以下资料来自英文网站

<https://tensorflow.google.cn/extend/>

是机器翻译的结果，请对照英文资料，学习并运行所有代码，修改文字使之通顺流畅。

（使用Word的‘审阅’修订功能进行修改，保留所有修改痕迹）

TensorFlow架构

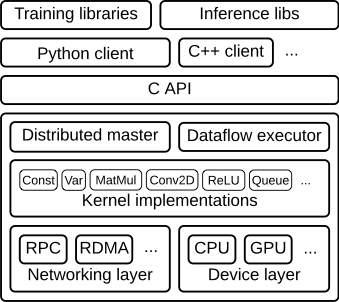
我们为大规模分布式培训和推理设计了TensorFlow，但它也足够灵活，可以支持新机器学习模型和系统级优化的实验。

本文档描述了可以实现规模和灵活性组合的系统架构。它假定您基本熟悉TensorFlow编程概念，例如计算图，操作和会话。有关这些主题的介绍，请参阅[此文档](https://tensorflow.google.cn/guide/low_level_intro)。熟悉[分布式TensorFlow](https://tensorflow.google.cn/deploy/distributed)也很有帮助。

本文档适用于希望以当前API不支持的某种方式扩展TensorFlow的开发人员，希望针对TensorFlow进行优化的硬件工程师，负责扩展和分发的机器学习系统的实施者，或任何想要了解Tensorflow的人员。在本文档的最后，您应该了解TensorFlow架构，以便读取和修改核心TensorFlow代码。

概观

TensorFlow运行时是一个跨平台库。图1说明了它的一般架构。AC API将不同语言的用户级代码与核心运行时分开。

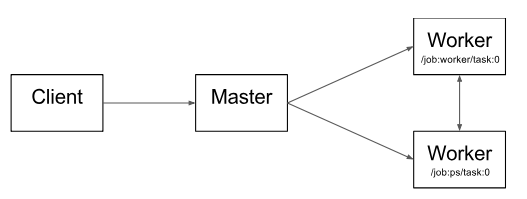


**图1**

本文档重点介绍以下几个方面：

* **客户**：
  + 将计算定义为数据流图。
  + 使用[**会话**](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/python/client/session.py)启动图执行。
* **分布式大师**
  + 修剪图中的特定子图，由Session.run（）的参数定义。
  + 将子图分成多个部分，这些部分在不同的进程和设备中运行。
  + 将图形片段分发给工作服务。
  + 通过工作服务启动图形块执行。
* **工作人员服务**（每个任务一个）
  + 使用适合于可用硬件（CPU，GPU等）的内核实现来安排图形操作的执行。
  + 向其他工作服务发送和接收操作结果。
* **内核实现**
  + 执行单个图形操作的计算。

图2说明了这些组件的交互。“/ job：worker / task：0”和“/ job：ps / task：0”都是具有工作服务的任务。“PS”代表“参数服务器”：负责存储和更新模型参数的任务。其他任务在优化参数时会对这些参数发送更新。任务之间的这种特殊分工不是必需的，但对于分布式培训来说是常见的。



**图2**

请注意，Distributed Master和Worker Service仅存在于分布式TensorFlow中。TensorFlow的单进程版本包含一个特殊的Session实现，它可以执行分布式主服务器执行的所有操作，但只与本地进程中的设备进行通信。

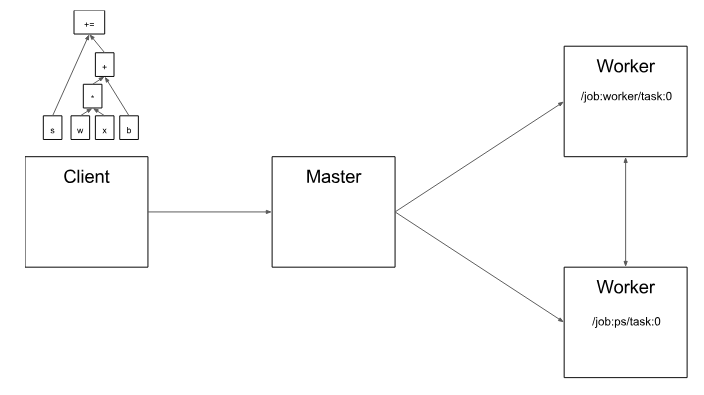
以下部分更详细地描述了核心TensorFlow层，并逐步处理示例图。

客户

用户编写构建计算图的客户端TensorFlow程序。该程序可以直接组成单独的操作，也可以使用Estimators API之类的便利库来组合神经网络层和其他更高级别的抽象。TensorFlow支持多种客户端语言，我们优先考虑Python和C ++，因为我们的内部用户最熟悉这些语言。随着功能变得更加成熟，我们通常将它们移植到C ++，以便用户可以从所有客户端语言访问优化的实现。大多数训练库仍然只支持Python，但C ++确实支持有效的推理。

客户端创建会话，该会话将图形定义作为[tf.GraphDef](https://tensorflow.google.cn/api_docs/python/tf/GraphDef) 协议缓冲区发送到分布式主节点。当客户端评估图中的一个或多个节点时，评估会触发对分布式主节点的调用以启动计算。

在图3中，客户端构建了一个图表，将权重（w）应用于特征向量（x），添加偏差项（b）并将结果保存在变量中。



**图3**

码

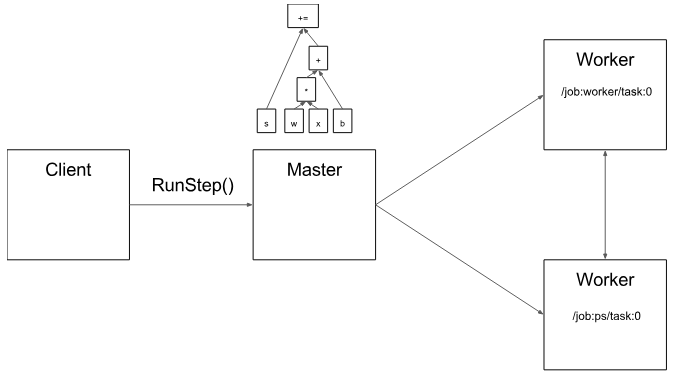
* [tf.Session](https://tensorflow.google.cn/api_docs/python/tf/Session)

