目錄

说明

TensorFlow 正式版本 V1.0 已经发布，api 较之前 V0.xx 版本发生了较大变化，Tutorial、HowTo 等文档也发生了很大变化。新的文档更加合理，对TensorFlow甚至机器学习的新手更加友好，更适合循序渐进的学习。

网络上流传较广的TensorFlow中文文档大多为 TensorFlow中文社区 的文档，翻译自 V0.5。

在此，选取版本 r1.1 的文档进行翻译，r1.1与r1.0的文档内容区别不大，结构做了一些调整。

具体内容查看 目录。

本项目同时更新于 GitBook 与 GitHub。

GitHub 项目地址：https://github.com/efeiefei/tensorflow_documents_zh/

GitBook 阅读地址：https://efeiefei.gitbooks.io/tensorflow_documents_zh/

欢迎联系，一起翻译！

安装TensorFlow

如下指南描述了如何安装TensorFlow的不同版本。

- 在 Ubuntu 上安装 TensorFlow
- 在 Mac OS X 上安装 TensorFlow
- 在 Windows 上安装 TensorFlow
- 从源码安装 TensorFlow

Python TensorFlow API 自版本 0.n 到 1.0 变化花了很多。如下指南描述了如何从老旧 TensorFlow 应用迁移到1.0版本。

迁移到 TensorFlow 1.0

如下指南描述了如何安装其他语言的TensorFlow库。这些API是为了在应用中使用TensorFlow模型，所以并不如Python API一样具有扩展性。

- 为 Java 安装 TensorFlow
- 为 C 安装 TensorFlow
- 为 GO 安装 TensorFlow

在 Ubuntu 上安装 TensorFlow

这篇指南描述了如何在 Ubuntu 上安装 TensorFlow。这些实例也可能在其他 Linux 版本生效，但我们只在 Ubuntu 14.04 及更高的版本上测试过。

确定 TensorFlow 版本

如下之中选择一种来安装:

- 只支持 **CPU** 的 **TensorFlow**。如果你的系统不支持 NVIDIA® GPU, 你必须安装这个版本。这个版本的 TensorFlow 通常安装起来比较简单（一般 5 到 10分钟），所以即使你拥有 NVIDIA GPU，我们也推荐首先安装这个版本。
- 支持 **GPU** 的 **TensorFlow**. TensorFlow 在 GPU 上通常比在 CPU 上的执行的更快。所以如果你有符合如下要求的 NVIDIA® GPU 并且需要注重性能，可以随后安装这个版本。

GPU support TensorFlow 的 NVIDIA 需求

需要事先安装如下 NVIDIA 软件。

- CUDA® Toolkit 8.0. 详见 NVIDIA's documentation。确保按照文档中描述的将 Cuda 相关路径加入到 `LD_LIBRARY_PATH` 环境变量中。
- CUDA Toolkit 8.0 相关的 NVIDIA 驱动。
- cuDNN v5.1。详见 NVIDIA's documentation。确保创建了 `CUDA_HOME` 环境变量。
- CUDA Compute Capability 3.0 或更高的 GPU 芯片。支持的 GPU 芯片详见 NVIDIA documentation 。
- libcupti-dev 库, 该库提供了高级的性能支持，按如下命令安装：

```
$ sudo apt-get install libcupti-dev
```

如果含有上述库但版本较老，先升级。如果不能升级，操作如下：

- 从源码安装 TensorFlow
- 安装或升级至少如下版本:
  - CUDA toolkit 7.0 或更高
  - cuDNN v3 或更高
  - CUDA Compute Capability 3.0 或更高的 GPU 芯片。

确定如何安装 TensorFlow

有如下选择：

- virtualenv
- "native" pip
- Docker
- Anaconda

推荐 **virtualenv** 安装 （略过四种方法的说明，自行查找）

virtualenv 安装

步骤如下：

1. 安装 pip 及 virtualenv：

   ```
   $ sudo apt-get install python-pip python-dev python-
   virtualenv
   ```

2. 建立 virtualenv 环境：

   ```
   $ virtualenv --system-site-packages targetDirectory
   ```

   `targetDirectory` 指明了 virtualenv 的位置。

3. 激活 virtualenv 环境：

   ```
   $ source ~/tensorflow/bin/activate # bash, sh, ksh, or zsh
   $ source ~/tensorflow/bin/activate.csh  # csh or tcsh
   ```

   如上操作会将提示符更改为如下：

   ```
    (tensorflow)$
   ```

4. 如下命令中选取一个安装 TensorFlow：

   ```
   (tensorflow)$ pip install --upgrade tensorflow      # for
   Python 2.7
   (tensorflow)$ pip3 install --upgrade tensorflow     # for
   Python 3.n
   (tensorflow)$ pip install --upgrade tensorflow-gpu  # for
   Python 2.7 and GPU
   (tensorflow)$ pip3 install --upgrade tensorflow-gpu # for
   Python 3.n and GPU
   ```

上述命令成功则跳过步骤5，否则执行步骤5。

5. (可选) 如果步骤4失败 (通常因为 pip 版本小于 8.1)：

```
(tensorflow)$ pip install --upgrade TF_PYTHON_URL   # Python 2.7
(tensorflow)$ pip3 install --upgrade TF_PYTHON_URL  # Python 3.N
```

`TF_PYTHON_URL` 指定了 python tensorflow 的包的地址。`TF_PYTHON_URL` 依赖于操作系统、Python 版本、GPU 支持，从 这里 找到合适的URL。如，安装 TensorFlow for Linux, Python 2.7、CPU-only，使用如下命令：

```
(tensorflow)$ pip3 install --upgrade \
https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-1.1.0-cp34-cp34m-linux_x86_64.whl
```

如果遇到安装问题，详见 常见安装问题.

