.
    return np.sinh(x)
inp = tf.placeholder(tf.float32, [...])
y = py_func(my_func, [inp], [tf.float32])
```

위의 스니펫은 그래프의 연산으로 numpy의 `sinh(x)`를 호출하는 tf 그래프를 구성합니다.

인자:

- `func` : python 함수.
- `inp` : `Tensor` 의 리스트.
- `Tout` : `func` 의 반환값을 나타내는 tensorflow 데이터 타입의 리스트.
- `name` : 연산의 명칭 (선택사항).

반환값:

`func` 를 통해 계산된 `Tensor` 의 리스트.

테스팅

유닛 테스트

TensorFlow는 `unittest.TestCase` 를 상속하고 TensorFlow 테스트와 관련된 메서드를 추가한 편리한 클래스를 제공합니다. 아래에 예시가 하나 있습니다.

```
import tensorflow as tf

class SquareTest(tf.test.TestCase):

    def testSquare(self):
        with self.test_session():
            x = tf.square([2, 3])
            self.assertAllEqual(x.eval(), [4, 9])

    if __name__ == '__main__':
        tf.test.main()
```

`tf.test.TestCase` 는 `unittest.TestCase` 를 상속하지만 추가적인 메서드가 더 있습니다. 우리는 곧 이 메서드들에 대해 문서화를 할 것입니다.

tf.test.main()

모든 유닛 테스트를 실행합니다.

유틸리티

tf.test.assert_equal_graph_def(actual, expected)

두 개의 `GraphDef` 가 (대부분) 같은지 확인합니다.

두 `GraphDef` 의 원형이 같은지를 비교하는데 버전과 노드의 순서, 속성 그리고 제어 입력은 무시합니다. 노드명은 두 그래프 사이의 노드를 매칭하는데 사용되기 때문에 노드의 네이밍은 일관되어야합니다.

인자:

- `actual` : 테스트를 위한 `GraphDef`.
- `expected` : 예상값 `GraphDef`.

예외:

- `AssertionError` : 두 `GraphDef` 가 매칭이 안될 경우 발생합니다.
- `TypeError` : 둘 중 하나라도 `GraphDef` 가 아닐 경우 발생합니다.

`tf.test.get_temp_dir()`

테스트중 사용할 임시 디렉토리를 반환합니다.

테스트 후 디렉토리를 삭제할 필요가 없습니다.

반환값:

임시 디렉토리.

`tf.test.is_built_with_cuda()`

TensorFlow의 CUDA (GPU) 지원 여부를 반환합니다.

그라디언트(Gradient) 확인

`compute_gradient` 와 `compute_gradient_error` 는 등록된 해석적 그라디언트와 그래프의 수치 미분의 비교를 수행합니다.

`tf.test.compute_gradient(x, x_shape, v, v_shape, x_init_value=None, delta=0.001, init_targets=None)`

이론 및 수치적 코비안을 계산한 후 반환합니다.

만약 `x` 또는 `y` 가 복소수이면, 코비안은 여전히 실수지만 대응되는 코비안 차원은 두 배가 될 것입니다. TensorFlow 그래프가 정칙(holomorphic)일 필요는 없기 때문에 비록 입력과 출력이 모두 복소수라고 해도 이는 필수적입니다. 그리고 이는 복소수로써 표현될 수 없는 그라디언트를 가질

것입니다. 예를 들면, `x` 가 `[m]` `shape`을 갖는 복소수이고 `y` 가 `[n]` `shape`을 갖는 복소수일 때, 각 코비안 `J` 는 다음과 같은 `[m * 2, n * 2]` `shape`을 갖게될 것입니다.

```
J[::m, ::n] = d(Re y)/d(Re x)
J[::m, n:] = d(Im y)/d(Re x)
J[m:, ::n] = d(Re y)/d(Im x)
J[m:, n:] = d(Im y)/d(Im x)
```

인자:

- `x` : 텐서 또는 텐서들의 리스트.
- `x_shape` : 정수형 튜플 또는 배열 형태의 `x`의 차원입니다. `x`가 리스트라면 이는 `shape`들의 리스트입니다.
- `y` : 텐서.
- `y_shape` : 정수형 튜플 또는 배열 형태의 `y`의 차원입니다.
- `x_init_value` : (선택적인) "x"와 같은 `shape`을 가진 numpy 배열로 `x`의 초기값을 나타냅니다. 만약 `x`가 리스트라면, 이는 numpy 배열의 리스트여야합니다. 만약 값이 없다면, 함수는 초기값 텐서를 랜덤으로 선택할 것입니다.
- `delta` : (선택적인) 섭동의 크기.
- `init_targets` : 모델 파라미터 초기화를 실행하기 위한 타겟들의 리스트. TODO(mrry): 이 인자를 없앱니다.

반환값:

`dy/dx`에 대한 이론 및 수치적 코비안을 나타내는 두 개의 이차원 numpy 배열. 각각은 `x`의 원소의 갯수인 "`x_size`"개의 행과 `y`의 원소의 갯수인 "`y_size`"개의 열을 가집니다. 만약 `x`가 리스트라면, 두 개의 numpy 배열의 리스트를 반환합니다.

```
tf.test.compute_gradient_error(x, x_shape, y,
                               y_shape, x_init_value=None, delta=0.001,
                               init_targets=None)
```

그라디언트 오차를 계산합니다.

계산된 코비안과 수치적으로 추정된 코비안간의 `dy/dx`에 대한 최대 오차를 계산합니다.

이 함수는 연산들을 추가함으로써 전달된 텐서를 변경시키고 따라서 입력 텐서의 연산을 사용하는 컨슈머들을 바꿉니다.

이 함수는 현재 세션에 연산들을 추가합니다. GPU 같은 특정한 디바이스를 사용하여 오차를 계산하기 위해선 디바이스 설정을 위한 표준 메서드를 사용합니다. (예를 들면 `with sess.graph.device()`를 사용하거나 세션 생성자에 디바이스 함수를 설정)

인자:

- `x` : 텐서 또는 텐서들의 리스트.
- `x_shape` : 정수형 튜플 또는 배열 형태의 `x`의 차원입니다. `x`가 리스트라면 이는 `shape`들의 리스트입니다.
- `y` : 텐서.
- `y_shape` : 정수형 튜플 또는 배열 형태의 `y`의 차원입니다.
- `x_init_value` : (선택적인) "x"와 같은 `shape`을 가진 numpy 배열로 `x`의 초기값을 나타냅니다. 만약 `x`가 리스트라면, 이는 numpy 배열의 리스트여야합니다. 만약 값이 없다면, 함수는 초기값 텐서를 랜덤으로 선택할 것입니다.
- `delta` : (선택적인) 섭동의 크기.
- `init_targets` : 모델 파라미터 초기화를 실행하기 위한 타겟들의 리스트. TODO(mrry): 이 인자를 없앱니다.

반환값:

두 코비안간의 최대 오차값.

Layers (contrib)

[TOC]

Ops for building neural network layers, regularizers, summaries, etc.

Higher level ops for building neural network layers.

This package provides several ops that take care of creating variables that are used internally in a consistent way and provide the building blocks for many common machine learning algorithms.

`tf.contrib.layers.convolution2d(*args, **kwargs)`

Adds a 2D convolution followed by an optional batch_norm layer.

`convolution2d` creates a variable called `weights`, representing the convolutional kernel, that is convolved with the `inputs` to produce a `Tensor` of activations. If a `normalizer_fn` is provided (such as `batch_norm`), it is then applied. Otherwise, if `normalizer_fn` is `None` and a `biases_initializer` is provided then a `biases` variable would be created and added the activations. Finally, if `activation_fn` is not `None`, it is applied to the activations as well.

Args:

- `inputs` : a 4-D tensor `[batch_size, height, width, channels]` .
- `num_outputs` : integer, the number of output filters.
- `kernel_size` : a list of length 2 `[kernel_height, kernel_width]` of the filters. Can be an int if both values are the same.
- `stride` : a list of length 2 `[stride_height, stride_width]` . Can be an int if both strides are the same. Note that presently both strides must have the same value.
- `padding` : one of `VALID` or `SAME` .
- `activation_fn` : activation function.
- `normalizer_fn` : normalization function to use instead of `biases` . If `normalize_fn` is provided then `biases_initializer` and `biases_regularizer` are ignored and `biases` are not created nor added.

- `normalizer_params` : normalization function parameters.
- `weights_initializer` : An initializer for the weights.
- `weights_regularizer` : Optional regularizer for the weights.
- `biases_initializer` : An initializer for the biases. If None skip biases.
- `biases_regularizer` : Optional regularizer for the biases.
- `reuse` : whether or not the layer and its variables should be reused. To be able to reuse the layer scope must be given.
- `variables_collections` : optional list of collections for all the variables or a dictionary containing a different list of collection per variable.
- `outputs_collections` : collection to add the outputs.
- `trainable` : If `True` also add variables to the graph collection `GraphKeys.TRAINABLE_VARIABLES` (see `tf.Variable`).
- `scope` : Optional scope for `variable_op_scope`.

Returns:

a tensor representing the output of the operation.

`tf.contrib.layers.fully_connected(*args, **kwargs)`

Adds a fully connected layer.

`fully_connected` creates a variable called `weights`, representing a fully connected weight matrix, which is multiplied by the `inputs` to produce a `Tensor` of hidden units. If a `normalizer_fn` is provided (such as `batch_norm`), it is then applied. Otherwise, if `normalizer_fn` is `None` and a `biases_initializer` is provided then a `biases` variable would be created and added the hidden units. Finally, if `activation_fn` is not `None`, it is applied to the hidden units as well.

Note: that if `inputs` have a rank greater than 2, then `inputs` is flattened prior to the initial matrix multiply by `weights`.

Args:

- `inputs` : A tensor of with at least rank 2 and value for the last dimension, i.e. `[batch_size, depth]`, `[None, None, None, channels]`.
- `num_outputs` : Integer, the number of output units in the layer.
- `activation_fn` : activation function.
- `normalizer_fn` : normalization function to use instead of `biases`. If `normalize_fn` is provided then `biases_initializer` and `biases_regularizer` are ignored and `biases`

are not created nor added.

- `normalizer_params` : normalization function parameters.
- `weights_initializer` : An initializer for the weights.
- `weights_regularizer` : Optional regularizer for the weights.
- `biases_initializer` : An initializer for the biases. If None skip biases.
- `biases_regularizer` : Optional regularizer for the biases.
- `reuse` : whether or not the layer and its variables should be reused. To be able to reuse the layer scope must be given.
- `variables_collections` : Optional list of collections for all the variables or a dictionary containing a different list of collections per variable.
- `outputs_collections` : collection to add the outputs.
- `trainable` : If `True` also add variables to the graph collection `GraphKeys.TRAINABLE_VARIABLES` (see `tf.Variable`).
- `scope` : Optional scope for variable_op_scope.

Returns:

the tensor variable representing the result of the series of operations.

Raises:

- `ValueError` : if `x` has rank less than 2 or if its last dimension is not set.

Aliases for `fully_connected` which set a default activation function are available: `relu` , `relu6` and `linear` .

Regularizers

Regularization can help prevent overfitting. These have the signature `fn(weights)` . The loss is typically added to `tf.GraphKeys.REGULARIZATION_LOSS`

`tf.contrib.layers.apply_regularization(regularizer, weights_list=None)`

Returns the summed penalty by applying `regularizer` to the `weights_list` .

Adding a regularization penalty over the layer weights and embedding weights can help prevent overfitting the training data. Regularization over layer biases is less common/useful, but assuming proper data preprocessing/mean subtraction, it usually shouldn't hurt much either.

Args:

- `regularizer` : A function that takes a single `Tensor` argument and returns a scalar `Tensor` output.
- `weights_list` : List of weights `Tensors` or `Variables` to apply `regularizer` over. Defaults to the `GraphKeys.WEIGHTS` collection if `None`.

Returns:

A scalar representing the overall regularization penalty.

Raises:

- `ValueError` : If `regularizer` does not return a scalar output.

tf.contrib.layers.l1_regularizer(scale)

Returns a function that can be used to apply L1 regularization to weights.

L1 regularization encourages sparsity.

Args:

- `scale` : A scalar multiplier `Tensor`. 0.0 disables the regularizer.

Returns:

A function with signature `l1(weights, name=None)` that apply L1 regularization.

Raises:

- `ValueError` : If scale is outside of the range [0.0, 1.0] or if scale is not a float.

tf.contrib.layers.l2_regularizer(scale)

Returns a function that can be used to apply L2 regularization to weights.

Small values of L2 can help prevent overfitting the training data.

Args:

- `scale` : A scalar multiplier `Tensor`. 0.0 disables the regularizer.

Returns:

A function with signature `l2(weights, name=None)` that applies L2 regularization.

Raises:

- `ValueError` : If scale is outside of the range [0.0, 1.0] or if scale is not a float.

`tf.contrib.layers.sum_regularizer(regularizer_list)`

Returns a function that applies the sum of multiple regularizers.

Args:

- `regularizer_list` : A list of regularizers to apply.

Returns:

A function with signature `sum_reg(weights, name=None)` that applies the sum of all the input regularizers.

Initializers

Initializers are used to initialize variables with sensible values given their size, data type, and purpose.

`tf.contrib.layers.xavier_initializer(uniform=True, seed=None, dtype=tf.float32)`

Returns an initializer performing "Xavier" initialization for weights.

This function implements the weight initialization from:

Xavier Glorot and Yoshua Bengio (2010): Understanding the difficulty of training deep feedforward neural networks. International conference on artificial intelligence and statistics.

This initializer is designed to keep the scale of the gradients roughly the same in all layers.

In uniform distribution this ends up being the range: $x = \sqrt{6. / (\text{in} + \text{out})}; [-x, x]$ and for normal distribution a standard deviation of $\sqrt{3. / (\text{in} + \text{out})}$ is used.

Args:

- `uniform` : Whether to use uniform or normal distributed random initialization.
- `seed` : A Python integer. Used to create random seeds. See [set_random_seed](#) for behavior.
- `dtype` : The data type. Only floating point types are supported.

Returns:

An initializer for a weight matrix.