分布式主人

分布式主机：

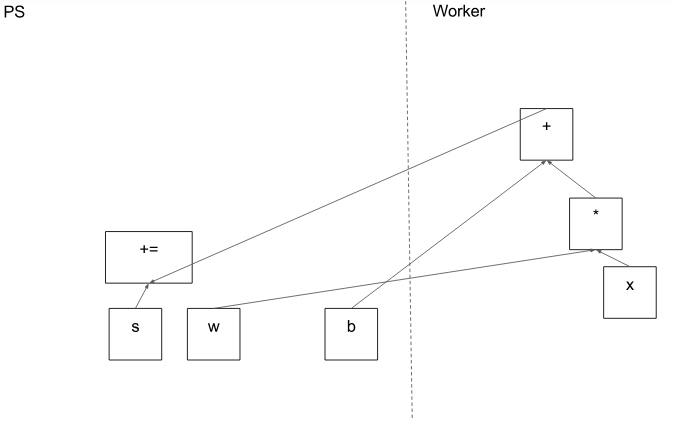
* 修剪图形以获得评估客户端请求的节点所需的子图，
* 对图表进行分区以获取每个参与设备的图形片段，以及
* 缓存这些部分，以便它们可以在后续步骤中重复使用。

由于主设备可以看到步骤的整体计算，因此它应用标准优化，例如公共子表达式消除和常量折叠。然后，它协调一组任务中优化子图的执行。



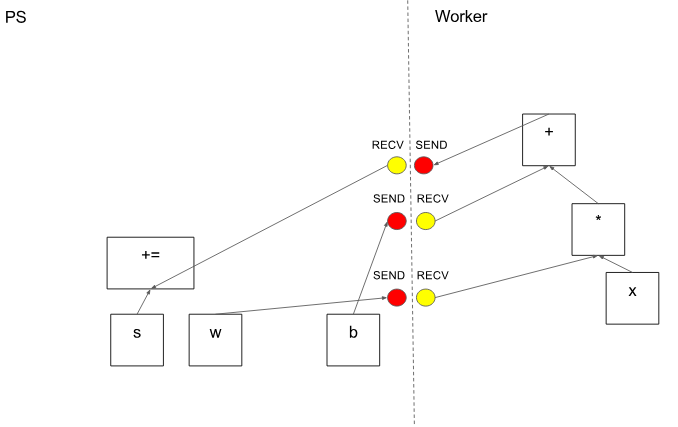
**图4**

图5显示了示例图的可能分区。分布式主服务器已对模型参数进行分组，以便将它们放在参数服务器上。



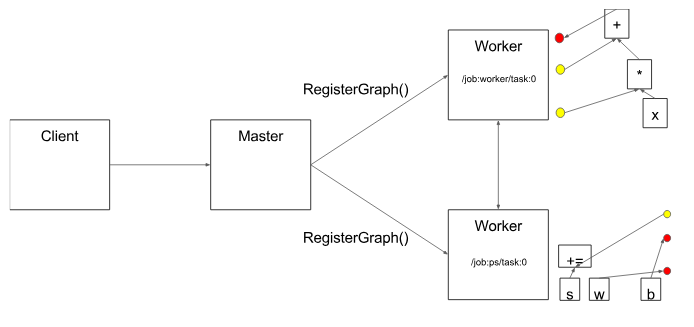
**图5**

在分区切割图形边缘的情况下，分布式主控插入发送和接收节点以在分布式任务之间传递信息（图6）。



**图6**

然后，分布式主服务器将图形片段传送到分布式任务。



**图7**

码

* [MasterService API定义](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/protobuf/master_service.proto)
* [主界面](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/distributed_runtime/master_interface.h)

工人服务

每项任务中的工作服务：

* 处理来自主人的请求，
* 为包含本地子图的操作安排内核的执行，和
* 调解任务之间的直接沟通。

我们优化工作服务，以便以较低的开销运行大型图形。我们当前的实现可以每秒执行数万个子图，这使得大量复制副本可以进行快速，细粒度的训练步骤。工作服务将内核分派给本地设备并在可能的情况下并行运行内核，例如通过使用多个CPU内核或GPU流。

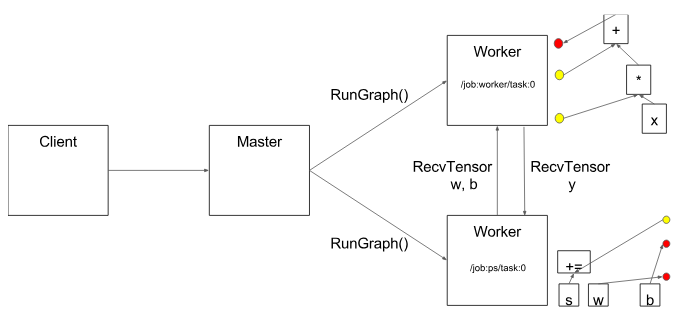
我们专门针对每对源设备和目标设备类型的Send和Recv操作：

* 本地CPU和GPU设备之间的传输使用 cudaMemcpyAsync()API来重叠计算和数据传输。
* 两个本地GPU之间的传输使用对等DMA，以避免通过主机CPU进行昂贵的复制。

对于任务之间的传输，TensorFlow使用多种协议，包括：

* gRPC over TCP。
* RDMA over Converged Ethernet。

我们还初步支持NVIDIA的NCCL多GPU通信库（参见 [tf.contrib.nccl](https://tensorflow.google.cn/api_docs/python/tf/contrib/nccl)）。



**图8**

码

* [WorkerService API定义](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/protobuf/worker_service.proto)
* [工人界面](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/distributed_runtime/worker_interface.h)
* [远程集合（用于Send和Recv实现）](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/distributed_runtime/rpc/rpc_rendezvous_mgr.h)

内核实现

运行时包含200多个标准操作，包括数学，数组操作，控制流和状态管理操作。这些操作中的每一个都可以具有针对各种设备优化的内核实现。许多操作内核都是使用Eigen :: Tensor实现的，它使用C ++模板为多核CPU和GPU生成高效的并行代码; 但是，我们自由地使用像cuDNN这样的库，可以实现更高效的内核实现。我们还实现了 [量化](https://tensorflow.google.cn/performance/quantization)，可以在移动设备和高吞吐量数据中心应用等环境中实现更快的推理，并使用 [gemmlowp](https://github.com/google/gemmlowp)低精度矩阵库来加速量化计算。