下一步

安装完毕之后： 验证安装结果。

注意，每次使用 TensorFlow 前需要先激活 virtualenv 环境，使用如下命令：

```
$ source ~/tensorflow/bin/activate      # bash, sh, ksh, or zsh
$ source ~/tensorflow/bin/activate.csh  # csh or tcsh
```

激活之后可从当前 shell 执行命令，此时提示符变成如下：

```
(tensorflow)$
```

使用完毕 TensorFlow 可用 `deactivate` 命令退出当前 virtualenv 环境：

```
(tensorflow)$ deactivate
```

此时提示符将会返回到默认提示符（在 `PS1` 环境变量中定义）。

卸载 TensorFlow

卸载 TensorFlow，删除相关目录即可：

```
$ rm -r targetDirectory
```

原生 pip 安装

注意： setup.py 的 REQUIRED_PACKAGES 部分 列出了 pip 将会安装或者升级的 TensorFlow 包。

前提：Python 及 Pip

Python 已经在 Ubuntu 中自动安装了。花谢时间确定（ `python -v` ）你的操作系统中含有如下 Python 版本中的一个：

- Python 2.7
- Python 3.3+

Pip 或 pip3 通常已经在 Ubuntu 中安装。花些时间确定（ `pip -V` 或 `pip3 -V` ）已经安装。强烈建议使用 8.1 或更高的版本。如果 8.1 或更高的版本没有安装，使用如下命令安装或升级到最新 pip 版本：

```
$ sudo apt-get install python-pip python-dev
```

安装 TensorFlow

假定已经安装了如上必要软件，如下步骤安装 TensorFlow：

1. 如下命令之一安装：

   ```
   $ pip install tensorflow      # Python 2.7; CPU support (no
   GPU support)
   $ pip3 install tensorflow     # Python 3.n; CPU support (no
   GPU support)
   $ pip install tensorflow-gpu  # Python 2.7;  GPU support
   $ pip3 install tensorflow-gpu # Python 3.n; GPU support
   ```

   如上命令执行完毕，可验证安装结果.

2. (可选) 如果步骤1失败，使用如下命令安装：

   ```
   $ sudo pip  install --upgrade TF_PYTHON_URL   # Python 2.7
   $ sudo pip3 install --upgrade TF_PYTHON_URL   # Python 3.N
   ```

`TF_PYTHON_URL` 指定了 python tensorflow 的包的地址。`TF_PYTHON_URL` 依赖于操作系统、Python 版本、GPU 支持，从 这里 找到合适的URL。 例如，安装 TensorFlow for Linux, Python 3.4、CPU-only，使用如下命令：

```
$ sudo pip3 install --upgrade \
https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-1.1.0-cp34-cp34m-linux_x86_64.whl
```

该步骤失败查询 常见安装问题.

下一步

安装完毕之后： 验证安装结果。

卸载 TensorFlow

如下命令卸载：

```
$ sudo pip uninstall tensorflow  # for Python 2.7
$ sudo pip3 uninstall tensorflow # for Python 3.n
```

Docker 安装

如下步骤通过 Docker 安装 TensorFlow：

1. 按描述在你的机器上安装 Docker Docker documentation.
2. 可选，建立名为 `docker` 的用户组以便不通过 sudo 来登陆 container， Docker documentation. （如果省略该步骤，每次启动 Docker 都需要使用 sudo。）
3. 为安装支持 GPU 的 TensorFlow 版本，需要首先 安装 nvidia-docker
4. 启动含有 TensorFlow binary images. 之一的 Docker 容器。

本章节的其余部分描述了如何启动一个 Docker 容器。

CPU-only

使用如下命令启动一个 CPU-only Docker 容器：

```
$ docker run -it -p hostPort:containerPort TensorFlowCPUImage
```

其中：

- *`-p hostPort:containerPort`* 可选。 如果计划从 shell 执行 TensorFlow 程序，忽略

该选项。 如果计划作为 `Jupyter notebooks` 执行 TensorFlow 程序，设定 *`hostPort`* 及 *`containerPort`* 均为 8888。如果计划在容器中启动 TensorBoard， 添加第二个 `-p` 参数, 设定 *hostPort* 及 *containerPort* 均为 6006.

- *`TensorFlowCPUImage`* 是必须的. 它指定了使用的容器，如下选项中选取一个：
  - `gcr.io/tensorflow/tensorflow`, TensorFlow CPU 镜像。
  - `gcr.io/tensorflow/tensorflow:latest-devel`, 最新的 TensorFlow CPU 镜像外加源代码。
  - `gcr.io/tensorflow/tensorflow:`*`version`*，指定版本（如1.0.1）。
  - `gcr.io/tensorflow/tensorflow:`*`version`*`-devel`，指定版本外加源代码。

  `gcr.io` 是 Google 的容器仓库。注意一些镜像同样可从 dockerhub获取。

例如，如下命令在一个容器中启动最新的 TensorFlow CPU 镜像，你可以在 shell 中执行 TensorFlow 程序：

```
$ docker run -it gcr.io/tensorflow/tensorflow bash
```

如下命令同样在一个容器中启动最新的 TensorFlow CPU 镜像。 但是在该容器中，你可以在 `Jupyter notebook` 中执行 TensorFlow 程序：

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

Docker 会在你第一次启动 TensorFlow 镜像时下载它。

GPU support

安装支持 GPU 的 TensorFlow 之前，确保你的系统满足 NVIDIA software requirements。To launch a Docker container 通过如下命令，启动一个支持 GPU 的 TensorFlow 的 Docker 容器

```
$ nvidia-docker run -it -p hostPort:containerPort
TensorFlowGPUImage
```

其中：

- *`-p hostPort:containerPort`* 可选。 如果计划从 shell 执行 TensorFlow 程序，忽略该选项。 如果计划作为 `Jupyter notebooks` 执行 TensorFlow 程序，设定 *`hostPort`* 及 *`containerPort`* 均为 8888。
- *TensorFlowCPUImage* 是必须的. 它指定了使用的容器，如下选项中选取一个：
  - `gcr.io/tensorflow/tensorflow:latest-gpu`，最新 TensorFlow GPU 镜像。
  - `gcr.io/tensorflow/tensorflow:latest-devel-gpu`，最新 TensorFlow

GPU 镜像外加源代码。
- `gcr.io/tensorflow/tensorflow:version-gpu`，指定版本的 TensorFlow GPU 镜像。
- `gcr.io/tensorflow/tensorflow:version-devel-gpu`，指定版本的 TensorFlow GPU 镜像外加源代码。

我们推荐安装一个 `最新` 版。如下命令可以在 Docker 容器中启动一个最新版本 TensorFlow GPU 镜像， 你可以在其 shell 中执行TensorFlow 程序：

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

如下命令同样在一个容器中启动最新的 TensorFlow GPU 镜像。 但是在该容器中，你可以在 `Jupyter notebook` 中执行 TensorFlow 程序：

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

如下命令启动一个老版本的 TensorFlow：

```
$ nvidia-docker run -it -p 8888:8888
gcr.io/tensorflow/tensorflow:0.12.1-gpu
```

Docker 会在你第一次启动 TensorFlow 镜像时下载它。详情 TensorFlow docker readme.