```
tf.contrib.layers.xavier_initializer_conv2d(uniform=True, seed=None, dtype=tf.float32)
```

Returns an initializer performing "Xavier" initialization for weights.

This function implements the weight initialization from:

Xavier Glorot and Yoshua Bengio (2010): Understanding the difficulty of training deep feedforward neural networks. International conference on artificial intelligence and statistics.

This initializer is designed to keep the scale of the gradients roughly the same in all layers.

In uniform distribution this ends up being the range: `x = sqrt(6. / (in + out)); [-x, x]` and for normal distribution a standard deviation of `sqrt(3. / (in + out))` is used.

Args:

- `uniform` : Whether to use uniform or normal distributed random initialization.
- `seed` : A Python integer. Used to create random seeds. See [set_random_seed](#) for behavior.
- `dtype` : The data type. Only floating point types are supported.

Returns:

An initializer for a weight matrix.

```
tf.contrib.layers.variance_scaling_initializer(factor=2.0, mode='FAN_IN', uniform=False, seed=None, dtype=tf.float32)
```

Returns an initializer that generates tensors without scaling variance.

When initializing a deep network, it is in principle advantageous to keep the scale of the input variance constant, so it does not explode or diminish by reaching the final layer. This initializer uses the following formula: if mode='FAN_IN': # Count only number of input connections. n = fan_in elif mode='FAN_OUT': # Count only number of output connections. n = fan_out elif mode='FAN_AVG': # Average number of inputs and output connections. n = (fan_in + fan_out)/2.0

```
truncated_normal(shape, 0.0, stddev=sqrt(factor / n))
```

To get <http://arxiv.org/pdf/1502.01852v1.pdf> use (Default):

- factor=2.0 mode='FAN_IN' uniform=False To get <http://arxiv.org/abs/1408.5093> use:
- factor=1.0 mode='FAN_IN' uniform=True To get <http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf> use:
- factor=1.0 mode='FAN_AVG' uniform=True. To get xavier_initializer use either:
- factor=1.0 mode='FAN_AVG' uniform=True.
- factor=1.0 mode='FAN_AVG' uniform=False.

Args:

- `factor` : Float. A multiplicative factor.
- `mode` : String. 'FAN_IN', 'FAN_OUT', 'FAN_AVG'.
- `uniform` : Whether to use uniform or normal distributed random initialization.
- `seed` : A Python integer. Used to create random seeds. See `set_random_seed` for behavior.
- `dtype` : The data type. Only floating point types are supported.

Returns:

An initializer that generates tensors with unit variance.

Raises:

- `ValueError` : if `dtype` is not a floating point type.
- `TypeError` : if `mode` is not in ['FAN_IN', 'FAN_OUT', 'FAN_AVG'].

Optimization

Optimize weights given a loss.

```
tf.contrib.layers.optimize_loss(loss,
global_step, learning_rate, optimizer,
gradient_noise_scale=None,
gradient_multipliers=None, clip_gradients=None,
moving_averages_decay=0.9,
learning_rate_decay_fn=None, update_ops=None,
variables=None, name=None)
```

Given loss and parameters for optimizer, returns a training op.

Args:

- `loss` : Tensor, 0 dimensional.
- `global_step` : Tensor, step counter for each update.
- `learning_rate` : float or Tensor, magnitude of update per each training step.
- `optimizer` : string, class or optimizer instance, used as trainer.

string should be name of optimizer, like 'SGD',
 'Adam', 'Adagrad'. Full list in OPTIMIZER_CLS_NAMES constant.
 class should be sub-class of tf.Optimizer that implements
 `compute_gradients` and `apply_gradients` functions.
 optimizer instance should be instantiation of tf.Optimizer sub-class
 and have `compute_gradients` and `apply_gradients` functions.

- `gradient_noise_scale` : float or None, adds 0-mean normal noise scaled by this value.

- `gradient_multipliers` : dict of variables or variable names to floats.

If present, gradients for specified variables will be multiplied by given constant.

- `clip_gradients` : float or `None`, clips gradients by this value.
- `moving_average_decay` : float or None, takes into account previous loss

to make learning smoother due to outliers.

- `learning_rate_decay_fn` : function, takes `learning_rate` and `global_step`

`Tensor`s, returns `Tensor`. Can be used to implement any learning rate decay functions. For example: `tf.train.exponential_decay`.

- `update_ops` : list of update `Operation`s to execute at each step. If `None`, uses elements of `UPDATE_OPS` collection.
- `variables` : list of variables to optimize or
`None` to use all trainable variables.
- `name` : The name for this operation is used to scope operations and summaries.

Returns:

Training op.

Raises:

- `ValueError` : if optimizer is wrong type.

Summaries

Helper functions to summarize specific variables or ops.

`tf.contrib.layers.summarize_activation(op)`

Summarize an activation.

This applies the given activation and adds useful summaries specific to the activation.

Args:

- `op` : The tensor to summarize (assumed to be a layer activation).

Returns:

The summary op created to summarize `op`.

`tf.contrib.layers.summarize_tensor(tensor, tag=None)`

Summarize a tensor using a suitable summary type.

This function adds a summary op for `tensor`. The type of summary depends on the shape of `tensor`. For scalars, a `scalar_summary` is created, for all other tensors, `histogram_summary` is used.

Args:

- `tensor` : The tensor to summarize
- `tag` : The tag to use, if None then use tensor's op's name.

Returns:

The summary op created or None for string tensors.

```
tf.contrib.layers.summarize_tensors(tensors,
summarizer=summarize_tensor)
```

Summarize a set of tensors.

```
tf.contrib.layers.summarize_collection(collect
ion, name_filter=None,
summarizer=summarize_tensor)
```

Summarize a graph collection of tensors, possibly filtered by name.

The layers module defines convenience functions `summarize_variables`, `summarize_weights` and `summarize_biases`, which set the `collection` argument of `summarize_collection` to `VARIABLES`, `WEIGHTS` and `BIASES`, respectively.

```
tf.contrib.layers.summarize_activations(name_f
ilter=None, summarizer=summarize_activation)
```

Summarize activations, using `summarize_activation` to summarize.

유틸리티 (contrib)

[TOC]

텐서를 처리하는 유틸리티

여러 유틸리티 함수

`tf.contrib.util.constant_value(tensor)`

만약 효율적으로 계산이 가능하다면 `tensor`의 상수 값을 반환합니다.

이 함수는 텐서가 주어지면 부분적으로 평가를 진행합니다. 성공하는 경우 `numpy ndarray` 값을 반환합니다.

하는일(marry): 이 함수는 손쉽게 확장이 가능하도록 `gradients`와 `ShapeFunctions`과 같은 등록 매커니즘을 고려합니다.

주의: 만약 `constant_value(tensor)` 가 `non-None` 결과를 반환하면 `tensor`에 별다른 값을 부여 할 수 없게 됩니다. 이 함수는 구축된 그래프에 영향을 미치도록 허용합니다. 그리고 정적 형상 최적화(permits static shape optimizations)을 허용합니다.

인자:

- `tensor` : 평가된 텐서

반환값:

`numpy ndarray`는 `tensor` 의 상수 값이거나 계산하지 않았다면 `None`을 포함합니다.

예외:

- `TypeError` : 텐서가 작동하지 않는 경우에 타입에러가 발생함.

`tf.contrib.util.make_tensor_proto(values, dtype=None, shape=None)`

TensorProto 생성.

인자:

- `values` : Values 를 TensorProto에 둡니다.
- `dtype` : Optional tensor_pb2 데이터 타입 값
- `shape` : 정수 리스트 형태로 텐서의 차원을 나타냅니다.

반환값:

TensorProto는 타입에 의존적입니다. TensorProto는 "tensor_content"를 포함합니다. 파이썬 프로그램에서 직접적으로 유용한 기능은 아닙니다. 값을 평가하기 위해 `tensor_util.MakeNdarray(proto)`를 이용해 proto를 다시 numpy ndarray로 변환해야합니다.

예외:

- `TypeError` : 타입이 제공되지 않은 경우.
- `ValueError` : 인자가 부적절한 값일 경우

`make_tensor_proto` 는 파이썬 스칼라 값인 "values" 를 받아들입니다. `values`는 파이썬의 리스트 형태입니다. numpy ndarray 와 numpy scalar와 같습니다.

만약 "values" 가 파이썬의 스칼라 혹은 리스트 형태라면, `make_tensor_proto first` 는 numpy ndarray로 변환됩니다. 만약 `dtype` 이 없다면, numpy 데이터형이 무엇인지 추론을 시도합니다. 달리 말하면 반환되는 numpy 배열은 주어진 `dtype`에 호환되는 데이터타입이 됩니다.

위의 두 경우에 있어서 numpy ndarray (호출자가 제공되거나, 자동 변환이 이뤄짐)는 반드시 `dtype`을 참고하여 타입이 호환되도록 해야합니다.

`make_tensor_proto` 는 numpy array에서 tensor proto로 변환을 담당합니다.

만약 "모양"이 None 일때 결과 텐서 proto는 numpy array로 정확히 표현할 수 있습니다.

다른말로 말하면 "모양"이 텐서의 모양으로 명시되면 numpy array는 더많은 엘리먼트가 필요 없게 됩니다.

`tf.contrib.util.make_ndarray(tensor)`

텐서로부터 numpy ndarray 를 생성합니다.

numpy ndarray를 생성할때 텐서와 동일한 모양과 데이터가 되도록 합니다.

인자:

- `tensor` : TensorProto

반환값:

텐서 컨텐츠로 이뤄진 numpy 배열

예외:

- `TypeError` : 텐서가 타입을 지원하지 않을 때 에러가 발생합니다.

`tf.contrib.util.ops_used_by_graph_def(graph_def)`

그래프에 사용된 ops의 리스트를 수집합니다.

ops가 모두 등록되었다면 검증하지 않습니다.

인자:

- `graph_def` : A `GraphDef` proto, as from `graph.as_graph_def()`.

반환값:

문자열 리스트를 반환합니다. 그래프에 사용된 각 op를 네이밍합니다.

`tf.contrib.util.stripped_op_list_for_graph(graph_def)`

Collect the stripped OpDefs for ops used by a graph. 그래프에서 사용된 ops에 대해 stripped OpDefs를 수집합니다.

이 함수는 `MetaGraphDef` 의 `stripped_op_list` 필드와 `protos`를 계산합니다. 결과는 생산자 (producer)에서 소비자(consumer)로 의사소통이 이뤄집니다. 이는 C++ 함수인 `RemoveNewDefaultAttrsFromGraphDef` 를 이용해 호환성이 향상될 수 있도록 합니다.

인자:

- `graph_def` : A `GraphDef` proto, as from `graph.as_graph_def()`.

반환값:

ops의 `opList` 는 그래프로서 사용됩니다.

예외:

- `ValueError` : 만약 등록되지 않은 op가 사용된 경우

텐서플로우 C++ 세션 API 레퍼런스 문서

0.5 버전의 텐서플로우의 퍼블릭 C++ API는 오직 그래프를 실행하는 API만을 포함합니다. C++로 부터 그래프 실행을 하는 것은 다음과 같습니다.

1. [Python API](#)를 이용해서 산출 그래프를 빌드합니다.
2. 그래프를 파일에 쓰기위해 `tf.train.write_graph()` 를 이용합니다.
3. C++ 세션 API를 이용해 그래프를 읽어옵니다. 예를 들면:

```
// Reads a model graph definition from disk, and creates a session object you
// can use to run it.
Status LoadGraph(string graph_file_name, Session** session) {
    GraphDef graph_def;
    TF_RETURN_IF_ERROR(
        ReadBinaryProto(Env::Default(), graph_file_name, &graph_def));
    TF_RETURN_IF_ERROR(NewSession(SessionOptions(), session));
    TF_RETURN_IF_ERROR((*session)->Create(graph_def));
    return Status::OK();
}
```

4. `session->Run()` 을 호출하여 그래프를 보여줍니다.

Env

- [tensorflow::Env](#)
- [tensorflow::RandomAccessFile](#)
- [tensorflow::WritableFile](#)
- [tensorflow::EnvWrapper](#)

Session

- [tensorflow::Session](#)
- [tensorflow::SessionOptions](#)

Status

- [tensorflow::Status](#)
- [tensorflow::Status::State](#)

Tensor

- [tensorflow::Tensor](#)
- [tensorflow::TensorShape](#)
- [tensorflow::TensorShapeDim](#)
- [tensorflow::TensorShapeUtils](#)
- [tensorflow::PartialTensorShape](#)
- [tensorflow::PartialTensorShapeUtils](#)
- [TF_Buffer](#)

Thread

- [tensorflow::Thread](#)
- [tensorflow::ThreadOptions](#)

class tensorflow::Env

An interface used by the tensorflow implementation to access operating system functionality like the filesystem etc.

Callers may wish to provide a custom Env object to get fine grain control.

All Env implementations are safe for concurrent access from multiple threads without any external synchronization.

Member Details

tensorflow::Env::Env()

virtual tensorflow::Env::~Env()=default

Status tensorflow::Env::GetFileSystemForFile(const string &fname, FileSystem **result)

Returns the FileSystem object to handle operations on the file specified by 'fname'. The FileSystem object is used as the implementation for the file system related (non-virtual) functions that follow. Returned FileSystem object is still owned by the Env object and will.

Status

tensorflow::Env::GetRegisteredFileSystemSchemes(std::vector<string> *schemes)

Returns the file system schemes registered for this Env .

Status tensorflow::Env::RegisterFileSystem(const string &scheme, FileSystemRegistry::Factory factory)

Status tensorflow::Env::NewRandomAccessFile(const string &fname, RandomAccessFile **result)

Creates a brand new random access read-only file with the specified name.

On success, stores a pointer to the new file in *result* and returns OK. On failure stores NULL in result and returns non-OK. If the file does not exist, returns a non-OK status.

The returned file may be concurrently accessed by multiple threads.

The ownership of the returned RandomAccessFile is passed to the caller and the object should be deleted when is not used. The file object shouldn't live longer than the Env object.

Status tensorflow::Env::NewWritableFile(const string &fname, WritableFile **result)

Creates an object that writes to a new file with the specified name.

Deletes any existing file with the same name and creates a new file. On success, stores a pointer to the new file in *result* and returns OK. On failure stores NULL in result and returns non-OK.

The returned file will only be accessed by one thread at a time.

The ownership of the returned WritableFile is passed to the caller and the object should be deleted when is not used. The file object shouldn't live longer than the Env object.

Status tensorflow::Env::NewAppendableFile(const string &fname, WritableFile **result)

Creates an object that either appends to an existing file, or writes to a new file (if the file does not exist to begin with).

On success, stores a pointer to the new file in *result* and returns OK. On failure stores NULL in result and returns non-OK.

The returned file will only be accessed by one thread at a time.

The ownership of the returned WritableFile is passed to the caller and the object should be deleted when is not used. The file object shouldn't live longer than the Env object.

Status tensorflow::Env::NewReadOnlyMemoryRegionFromFile(const string &fname, ReadOnlyMemoryRegion **result)

Creates a readonly region of memory with the file context.

On success, it returns a pointer to read-only memory region from the content of file fname. The ownership of the region is passed to the caller. On failure stores nullptr in *result and returns non-OK.

The returned memory region can be accessed from many threads in parallel.

The ownership of the returned ReadOnlyMemoryRegion is passed to the caller and the object should be deleted when is not used. The memory region object shouldn't live longer than the Env object.

bool tensorflow::Env::FileExists(const string &fname)

Returns true iff the named file exists.

Status tensorflow::Env::GetChildren(const string &dir, std::vector< string > *result)

Stores in *result the names of the children of the specified directory. The names are relative to "dir".

Original contents of *results are dropped.

Status tensorflow::Env::DeleteFile(const string &fname)

Deletes the named file.

Status tensorflow::Env::CreateDir(const string &dirname)

Creates the specified directory.

Status tensorflow::Env::DeleteDir(const string &dirname)

Deletes the specified directory.

Status tensorflow::Env::GetFileSize(const string &fname, uint64 *file_size)

Stores the size of fname in *file_size .

Status tensorflow::Env::RenameFile(const string &src, const string &target)

Renames file src to target. If target already exists, it will be replaced.

virtual uint64 tensorflow::Env::NowMicros()=0

Returns the number of micro-seconds since some fixed point in time. Only useful for computing deltas of time.

virtual void tensorflow::Env::SleepForMicroseconds(int micros)=0

Sleeps/delays the thread for the prescribed number of micro-seconds.

```
virtual Thread* tensorflow::Env::StartThread(const ThreadOptions &thread_options, const string &name, std::function< void()> fn) TF_MUST_USE_RESULT=0
```

Returns a new thread that is running fn() and is identified (for debugging/performance-analysis) by "name".

Caller takes ownership of the result and must delete it eventually (the deletion will block until fn() stops running).

```
virtual void tensorflow::Env::SchedClosure(std::function< void()> closure)=0
```

```
virtual void tensorflow::Env::SchedClosureAfter(int micros, std::function< void()> closure)=0
```

```
virtual Status tensorflow::Env::LoadLibrary(const char *library_filename, void **handle)=0
```

```
virtual Status tensorflow::Env::GetSymbolFromLibrary(void *handle, const char *symbol_name, void **symbol)=0
```

```
static Env* tensorflow::Env::Default()
```

Returns a default environment suitable for the current operating system.

Sophisticated users may wish to provide their own Env implementation instead of relying on this default environment.

The result of Default() belongs to this library and must never be deleted.

class tensorflow::RandomAccessFile

A file abstraction for randomly reading the contents of a file.

Member Details

tensorflow::RandomAccessFile::RandomAccessFile()

tensorflow::RandomAccessFile::~RandomAccessFile()

```
virtual Status tensorflow::RandomAccessFile::Read(uint64 offset, size_t n, StringPiece *result, char *scratch) const =0
```

Reads up to `n` bytes from the file starting at `offset`.

`scratch[0..n-1]` may be written by this routine. Sets `*result` to the data that was read (including if fewer than `n` bytes were successfully read). May set `*result` to point at data in `scratch[0..n-1]`, so `scratch[0..n-1]` must be live when `*result` is used.

On OK returned status: `n` bytes have been stored in `*result`. On non-OK returned status: `[0..n]` bytes have been stored in `*result`.

Returns `OUT_OF_RANGE` if fewer than `n` bytes were stored in `*result` because of EOF.

Safe for concurrent use by multiple threads.

class tensorflow::WritableFile

A file abstraction for sequential writing.

The implementation must provide buffering since callers may append small fragments at a time to the file.

Member Details

`tensorflow::WritableFile::WritableFile()`

`tensorflow::WritableFile::~WritableFile()`

`virtual Status tensorflow::WritableFile::Append(const StringPiece &data)=0`

`virtual Status tensorflow::WritableFile::Close()=0`

`virtual Status tensorflow::WritableFile::Flush()=0`

`virtual Status tensorflow::WritableFile::Sync()=0`

class tensorflow::EnvWrapper

An implementation of Env that forwards all calls to another Env .

May be useful to clients who wish to override just part of the functionality of another Env .

Member Details

tensorflow::EnvWrapper::EnvWrapper(Env *t)

Initializes an EnvWrapper that delegates all calls to *t.

tensorflow::EnvWrapper::~EnvWrapper()

Env* tensorflow::EnvWrapper::target() const

Returns the target to which this Env forwards all calls.

Status tensorflow::EnvWrapper::GetFileSystemForFile(const string &fname, FileSystem **result) override

Returns the FileSystem object to handle operations on the file specified by 'fname'. The FileSystem object is used as the implementation for the file system related (non-virtual) functions that follow. Returned FileSystem object is still owned by the Env object and will.

Status tensorflow::EnvWrapper::GetRegisteredFileSystemSchemes(std::vector< string > *schemes) override

Returns the file system schemes registered for this Env .

Status tensorflow::EnvWrapper::RegisterFileSystem(const string &scheme, FileSystemRegistry::Factory factory) override

uint64 tensorflow::EnvWrapper::NowMicros() override

Returns the number of micro-seconds since some fixed point in time. Only useful for computing deltas of time.

class tensorflow::EnvWrapper

```
void tensorflow::EnvWrapper::SleepForMicroseconds(int  
micros) override
```

Sleeps/delays the thread for the prescribed number of micro-seconds.

```
Thread* tensorflow::EnvWrapper::StartThread(const  
ThreadOptions &thread_options, const string &name,  
std::function< void()> fn) override
```

Returns a new thread that is running fn() and is identified (for debugging/performance-analysis) by "name".

Caller takes ownership of the result and must delete it eventually (the deletion will block until fn() stops running).

```
void tensorflow::EnvWrapper::SchedClosure(std::function<  
void()> closure) override
```

```
void tensorflow::EnvWrapper::SchedClosureAfter(int micros,  
std::function< void()> closure) override
```

```
Status tensorflow::EnvWrapper::LoadLibrary(const char  
*library_filename, void **handle) override
```

```
Status tensorflow::EnvWrapper::GetSymbolFromLibrary(void  
*handle, const char *symbol_name, void **symbol) override
```

class tensorflow::Session

A Session instance lets a caller drive a TensorFlow graph computation.

When a Session is created with a given target, a new Session object is bound to the universe of resources specified by that target. Those resources are available to this session to perform computation described in the GraphDef. After extending the session with a graph, the caller uses the Run() API to perform the computation and potentially fetch outputs as Tensors.

Example:

```
```c++
tensorflow::GraphDef graph; // ... Create or load graph into "graph".

// This example uses the default options which connects // to a local runtime.
tensorflow::SessionOptions options; std::unique_ptr<
session(tensorflow::NewSession(options));

// Create the session with this graph. tensorflow::Status s = session->Create(graph); if
(!s.ok()) { ... }

// Run the graph and fetch the first output of the "output" // operation, and also run to but do
not return anything // for the "update_state" operation. std::vector<outputs>; s = session-
>Run({}, {"output:0"}, {"update_state"}, &outputs); if (!s.ok()) { ... }

// Map the output as a flattened float tensor, and do something // with it. auto output_tensor =
outputs[0].flat(); if (output_tensor(0) > 0.5) { ... }

// Close the session to release the resources associated with // this session. session-
>Close();

...
```

```

A Session allows concurrent calls to Run() , though a Session must be created / extended by a single thread.

Only one thread must call Close() , and Close() must only be called after all other calls to Run() have returned.

Member Details

tensorflow::Session::Session()

virtual tensorflow::Session::~Session()

virtual Status tensorflow::Session::Create(const GraphDef &graph)=0

Create the graph to be used for the session.

Returns an error if this session has already been created with a graph. To re-use the session with a different graph, the caller must Close() the session first.

virtual Status tensorflow::Session::Extend(const GraphDef &graph)=0

Adds operations to the graph that is already registered with the Session .

The names of new operations in "graph" must not exist in the graph that is already registered.

virtual Status tensorflow::Session::Run(const std::vector<std::pair< string, Tensor > > &inputs, const std::vector< string > &output_tensor_names, const std::vector< string > &target_node_names, std::vector< Tensor > *outputs)=0

Runs the graph with the provided input tensors and fills `outputs` for the endpoints specified in `output_tensor_names` . Runs to but does not return Tensors for the nodes in `target_node_names` .

The order of tensors in `outputs` will match the order provided by `output_tensor_names` .

If `Run` returns `OK()` , then `outputs->size()` will be equal to `output_tensor_names.size()` . If `Run` does not return `OK()` , the state of `outputs` is undefined.

REQUIRES: The name of each Tensor of the input or output must match a "Tensor endpoint" in the `GraphDef` passed to `Create()` .

REQUIRES: At least one of `output_tensor_names` and `target_node_names` must be non-empty.

REQUIRES: `outputs` is not nullptr if `output_tensor_names` is non-empty.

virtual Status tensorflow::Session::Create(const RunOptions &run_options, const GraphDef &graph)

Implementations which support `RunOptions` .

NOTE: This API is still experimental and may change.