如果将子计算表示为操作组合是困难或低效的，则用户可以注册提供用C ++编写的有效实现的其他内核。例如，我们建议为一些性能关键操作注册自己的融合内核，例如ReLU和Sigmoid激活函数及其相应的渐变。该[XLA编译器](https://tensorflow.google.cn/performance/xla/index)有一个实验实现自动内核融合。

码

* [OpKernel 接口](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/op_kernel.h)

添加新的操作

**注意：**默认情况下，[www.tensorflow.org](https://tensorflow.google.cn/)显示最新稳定版本的文档。本文档中的说明需要从源代码构建。您可能希望从**master**tensorflow 的版本构建。因此，您应确保遵循[**master**此文档](https://tensorflow.google.cn/versions/master/extend/adding_an_op)的 [版本](https://tensorflow.google.cn/versions/master/extend/adding_an_op)，以防发生任何更改。

如果您想创建一个现有TensorFlow库未涵盖的操作，我们建议您首先尝试将Python中的操作编写为现有Python操作或函数的组合。如果无法做到这一点，您可以创建自定义C ++操作。您可能希望创建自定义C ++操作有几个原因：

* 将您的操作表达为现有操作的组合并不容易或不可能。
* 将您的操作表达为现有基元的组合并不高效。
* 您希望手工融合未来编译器难以融合的基元组合。

例如，想象一下你想要实现像“中位池”这样的东西，类似于“MaxPool”运算符，但是计算中位数而非滑动窗口而不是最大值。可以使用操作组合来执行此操作（例如，使用ExtractImagePatches和TopK），但可能不像本机操作那样具有性能或内存效率，您可以在单个融合操作中执行更聪明的操作。与往常一样，通常首先要尝试使用运算符组合来表达您想要的内容，只有在证明难以实现或效率低下时才选择添加新操作。

要合并您的自定义操作，您需要：

1. 在C ++文件中注册新的op。操作注册定义了操作功能的接口（规范），它独立于操作的实现。例如，op注册定义了op的名称和op的输入和输出。它还定义了用于张量形状推断的形状函数。
2. 用C ++实现op。op的实现称为内核，它是您在步骤1中注册的规范的具体实现。可以有多个内核用于不同的输入/输出类型或体系结构（例如，CPU，GPU）。
3. 创建一个Python包装器（可选）。这个包装器是用于在Python中创建op的公共API。从op注册生成默认包装器，可以直接使用或添加到。
4. 编写一个函数来计算op的梯度（可选）。
5. 测试操作。为方便起见，我们通常在Python中执行此操作，但您也可以在C ++中测试op。如果定义渐变，则可以使用Python验证它们[tf.test.compute\_gradient\_error](https://tensorflow.google.cn/api_docs/python/tf/test/compute_gradient_error)。请参阅 [relu\_op\_test.py](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/python/kernel_tests/relu_op_test.py)测试Relu-like运算符及其渐变的正向函数的示例。

先决条件：

* 熟悉C ++。
* 必须已安装 [TensorFlow二进制文件](https://tensorflow.google.cn/install)，或者必须已 [下载TensorFlow源](https://tensorflow.google.cn/install/source)，并且能够构建它。

定义op的界面

您可以通过将其注册到TensorFlow系统来定义操作的接口。在注册中，您可以指定op的名称，输入（类型和名称）和输出（类型和名称），以及文档字符串和操作可能需要的任何[attrs](https://tensorflow.google.cn/extend/adding_an_op#attrs)。

为了了解它是如何工作的，假设您想要创建一个采用张量int32s 的op 并输出张量的 副本，除了第一个元素设置为零之外的所有元素。为此，请创建一个名为的文件zero\_out.cc。然后添加一个调用 REGISTER\_OP宏来定义op的接口：

#include "tensorflow/core/framework/op.h"  
#include "tensorflow/core/framework/shape\_inference.h"  
  
using namespace tensorflow;  
  
REGISTER\_OP("ZeroOut")  
    .Input("to\_zero: int32")  
    .Output("zeroed: int32")  
    .SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {  
      c->set\_output(0, c->input(0));  
      return Status::OK();  
    });

该ZeroOut操作采用一个to\_zero32位整数的张量作为输入，并输出一个zeroed32位整数的张量。op还使用形状函数来确保输出张量与输入张量的形状相同。例如，如果输入是形状的张量[10,20]，则此形状函数指定输出形状也是[10,20]。

关于命名的注释：操作名称必须在CamelCase中，并且在二进制文件中注册的所有其他操作中必须是唯一的。

为op实现内核

定义接口后，提供op的一个或多个实现。要创建其中一个内核，请创建一个扩展OpKernel和覆盖该Compute方法的类。该Compute方法提供了一个context 类型的参数OpKernelContext\*，您可以从中访问输入和输出张量等有用的东西。

将您的内核添加到您在上面创建的文件中。内核可能看起来像这样：

#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\_flat = output\_tensor->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\_flat(i) = 0;  
    }  
  
    // Preserve the first input value if possible.  
    if (N > 0) output\_flat(0) = input(0);  
  }  
};

实现内核后，将其注册到TensorFlow系统。在注册中，您指定运行此内核的不同约束。例如，您可能有一个内核用于CPU，另一个内核用于GPU。

要对ZeroOutop 执行此操作，请将以下内容添加到zero\_out.cc：

REGISTER\_KERNEL\_BUILDER(Name("ZeroOut").Device(DEVICE\_CPU), ZeroOutOp);

**重要提示：**可以同时访问OpKernel的实例。您的**Compute**方法必须是线程安全的。使用互斥锁保护对类成员的任何访问权限。或者更好的是，不要通过班级成员分享国家！考虑使用a [**ResourceMgr**](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/resource_mgr.h) 来跟踪op状态。

多线程CPU内核

要编写多线程CPU内核，[work\_sharder.h](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/util/work_sharder.h) 可以使用Shard函数 。此函数在配置为用于操作内线程的线程之间分割计算函数（请参阅intra\_op\_parallelism\_threads in [config.proto](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/protobuf/config.proto)）。