下一步

安装完毕之后： 验证安装结果。

Anaconda 安装

按照如下步骤在 Anaconda 环境中安装 TensorFlow：

1. 按说明下载并安装 Anaconda： Anaconda download site
2. 建立一个 conda 环境，命名为 `tensorflow`，以便运行某个 Python 版本：

   ```
   $ conda create -n tensorflow
   ```

3. 激活 anaconda 环境：

```
$ source activate tensorflow
(tensorflow)$  # 你的提示符应变化
```

4. 在你的 conda 环境中安装 TensorFlow：

```
(tensorflow)$ pip install --ignore-installed --upgrade
TF_PYTHON_URL
```

其中 `TF_PYTHON_URL` 是 TensorFlow Python 包地址. 比如：如下命令可以为 Python 3.4 安装 CPU-only 版本的 TensorFlow：

```
(tensorflow)$ pip install --ignore-installed --upgrade \
https://storage.googleapis.com/tensorflow/linux/cpu/tensorflo
w-1.1.0-cp34-cp34m-linux_x86_64.whl
```

验证安装结果

按如下操作验证 TensorFlow 安装结果：

1. 确保准备环境完备
2. 执行一个简短的 TensorFlow 程序

准备环境

If you installed on native pip, virtualenv, or Anaconda, then do the following: 如果通过原生 pip、virtualenv、Anaconda 安装，做如下操作：

1. 启动一个 terminal
2. 如果通过 virtualenv 或 Anaconda 安装，激活容器
3. 如果你安装了 TensorFlow 源码，定位到不含源码的任一目录中

如果通过 Docker 安装，启动一个可以通过 bash 操作的 Docker 容器：

```
$ docker run -it gcr.io/tensorflow/tensorflow bash
```

执行一个简短的 TensorFlow 程序

在 shell 中调用 Python：

```
$ python
```

在 Python 交互式环境中输入如下命令：

```
>>> import tensorflow as tf
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()
>>> print(sess.run(hello))
```

如果系统输出如下，则安装成功：

```
Hello, TensorFlow!
```

如果你新接触 TensorFlow，参考初识 TensorFlow进行下一步学习。

如果系统输出错误信息而非欢迎信息，查看常见安装问题。

常见安装问题

我们依靠 Stack Overflow 来编写 TensorFlow 安装问题及解决方案的文档。 如下表格包含了 Stack Overflow 上比较常见的安装问题的连接。 如果你遇到了不在列表中的新的错误信息或者其他安装问题，请在 Stack Overflow 上搜索。 如果搜索不到，请在 Stack Overflow 上提出一个新的问题，并打上 `tensorflow` 的标签。

Stack Overflow Link	Error Message
36159194	`ImportError: libcudart.so.Version: cannot open shared object file: No such file or directory`
41991101	`ImportError: libcudnn.Version: cannot open shared objec file: No such file or directory`
36371137 and here	`libprotobuf ERROR google/protobuf/src/google/protobuf/io/coded_stream.cc: A protocol message was rejected because it was too big (more than 67108864 bytes). To increase the limit (or to disable these warnings), CodedInputStream::SetTotalBytesLimit() in google/protobuf/io/coded_stream.h.`

35252888	`Error importing tensorflow. Unless you are using bazel, should not try to import tensorflow from its source director please exit the tensorflow source tree, and relaunch your python interpreter from there.`
33623453	`IOError: [Errno 2] No such file or directory: '/tmp/pip-o6Tpui-build/setup.py'`
42006320	`ImportError: Traceback (most recent call last): File ".../tensorflow/core/framework/graph_pb2.py", li 6, in from google.protobuf import descriptor as _descriptor ImportError: cannot import name 'descriptor'`
35190574	`SSLError: [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed`
42009190	`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/.../lib/python/_markerlib'`
36933958	`... 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'`

TensorFlow Python 包地址（`TF_PYTHON_URL`）

一些安装方法需要 TensorFlow Python 包，它的地址依赖于几个方面：

- 操作系统
- Python 版本
- CPU-only 还是 GPU support

Python 2.7

CPU only:

```
https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-1.1.0-cp27-none-linux_x86_64.whl
```

GPU support:

```
https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow_gpu-1.1.0-cp27-none-linux_x86_64.whl
```

注意 GPU 支持需要满足 NVIDIA 硬件需求以及在 GPU support TensorFlow 的 NVIDIA 需求中描述的软件。

Python 3.4

CPU only:

```
https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-1.1.0-cp34-cp34m-linux_x86_64.whl
```

GPU support:

```
https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow_gpu-1.1.0-cp34-cp34m-linux_x86_64.whl
```

注意 GPU 支持需要满足 NVIDIA 硬件需求以及在 GPU support TensorFlow 的 NVIDIA 需求中描述的软件。

Python 3.5

CPU only:

```
https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-
1.1.0-cp35-cp35m-linux_x86_64.whl
```

GPU support:

```
https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow_g
pu-1.1.0-cp35-cp35m-linux_x86_64.whl
```

注意 GPU 支持需要满足 NVIDIA 硬件需求以及在 GPU support TensorFlow 的 NVIDIA 需求中描述的软件。

Python 3.6

CPU only:

```
https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-
1.1.0-cp36-cp36m-linux_x86_64.whl
```

GPU support:

```
https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow_g
pu-1.1.0-cp36-cp36m-linux_x86_64.whl
```

注意 GPU 支持需要满足 NVIDIA 硬件需求以及在 GPU support TensorFlow 的 NVIDIA 需求中描述的软件。

Protobuf pip 包 3.1

如果没有遇到和 protobuf pip 包相关的问题，你可以跳过这个部分。

注意： 如果你的 TensorFlow 运行缓慢，你可能遇到了一个 protobuf pip 包相关的问题。

TensorFlow pip 包依赖于 protobuf pip 包 3.1。 从 PyPI 下载的 protobuf 是 proto 序列化/反序列化的纯 Python 实现的库， 其速度比 C++ 实现慢**10-50**倍。Protobuf 支持二进制扩展，速度更快，基于 C++。 但该扩展无法在纯Python实现的 pip 包中获取。我们制作了包含该二进制扩展的protobuf pip 包。 如下命令可安装该自定义的 protobuf pip 包：

- for Python 2.7:

```
$ pip install --upgrade \
https://storage.googleapis.com/tensorflow/linux/cpu/protobuf-
3.1.0-cp27-none-linux_x86_64.whl
```

- for Python 3.5:

```
$ pip3 install --upgrade \
https://storage.googleapis.com/tensorflow/linux/cpu/protobuf-
3.1.0-cp35-none-linux_x86_64.whl
```

安装这个 protobuf 包会覆盖已经存在的 protobuf 包。 注意该二进制 pip 包已经支持大于 64MB 的问题，修复了如下问题：

```
[libprotobuf ERROR
google/protobuf/src/google/protobuf/io/coded_stream.cc:207]
Protocol message 被拒绝，因为太大（大于 67108864 字节）。
为增大限制或禁用报警，
在 google/protobuf/io/coded_stream.h 中查看
CodedInputStream::SetTotalBytesLimit()
```