```
virtual Status tensorflow::Session::Extend(const RunOptions  
&run_options, const GraphDef &graph)
```

```
virtual Status tensorflow::Session::Close(const RunOptions  
&run_options)
```

```
virtual Status tensorflow::Session::Run(const RunOptions  
&run_options, const std::vector< std::pair< string, Tensor >  
> &inputs, const std::vector< string > &output_tensor_names,  
const std::vector< string > &target_node_names, std::vector<  
Tensor > *outputs, RunMetadata *run_metadata)
```

Like `Run`, but allows users to pass in a `RunOptions` proto and to retrieve non-Tensor metadata output via a `RunMetadata` proto for this step. `run_metadata` may be `nullptr`, in which case any metadata output is discarded. NOTE: This API is still experimental and may change.

```
virtual Status tensorflow::Session::PRunSetup(const  
std::vector< string > &input_names, const std::vector<  
string > &output_names, const std::vector< string >  
&target_nodes, string *handle)
```

Sets up a graph for partial execution. All future feeds and fetches are specified by `input_names` and `output_names`. Returns `handle` that can be used to perform a sequence of partial feeds and fetches. NOTE: This API is still experimental and may change.

```
virtual Status tensorflow::Session::PRun(const string  
&handle, const std::vector< std::pair< string, Tensor > >  
&inputs, const std::vector< string > &output_names,  
std::vector< Tensor > *outputs)
```

Continues the pending execution specified by `handle` with the provided input tensors and fills `outputs` for the endpoints specified in `output_names`. NOTE: This API is still experimental and may change.

```
virtual Status tensorflow::Session::Close()=0
```

Closes this session.

Closing a session releases the resources used by this session on the TensorFlow runtime (specified during session creation by the `SessionOptions::target` field).

struct tensorflow::SessionOptions

Configuration information for a Session .

Member Details

`Env* tensorflow::SessionOptions::env`

The environment to use.

`string tensorflow::SessionOptions::target`

The TensorFlow runtime to connect to.

If 'target' is empty or unspecified, the local TensorFlow runtime implementation will be used. Otherwise, the TensorFlow engine defined by 'target' will be used to perform all computations.

"target" can be either a single entry or a comma separated list of entries. Each entry is a resolvable address of the following format: local ip:port host:port ... other system-specific formats to identify tasks and jobs ...

NOTE: at the moment 'local' maps to an in-process service-based runtime.

Upon creation, a single session affines itself to one of the remote processes, with possible load balancing choices when the "target" resolves to a list of possible processes.

If the session disconnects from the remote process during its lifetime, session calls may fail immediately.

`ConfigProto tensorflow::SessionOptions::config`

Configuration options.

`tensorflow::SessionOptions::SessionOptions()`

class tensorflow::Status

Member Details

tensorflow::Status::Status()

Create a success status.

tensorflow::Status::~Status()

**tensorflow::Status::Status(tensorflow::error::Code code,
tensorflow::StringPiece msg)**

Create a status with the specified error code and msg as a human-readable string containing more detailed information.

tensorflow::Status::Status(const Status &s)

Copy the specified status.

void tensorflow::Status::operator=(const Status &s)

bool tensorflow::Status::ok() const

Returns true iff the status indicates success.

tensorflow::error::Code tensorflow::Status::code() const

const string& tensorflow::Status::error_message() const

bool tensorflow::Status::operator==(const Status &x) const

bool tensorflow::Status::operator!=(const Status &x) const

void tensorflow::Status::Update(const Status &new_status)

If `ok()`, stores `new_status` into `*this`. If `!ok()`, preserves the current status, but may augment with additional information about `new_status`.

Convenient way of keeping track of the first error encountered. Instead of:

```
if  
(overall_status.ok()) overall_status = new_status Use:  
overall_status.Update(new_status);
```

string tensorflow::Status::ToString() const

Return a string representation of this status suitable for printing. Returns the string "OK" for success.

```
return tensorflow::Status::OK()
```

struct tensorflow::Status::State

Member Details

`tensorflow::error::Code tensorflow::Status::State::code`

`string tensorflow::Status::State::msg`

class tensorflow::Tensor

Represents an n-dimensional array of values.

Member Details

tensorflow::Tensor::Tensor()

Default Tensor constructor. Creates a 1-dimension, 0-element float tensor.

tensorflow::Tensor::Tensor(DataType type, const TensorShape &shape)

Creates a Tensor of the given `type` and `shape`. If `LogMemory::IsEnabled()` the allocation is logged as coming from an unknown kernel and step. Calling the Tensor constructor directly from within an Op is deprecated: use the `OpKernelConstruction/OpKernelContext allocate_*` methods to allocate a new tensor, which record the kernel and step.

The underlying buffer is allocated using a `CPUAllocator`.

tensorflow::Tensor::Tensor(Allocator *a, DataType type, const TensorShape &shape)

Creates a tensor with the input `type` and `shape`, using the allocator `a` to allocate the underlying buffer. If `LogMemory::IsEnabled()` the allocation is logged as coming from an unknown kernel and step. Calling the Tensor constructor directly from within an Op is deprecated: use the `OpKernelConstruction/OpKernelContext allocate_*` methods to allocate a new tensor, which record the kernel and step.

`a` must outlive the lifetime of this Tensor.

tensorflow::Tensor::Tensor(Allocator *a, DataType type, const TensorShape &shape, const AllocationAttributes &allocation_attr)

Creates a tensor with the input `type` and `shape`, using the allocator `a` and the specified "allocationattr" to allocate the underlying buffer. If the kernel and step are known `allocation_attr.allocation_will_be_logged` should be set to true and `LogMemory::RecordTensorAllocation` should be called after the tensor is constructed.

Calling the Tensor constructor directly from within an Op is deprecated: use the OpKernelConstruction/OpKernelContext allocate methods to allocate a new tensor, which record the kernel and step.*

a must outlive the lifetime of this Tensor .

tensorflow::Tensor::Tensor(DataType type)

Creates an uninitialized Tensor of the given data type.

tensorflow::Tensor::Tensor(const Tensor &other)

tensorflow::Tensor::~Tensor()

Copy constructor.

DataType tensorflow::Tensor::dtype() const

Returns the data type.

const TensorShape& tensorflow::Tensor::shape() const

Returns the shape of the tensor.

int tensorflow::Tensor::dims() const

Convenience accessor for the tensor shape.

For all shape accessors, see comments for relevant methods of `TensorShape` in `tensor_shape.h`.

int64 tensorflow::Tensor::dim_size(int d) const

Convenience accessor for the tensor shape.

int64 tensorflow::Tensor::NumElements() const

Convenience accessor for the tensor shape.

bool tensorflow::Tensor::IsSameSize(const Tensor &b) const

bool tensorflow::Tensor::SharesBufferWith(const Tensor &b) const

class tensorflow::Tensor

size_t tensorflow::Tensor::BufferHash() const

bool tensorflow::Tensor::IsInitialized() const

Has this Tensor been initialized?

size_t tensorflow::Tensor::TotalBytes() const

Returns the estimated memory usage of this tensor.

bool tensorflow::Tensor::IsAligned() const

Returns true iff this tensor is aligned.

Tensor& tensorflow::Tensor::operator=(const Tensor &other)

Assign operator. This tensor shares other's underlying storage.

**bool tensorflow::Tensor::CopyFrom(const Tensor &other,
const TensorShape &shape) TF_MUST_USE_RESULT**

Copy the other tensor into this tensor and reshape it.

This tensor shares other's underlying storage. Returns `true` iff `other.shape()` has the same number of elements of the given `shape`.

**Tensor tensorflow::Tensor::Slice(int64 dim0_start, int64
dim0_limit) const**

Slice this tensor along the 1st dimension.

I.e., the returned tensor satisfies `returned[i, ...] == this[dim0_start + i, ...]`. The returned tensor shares the underlying tensor buffer with this tensor.

NOTE: The returned tensor may not satisfies the same alignment requirement as this tensor depending on the shape. The caller must check the returned tensor's alignment before calling certain methods that have alignment requirement (e.g., `flat()`, `tensor()`).

REQUIRES: `dims() >= 1` REQUIRES: `0 <= dim0_start <= dim0_limit <= dim_size(0)`

**bool tensorflow::Tensor::FromProto(const TensorProto
&other) TF_MUST_USE_RESULT**

Parse `other` and construct the tensor.

Returns `true` iff the parsing succeeds. If the parsing fails, the state of `*this` is unchanged.

```
bool tensorflow::Tensor::FromProto(Allocator *a, const  
TensorProto &other) TF_MUST_USE_RESULT
```

```
void tensorflow::Tensor::AsProtoField(TensorProto *proto)  
const
```

Fills in `proto` with `*this` tensor's content.

`AsProtoField()` fills in the repeated field for `proto.dtype()`, while `AsProtoTensorContent()` encodes the content in `proto.tensor_content()` in a compact form.

```
void tensorflow::Tensor::AsProtoTensorContent(TensorProto  
*proto) const
```

```
TTypes<T>::Vec tensorflow::Tensor::vec()
```

Return the tensor data as an `Eigen::Tensor` with the type and sizes of this `Tensor`.

Use these methods when you know the data type and the number of dimensions of the Tensor and you want an `Eigen::Tensor` automatically sized to the `Tensor` sizes. The implementation check fails if either type or sizes mismatch.

Example:

```
```c++  
typedef float T;
Tensor my_mat(...built with Shape{rows: 3, cols: 5}...);
auto mat = my_mat.matrix(); // 2D Eigen::Tensor, 3 x 5.
auto mat = my_mat.tensor(); // 2D Eigen::Tensor, 3 x 5.
auto vec = my_mat.vec(); // CHECK fails as my_mat is 2D.
auto vec = my_mat.tensor(); // CHECK fails as my_mat is 2D.
auto mat = my_mat.matrix(); // CHECK fails as type mismatch.
```

```
`TTypes<T>::Matrix tensorflow::Tensor::matrix()` `[#TTypes_T_Matrix_tensorflow_Tensor_matrix]

`TTypes< T, NDIMS >::Tensor tensorflow::Tensor::tensor()` `[#TTypes_T_NDIMS_Tensor_tensorflow_Tensor_tensor]

`TTypes<T>::Flat tensorflow::Tensor::flat()` `[#TTypes_T_Flat_tensorflow_Tensor_flat}
```

Return the tensor data as an `Eigen::Tensor` of the data type and a specified shape.

These methods allow you to access the data with the dimensions and sizes of your choice. You do not need to know the number of dimensions of the Tensor to call them. However, they `CHECK` that the type matches and the dimensions requested creates an `Eigen::Tensor` with the same number of elements as the tensor.

Example:

```
```c++
typedef float T;
Tensor my_ten(...built with Shape{planes: 4, rows: 3, cols: 5}...);
// 1D Eigen::Tensor, size 60:
auto flat = my_ten.flat<T>();
// 2D Eigen::Tensor 12 x 5:
auto inner = my_ten.flat_inner_dims<T>();
// 2D Eigen::Tensor 4 x 15:
auto outer = my_ten.shaped<T, 2>({4, 15});
// CHECK fails, bad num elements:
auto outer = my_ten.shaped<T, 2>({4, 8});
// 3D Eigen::Tensor 6 x 5 x 2:
auto weird = my_ten.shaped<T, 3>({6, 5, 2});
// CHECK fails, type mismatch:
auto bad   = my_ten.flat<int32>();
```

TTypes<T>::UnalignedFlat tensorflow::Tensor::unaligned_flat()

TTypes< T, NDIMS >::Tensor tensorflow::Tensor::flat_inner_dims()

Returns the data as an Eigen::Tensor with NDIMS dimensions, collapsing all Tensor dimensions but the last NDIMS-1 into the first dimension of the result. If NDIMS > dims() then leading dimensions of size 1 will be added to make the output rank NDIMS.

```
TTypes< T, NDIMS >::Tensor  
tensorflow::Tensor::flat_outer_dims()
```

Returns the data as an Eigen::Tensor with NDIMS dimensions, collapsing all Tensor dimensions but the first NDIMS-1 into the last dimension of the result. If NDIMS > dims() then trailing dimensions of size 1 will be added to make the output rank NDIMS.

```
TTypes< T, NDIMS >::Tensor  
tensorflow::Tensor::shaped(gtl::ArraySlice< int64 >  
new_sizes)
```

```
TTypes< T, NDIMS >::UnalignedTensor  
tensorflow::Tensor::unaligned_shaped(gtl::ArraySlice< int64  
> new_sizes)
```

```
TTypes< T >::Scalar tensorflow::Tensor::scalar()
```

Return the Tensor data as a `TensorMap` of fixed size 1: `TensorMap<TensorFixedSize<T, 1>>`.

Using `scalar()` allows the compiler to perform optimizations as the size of the tensor is known at compile time.