GPU内核

GPU内核分为两部分实现：OpKernel和CUDA内核及其启动代码。

有时，OpKernel实现在CPU和GPU内核之间很常见，例如检查输入和分配输出。在这种情况下，建议的实施是：

1. 定义在Device上模板化的OpKernel和张量的基元类型。
2. 要进行输出的实际计算，Compute函数调用模板化的functor结构。
3. CPUDevice的该仿函数的专门化在同一文件中定义，但GPUDevice的特化在.cu.cc文件中定义，因为它将使用CUDA编译器进行编译。

这是一个示例实现。

// kernel\_example.h  
#ifndef KERNEL\_EXAMPLE\_H\_  
#define KERNEL\_EXAMPLE\_H\_  
  
template <typename Device, typename T>  
struct ExampleFunctor {  
  void operator()(const Device& d, int size, const T\* in, T\* out);  
};  
  
#if GOOGLE\_CUDA  
// Partially specialize functor for GpuDevice.  
template <typename Eigen::GpuDevice, typename T>  
struct ExampleFunctor {  
  void operator()(const Eigen::GpuDevice& d, int size, const T\* in, T\* out);  
};  
#endif  
  
#endif KERNEL\_EXAMPLE\_H\_

// kernel\_example.cc  
#include "example.h"  
#include "tensorflow/core/framework/op\_kernel.h"  
  
using namespace tensorflow;  
  
using CPUDevice = Eigen::ThreadPoolDevice;  
using GPUDevice = Eigen::GpuDevice;  
  
// CPU specialization of actual computation.  
template <typename T>  
struct ExampleFunctor<CPUDevice, T> {  
  void operator()(const CPUDevice& d, int size, const T\* in, T\* out) {  
    for (int i = 0; i < size; ++i) {  
      out[i] = 2 \* in[i];  
    }  
  }  
};  
  
// OpKernel definition.  
// template parameter <T> is the datatype of the tensors.  
template <typename Device, typename T>  
class ExampleOp : public OpKernel {  
 public:  
  explicit ExampleOp(OpKernelConstruction\* context) : OpKernel(context) {}  
  
  void Compute(OpKernelContext\* context) override {  
    // Grab the input tensor  
    const Tensor& input\_tensor = context->input(0);  
  
    // Create an output tensor  
    Tensor\* output\_tensor = NULL;  
    OP\_REQUIRES\_OK(context, context->allocate\_output(0, input\_tensor.shape(),  
                                                     &output\_tensor));  
  
    // Do the computation.  
    OP\_REQUIRES(context, input\_tensor.NumElements() <= tensorflow::kint32max,  
                errors::InvalidArgument("Too many elements in tensor"));  
    ExampleFunctor<Device, T>()(  
        context->eigen\_device<Device>(),  
        static\_cast<int>(input\_tensor.NumElements()),  
        input\_tensor.flat<T>().data(),  
        output\_tensor->flat<T>().data());  
  }  
};  
  
// Register the CPU kernels.  
#define REGISTER\_CPU(T)                                          \  
  REGISTER\_KERNEL\_BUILDER(                                       \  
      Name("Example").Device(DEVICE\_CPU).TypeConstraint<T>("T"), \  
      ExampleOp<CPUDevice, T>);  
REGISTER\_CPU(float);  
REGISTER\_CPU(int32);  
  
// Register the GPU kernels.  
#ifdef GOOGLE\_CUDA  
#define REGISTER\_GPU(T)                                          \  
  /\* Declare explicit instantiations in kernel\_example.cu.cc. \*/ \  
  extern template ExampleFunctor<GPUDevice, T>;                  \  
  REGISTER\_KERNEL\_BUILDER(                                       \  
      Name("Example").Device(DEVICE\_GPU).TypeConstraint<T>("T"), \  
      ExampleOp<GPUDevice, T>);  
REGISTER\_GPU(float);  
REGISTER\_GPU(int32);  
#endif  // GOOGLE\_CUDA

// kernel\_example.cu.cc  
#ifdef GOOGLE\_CUDA  
#define EIGEN\_USE\_GPU  
#include "example.h"  
#include "tensorflow/core/util/cuda\_kernel\_helper.h"  
  
using namespace tensorflow;  
  
using GPUDevice = Eigen::GpuDevice;  
  
// Define the CUDA kernel.  
template <typename T>  
\_\_global\_\_ void ExampleCudaKernel(const int size, const T\* in, T\* out) {  
  for (int i = blockIdx.x \* blockDim.x + threadIdx.x; i < size;  
       i += blockDim.x \* gridDim.x) {  
    out[i] = 2 \* ldg(in + i);  
  }  
}  
  
// Define the GPU implementation that launches the CUDA kernel.  
template <typename T>  
void ExampleFunctor<GPUDevice, T>::operator()(  
    const GPUDevice& d, int size, const T\* in, T\* out) {  
  // Launch the cuda kernel.  
  //  
  // See core/util/cuda\_kernel\_helper.h for example of computing  
  // block count and thread\_per\_block count.  
  int block\_count = 1024;  
  int thread\_per\_block = 20;  
  ExampleCudaKernel<T>  
      <<<block\_count, thread\_per\_block, 0, d.stream()>>>(size, in, out);  
}  
  
// Explicitly instantiate functors for the types of OpKernels registered.  
template struct ExampleFunctor<GPUDevice, float>;  
template struct ExampleFunctor<GPUDevice, int32>;  
  
#endif  // GOOGLE\_CUDA

构建op库

使用系统编译器编译op（TensorFlow二进制安装）

你应该能够编译zero\_out.cc一个C++编译器，如g++ 或clang您的系统上。二进制PIP包安装了在特定于系统的位置编译op所需的头文件和库。但是，TensorFlow python库提供了get\_include获取头目录的 功能，并且该get\_lib目录有一个要链接的共享对象。以下是Ubuntu机器上这些功能的输出。

$ python  
>>> import tensorflow as tf  
>>> tf.sysconfig.get\_include()  
'/usr/local/lib/python2.7/site-packages/tensorflow/include'  
>>> tf.sysconfig.get\_lib()  
'/usr/local/lib/python2.7/site-packages/tensorflow'

假设您已经g++安装，可以使用以下命令序列将op编译为动态库。

TF\_CFLAGS=( $(python -c 'import tensorflow as tf; print(" ".join(tf.sysconfig.get\_compile\_flags()))') )  
TF\_LFLAGS=( $(python -c 'import tensorflow as tf; print(" ".join(tf.sysconfig.get\_link\_flags()))') )  
g++ -std=c++11 -shared zero\_out.cc -o zero\_out.so -fPIC ${TF\_CFLAGS[@]} ${TF\_LFLAGS[@]} -O2