在 Windows 上安装 TensorFlow

这篇指南描述了如何在 Windows 上安装 TensorFlow。

确定 TensorFlow 版本

如下之中选择一种来安装:

- 只支持 **CPU** 的 **TensorFlow**。如果你的系统不支持 NVIDIA® GPU, 你必须安装这个版本。这个版本的 TensorFlow 通常安装起来比较简单（一般 5 到 10分钟），所以即使你拥有 NVIDIA GPU，我们也推荐首先安装这个版本。
- 支持 **GPU** 的 **TensorFlow**. TensorFlow 在 GPU 上通常比在 CPU 上的执行的更快。所以如果你有符合如下要求的 NVIDIA® GPU 并且需要注重性能，可以随后安装这个版本。

GPU support TensorFlow 的 NVIDIA 需求

需要事先安装如下软件：

- CUDA® Toolkit 8.0。详见 NVIDIA's documentation。确保按照文档中描述的将 Cuda 相关路径加入到 `%PATH%` 环境变量中。
- CUDA Toolkit 8.0 相关的 NVIDIA 驱动。
- cuDNN v5.1。详见 NVIDIA's documentation。注意：cuDNN 通常与其他 CUDA DLLs 安装的位置不同。确保将 cuDNN 库的安装目录加入到了 `%PATH%` 中。
- CUDA Compute Capability 3.0 或更高的 GPU 芯片。支持的 GPU 芯片详见 NVIDIA documentation 。

如果上述软件版本较老，请将其升级到指定版本。

确定如何安装 TensorFlow

有如下选择：

- "native" pip
- Anaconda

原生 pip 直接在系统中安装 TensorFlow，而不使用虚拟环境。 因为原生 pip 安装没有使用独立的容器隔离开，所以可能干扰其他基于Python的安装。 不过，如果你理解 pip 和 Python 环境，原生 pip 安装通常只需要一个命令！ 如果使用原生 pip 安装，用户可在任何目录中执行 TensorFlow 程序。

在 Anaconda 中，你可以通过 conda 创建一个虚拟环境。 然而，我们推荐使用 `pip install` 安装 TensorFlow，而非 `conda install` 。

注意：conda 包是社区支持而非官方支持。也就是说 TensorFlow 团队没有测试也没有管理过 conda 包。使用这个包需要自行承担风险。

原生 pip 安装

如果如下版本的 Python 没有安装，先安装：

- Python 3.5.x from python.org

TensorFlow 在 Windows 上支持 Python 3.5.x。注意 Python 3.5.x 使用 pip3，我们用 pip3 来安装 TensorFlow。

在 terminal 中输入如下命令安装只支持 CPU 的 TensorFlow：

```
C:\> pip3 install --upgrade tensorflow
```

安装支持 GPU 的 TensorFlow，使用如下命令：

```
C:\> pip3 install --upgrade tensorflow-gpu
```

Anaconda 安装

Anaconda 安装是社区支持，而非官方支持

按照如下步骤在 Anaconda 环境中安装 TensorFlow：

1. 按说明下载并安装 Anaconda： Anaconda download site
2. 建立一个 conda 环境，命名为 `tensorflow`，以便运行某个 Python 版本：

   ```
   C:\> conda create -n tensorflow
   ```

3. 激活 anaconda 环境：

   ```
   C:\> activate tensorflow
   (tensorflow)C:\>  # 你的提示符应该发生变化
   ```

4. 在你的 conda 环境中安装只支持 CPU 的 TensorFlow（写在一行）：

   ```
   (tensorflow)C:\> pip install --ignore-installed --upgrade
   https://storage.googleapis.com/tensorflow/windows/cpu/tensorf
   low-1.1.0-cp35-cp35m-win_amd64.whl
   ```

安装支持 GPU 的 TensorFlow（写在一行）：

```
(tensorflow)C:\> pip install --ignore-installed --upgrade
https://storage.googleapis.com/tensorflow/windows/gpu/tensorf
low_gpu-1.1.0-cp35-cp35m-win_amd64.whl
```

验证安装结果

启动 terminal。

如果通过 Anaconda 安装，激活 Anaconda 环境。

启动 Python：

```
$ python
```

在 Python 交互式环境中输入

```
>>> import tensorflow as tf
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()
>>> print(sess.run(hello))
```

如果系统输出如下，则安装成功：

```
Hello, TensorFlow!
```

如果你新接触 TensorFlow，参考初识 TensorFlow进行下一步学习。

如果系统输出错误信息而非欢迎信息，查看常见安装问题。

常见安装问题

我们依靠 Stack Overflow 来编写 TensorFlow 安装问题及解决方案的文档。 如下表格包含了 Stack Overflow 上比较常见的安装问题的连接。 如果你遇到了不在列表中的新的错误信息或者其他安装问题，请在 Stack Overflow 上搜索。 如果搜索不到，请在 Stack Overflow 上提出一个新的问题，并打上 `tensorflow` 的标签。

Stack Overflow Link	Error Message
41007279	`[...\stream_executor\dso_loader.cc] Couldn't open CUDA library nvcuda.dll`
41007279	`[...\stream_executor\cuda\cuda_dnn.cc] Unable to load cuDNN DSO`
42006320	`ImportError: Traceback (most recent call last): File "...\tensorflow\core\framework\graph_pb2.py", line 6, in from google.protobuf import descriptor as _descriptor ImportError: cannot import name 'descriptor'`
42011070	`No module named "pywrap_tensorflow"`
42217532	`OpKernel ('op: "BestSplits" device_type: "CPU"') for unknown op: BestSplits`
43134753	`The TensorFlow library wasn't compiled to use SSE instructions`

开始

查看如下指南可对 TensorFlow 程序有个简要的概览：

- 初识 TensorFlow

MINIST 是实验新的机器学习工具的经典数据集。我们提供了三篇指南，每篇介绍了一种不同的方法在 TensorFlow 上训练 MNIST 模型。

- MNIST-机器学习初学者，通过高级 API 介绍了 MNIST。
- MNIST-机器学习专家，比“MNIST-机器学习初学者”更加深入， 假设读者对机器学习的概念有所了解。
- TensorFlow 机制 101，通过底层 API 介绍了 MNIST。

对刚接触 TensorFlow 的开发者来说，可以从高级 API 开始。下面的指南可以帮助读者学习高级 API：

- tf.contrib.learn 快速开始，介绍该 API。
- 通过 tf.contrib.learn 构建输入函数，让你对该 API 更深入的使用。
- 通过 tf.contrib.learn 打日志并做监控，解释了如何观察(audit)训练速度。