```
TTypes<T>::ConstVec tensorflow::Tensor::vec() const
```

Const versions of all the methods above.

```
TTypes<T>::ConstMatrix tensorflow::Tensor::matrix() const
```

```
TTypes< T, NDIMS >::ConstTensor  
tensorflow::Tensor::tensor() const
```

```
TTypes<T>::ConstFlat tensorflow::Tensor::flat() const
```

```
TTypes<T>::UnalignedConstFlat  
tensorflow::Tensor::unaligned_flat() const
```

```
TTypes< T, NDIMS >::ConstTensor  
tensorflow::Tensor::shaped(gtl::ArraySlice< int64 >  
new_sizes) const
```

class tensorflow::Tensor

```
TTypes< T, NDIMS >::UnalignedConstTensor  
tensorflow::Tensor::unaligned_shaped(gtl::ArraySlice< int64  
> new_sizes) const
```

```
TTypes< T >::ConstScalar tensorflow::Tensor::scalar()  
const
```

```
TTypes< T, NDIMS >::ConstTensor  
tensorflow::Tensor::flat_inner_dims() const
```

```
TTypes< T, NDIMS >::ConstTensor  
tensorflow::Tensor::flat_outer_dims() const
```

```
string tensorflow::Tensor::SummarizeValue(int64  
max_entries) const
```

Render the first `max_entries` values in `*this` into a string.

```
string tensorflow::Tensor::DebugString() const
```

A human-readable summary of the tensor suitable for debugging.

```
void tensorflow::Tensor::FillDescription(TensorDescription  
*description) const
```

Fill in the `TensorDescription` proto with metadata about the tensor that is useful for monitoring and debugging.

```
StringPiece tensorflow::Tensor::tensor_data() const
```

Returns a `StringPiece` mapping the current tensor's buffer.

The returned `StringPiece` may point to memory location on devices that the CPU cannot address directly.

NOTE: The underlying tensor buffer is refcounted, so the lifetime of the contents mapped by the `StringPiece` matches the lifetime of the buffer; callers should arrange to make sure the buffer does not get destroyed while the `StringPiece` is still used.

REQUIRES: `DataTypeCanUseMemcpy(dtype())`.

```
void tensorflow::Tensor::UnsafeCopyFromInternal(const  
Tensor &, const TensorShape &)
```

Copy the other tensor into this tensor and reshape it and reinterpret the buffer's datatype.

This tensor shares other's underlying storage.

class tensorflow::TensorShape

Member Details

`uint8 tensorflow::TensorShape::buf[16][16]`

`Rep64* tensorflow::TensorShape::unused_aligner`

`tensorflow::TensorShape::TensorShape(gtl::ArraySlice< int64 > dim_sizes)`

Construct a `TensorShape` from the provided sizes. REQUIRES: `dim_sizes[i] >= 0`

`tensorflow::TensorShape::TensorShape(std::initializer_list< int64 > dim_sizes)`

`tensorflow::TensorShape::TensorShape(const TensorShapeProto &proto)`

REQUIRES: `IsValid(proto)`

`tensorflow::TensorShape::TensorShape()`

Create a tensor shape with no dimensions and one element, which you can then call `AddDim()` on.

`tensorflow::TensorShape::~TensorShape()`

`tensorflow::TensorShape::TensorShape(const TensorShape &b)`

Copy the specified shape.

`void tensorflow::TensorShape::operator=(const TensorShape &b)`

`void tensorflow::TensorShape::Clear()`

Clear a tensor shape.

`void tensorflow::TensorShape::AddDim(int64 size)`

class tensorflow::TensorShape

Add a dimension to the end ("inner-most"). REQUIRES: `size >= 0`

void tensorflow::TensorShape::AppendShape(const TensorShape &shape)

Appends all the dimensions from `shape`.

void tensorflow::TensorShape::InsertDim(int d, int64 size)

Insert a dimension somewhere in the `TensorShape`. REQUIRES: `0 <= d <= dims()`
REQUIRES: `size >= 0`

void tensorflow::TensorShape::set_dim(int d, int64 size)

Modifies the size of the dimension `d` to be `size` REQUIRES: `0 <= d < dims()`
REQUIRES: `size >= 0`

void tensorflow::TensorShape::RemoveDim(int d)

Removes dimension `d` from the `TensorShape`. REQUIRES: `0 <= d < dims()`

int tensorflow::TensorShape::dims() const

Return the number of dimensions in the tensor.

int64 tensorflow::TensorShape::dim_size(int d) const

Returns the number of elements in dimension `d`. REQUIRES: `0 <= d < dims()`

**gtl::InlinedVector< int64, 4 >
tensorflow::TensorShape::dim_sizes() const**

Returns sizes of all dimensions.

int64 tensorflow::TensorShape::num_elements() const

Returns the number of elements in the tensor.

We use `int64` and not `size_t` to be compatible with `Eigen::Tensor` which uses `ptrdiff_t`.

bool tensorflow::TensorShape::IsSameSize(const TensorShape &b) const

```
class tensorflow::TensorShape
```

Returns true if `*this` and `b` have the same sizes. Ignores dimension names.

```
bool tensorflow::TensorShape::operator==(const TensorShape &b) const
```

```
bool tensorflow::TensorShape::operator!=(const TensorShape &b) const
```

```
void tensorflow::TensorShape::AsProto(TensorShapeProto *proto) const
```

Fill `*proto` from `*this`.

```
Eigen::DSizes< Eigen::DenseIndex, NDIMS >
tensorflow::TensorShape::AsEigenDSizes() const
```

Fill `*dsizes` from `*this`.

```
Eigen::DSizes< Eigen::DenseIndex, NDIMS >
tensorflow::TensorShape::AsEigenDSizesWithPadding() const
```

Same as `AsEigenDSizes()` but allows for `NDIMS > dims()` in which case we pad the rest of the sizes with 1.

```
TensorShapeIter tensorflow::TensorShape::begin() const
```

For iterating through the dimensions.

```
TensorShapeIter tensorflow::TensorShape::end() const
```

```
string tensorflow::TensorShape::DebugString() const
```

For error messages.

```
void tensorflow::TensorShape::DumpRep() const
```

```
bool tensorflow::TensorShape::IsValid(const TensorShapeProto &proto)
```

Returns `true` iff `proto` is a valid tensor shape.

```
Status tensorflow::TensorShape::IsValidShape(const TensorShapeProto &proto)
```

```
class tensorflow::TensorShape
```

Returns `ok` iff `proto` is a valid tensor shape, and a descriptive error status otherwise.

```
static constexpr int  
tensorflow::TensorShape::MaxDimensions()
```

```
string tensorflow::TensorShape::DebugString(const  
TensorShapeProto &proto)
```

Same as `TensorShape(proto).DebugString()` but doesn't crash for invalid protos.

struct tensorflow::TensorShapeDim

Member Details

`int64 tensorflow::TensorShapeDim::size`

`tensorflow::TensorShapeDim::TensorShapeDim(int64 s)`

class tensorflow::TensorShapeUtils

Static helper routines for `TensorShape`. Includes a few common predicates on a tensor shape.

Member Details

```
static bool tensorflow::TensorShapeUtils::IsScalar(const TensorShape &shape)
```

```
static bool tensorflow::TensorShapeUtils::IsVector(const TensorShape &shape)
```

```
static bool tensorflow::TensorShapeUtils::IsVectorOrHigher(const TensorShape &shape)
```

```
static bool tensorflow::TensorShapeUtils::IsMatrix(const TensorShape &shape)
```

```
static bool tensorflow::TensorShapeUtils::IsMatrixOrHigher(const TensorShape &shape)
```

```
static Status tensorflow::TensorShapeUtils::MakeShape(const int32 *dims, int n, TensorShape *out)
```

Returns a `TensorShape` whose dimensions are `dims[0]`, `dims[1]`, ..., `dims[n-1]`.

```
static Status tensorflow::TensorShapeUtils::MakeShape(const int64 *dims, int n, TensorShape *out)
```

```
string tensorflow::TensorShapeUtils::ShapeListString(const gtl::ArraySlice< TensorShape > &shapes)
```

```
bool tensorflow::TensorShapeUtils::StartsWith(const TensorShape &shape0, const TensorShape &shape1)
```

class tensorflow::PartialTensorShape

Manages the partially known dimensions of a Tensor and their sizes.

Member Details

`tensorflow::PartialTensorShape::PartialTensorShape()`

Construct an unknown `PartialTensorShape`.

`tensorflow::PartialTensorShape::PartialTensorShape(gtl::ArraySlice< int64 > dim_sizes)`

Construct a `PartialTensorShape` from the provided sizes. REQUIRES: `dim_sizes[i] >= 0`

`tensorflow::PartialTensorShape::PartialTensorShape(std::initializer_list< int64 > dim_sizes)`

`tensorflow::PartialTensorShape::PartialTensorShape(const TensorShapeProto &proto)`

REQUIRES: `IsValid(proto)`

`PartialTensorShape`

`tensorflow::PartialTensorShape::Concatenate(int64 size) const`

Add a dimension to the end ("inner-most"), returns a new `PartialTensorShape`. REQUIRES: `size >= -1`, where -1 means unknown

`PartialTensorShape`

`tensorflow::PartialTensorShape::Concatenate(const PartialTensorShape &shape) const`

Appends all the dimensions from `shape`. Returns a new `PartialTensorShape`.

`Status tensorflow::PartialTensorShape::MergeWith(const PartialTensorShape &shape, PartialTensorShape *result) const`

Merges all the dimensions from `shape`. Returns `InvalidArgument` error if either `shape` has a different rank or if any of the dimensions are incompatible.

```
class tensorflow::PartialTensorShape
```

```
int tensorflow::PartialTensorShape::dims() const
```

Return the number of dimensions in the tensor. If the number of dimensions is unknown, return -1.

```
bool tensorflow::PartialTensorShape::IsFullyDefined() const
```

Return true iff the rank and all of the dimensions are well defined.

```
bool tensorflow::PartialTensorShape::IsCompatibleWith(const PartialTensorShape &shape) const
```

Return true iff the ranks match, and if the dimensions all either match or one is unknown.

```
bool tensorflow::PartialTensorShape::IsCompatibleWith(const TensorShape &shape) const
```

Return true iff the dimensions of `shape` are compatible with `*this`.

```
int64 tensorflow::PartialTensorShape::dim_size(int d) const
```

Returns the number of elements in dimension `d`. REQUIRES: `0 <= d < dims()`

```
gtl::ArraySlice<int64> tensorflow::PartialTensorShape::dim_sizes() const
```

Returns sizes of all dimensions.

```
void tensorflow::PartialTensorShape::AsProto(TensorShapeProto *proto) const
```

Fill `*proto` from `*this`.

```
bool tensorflow::PartialTensorShape::AsTensorShape(TensorShape *tensor_shape) const
```

```
string tensorflow::PartialTensorShape::DebugString() const
```

For error messages.

class tensorflow::PartialTensorShape

```
bool tensorflow::PartialTensorShape::IsValid(const  
TensorShapeProto &proto)
```

Returns `true` iff `proto` is a valid partial tensor shape.

```
Status tensorflow::PartialTensorShape::IsValidShape(const  
TensorShapeProto &proto)
```

Returns `ok` iff `proto` is a valid tensor shape, and a descriptive error status otherwise.

```
string tensorflow::PartialTensorShape::DebugString(const  
TensorShapeProto &proto)
```

```
static Status  
tensorflow::PartialTensorShape::MakePartialShape(const int32  
*dims, int n, PartialTensorShape *out)
```

Returns a `PartialTensorShape` whose dimensions are `dims[0]`, `dims[1]`, ..., `dims[n-1]`.
Values of -1 are considered "unknown".