在Mac OS X上，构建.so文件时需要附加标志“-undefined dynamic\_lookup” 。

关于gcc版本的注意事项>=5：gcc 从版本开始使用新的C ++ [ABI](https://gcc.gnu.org/gcc-5/changes.html#libstdcxx)5。TensorFlow网站上提供的二进制pip包gcc4是使用旧的ABI 构建的。如果使用编译op库gcc>=5，请添加 -D\_GLIBCXX\_USE\_CXX11\_ABI=0到命令行以使库与旧的abi兼容。

使用bazel编译op（TensorFlow源代码安装）

如果安装了TensorFlow源，则可以使用TensorFlow的构建系统来编译操作。在[tensorflow/core/user\_ops](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/user_ops/)目录中放置一个带有以下Bazel构建规则的BUILD文件。

load("//tensorflow:tensorflow.bzl", "tf\_custom\_op\_library")  
  
tf\_custom\_op\_library(  
    name = "zero\_out.so",  
    srcs = ["zero\_out.cc"],  
)

运行以下命令进行构建zero\_out.so。

$ bazel build --config opt //tensorflow/core/user\_ops:zero\_out.so

如上所述，如果使用gcc> = 5进行编译，请添加--cxxopt="-D\_GLIBCXX\_USE\_CXX11\_ABI=0"到bazel命令行。

**注意：**虽然您可以**.so**使用标准**cc\_library**规则创建共享库（文件），但我们强烈建议您使用该 **tf\_custom\_op\_library**宏。它添加了一些必需的依赖项，并执行检查以确保共享库与TensorFlow的插件加载机制兼容。

在Python中使用op

TensorFlow Python API提供了[tf.load\_op\_library](https://tensorflow.google.cn/api_docs/python/tf/load_op_library)加载动态库并使用TensorFlow框架注册op 的 功能。load\_op\_library返回一个Python模块，其中包含op和内核的Python包装器。因此，一旦构建了op，就可以执行以下操作从Python运行它：

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()  
  
# Prints  
array([[1, 0], [0, 0]], dtype=int32)

请记住，生成的函数将被赋予snake\_case名称（符合[PEP8](https://www.python.org/dev/peps/pep-0008/)）。因此，如果您的操作ZeroOut在C ++文件中命名，则将调用python函数zero\_out。

要使op作为importPython模块中的常规函数​​可用，load\_op\_library在Python源文件中进行调用可能很有用，如下所示：

import tensorflow as tf  
  
zero\_out\_module = tf.load\_op\_library('./zero\_out.so')  
zero\_out = zero\_out\_module.zero\_out

验证操作是否正常

验证您是否已成功实施op的一种好方法是为其编写测试。zero\_out\_op\_test.py使用以下内容创建文件 ：

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])  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  tf.test.main()

然后运行测试（假设您已安装tensorflow）：

$ python zero\_out\_op\_test.py

在您的操作中构建高级功能

既然您已经知道如何构建一个基本的（并且受到限制的）操作和实现，我们将会看到您通常需要构建到op中的一些更复杂的事情。这包括：

* [有条件的检查和验证](https://tensorflow.google.cn/extend/adding_an_op#conditional_checks_and_validation)
* [操作注册](https://tensorflow.google.cn/extend/adding_an_op#op_registration)
  + [ATTRS](https://tensorflow.google.cn/extend/adding_an_op#attrs)
  + [Attr类型](https://tensorflow.google.cn/extend/adding_an_op#attr_types)
  + [多态性](https://tensorflow.google.cn/extend/adding_an_op#polymorphism)
  + [输入和输出](https://tensorflow.google.cn/extend/adding_an_op#inputs_and_outputs)
  + [向后兼容性](https://tensorflow.google.cn/extend/adding_an_op#backwards_compatibility)
* [GPU支持](https://tensorflow.google.cn/extend/adding_an_op#gpu_support)
  + [编译GPU设备的内核](https://tensorflow.google.cn/extend/adding_an_op#compiling_the_kernel_for_the_gpu_device)
* [在Python中实现渐变](https://tensorflow.google.cn/extend/adding_an_op#implement_the_gradient_in_python)
* [C ++中的Shape函数](https://tensorflow.google.cn/extend/adding_an_op#shape_functions_in_c)

有条件的检查和验证

上面的例子假设op适用于任何形状的张量。如果它只适用于矢量怎么办？这意味着在上面的OpKernel实现中添加一个检查。

  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."));  
    // ...  
  }

这断言输入是一个向量，InvalidArgument如果不是则返回已设置 状态。该 [OP\_REQUIRES宏](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/lib/core/errors.h)有三个参数：

* 的context，其可以是一个OpKernelContext或 OpKernelConstruction指针（见[tensorflow/core/framework/op\_kernel.h](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/op_kernel.h)），对于其SetStatus()方法。
* 条件。例如，有用于验证张量形状的函数 [tensorflow/core/framework/tensor\_shape.h](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/tensor_shape.h)
* 错误本身，由Status对象表示，请参阅 [tensorflow/core/lib/core/status.h](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/lib/core/status.h)。AStatus既有类型（经常InvalidArgument，但看到类型列表）和消息。可以在中找到构造错误的函数 [tensorflow/core/lib/core/errors.h](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/lib/core/errors.h)。

或者，如果要测试Status从某个函数返回的对象是否为错误，如果是，则返回它，请使用[OP\_REQUIRES\_OK](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/lib/core/errors.h)。这两个宏都会在出错时从函数返回。

操作注册

ATTRS

Ops可以有attrs，其值在op添加到图形时设置。这些用于配置op，并且可以在内核实现中以及op注册中的输入和输出类型中访问它们的值。在可能的情况下，首选使用输入而不是attr，因为输入更灵活。这是因为attrs是常量，必须在图形构造时定义。相反，输入是Tensors，其值可以是动态的; 也就是说，输入可以改变每一步，使用馈送等设置。吸附器用于无法通过输入完成的任务：任何影响签名的配置（输入或输出的数量或类型）或者可以'从一步到步改变。

您在注册op时定义了一个attr，Attr方法是使用方法指定其名称和类型，该方法需要以下形式的规范：

<name>: <attr-type-expr>

其中<name>以字母开头并且可以由字母数字字符和下划线，并且<attr-type-expr>是具有以下形式的式表达[如下所述](https://tensorflow.google.cn/extend/adding_an_op#attr_types)。