TensorBoard 是可视化机器学习的各个方面的工具。如下指南描述了如何使用 TensorBoard。

- TensorBoard：可视化学习，让你开始。
- TensorBoard：Embedding 可视化，演示了如何查看高维数据并与之交互。
- TensorBoard：图可视化，演示了如何可视化计算图。图可视化一般对使用底层 API 的开发者更有用。

初识 TensorFlow

本指南让你开始进行 TensorFlow 编程。开始之前，安装 TensorFlow， 为了充分利用本指南，您应该了解以下内容：

- Python 如何编程。
- 至少对数组了解一点。
- 最好了解一些机器学习。然而，如果你对机器学习了解很少或不了解，你仍然应该首先阅读本指南。

TensorFlow 提供了多种API。最底层的 API--TensorFlow Core-- 可以让你完全控制自己的程序。 我们推荐机器学习研究人员和其他需要很好控制他们的模型的人员使用TensorFlow Core。 更高级别的API构建在 TensorFlow Core 之上。这些更高级的API通常比 TensorFlow Core 更容易学习和使用。此外，较高级别的 API 使重复任务更容易，不同用户之间更一致。像 tf.contrib.learn 这样的高级API可以帮助您管理数据集，进行估计，训练和推理。注意：一些高级 API（名称包含 `contrib` 的API）仍然在开发中。 有些 `contrib` 方法之后可能会改变或在随后的 TensorFlow 版本中废弃。

本指南从 TensorFlow Core 的教程开始。随后我们会演示如何通过 tf.contrib.learn 实现相同的模型。 了解 TensorFlow Core 的原理将你您提供一个 *mental model*，以便你在使用更紧凑更高级的 API 时理解内部是如何工作的

Tensors

The central unit of data in TensorFlow is the **tensor**. A tensor consists of a set of primitive values shaped into an array of any number of dimensions. A tensor's **rank** is its number of dimensions. Here are some examples of tensors:

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

TensorFlow Core tutorial

Importing TensorFlow

The canonical import statement for TensorFlow programs is as follows:

```
import tensorflow as tf
```

This gives Python access to all of TensorFlow's classes, methods, and symbols. Most of the documentation assumes you have already done this.

The Computational Graph

You might think of TensorFlow Core programs as consisting of two discrete sections:

1. Building the computational graph.
2. Running the computational graph.

A **computational graph** is a series of TensorFlow operations arranged into a graph of nodes. Let's build a simple computational graph. Each node takes zero or more tensors as inputs and produces a tensor as an output. One type of node is a constant. Like all TensorFlow constants, it takes no inputs, and it outputs a value it stores internally. We can create two floating point Tensors `node1` and `node2` as follows:

```
node1 = tf.constant(3.0, tf.float32)
node2 = tf.constant(4.0) # also tf.float32 implicitly
print(node1, node2)
```

The final print statement produces

```
Tensor("Const:0", shape=(), dtype=float32) Tensor("Const_1:0", shape=(), dtype=float32
)
```

Notice that printing the nodes does not output the values `3.0` and `4.0` as you might expect. Instead, they are nodes that, when evaluated, would produce 3.0 and 4.0, respectively. To actually evaluate the nodes, we must run the computational graph within a **session**. A session encapsulates the control and state of the TensorFlow runtime.

The following code creates a `Session` object and then invokes its `run` method to run enough of the computational graph to evaluate `node1` and `node2` . By running the computational graph in a session as follows:

```
sess = tf.Session()
print(sess.run([node1, node2]))
```

we see the expected values of 3.0 and 4.0:

```
[3.0, 4.0]
```

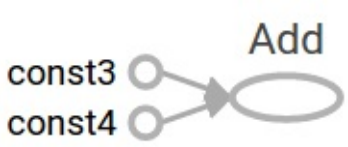
We can build more complicated computations by combining `Tensor` nodes with operations (Operations are also nodes.). For example, we can add our two constant nodes and produce a new graph as follows:

```
node3 = tf.add(node1, node2)
print("node3: ", node3)
print("sess.run(node3): ",sess.run(node3))
```

The last two print statements produce

```
node3:  Tensor("Add_2:0", shape=(), dtype=float32)
sess.run(node3):  7.0
```

TensorFlow provides a utility called TensorBoard that can display a picture of the computational graph. Here is a screenshot showing how TensorBoard visualizes the graph:



As it stands, this graph is not especially interesting because it always produces a constant result. A graph can be parameterized to accept external inputs, known as **placeholders**. A **placeholder** is a promise to provide a value later.

```
a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
adder_node = a + b  # + provides a shortcut for tf.add(a, b)
```

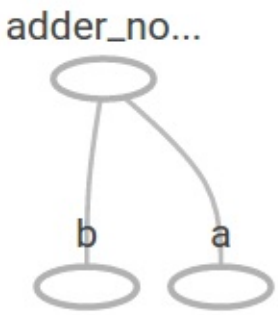
The preceding three lines are a bit like a function or a lambda in which we define two input parameters (a and b) and then an operation on them. We can evaluate this graph with multiple inputs by using the feed_dict parameter to specify Tensors that provide concrete values to these placeholders:

```
print(sess.run(adder_node, {a: 3, b:4.5}))
print(sess.run(adder_node, {a: [1,3], b: [2, 4]}))
```

resulting in the output

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

In TensorBoard, the graph looks like this:



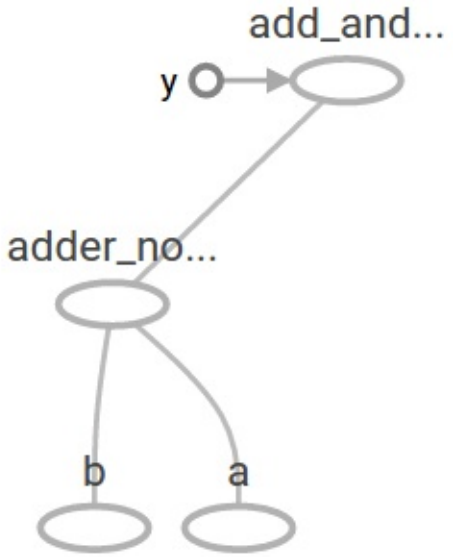
We can make the computational graph more complex by adding another operation. For example,

```
add_and_triple = adder_node * 3.
print(sess.run(add_and_triple, {a: 3, b:4.5}))
```

produces the output

```
22.5
```

The preceding computational graph would look as follows in TensorBoard:



In machine learning we will typically want a model that can take arbitrary inputs, such as the one above. To make the model trainable, we need to be able to modify the graph to get new outputs with the same input. **Variables** allow us to add trainable parameters to a graph. They are constructed with a type and initial value:

```
W = tf.Variable([.3], tf.float32)
b = tf.Variable([-.3], tf.float32)
x = tf.placeholder(tf.float32)
linear_model = W * x + b
```

Constants are initialized when you call `tf.constant`, and their value can never change. By contrast, variables are not initialized when you call `tf.Variable`. To initialize all the variables in a TensorFlow program, you must explicitly call a special operation as follows:

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

It is important to realize `init` is a handle to the TensorFlow sub-graph that initializes all the global variables. Until we call `sess.run`, the variables are uninitialized.