```
static Status  
tensorflow::PartialTensorShape::MakePartialShape(const int64  
*dims, int n, PartialTensorShape *out)
```

class tensorflow::PartialTensorShapeUtils

Static helper routines for `PartialTensorShape`. Includes a few common predicates on a partially known tensor shape.

Member Details

```
string
tensorflow::PartialTensorShapeUtils::PartialShapeListString(
const gtl::ArraySlice< PartialTensorShape > &shapes)
```

```
bool
tensorflow::PartialTensorShapeUtils::AreCompatible(const
gtl::ArraySlice< PartialTensorShape > &shapes0, const
gtl::ArraySlice< PartialTensorShape > &shapes1)
```

class tensorflow::Thread

Member Details

tensorflow::Thread::Thread()

tensorflow::Thread::~Thread()

Blocks until the thread of control stops running.

struct tensorflow::ThreadOptions

Options to configure a Thread .

Note that the options are all hints, and the underlying implementation may choose to ignore it.

Member Details

size_t tensorflow::ThreadOptions::stack_size

Thread stack size to use (in bytes).

size_t tensorflow::ThreadOptions::guard_size

Guard area size to use near thread stacks to use (in bytes)

Additional Resources

TensorFlow WhitePaper

Additional details about the TensorFlow programming model and the underlying implementation can be found in our white paper:

- [TensorFlow: Large-scale machine learning on heterogeneous systems](#)

Citation

If you use TensorFlow in your research and would like to cite the TensorFlow system, we suggest you cite the paper above. You can use this [BibTeX entry](#). As the project progresses, we may update the suggested citation with new papers.

Please only use the TensorFlow name and marks when accurately referencing this software distribution, and do not use our marks in a way that suggests you are endorsed by or otherwise affiliated with Google. When referring to our marks, please include the following attribution statement: "TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc."

What is TensorFlow used for?

TensorFlow enables researchers to build machine learning models. We collect such models in our [Zoo](#). If you have built a model with TensorFlow, you may consider publishing it there.

We keep a list of projects that use TensorFlow [here](#). If you made something amazing with TensorFlow, we'd like to hear about it!

Community

The TensorFlow community has created many great projects around TensorFlow, including:

- [TensorFlow tutorials](#)
- [Scikit Flow - Simplified Interface for TensorFlow](#)
- [Caffe to TensorFlow model converter](#)

Development

The source code for TensorFlow is hosted on GitHub:

<https://github.com/tensorflow/tensorflow>.

If you are interested in contributing to TensorFlow please [review the contributing guide](#).

Help / Support / How do I?

For help and support, technical or algorithmic questions, please submit your questions to Stack Overflow: <https://stackoverflow.com/questions/tagged/tensorflow>. You may also find answers in our [FAQ](#), our [glossary](#), or in the [shapes, sizes and types guide](#). Please do not use the mailing list or issue tracker for support.

Discussions

For general discussions, please join the [TensorFlow discuss mailing list](#). This list is intended for general discussions about TensorFlow development and directions, not as a help forum. Instead, direct your questions to [Stack Overflow](#), and report issues on [GitHub](#).

Report Issues

Please report bugs, feature requests and installation / compatibility issues on the [TensorFlow issues tracker](#) on GitHub. If you need help with using TensorFlow, please do not use the issue tracker for that. Instead, direct your questions to [Stack Overflow](#).

Versioning

TensorFlow uses [Semantic Versioning 2.0](#). For details on the versioning of our public API and binary compatibility, see the [versioning document](#).

Roadmap

A roadmap containing what we're working on at the moment is [here](#).

텐서플로우 백서

당신의 연구에서 텐서플로우를 사용하면서 텐서플로우 시스템을 인용하려는 경우에는 이 [백서](#)를 인용하시기 바랍니다.

```

@misc{tensorflow2015-whitepaper,
  title={{TensorFlow}: Large-Scale Machine Learning on Heterogeneous Systems},
  url={http://tensorflow.org/},
  note={Software available from tensorflow.org},
  author={
    Mart\'in~Abadi and
    Ashish~Agarwal and
    Paul~Barham and
    Eugene~Brevdo and
    Zhifeng~Chen and
    Craig~Citro and
    Greg~S.~Corrado and
    Andy~Davis and
    Jeffrey~Dean and
    Matthieu~Devin and
    Sanjay~Ghemawat and
    Ian~Goodfellow and
    Andrew~Harp and
    Geoffrey~Irving and
    Michael~Isard and
    Yangqing~Jia and
    Rafal~Jozefowicz and
    Lukasz~Kaiser and
    Manjunath~Kudlur and
    Josh~Levenberg and
    Dan~Man\'e and
    Rajat~Monga and
    Sherry~Moore and
    Derek~Murray and
    Chris~Olah and
    Mike~Schuster and
    Jonathon~Shlens and
    Benoit~Steiner and
    Ilya~Sutskever and
    Kunal~Talwar and
    Paul~Tucker and
    Vincent~Vanhoucke and
    Vijay~Vasudevan and
    Fernanda~Vi\'egas and
    Oriol~Vinyals and
    Pete~Warden and
    Martin~Wattenberg and
    Martin~Wicke and
    Yuan~Yu and
    Xiaoqiang~Zheng},
  year={2015},
}

```

텍스트 형태:

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo,
Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis,
Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow,
Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia,
Łukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster,
Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens,
Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker,
Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas,
Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke,
Yuan Yu, and Xiaoqiang Zheng.
TensorFlow: Large-scale machine learning on heterogeneous systems,
2015. Software available from tensorflow.org.

TensorFlow의 용도

이 페이지는 TensorFlow 시스템의 현재 용도를 설명한다.

TensorFlow를 연구용, 교육용, 또는 제품 개발용으로 사용하고 계시다면, 그 내용을 이곳에 추가하고자 합니다. 당신의 TensorFlow의 용도를 간략히 기술해서 이메일로 보내주십시오. 이 파일에 내용을 추가해서 Pull Request로 보내 주시면 더욱 좋습니다.

아래는 TensorFlow의 다양한 용도 중에서 몇 개를 나열한 것입니다.

- **랭크브레인 (RankBrain)**

- 기관: 구글
- 영역: 정보 검색
- 내용: www.google.com 검색 랭킹용 딥 뉴럴 네트 대규모 배포
- 추가정보: "[구글은 자사의 고수익을 내는 검색을 AI 머신으로 교체하고 있다.](#)"

- **인셉션 이미지 분류 모델 (Inception Image Classification Model)**

- 기관: 구글
- 내용: 베이스라인 모델과 2014 이미지 분류 챌린지에서 우승한 모델로 시작한 고정확도 컴퓨터 비전 모델 연구를 따름.
- 추가정보: 베이스라인 모델 [Arxiv paper](#)

- **스마트 응답 (SmartReply)**

- 기관: 구글
- 내용: 자동 이메일 응답 생성용 딥 LSTM 모델
- 추가정보: [구글 리서치 블로그 포스트](#)

- **마약 검색용 대규모 멀티태스크 네트워크 (Massively Multitask Networks for Drug Discovery)**

- 기관: 구글, 스탠포드 대학
- 영역: 마약 검색
- 내용: 마약일 가망성이 높은 것을 인식하는 딥 뉴럴 네트워크 모델
- 추가정보: [Arxiv paper](#)

- **OCR용 온-디바이스 컴퓨터 비전 (On-Device Computer Vision for OCR)**

- 기관: 구글
- 내용: 실시간 번역을 가능하게 하는 '광학 문자 인식'용 온-디바이스 컴퓨터 비전 모델
- 추가 정보: [구글 리서치 블로그 포스트](#)

자주 하는 질문 (FAQ)

이 문서는 TensorFlow에 대해 자주 하는 몇몇 질문의 답을 제공한다. 찾는 질문이 이 곳에 없으면, 아마도 TensorFlow [커뮤니티 리소스](#)에서 찾을 수 있을 것이다.

[TOC]

특징과 호환성

여러 대 컴퓨터에서 분산 트레이닝을 할 수 있는가?

할 수 있다! TensorFlow는 버전 0.8 부터 [분산 컴퓨테이션 지원](#). TensorFlow는 하나 이상의 컴퓨터에서 멀티 디바이스(CPU와 GPU)를 지원한다.

TensorFlow는 Python 3에서 동작 하는가?

0.6.0 릴리즈(2015년 12월 초)부터 Python 3.3+를 지원한다.

TensorFlow 그래프 작성

[그래프 작성 API 문서 참조.](#)

c = tf.matmul(a, b) 는 왜 매트릭스 곱 연산을 바로 실행하지 않는가??

TensorFlow 파이썬 API에서 `a`, `b`, `c` 는 `Tensor` 오브젝트다. `Tensor` 오브젝트는 오퍼레이션 결과에 대한 심볼릭 핸들이지만, 실제로 오퍼레이션의 결과 값을 갖고 있지는 않다. 대신, TensorFlow는 사용자가 복잡한 표현(전체 뉴럴 네트워크와 그래디언트 같은)을 데이터플로우 그래프로 작성하는 것을 권장한다. 그렇게 해 놓으면 전체 데이터플로우 그래프(또는 이의 하위그래프)의 컴퓨테이션을 TensorFlow `Session` 이 실행하게 할 수 있다. 이렇게 하면 오퍼레이션을 하나씩 실행 하는 것 보다, 전체 컴퓨테이션을 보다 효율적으로 실행 할 수 있다.

디바이스명은 어떻게 불리는가?

CPU 디바이스는 `"/device:CPU:0"` (또는 `"/cpu:0"`)로, i th GPU 디바이스는 `"/device:GPU: i "` (또는 `"/gpu: i "`)로 지원된다.

특정 디바이스에 오퍼레이션을 배치하려면 어떻게 해야 하는가?

한 디바이스에 오퍼레이션 그룹을 배치하려면 이들을 `with tf.device(name):` 컨텍스트 안에서 생성해라. TensorFlow에서 오퍼레이션을 디바이스에 할당하는 방법에 대한 자세한 설명 [TensorFlow를 GPU와 함께 사용하기](#), 그리고 다수의 GPU를 사용하는 모델에 대한 예 [CIFAR-10 tutorial](#)에 대한 how-to 문서를 참조해라.

사용한 텐서의 다른 타입에는 무엇이 있는가?

TensorFlow는 다양한 다른 종류의 데이터 타입과 텐서 쉐이프(shape)를 지원한다. 보다 자세한 내용은 [ranks, shapes, and types reference](#)를 참조해라.

TensorFlow 컴퓨테이션 실행하기

[API documentation on running graphs](#)를 참조해라.

피딩(feeding)과 플레이스홀더 placeholder)는 어떤 관계인가?

피딩은 TensorFlow 세션 API에 있는 메커니즘으로, 실행 시간에 하나 이상의 텐서를 위해 다른 값들로 교체하는 것을 허용한다. `Session.run()`에 대한 `feed_dict` 인자는 `Tensor` 오브젝트를 numpy(또는 다른 타입) 배열로 맵(map)하는 딕셔너리(dictionary)이고, 이들은 단계(step) 실행 시에 텐서의 값으로 사용될 것이다.

입력처럼, 항상 있어줘야 하는 텐서가 있다. `tf.placeholder()` op는 반드시 있어줘야 하는 텐서를 정의할 수 있도록 해주며, 옵션으로 모양(shape)의 제약(contrain) 또한 허용한다.

Session.run() 과 Tensor.eval() 의 차이는 무엇인가?

`t` 가 `Tensor` 오브젝트라면, `t.eval()` 은 `sess.run(t)` 의 속기 표현이다. (`sess` 가 현재의 디폴트 세션인 곳에서)

```
# `Session.run()` 사용.
sess = tf.Session()
c = tf.constant(5.0)
print sess.run(c)