例如，如果您希望ZeroOutop保留用户指定的索引，而不是仅保留第0个元素，则可以像这样注册op：

REGISTER\_OP("ZeroOut")  
    .Attr("preserve\_index: int")  
    .Input("to\_zero: int32")  
    .Output("zeroed: int32");

（请注意，[属性类型](https://tensorflow.google.cn/extend/adding_an_op#attr_types)集与[tf.DType](https://tensorflow.google.cn/api_docs/python/tf/DType)用于输入和输出的[属性类型](https://tensorflow.google.cn/extend/adding_an_op#attr_types)集不同 。）

然后，您的内核可以通过以下context 参数在其构造函数中访问此attr ：

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\_;  
};

然后可以在Compute方法中使用：

  void Compute(OpKernelContext\* context) override {  
    // ...  
  
    // We're using saved attr to validate potentially dynamic input  
    // So we 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\_);  
  }

Attr类型

attr支持以下类型：

* string：任何字节序列（不需要是UTF8）。
* int：有符号整数。
* float：浮点数。
* bool： 对或错。
* type：其中一个（非参考）值[DataType](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/types.cc)。
* shape：A [TensorShapeProto](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/tensor_shape.proto)。
* tensor：A [TensorProto](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/tensor.proto)。
* list(<type>)：列表<type>，其中<type>一个是上述类型之一。注意list(list(<type>))无效。

另见：[op\_def\_builder.cc:FinalizeAttr](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/op_def_builder.cc)最终清单。

**默认值和约束**

Attrs可能有默认值，某些类型的attrs可能有约束。要使用约束定义attr，可以使用以下<attr-type-expr>s：

* {'<string1>', '<string2>'}：值必须是具有值<string1>或的字符串<string2>。string使用此语法时隐含了类型的名称。这模拟了一个枚举：

REGISTER\_OP("EnumExample")  
    .Attr("e: {'apple', 'orange'}");

* {<type1>, <type2>}：值是类型type，并且必须是<type1>或中的一个 <type2>，其中<type1>和<type2>支持 [tf.DType](https://tensorflow.google.cn/api_docs/python/tf/DType)。您没有指定attr的类型type。如果您有类型列表，则表示这意味着{...}。例如，在这种情况下，attr t是一个必须是a int32，a float或a的类型bool：

REGISTER\_OP("RestrictedTypeExample")  
    .Attr("t: {int32, float, bool}");

* 常见类型约束有快捷方式：
  + numbertype：类型type限制为数字（非字符串和非布尔）类型。
  + realnumbertype：喜欢numbertype没有复杂的类型。
  + quantizedtype：numbertype只是量化的数字类型。

这些类型允许的特定类型列表由函数（如NumberTypes()）中 定义[tensorflow/core/framework/types.h](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/types.h)。在此示例中，attr t必须是数字类型之一：

REGISTER\_OP("NumberType")  
    .Attr("t: numbertype");

对于这个操作：

tf.number\_type(t=tf.int32)  # Valid  
tf.number\_type(t=tf.bool)   # Invalid

列表可以与其他列表和单个类型组合使用。以下操作允许attr t为任何数字类型或bool类型：

REGISTER\_OP("NumberOrBooleanType")  
    .Attr("t: {numbertype, bool}");

对于这个操作：

tf.number\_or\_boolean\_type(t=tf.int32)  # Valid  
tf.number\_or\_boolean\_type(t=tf.bool)   # Valid  
tf.number\_or\_boolean\_type(t=tf.string) # Invalid

* int >= <n>：值必须是int，其值大于或等于 <n>，其中<n>是自然数。

例如，以下op注册指定attr a必须至少具有以下值2：

REGISTER\_OP("MinIntExample")  
    .Attr("a: int >= 2");

* list(<type>) >= <n>：<type>长度大于或等于的类型列表<n>。

例如，下面的运算登记指定的ATTR a是一个类型列表（无论是int32或float），并且必须有它们中的至少3：

REGISTER\_OP("TypeListExample")  
    .Attr("a: list({int32, float}) >= 3");

要设置attr的默认值（使其在生成的代码中可选），请添加= <default>到结尾，如：

REGISTER\_OP("AttrDefaultExample")  
    .Attr("i: int = 0");

支持的默认值语法是在生成的GraphDef定义的proto表示中使用的语法。

以下是如何为所有类型指定默认值的示例：

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]");

特别注意类型type 使用的值[tf.DType](https://tensorflow.google.cn/api_docs/python/tf/DType)。

多态性

**类型多态性**

对于可采取不同类型的输入或产生不同的输出类型的OPS，可以指定[一个ATTR](https://tensorflow.google.cn/extend/adding_an_op#attrs)在 [输入或输出型](https://tensorflow.google.cn/extend/adding_an_op#inputs_and_outputs)的运算注册。通常，您会OpKernel为每种支持的类型注册一个。

例如，如果除了s 之外你还希望ZeroOut操作s，你的操作注册可能如下所示：floatint32

REGISTER\_OP("ZeroOut")  
    .Attr("T: {float, int32}")  
    .Input("to\_zero: T")  
    .Output("zeroed: T");

您的op注册现在指定输入的类型必须是float，或者 int32，它的输出将是相同的类型，因为它们都具有类型T。

关于命名的注释：输入，输出和attrs通常应该给出snake\_case名称。一个例外是用作输入类型或输入类型的attrs。当op添加到图形中时，可以推断出这些attrs，因此不会出现在op的函数中。例如，ZeroOut的最后一个定义将生成一个Python函数，如下所示：

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`.  
  """

如果to\_zero传递int32张量，则T自动设置为 int32（实际上DT\_INT32）。这些推断的attrs被赋予大写或CamelCase名称。

将其与具有确定输出类型的类型attr的op进行比较：

REGISTER\_OP("StringToNumber")  
    .Input("string\_tensor: string")  
    .Output("output: out\_type")  
    .Attr("out\_type: {float, int32} = DT\_FLOAT");  
    .Doc(R"doc(  
Converts each string in the input Tensor to the specified numeric type.  
)doc");