Since `x` is a placeholder, we can evaluate `linear_model` for several values of `x` simultaneously as follows:

```
print(sess.run(linear_model, {x:[1,2,3,4]}))
```

to produce the output

```
[ 0.          0.30000001  0.60000002  0.90000004]
```

We've created a model, but we don't know how good it is yet. To evaluate the model on training data, we need a `y` placeholder to provide the desired values, and we need to write a loss function.

A loss function measures how far apart the current model is from the provided data. We'll use a standard loss model for linear regression, which sums the squares of the deltas between the current model and the provided data. `linear_model - y` creates a vector where each element is the corresponding example's error delta. We call `tf.square` to square that error. Then, we sum all the squared errors to create a single scalar that abstracts the error of all examples using `tf.reduce_sum`:

```
y = tf.placeholder(tf.float32)
squared_deltas = tf.square(linear_model - y)
loss = tf.reduce_sum(squared_deltas)
print(sess.run(loss, {x:[1,2,3,4], y:[0,-1,-2,-3]}))
```

producing the loss value

```
23.66
```

We could improve this manually by reassigning the values of `W` and `b` to the perfect values of -1 and 1. A variable is initialized to the value provided to `tf.Variable` but can be changed using operations like `tf.assign`. For example, `W=-1` and `b=1` are the optimal parameters for our model. We can change `W` and `b` accordingly:

```
fixW = tf.assign(W, [-1.])
fixb = tf.assign(b, [1.])
sess.run([fixW, fixb])
print(sess.run(loss, {x:[1,2,3,4], y:[0,-1,-2,-3]}))
```

The final print shows the loss now is zero.

```
0.0
```

We guessed the "perfect" values of `W` and `b`, but the whole point of machine learning is to find the correct model parameters automatically. We will show how to accomplish this in the next section.

tf.train API

A complete discussion of machine learning is out of the scope of this tutorial. However, TensorFlow provides **optimizers** that slowly change each variable in order to minimize the loss function. The simplest optimizer is **gradient descent**. It modifies each variable according to the magnitude of the derivative of loss with respect to that variable. In general, computing symbolic derivatives manually is tedious and error-prone. Consequently, TensorFlow can automatically produce derivatives given only a description of the model using the function `tf.gradients`. For simplicity, optimizers typically do this for you. For example,

```
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
```

```
sess.run(init) # reset values to incorrect defaults.
for i in range(1000):
  sess.run(train, {x:[1,2,3,4], y:[0,-1,-2,-3]})

print(sess.run([W, b]))
```

results in the final model parameters:

```
[array([-0.9999969], dtype=float32), array([ 0.99999082],
 dtype=float32)]
```

Now we have done actual machine learning! Although doing this simple linear regression doesn't require much TensorFlow core code, more complicated models and methods to feed data into your model necessitate more code. Thus TensorFlow provides higher level abstractions for common patterns, structures, and functionality. We will learn how to use some of these abstractions in the next section.

Complete program

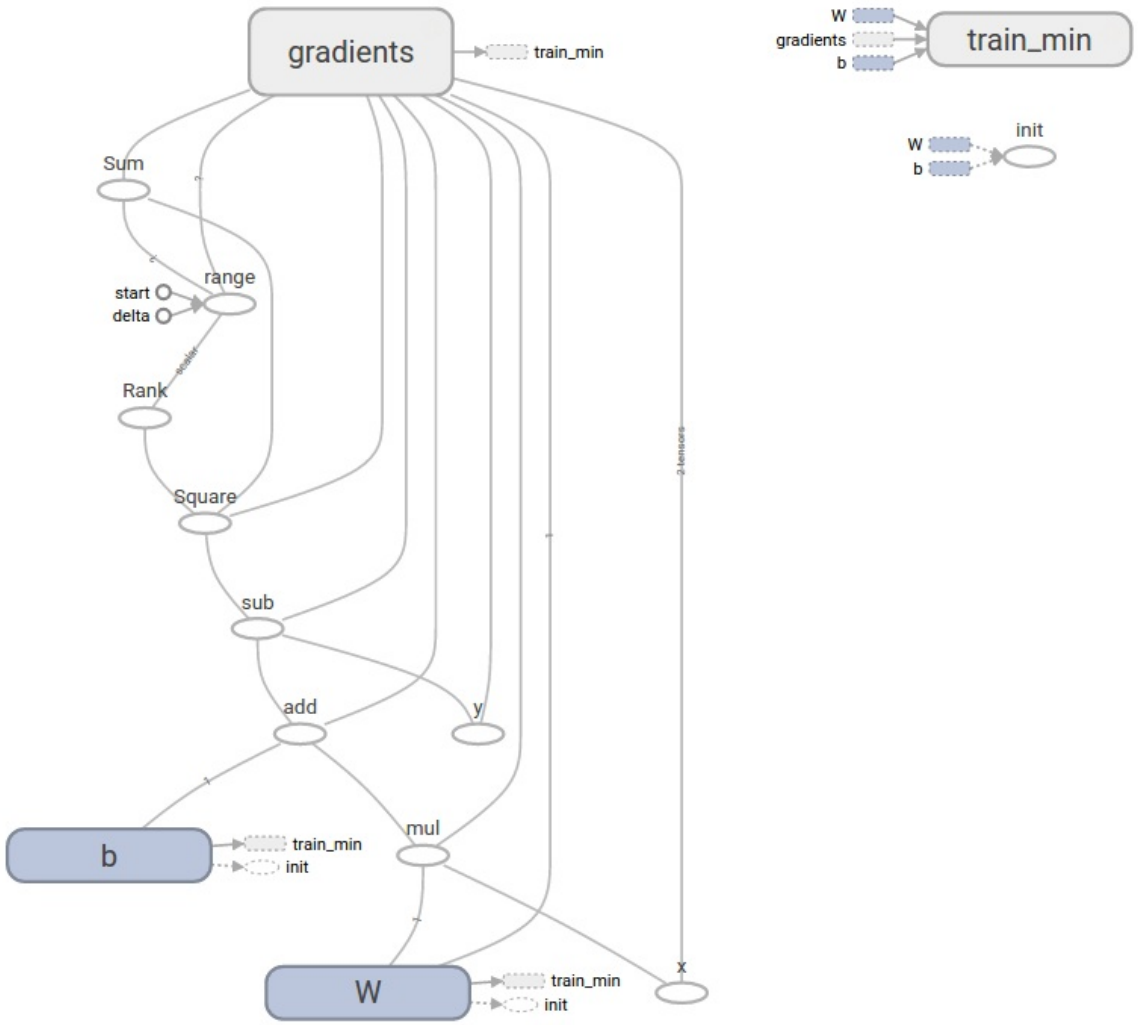
The completed trainable linear regression model is shown here:

```
import numpy as np
import tensorflow as tf

# Model parameters
W = tf.Variable([.3], tf.float32)
b = tf.Variable([-.3], tf.float32)
# Model input and output
x = tf.placeholder(tf.float32)
linear_model = W * x + b
y = tf.placeholder(tf.float32)
# loss
loss = tf.reduce_sum(tf.square(linear_model - y)) # sum of the squares
# optimizer
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
# training data
x_train = [1,2,3,4]
y_train = [0,-1,-2,-3]
# training loop
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init) # reset values to wrong
for i in range(1000):
  sess.run(train, {x:x_train, y:y_train})

# evaluate training accuracy
curr_W, curr_b, curr_loss  = sess.run([W, b, loss], {x:x_train, y:y_train})
print("W: %s b: %s loss: %s"%(curr_W, curr_b, curr_loss))
```