# `Tensor.eval()` 사용.
c = tf.constant(5.0)
with tf.Session():
    print c.eval()
```

두번째 예제에서, 세션은 [컨텍스트 관리자](#)처럼 동작하며, 이는 `with` 블럭 내의 디폴트 세션으로 설정하는 효과가 있다. 컨텍스트 관리자 접근법은 간단한 유즈케이스(단위 테스트 같은)에서 보다 간결한 코드를 만들 수 있다. 다중 그래프와 세션을 다루는 코드에서는 `Session.run()` 을 명시적으로 호출하는것이 보다 직관적일 수 있다.

세션에 라이프타임이 있는가? 중간에 있는 텐서는 어떻게 되는가?

세션은 [variables](#), [queues](#), [readers](#) 같은 리소스를 가질 수 있다. 그리고 이런 리소스는 상당히 많은 메모리를 사용할 수 있다. 이런 리소스(그리고 관련된 메모리)는 세션이 닫힐 때 `Session.close()` 호출에 의해 릴리즈 된다.

`Session.run()` 호출의 일부로 생성된 중간 과정 텐서는 호출의 마지막 단계 또는 그 이전에 해제(free)될 것이다.

런타임은 그래프의 일부분을 병렬로 실행하는가?

텐서 런타임은 다수의 다양한 차원에 대해서 그래프 실행을 병렬화 한다:

- 각 ops는 멀티코어 CPU 또는 GPU에서 멀티쓰레드를 사용하는 병렬처리 구현을 갖고 있다.
- 텐서 그래프에 있는 각 노드는 멀티 노드에서 병렬로 처리 될 수 있고, 이는 스피드업을 가능하게 만든다 [CIFAR-10 멀티 GPU를 사용한 트레이닝](#).
- 세션 API는 병렬처리에서 다수의 동시적 스텝(step)(예, `Session.run()`)을 허용한다. 만약 하나의 스텝이 모든 리소스를 사용하지 않는다면, 이는 런타임의 처리량 향상을 가능하게 한다.

TensorFlow에서 지원하는 클라이언트 언어는?

TensorFlow는 다중 클라이언트 언어를 지원하도록 설계되었다. 현재, 가장 잘 지원 되는 클라이언트 언어는 [Python](#)이다. [C++ client API](#)는 그래프 론칭과 스텝을 돌리기 위한 인터페이스를 제공한다; 또한 [building graphs in C++](#)라는 실험용 API도 있다.

커뮤니티의 관심도에 따라서 더 많은 클라이언트 언어를 지원하고자 한다. TensorFlow에는 [C-based client API](#)가 있으며, 이는 많은 다른 언어에 대한 클라이언트 제작을 쉽게 해준다. 우리는 새로운 언어로 바인딩 하는 공헌을 바라고 있다.

TensorFlow는 디바이스에 있는 GPU와 CPU를 모두 사용하는가?

TensorFlow는 멀티 GPU와 CPU를 지원한다. TensorFlow가 오퍼레이션을 디바이스에 어떻게 할당하는지에 대한 자세한 방법과 다중 GPU를 사용하는 모델의 예제를 보려면 [using GPUs with TensorFlow](#) 문서를 참조해라.

TensorFlow는 계산능력(Compute Capability)가 3.5 이상인 디바이스만 사용한다는 것에 주의하라.

reader나 queue 사용시 session.run() 은 왜 멈추는가(Hang)?

reader와 queue 클래스는 입력(또는 큐의 메모리 공간)이 가능해 질 때 까지 *block* 할 수 있는 특별한 오퍼레이션을 제공한다. 이러한 오퍼레이션들은 TensorFlow 컴퓨테이션을 다소 많이 복잡하게 하는 비용을 감수 하고서라도, 복잡한 *input pipelines*를 만들 수 있도록 한다. 사용법에 대한 더 많은 정보를 원하면 [using QueueRunner objects to drive queues and readers](#) 를 참조해라.

변수

[variables](#), [variable scopes](#), [the API documentation for variables](#) 에 있는 how-to 문서 또한 참조해라.

변수의 라이프타임은 무엇인가?

변수는 세션에서 `tf.Variable.initializer` 오퍼레이션을 처음 실행할 때 생긴다. `session is closed` 될 때 변수는 삭제된다.

변수에 동시 접근 할 때 (변수는) 어떻게 되는가?

변수는 동시에 읽고 쓰는 오퍼레이션을 허용한다. 읽어온 변수가 동시에 업데이트 되는 경우, 변수는 변경될 수 있다. 기본적으로, 변수에 대한 동시할당 오퍼레이션은 상호배제(mutual exclusion)를 사용하지 않아도 허용된다. 변수에 값 할당 시 락(lock)을 얻어 오려면 `use_lock=True` 를 `Variable.assign()` 에 전달하라.

텐서 쉐이프(shape)

[TensorShape API documentation](#) 도 참조하라.

파이썬 환경에서 텐서 쉐이프를 어떻게 결정 할 수 있는가?

TensorFlow에서, 텐서는 정적 (inferred) 쉐이프와 동적 (true) 쉐이프를 갖고 있다. 정적 쉐이프는 `tf.Tensor.get_shape()` 메소드로 읽을 수 있다. 이 쉐이프는 텐서를 사용하는데 사용된 오퍼레이션으로부터 추론할 수 있다. 그리고 [partially complete](#)일 것이다. 정적 쉐이프가 완전히 정의되지 않았다면, 동적 쉐이프 `Tensor t` 가 `tf.shape(t)` 의 평가에 의해 결정 될 수 있다.

x.set_shape() 와 x = tf.reshape(x) 간의 차이는 무엇인가?

`tf.Tensor.set_shape()` 메소드는 `Tensor` 오브젝트의 정적 쉐이프를 업데이트 한다. 그리고 전형적으로 쉐이프를 직접 추론할 수 없을 때 추가 쉐이프 정보를 제공하기 위해 사용한다. 이는 텐서의 동적 쉐이프를 변경하지 않는다.

`tf.reshape()` 오퍼레이션은 다른 동적 쉐이프의 새로운 텐서를 생성한다.

변수 배치 크기로 동작하는 그래프는 어떻게 만들어야 하는가?

변수 배치 크기로 동작하는 그래프를 만들면 유용한 경우가 많다. 예를 들면, 같은 코드로 (미니-)배치 트레이닝과 싱글-인스턴스 추론을 할 수 있다. 결과 그래프는 [saved as a protocol buffer](#)와 [imported into another program](#)이 될 수 있다.

변수 크기 그래프를 만들 때, 가장 중요한 것은 파이썬 상수로 배치 크기를 인코드 하면 안되고, 대신 심볼릭 `Tensor` 를 사용해야 한다. 아래에 유용한 팁이 있다.

- `input` 으로 불리는 `Tensor` 로 부터 배치 차원을 뽑아내기 위해 `batch_size = tf.shape(input)[0]` 을 사용해라. 그리고 `batch_size` 로 불리는 `Tensor` 에 저장해라.
- `tf.reduce_sum(...)` / `batch_size` 대신 `tf.reduce_mean()` 를 사용해라.
- `placeholders for feeding input`를 사용한다면 `tf.placeholder(..., shape=[None, ...])` 로 플레이스홀더(placeholder)를 생성해서 변수 배치 차원을 명시 할 수 있다. 쉐이프의 `None` 엘리먼트는 변수 크기 차원에 대응된다.

TensorBoard

TensorFlow 그래프를 어떻게 가시화 할 수 있는가?

[graph visualization tutorial](#)를 참조해라.

TensorBoard에 데이터를 보내는 가장 간단한 방법은 무엇인가?

TensorFlow 그래프에 요약 ops를 추가하고, 요약 내용을 로그 디렉토리에 쓰기 위해 `SummaryWriter` 를 사용해라. 그리고 나서, TensorBoard를 다음 명령으로 시작해라.

```
python tensorflow/tensorboard/tensorboard.py --logdir=path/to/log-directory
```

보다 자세한 내용은 [Summaries and TensorBoard tutorial](#) 를 참조해라.

TensorBoard를 띠울 때마다, 네트워크 보안 팝업이 뜬다.

--host=localhost 옵션을 줘서, '0.0.0.0'이 아닌 localhost로 서비스 되도록 변경할 수 있다. 이렇게 하면 보안 경고가 안 나올 것이다.

TensorFlow 확장

how-to 문서인 [adding a new operation to TensorFlow](#) 를 참조해라.

내가 가진 데이터는 커스텀 포맷이다. TensorFlow로 읽으려면 어떻게 해야 하는가?

커스텀 포맷 데이터를 처리하는 옵션이 두 가지 있다.

쉬운 옵션은 파이썬으로 파싱 코드를 작성해서 데이터를 numpy 배열로 변경하고

`tf.placeholder()` 에 그 데이터 텐서를 넣는 것이다.

보다 자세한 내용은 [using placeholders for input](#)를 봐라. 이 접근법은 빨리 만들어서 돌려보기 좋지만, 파싱이 성능 병목이 될 수 있다.

보다 효율적인 옵션은 [add a new op written in C++](#) 를 이용해서 당신의 데이터 포맷을 파싱하는 op를 추가하는 것이다. [guide to handling new data formats](#)에 이를 처리하는 절차에 대한 더 많은 정보가 있다.

입력의 개수가 변경되는 것을 처리하는 오퍼레이션을 어떻게 정의해야 하는가?

TensorFlow op 등록 메커니즘은 입력 정의를 허용한다. 그런 입력에는 싱글 텐서, 동일 타입의 텐서 리스트 (예를 들면, 가변 길이 목록의 텐서를 같이 등록할 때), 또는 이종 타입의 텐서 리스트 (예를 들면, 큐에 텐서 튜플을 넣을 때). 이종 입력 타입을 정의하는 방법에 대한 보다 자세한 내용은 [adding an op with a list of inputs or outputs](#) 를 참조해라.

Miscellaneous

기타사항

TensorFlow의 코딩 스타일 컨벤션은 무엇인가?

TensorFlow 파이썬 API는 [PEP8](#) 컨벤션을 따른다.* 특히, 클래스는 `CamelCase`, 함수와 속성은 `snake_case` 를 사용한다. 또한, [Google Python style guide](#) 도 따른다.

TensorFlow C++ 코드 베이스는 [Google C++ style guide](#) 를 따른다.

*

(* 하나의 예외사항: 4-space 들여쓰기 대신 2-space 들여쓰기를 사용한다.)

어휘 (Glossary)

Broadcasting operation

텐서 인자의 구조(shape)와의 호환을 위해 [numpy-style broadcasting](#)을 사용하는 연산입니다.

Device

계산을 실행할 수 있으며 자체 주소 공간을 갖는 GPU나 CPU와 같은 하드웨어의 일부입니다.

eval

값을 결정하는데 필요한 그래프 계산을 트리거링하며 `Tensor`의 값을 반환하는 `Tensor`의 메서드입니다. 세션에서 시작된 그래프에서 `Tensor`의 `eval()`을 호출하기만 하면됩니다.

Feed

세션에서 시작된 그래프에 있는 노드로 직접 텐서를 패치하는 TensorFlow의 메커니즘입니다. 그래프를 생성하는 때가 아닌 그래프의 실행을 트리거링할 때 피드를 적용합니다. 피드는 임시로 노드를 텐서값으로 바꿉니다. 계산을 시작하는 `run()` 또는 `eval()` 호출에 피드 데이터를 인자로 써 공급합니다. 실행 후에는 피드가 사라지며 원래 노드의 정의가 남습니다. 일반적으로 그것들을 생성하기위해 `tf.placeholder()`를 사용하여 "feed" 노드가 될 특정한 노드들을 지정합니다. 좀 더 자세한 내용은 [Basic Usage](#)를 보십시오.

Fetch

세션에서 시작된 그래프에서 텐서를 검색하기위한 TensorFlow의 메커니즘입니다. 그래프를 생성하는 때가 아닌 그래프의 실행을 트리거링할 때 폐치를 검색합니다. 노드 또는 노드들의 텐서값을 가져오기위해 `Session` 객체에서 `run()`을 호출하여 그래프를 실행시키고 검색할 노드명 리스트를 전달합니다. 좀 더 자세한 내용은 [Basic Usage](#)를 보십시오.

Graph

방향성 비순환 그래프로 계산을 나타냅니다. 그래프의 노드들은 수행되어야 하는 연산들을 나타냅니다. 그래프의 엣지는 데이터 또는 제어 종속성을 나타냅니다. `GraphDef` 는 그래프를 시스템 (API)에 서술하기위해 사용되는 프로토콜 버퍼이며 `NodeDef` (아래를 보십시오.)의 컬렉션으로 이루어져 있습니다. `GraphDef` 는 조작하기 쉬운 (C++) `Graph` 객체로 변환될 수 있습니다.

IndexedSlices

파이썬 API에서, 텐서의 첫번째 차원만을 따르는 희소 텐서를 나타내는 TensorFlow의 표현식입니다. 만약 텐서가 `k` 차원이면, `IndexedSlices` 인스턴스는 논리적으로 텐서의 첫번째 차원을 따르는 `(k-1)` 차원 슬라이스들의 컬렉션을 나타냅니다. 슬라이스의 인덱스는 1차원 벡터와 합쳐진 상

태로 저장이 되고, 각 슬라이스는 하나의 `k` 차원 텐서의 형태로 합쳐집니다. 만약 첫번째 차원에서 희소성이 제한되지 않는다면 `sparseTensor` 를 사용하십시오.

Node

그래프의 요소.

연산을 설정하는데 필요한 `attrs` 의 값을 포함해 특정한 계산 `Graph` 에서 하나의 노드로써 특정한 연산을 어떻게 실행시키는지에 대한 방법을 서술합니다. 다형적 연산을 위해 `attrs` 는 `Node` 의 시그니처를 완전히 결정지을 수 있는 충분한 정보를 포함합니다. 자세한 건 `graph.proto` 를 보십시오.

Op (operation)

TensorFlow 런타임에서: `add` 나 `matmul` 또는 `concat` 과 같은 연산의 타입입니다. [how to add an op](#)에 설명된대로 런타임에 새로운 연산을 추가할 수 있습니다.

파이썬 API에서: 그래프의 노드입니다. 연산은 `tf.Operation` 클래스의 인스턴스로 나타냅니다. `Operation` 의 `type` 프로퍼티는 `add` 나 `matmul` 과 같은 노드에 대한 실행 연산을 나타냅니다.

Run

시작된 그래프에서 연산 실행의 액션입니다. 그래프가 `Session` 에서 시작되어야 합니다.

파이썬 API에서: `Session` 클래스 `tf.Session.run` 의 메서드입니다. 피드와 폐치를 하기 위한 텐서를 `run()` 호출에 전달할 수 있습니다.

C++ API에서: `tensorflow::Session` 의 메서드입니다.

Session

시작된 그래프를 나타내는 런타임 객체입니다. 그래프에서 연산을 실행하기 위한 메서드들을 제공합니다.

파이썬 API에서: `tf.Session`

C++ API에서: 그래프를 시작하고 연산을 실행할 때 사용되는 클래스 `tensorflow::Session`

Shape

텐서의 차원과 크기.

시작된 그래프에서: 노드 사이를 흐르는 텐서의 프로퍼티. 몇몇 연산은 그들의 입력의 구조(shape)에 대한 엄격한 요구조건을 가지고 있으며, 안맞을시 런타임에서 에러를 리포팅합니다.

파이썬 API에서: 그래프 생성 API에 있는 파이썬 `Tensor` 의 속성. 생성 도중 텐서의 구조(shape)는 부분적으로만 알 수 있거나 아예 모를수도 있습니다. `tf.TensorShape` 를 보십시오.

C++ API에서: 텐서의 구조(shape)을 나타내는데 사용되는 클래스 `tensorflow::TensorShape`

SparseTensor

파이썬 API에서, 임의의 위치에서 드물게 존재하는 텐서를 나타내는 TensorFlow의 표현식입니다. `SparseTensor` 는 딕셔너리-키 포맷을 사용해 값의 인덱스에 따라 비어 있지 않는 값들만을 저장합니다. 다시 말하면, m 개의 비어 있지 않는 값들이 있을 때, 이는 길이가 m 인 값의 벡터와 m 개의 인덱스의 행을 갖는 행렬을 유지합니다. 효율성을 위해, `sparseTensor` 는 차원수가 증가함에 따라 정렬된 (즉, 행 위주 순서) 인덱스가 필요합니다. 만약 희소성이 첫번째 차원만 따르는 경우엔 `IndexedSlices` 를 사용하십시오.

Tensor

`Tensor` 는 타입을 가진 다차원 배열입니다. 예를 들면, `[batch, height, width, channel]` 차원을 갖는 이미지의 작은 배치를 나타내는 실수형 숫자들의 4차원 배열이 있습니다.

시작된 그래프에서: 노드 사이를 흐르는 데이터의 타입.

파이썬 API에서: 그래프에 추가된 연산의 출력과 입력을 나타내는데 사용되는 클래스 `tf.Tensor`. 이 클래스의 인스턴스는 데이터를 저장하지 않습니다.

C++ API에서: `tensorflow::Tensor` 를 호출하는 `Session::Run()` 에서 반환되는 텐서를 나타내는데 사용되는 클래스. 이 클래스의 인스턴스는 데이터를 가지고 있습니다.

Tensor Ranks, Shapes, Type

TensorFlow 프로그램은 모든 데이터를 tensor 데이터 구조를 사용해서 표현한다. TensorFlow의 tensor는 n-차원 배열 또는 리스트라고 생각해도 된다. 하나의 tensor는 정적 타입과 동적 차원을 갖고 있다. 컴퓨테이션 그래프의 노드들은 오직 tensor만을 전달 할 수 있다.

Rank

TensorFlow 시스템에서, tensor는 rank라는 차원 단위로 표현된다. Tensor rank는 행렬의 rank와 다르다. Tensor rank(*order*, *degree*, *-n_dimension* 으로도 언급됨)는 tensor의 차원수다. 예를 들어, 아래 tensor(Python 리스트로 정의)의 rank는 2다.