在这种情况下，用户必须指定输出类型，如生成的Python中所示：

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 <a href="./../api\_docs/python/tf/DType"><code>tf.DType</code></a> from: `tf.float32, tf.int32`.  
      Defaults to <a href="./../api\_docs/python/tf#float32"><code>tf.float32</code></a>.  
    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"),  
    ZeroOutOpInt32);  
REGISTER\_KERNEL\_BUILDER(  
    Name("ZeroOut")  
    .Device(DEVICE\_CPU)  
    .TypeConstraint<float>("T"),  
    ZeroOutFloatOp);

要保持[向后兼容性](https://tensorflow.google.cn/extend/adding_an_op#backwards_compatibility)，应在将attr添加到现有op时指定[默认值](https://tensorflow.google.cn/extend/adding_an_op#default_values_constraints)：

REGISTER\_OP("ZeroOut")  
  .Attr("T: {float, int32} = DT\_INT32")  
  .Input("to\_zero: T")  
  .Output("zeroed: T")

假设您要添加更多类型，请说double：

REGISTER\_OP("ZeroOut")  
    .Attr("T: {float, double, int32}")  
    .Input("to\_zero: T")  
    .Output("zeroed: T");

OpKernel通常您可以使用C ++模板，而不是使用上面的冗余代码编写另一个代码。REGISTER\_KERNEL\_BUILDER每次重载仍然会有一个内核注册（调用）。

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>);

如果您有多个重载，则可以将注册放在宏中。

#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

根据您为其注册内核的类型列表，您可以使用以下提供的宏[tensorflow/core/framework/register\_types.h](https://www.github.com/tensorflow/tensorflow/blob/master/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

**列出输入和输出**

除了能够接受或产生不同类型之外，操作可以消耗或产生可变数量的张量。

在下一个示例中，attr T包含类型*列表*，并用作输入in和输出的类型out。输入和输出是该类型的张量列表（输出中张量的数量和类型与输入相同，因为两者都有类型T）。

REGISTER\_OP("PolymorphicListExample")  
    .Attr("T: list(type)")  
    .Input("in: T")  
    .Output("out: T");

您还可以对列表中指定的类型进行限制。在下一种情况下，输入是一个列表float和double张量。例如，op接受输入类型(float, double, float)，在这种情况下输出类型也是(float, double, float)。

REGISTER\_OP("ListTypeRestrictionExample")  
    .Attr("T: list({float, double})")  
    .Input("in: T")  
    .Output("out: T");

如果您希望列表中的所有张量具有相同的类型，您可能会执行以下操作：

REGISTER\_OP("IntListInputExample")  
    .Attr("N: int")  
    .Input("in: N \* int32")  
    .Output("out: int32");

这接受int32张量列表，并使用intattr N指定列表的长度。

这也可以是[多态的](https://tensorflow.google.cn/extend/adding_an_op#type_polymorphism)。在下一个示例中，输入是具有"N"相同（但未指定）类型（"T"）的张量（具有长度）的列表，并且输出是匹配类型的单个张量：

REGISTER\_OP("SameListInputExample")  
    .Attr("N: int")  
    .Attr("T: type")  
    .Input("in: N \* T")  
    .Output("out: T");

默认情况下，张名单有1的最小长度您可以更改使用默认的 [一">="对相应的ATTR约束](https://tensorflow.google.cn/extend/adding_an_op#default_values_constraints)。在下一个示例中，输入是至少2个int32张量的列表：

REGISTER\_OP("MinLengthIntListExample")  
    .Attr("N: int >= 2")  
    .Input("in: N \* int32")  
    .Output("out: int32");

相同的语法适用于"list(type)"attrs：

REGISTER\_OP("MinimumLengthPolymorphicListExample")  
    .Attr("T: list(type) >= 3")  
    .Input("in: T")  
    .Output("out: T");

输入和输出

总结一下，op注册可以有多个输入和输出：

REGISTER\_OP("MultipleInsAndOuts")  
    .Input("y: int32")  
    .Input("z: float")  
    .Output("a: string")  
    .Output("b: int32");

每个输入或输出规范的形式如下：

<name>: <io-type-expr>

其中<name>以字母开头并且可以由字母数字字符和下划线。<io-type-expr>是以下类型表达式之一：

* <type>，其中<type>是一个支持的输入类型（例如float，int32， string）。这指定了给定类型的单个张量。

见 [tf.DType](https://tensorflow.google.cn/api_docs/python/tf/DType)。

REGISTER\_OP("BuiltInTypesExample")  
    .Input("integers: int32")  
    .Input("complex\_numbers: complex64");

* <attr-type>，其中<attr-type>是具有类型 的[Attr](https://tensorflow.google.cn/extend/adding_an_op#attrs)的名称type或list(type)（具有可能的类型限制）。此语法允许[多态操作](https://tensorflow.google.cn/extend/adding_an_op#polymorphism)。

REGISTER\_OP("PolymorphicSingleInput")  
    .Attr("T: type")  
    .Input("in: T");  
  
REGISTER\_OP("RestrictedPolymorphicSingleInput")  
    .Attr("T: {int32, int64}")  
    .Input("in: T");

引用类型的attr list(type)允许您接受一系列张量。

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");

请注意，输出中张量的数量和类型与输入out中的相同in，因为两者都是类型T。

* 对于具有相同类型的张量序列：<number> \* <type>，其中 <number>是具有类型的[Attr](https://tensorflow.google.cn/extend/adding_an_op#attrs)的名称int。的<type>可以是一个[tf.DType](https://tensorflow.google.cn/api_docs/python/tf/DType)，或者与一个类型attr的名称type。作为第一个示例，此操作接受int32张量列表：

REGISTER\_OP("Int32SequenceExample")  
    .Attr("NumTensors: int")  
    .Input("in: NumTensors \* int32")

虽然这个op接受任何类型的张量列表，只要它们都是相同的：

REGISTER\_OP("SameTypeSequenceExample")  
    .Attr("NumTensors: int")  
    .Attr("T: type")  
    .Input("in: NumTensors \* T")

* 对于张量的引用：Ref(<type>)，其中<type>一个是先前的类型。

关于命名的注释：将推断出在输入类型中使用的任何attr。按照惯例，这些推断的attrs使用大写名称（如T或N）。否则输入，输出和attrs具有类似函数参数的名称（例如num\_outputs）。有关更多详细信息，请参阅 [前面有关命名的说明](https://tensorflow.google.cn/extend/adding_an_op#naming)。