When run, it produces

```
W: [-0.9999969] b: [ 0.99999082] loss: 5.69997e-11
```

This more complicated program can still be visualized in TensorBoard



`tf.contrib.learn`

`tf.contrib.learn` is a high-level TensorFlow library that simplifies the mechanics of machine learning, including the following:

- running training loops
- running evaluation loops
- managing data sets
- managing feeding

tf.contrib.learn defines many common models.

Basic usage

Notice how much simpler the linear regression program becomes with `tf.contrib.learn`:

```
import tensorflow as tf
# NumPy is often used to load, manipulate and preprocess data.
import numpy as np

# Declare list of features. We only have one real-valued feature. There are many
# other types of columns that are more complicated and useful.
features = [tf.contrib.layers.real_valued_column("x", dimension=1)]

# An estimator is the front end to invoke training (fitting) and evaluation
# (inference). There are many predefined types like linear regression,
# logistic regression, linear classification, logistic classification, and
# many neural network classifiers and regressors. The following code
# provides an estimator that does linear regression.
estimator = tf.contrib.learn.LinearRegressor(feature_columns=features)

# TensorFlow provides many helper methods to read and set up data sets.
# Here we use `numpy_input_fn`. We have to tell the function how many batches
# of data (num_epochs) we want and how big each batch should be.
x = np.array([1., 2., 3., 4.])
y = np.array([0., -1., -2., -3.])
input_fn = tf.contrib.learn.io.numpy_input_fn({"x":x}, y, batch_size=4,
                                              num_epochs=1000)

# We can invoke 1000 training steps by invoking the `fit` method and passing the
# training data set.
estimator.fit(input_fn=input_fn, steps=1000)

# Here we evaluate how well our model did. In a real example, we would want
# to use a separate validation and testing data set to avoid overfitting.
print(estimator.evaluate(input_fn=input_fn))
```

When run, it produces

```
    {'global_step': 1000, 'loss': 1.9650059e-11}
```

A custom model

`tf.contrib.learn` does not lock you into its predefined models. Suppose we wanted to create a custom model that is not built into TensorFlow. We can still retain the high level abstraction of data set, feeding, training, etc. of `tf.contrib.learn`. For illustration, we will show how to implement our own equivalent model to `LinearRegressor` using our knowledge of the lower level TensorFlow API.

To define a custom model that works with `tf.contrib.learn`, we need to use `tf.contrib.learn.Estimator`. `tf.contrib.learn.LinearRegressor` is actually a sub-class of `tf.contrib.learn.Estimator`. Instead of sub-classing `Estimator`, we simply provide `Estimator` a function `model_fn` that tells `tf.contrib.learn` how it can evaluate predictions, training steps, and loss. The code is as follows:

```
import numpy as np
import tensorflow as tf
# Declare list of features, we only have one real-valued feature
def model(features, labels, mode):
  # Build a linear model and predict values
  W = tf.get_variable("W", [1], dtype=tf.float64)
  b = tf.get_variable("b", [1], dtype=tf.float64)
  y = W*features['x'] + b
  # Loss sub-graph
  loss = tf.reduce_sum(tf.square(y - labels))
  # Training sub-graph
  global_step = tf.train.get_global_step()
  optimizer = tf.train.GradientDescentOptimizer(0.01)
  train = tf.group(optimizer.minimize(loss),
                   tf.assign_add(global_step, 1))
  # ModelFnOps connects subgraphs we built to the
  # appropriate functionality.
  return tf.contrib.learn.ModelFnOps(
      mode=mode, predictions=y,
      loss=loss,
      train_op=train)

estimator = tf.contrib.learn.Estimator(model_fn=model)
# define our data set
x = np.array([1., 2., 3., 4.])
y = np.array([0., -1., -2., -3.])
input_fn = tf.contrib.learn.io.numpy_input_fn({"x": x}, y, 4, num_epochs=1000)

# train
estimator.fit(input_fn=input_fn, steps=1000)
# evaluate our model
print(estimator.evaluate(input_fn=input_fn, steps=10))
```

When run, it produces

```
{'loss': 5.9819476e-11, 'global_step': 1000}
```

Notice how the contents of the custom `model()` function are very similar to our manual model training loop from the lower level API.

Next steps

Now you have a working knowledge of the basics of TensorFlow. We have several more tutorials that you can look at to learn more. If you are a beginner in machine learning see @{$beginners$MNIST for beginners}, otherwise see @{$pros$Deep MNIST for experts}.

Programmer's Guide

The documents in this unit dive into the details of writing TensorFlow code. This section begins with the following guides, each of which explain a particular aspect of TensorFlow:

- @{$variables$Variables: Creation, Initialization, Saving, and Loading}, which details the mechanics of TensorFlow Variables.
- @{$dims_types$Tensor Ranks, Shapes, and Types}, which explains Tensor rank (the number of dimensions), shape (the size of each dimension), and datatypes.
- @{$variable_scope$Sharing Variables}, which explains how to share and manage large sets of variables when building complex models.
- @{$threading_and_queues$Threading and Queues}, which explains TensorFlow's rich queuing system.
- @{$reading_data$Reading Data}, which documents three different mechanisms for getting data into a TensorFlow program.