```
t = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

rank 2인 tensor는 행렬, rank 1인 tensor는 벡터로 생각할 수 있다. rank 2인 tensor는 `t[i, j]` 형식으로 원소에 접근할 수 있다. rank 3인 tensor는 `t[i, j, k]` 형식으로 원소를 지정할 수 있다.

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)

Shape

TensorFlow 문서는 tensor 차원을 표현할 때 세 가지 기호를 사용한다. rank, shape, 차원수. 아래 표는 그 세 가지의 관계를 보여준다:

Rank	Shape	Dimension number	Example
0	[]	0-D	A 0-D tensor. A scalar.
1	[D0]	1-D	A 1-D tensor with shape [5].
2	[D0, D1]	2-D	A 2-D tensor with shape [3, 4].
3	[D0, D1, D2]	3-D	A 3-D tensor with shape [1, 4, 3].
n	[D0, D1, ... Dn-1]	n-D	A tensor with shape [D0, D1, ... Dn-1].

Shape는 Python 리스트 / 정수형 튜플 또는 [TensorShape class](#)로 표현 할 수 있다.

Data types

Tensor는 차원 말고도 데이터 타입도 갖는다. 아래의 데이터 타입을 tensor에 지정할 수 있다.

Data type	Python type	Description
DT_FLOAT	tf.float32	32 비트 부동 소수.
DT_DOUBLE	tf.float64	64 비트 부동 소수.
DT_INT8	tf.int8	8 비트 부호 있는 정수.
DT_INT16	tf.int16	16 비트 부호 있는 정수.
DT_INT32	tf.int32	32 비트 부호 있는 정수.
DT_INT64	tf.int64	64 비트 부호 있는 정수.
DT_UINT8	tf.uint8	8 비트 부호 없는 정수.
DT_STRING	tf.string	가변 길이 바이트 배열. Tensor의 각 원소는 바이트 배열.
DT_BOOL	tf.bool	불리언.
DT_COMPLEX64	tf.complex64	2개의 32 비트 부동 소수로 만든 복소수 : 실수부 + 허수부
DT_COMPLEX128	tf.complex128	2개의 64 비트 부동 소수로 만든 복소수 : 실수부 + 허수부
DT_QINT8	tf.qint8	8 비트 부호 있는 정수로 quantized Ops에서 사용.
DT_QINT32	tf.qint32	32 비트 부호 있는 정수로 quantized Ops에서 사용.
DT_QUINT8	tf.quint8	8 비트 부호 없는 정수로 quantized Ops에서 사용.

(역주: `quantized op` 는 fixed-point 데이터로써, [quantized op](#)에 의하면 아직 문서화 되지 않은 기능.)

TensorFlow Version Semantics

Semantic Versioning 2.0

Once we reach version 1.0, TensorFlow will follow Semantic Versioning 2.0 (semver). For details, see <http://semver.org>. Each release version of TensorFlow has the form

`MAJOR.MINOR.PATCH`. Changes to the each number have the following meaning:

- **MAJOR**: Backwards incompatible changes. Code and data that worked with a previous major release will not necessarily work with a new release. However, in some cases existing TensorFlow data (graphs, checkpoints, and other protobufs) may be migratable to the newer release; see below for details on data compatibility.
- **MINOR**: Backwards compatible features, speed improvements, etc. Code and data that worked with a previous minor release *and* which depends only the public API will continue to work unchanged. For details on what is and is not the public API, see below.
- **PATCH**: Backwards compatible bug fixes.

Before 1.0, semver allows backwards incompatible changes at any time. However, to support users now, we will use the format `0.MAJOR.MINOR` (shifted one step to the right).

Thus 0.5.0 to 0.6.0 may be backwards incompatible, but 0.6.0 to 0.6.1 will include only backwards compatible features and bug fixes.

At some point (especially as we approach 1.0) we will likely use prerelease versions such as X.Y.Z-alpha.1, but we do not yet have specific plans (beyond the restrictions of semver).

Public API

Only the public API of TensorFlow is backwards compatible across minor and patch versions. The public API consists of

- The documented [C++ and Python APIs](#).
- The following protocol buffer files: `attr_value`, `config`, `event`, `graph`, `op_def`, `reader_base`, `summary`, `tensor`, `tensor_shape`, and `types`.

The public C++ API is exposed through the header files in `tensorflow/core/public`. The public Python API is unfortunately **not** everything available through the tensorflow python module and its submodules, since we do not yet use `__all__` everywhere ([#421](#)). Please

refer to the documentation to determine whether a given Python feature is part of the public API. For now, the protocol buffers are defined in [tensorflow/core/framework/*.proto](#) (#484).

Details That Are Not Public

The following are specifically **not** part of the public API: they are allowed to change without notice across minor releases and even patch releases if bug fixes require it:

- **Details of composite ops:** Many public functions in Python expand to several primitive ops in the graph, and these details will be part of any graphs saved to disk as GraphDefs. These details are allowed to change for minor releases. In particular, regressions tests that check for exact matching between graphs are likely to break across minor releases, even though the behavior of the graph should be unchanged and existing checkpoints will still work.
- **Floating point numerical details:** The specific floating point values computed by ops may change at any time: users should rely only on approximate accuracy and numerical stability, not on the specific bits computed. Changes to numerical formulas in minor and patch releases should result in comparable or improved accuracy, with the caveat that in machine learning improved accuracy of specific formulas may result in worse accuracy for the overall system.
- **Random numbers:** The specific random numbers computed by the [random ops](#) may change at any time: users should rely only on approximately correct distributions and statistical strength, not the specific bits computed. However, we will make changes to random bits rarely and ideally never for patch releases, and all such intended changes will be documented.

Compatibility for Graphs and Checkpoints

Many users of TensorFlow will be saving graphs and trained models to disk for later evaluation or more training, often changing versions of TensorFlow in the process. First, following semver, any graph or checkpoint written out with one version of TensorFlow can be loaded and evaluated with a later version of TensorFlow with the same major release. However, we will endeavour to preserve backwards compatibility even across major releases when possible, so that the serialized files are usable over long periods of time.

There are two main classes of saved TensorFlow data: graphs and checkpoints. Graphs describe the data flow graphs of ops to be run during training and inference, and checkpoints contain the saved tensor values of variables in a graph.

Graphs are serialized via the `GraphDef` protocol buffer. To facilitate (rare) backwards incompatible changes to graphs, each `GraphDef` has an integer version separate from the TensorFlow version. The semantics are:

- Each version of TensorFlow supports an interval of `GraphDef` versions. This interval will be constant across patch releases, and will only grow across minor releases. Dropping support for a `GraphDef` version will only occur for a major release of TensorFlow.
- Newly created graphs use the newest `GraphDef` version.
- If a given version of TensorFlow supports the `GraphDef` version of a graph, it will load and evaluate with the same behavior as when it was written out (except for floating point numerical details and random numbers), regardless of the major version of TensorFlow. In particular, all checkpoint files will be compatible.
- If the `GraphDef` upper bound is increased to X in a (minor) release, there will be at least six months before the lower bound is increased to X.

For example (numbers and versions hypothetical), TensorFlow 1.2 might support `GraphDef` versions 4 to 7. TensorFlow 1.3 could add `GraphDef` version 8 and support versions 4 to 8. At least six months later, TensorFlow 2.0.0 could drop support for versions 4 to 7, leaving version 8 only.

Finally, when support for a `GraphDef` version is dropped, we will attempt to provide tools for automatically converting graphs to a newer supported `GraphDef` version.

C++ API Compatibility

Only patch releases will be binary compatible at the C++ level. That is, minor releases are backwards compatible in terms of behavior but may require a recompile for downstream C++ code. As always, backwards compatibility is only provided for the public C++ API.

로드맵

마지막 업데이트: 2016년 6월 3일

TensorFlow는 빠르게 움직이는 프로젝트입니다. 커뮤니티가 가까운 미래를 이해할 수 있도록, 우리가 내부에서 작업 중인 것을 문서로 공유합니다. 이런 기능 대부분은 커뮤니티에서 요청됐고, 우리는 [기여](#)를 환영합니다.

이 목록에 있는 기능은 다음 몇 개월을 목표로 하고 있습니다. 이 시점에, 우리는 이러한 기능에 대한 일정이 없습니다.

C++ 사용자만을 위한 지원 개선

- 그래프(Graph) 구조
- 그라디언트
- 모양(Shape) 유추

TensorFlow를 보다 쉽게 사용할 수 있도록 제작

- training jobs을 분산하기 위한 쉬운 설정

성능

- 속도와 메모리 벤치 마크
- 성능과 메모리 사용량 개선

핵심 기능

- 반복되는 일부 그래프 평가 ([#672](#))
- 자동 오퍼레이션 배치 ([#2126](#))

플랫폼

- OpenCL 지원 ([#22](#))
- Windows 지원 ([#17](#))

커뮤니티

- 더 많은 교육 지원.
- 오픈소스 빅데이터 생태계에 TensorFlow의 더 나은 통합 ([#1996](#), [#2218](#), [#2655](#))

- 모델 벤치마킹과 비교 도구