The following guide is helpful when training a complex model over multiple days:

- @{$supervisor$Supervisor: Training Helper for Days-Long Trainings}, which explains how to gracefully handle system crashes during a lengthy training session.

TensorFlow provides a debugger named `tfdbg`, which is documented in the following two guides:

- @{$debugger$TensorFlow Debugger (tfdbg) Command-Line-Interface Tutorial: MNIST}, which walks you through the use of `tfdbg` within an application written in the low-level TensorFlow API.
- @{$tfdbg-tflearn$How to Use TensorFlow Debugger (tfdbg) with tf.contrib.learn}, which demonstrates how to use `tfdbg` within the Estimators API.

A `MetaGraph` consists of both a computational graph and its associated metadata. A `MetaGraph` contains the information required to continue training, perform evaluation, or run inference on a previously trained graph. The following guide details `MetaGraph` objects:

- @{$meta_graph$Exporting and Importing a MetaGraph}.

To learn about the TensorFlow versioning scheme, consult the following two guides:

- @{$version_semantics$TensorFlow Version Semantics}, which explains TensorFlow's versioning nomenclature and compatibility rules.
- @{$data_versions$TensorFlow Data Versioning: GraphDefs and Checkpoints}, which explains how TensorFlow adds versioning information to computational graphs and checkpoints in order to support compatibility across versions.

We conclude this section with a FAQ about TensorFlow programming:

- @{faqFrequently Asked Questions}

Tutorials

This section contains tutorials demonstrating how to do specific tasks in TensorFlow. If you are new to TensorFlow, we recommend reading the documents in the "Get Started" section before reading these tutorials.

The following tutorial explains the interaction of CPUs and GPUs on a TensorFlow system:

- @{$using_gpu$Using GPUs}

The following tutorials cover different aspects of image recognition:

- @{$image_recognition$Image Recognition}, which introduces the field of image recognition and a model (Inception) for recognizing images.
- @{$image_retraining$How to Retrain Inception's Final Layer for New Categories}, which has a wonderfully self-explanatory title.
- @{$layers$A Guide to TF Layers: Building a Convolutional Neural Network}, which introduces convolutional neural networks (CNNs) and demonstrates how to build a CNN in TensorFlow.
- @{$deep_cnn$Convolutional Neural Networks}, which demonstrates how to build a small CNN for recognizing images. This tutorial is aimed at advanced TensorFlow users.

The following tutorials focus on machine learning problems in human language:

- @{$word2vec$Vector Representations of Words}, which demonstrates how to create an embedding for words.
- @{$recurrent$Recurrent Neural Networks}, which demonstrates how to use a recurrent neural network to predict the next word in a sentence.
- @{$seq2seq$Sequence-to-Sequence Models}, which demonstrates how to use a sequence-to-sequence model to translate text from English to French.

The following tutorials focus on linear models:

- @{$linear$Large-Scale Linear Models with TensorFlow}, which introduces linear models and demonstrates how to build them with the high-level API.
- @{$wide$TensorFlow Linear Model Tutorial}, which demonstrates how to solve a binary classification problem in TensorFlow.
- @{$wide_and_deep$TensorFlow Wide & Deep Learning Tutorial}, which explains how to use the high-level API to jointly train both a wide linear model and a deep feed-forward neural network.

Although TensorFlow specializes in machine learning, you may also use TensorFlow to solve other kinds of math problems. For example:

- @{$mandelbrot$Mandelbrot Set}

- @{$pdes$Partial Differential Equations}

Performance

Performance is often a significant issue when training a machine learning model. This section explains various ways to optimize performance. Start your investigation with the following guide:

- @{$performance_guide$Performance}, which contains a collection of best practices for optimizing your TensorFlow code.

XLA (Accelerated Linear Algebra) is an experimental compiler for linear algebra that optimizes TensorFlow computations. The following guides explore XLA:

- @{xlaXLA Overview}, which introduces XLA.
- @{$broadcasting$Broadcasting Semantics}, which describes XLA's broadcasting semantics.
- @{$developing_new_backend$Developing a new back end for XLA}, which explains how to re-target TensorFlow in order to optimize the performance of the computational graph for particular hardware.
- @{jitUsing JIT Compilation}, which describes the XLA JIT compiler that compiles and runs parts of TensorFlow graphs via XLA in order to optimize performance.
- @{$operation_semantics$Operation Semantics}, which is a reference manual describing the semantics of operations in the `ComputationBuilder` interface.
- @{$shapes$Shapes and Layout}, which details the `Shape` protocol buffer.
- @{$tfcompile$Using AOT compilation}, which explains `tfcompile`, a standalone tool that compiles TensorFlow graphs into executable code in order to optimize performance.

And finally, we offer the following guide:

- @{$quantization$How to Quantize Neural Networks with TensorFlow}, which can explains how to use quantization to reduce model size, both in storage and at runtime. Quantization can improve performance, especially on mobile hardware.

Deploy

This section focuses on deploying real-world models. It contains the following documents:

- @{$distributed$Distributed TensorFlow}, which explains how to create a cluster of TensorFlow servers.
- @{$tfserve$TensorFlow Serving}, which describes TensorFlow Serving--an open-source serving system for machine learning models. This document provides a short introduction to TensorFlow Serving; the bulk of the documentation about TensorFlow Serving is in a separate website.
- @{$hadoop$How to run TensorFlow on Hadoop}, which has a highly self-explanatory title.

Extend

This section explains how developers can add functionality to TensorFlow's capabilities. Begin by reading the following architectural overview:

- @{$architecture$TensorFlow Architecture}

The following guides explain how to extend particular aspects of TensorFlow:

- @{$adding_an_op$Adding a New Op}, which explains how to create your own operations.
- @{$add_filesys$Adding a Custom Filesystem Plugin}, which explains how to add support for your own shared or distributed filesystem.
- @{$new_data_formats$Custom Data Readers}, which details how to add support for your own file and record formats.
- @{$estimators$Creating Estimators in tf.contrib.learn}, which explains how to write your own custom Estimator. For example, you could build your own Estimator to implement some variation on standard linear regression.

Python is currently the only language supported by TensorFlow's API stability promises. However, TensorFlow also provides functionality in C++, Java, and Go, plus community support for Haskell and Rust. If you'd like to create or develop TensorFlow features in a language other than these languages, read the following guide:

- @{$language_bindings$TensorFlow in Other Languages}

To create tools compatible with TensorFlow's model format, read the following guide:

- @{$tool_developers$A Tool Developer's Guide to TensorFlow Model Files}