有关详细信息，请参阅 [tensorflow/core/framework/op\_def\_builder.h](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/op_def_builder.h)。

向后兼容性

假设您已经编写了一个很好的自定义操作并与其他人共享，因此您可以使用您的操作来满足客户的需求。但是，您希望以某种方式对op进行更改。

通常，对现有的签入规范的更改必须向后兼容：更改op的规范不得破坏先前GraphDef由旧规范构造的序列化协议缓冲区。的细节GraphDef兼容性是 [这里描述](https://tensorflow.google.cn/guide/version_compat#compatibility_of_graphs_and_checkpoints)。

有几种方法可以保持向后兼容性。

1. 添加到操作的任何新attrs必须定义默认值，并且使用该默认值，op必须具有原始行为。要将操作从非多态更改为多态，*必须*为新类型attr提供默认值，以便默认保留原始签名。例如，如果您的操作是：

REGISTER\_OP（“MyGeneralUnaryOp”）。Input（“in：float”）。Output（“out：float”）;

您可以使用以下方式以向后兼容的方式使其变为多态：

REGISTER\_OP（“MyGeneralUnaryOp”）。输入（“in：T”）。输出（“out：T”）。Attr（“T：numerictype = DT\_FLOAT”）;

1. 您可以安全地对attr进行约束，从而减少限制。例如，您可以更改{int32, int64}为{int32, int64, float}或type。或者你可以{"apple", "orange"}改为{"apple", "banana", "orange"}或string。
2. 只要列表类型的默认值与旧签名匹配，您就可以将单个输入/输出更改为列表输入/输出。
3. 如果默认为空，则可以添加新的列表输入/输出。
4. 命名空间您创建的任何新操作，通过在操作名称前添加项目唯一的内容。这可以避免您的op与可能包含在TensorFlow的未来版本中的任何操作发生冲突。
5. 未雨绸缪！尝试预测op的未来用途。某些签名更改无法以兼容的方式完成（例如，将相同类型的列表放入不同类型的列表中）。

有关安全和不安全更改的完整列表，请参阅[tensorflow/core/framework/op\_compatibility\_test.cc](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/op_compatibility_test.cc)。如果无法对向后兼容的操作进行更改，则使用新语义创建具有新名称的新操作。

另请注意，虽然这些更改可以保持GraphDef兼容性，但生成的Python代码可能会以与旧调用方不兼容的方式更改。Python API可以通过手写Python包装器中的仔细更改保持兼容，保留旧签名，除了可能在末尾添加新的可选参数。通常，只有在TensorFlow更改主要版本时才会进行不兼容的更改，并且必须符合 [GraphDef版本语义](https://tensorflow.google.cn/guide/version_compat#compatibility_of_graphs_and_checkpoints)。

GPU支持

您可以实现不同的OpKernel并为CPU注册一个，为GPU注册另一个，就像您可以[为不同类型注册内核一样](https://tensorflow.google.cn/extend/adding_an_op#polymorphism)。有几个内核支持GPU的例子 [tensorflow/core/kernels/](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/)。请注意，某些内核在.cc文件中具有CPU版本，在文件中以结尾的GPU版本以及在文件中\_gpu.cu.cc共享的一些代码.h。

例如，[tf.pad](https://tensorflow.google.cn/api_docs/python/tf/pad)除了GPU内核之外，其他东西都有[tensorflow/core/kernels/pad\_op.cc](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/pad_op.cc)。GPU内核在 [tensorflow/core/kernels/pad\_op\_gpu.cu.cc](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/pad_op_gpu.cu.cc)，并且共享代码是在其中定义的模板类[tensorflow/core/kernels/pad\_op.h](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/pad_op.h)。我们以这种方式组织代码有两个原因：它允许您在CPU和GPU实现之间共享公共代码，并且它将GPU实现放入单独的文件中，以便它只能由GPU编译器编译。

有一点需要注意，即使使用GPU内核版本pad，它仍然需要"paddings"在CPU内存中输入。要标记输入或输出保留在CPU上，请添加HostMemory()对内核注册的调用，例如：

#define REGISTER\_GPU\_KERNEL(T)                         \  
  REGISTER\_KERNEL\_BUILDER(Name("Pad")                  \  
                              .Device(DEVICE\_GPU)      \  
                              .TypeConstraint<T>("T")  \  
                              .HostMemory("paddings"), \  
                          PadOp<GPUDevice, T>)

编译GPU设备的内核

查看 [cuda\_op\_kernel.cu.cc](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/adding_an_op/cuda_op_kernel.cu.cc) 以获取使用CUDA内核实现操作的示例。在 tf\_custom\_op\_library接受一个gpu\_srcs在其中含有CUDA内核（源文件的列表参数\*.cu.cc可指定文件）。为了与TensorFlow的二进制安装一起使用，必须使用NVIDIA的nvcc编译器编译CUDA内核。以下是可用于将[cuda\_op\_kernel.cu.cc](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/adding_an_op/cuda_op_kernel.cu.cc) 和 [cuda\_op\_kernel.cc](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/adding_an_op/cuda_op_kernel.cc)编译 为单个可动态加载库的命令序列 ：

nvcc -std=c++11 -c -o cuda\_op\_kernel.cu.o cuda\_op\_kernel.cu.cc \  
  ${TF\_CFLAGS[@]} -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 ${TF\_CFLAGS[@]} -fPIC -lcudart ${TF\_LFLAGS[@]}

cuda\_op\_kernel.so上面生成的可以像往常一样在Python中使用[tf.load\_op\_library](https://tensorflow.google.cn/api_docs/python/tf/load_op_library)函数加载 。

请注意，如果未安装CUDA库，则/usr/local/lib64需要在上面的第二个（g ++）命令中明确指定路径。例如，-L /usr/local/cuda-8.0/lib64/如果安装了CUDA ，请添加/usr/local/cuda-8.0。

请注意，在某些Linux设置中，nvcc需要编译步骤的其他选项。添加-D\_MWAITXINTRIN\_H\_INCLUDED到nvcc命令行以避免错误mwaitxintrin.h。

在Python中实现渐变

给定操作图，TensorFlow使用自动微分（反向传播）来添加表示相对于现有操作的[渐变](https://tensorflow.google.cn/api_guides/python/train#Gradient_Computation)的新操作（参见 [梯度计算](https://tensorflow.google.cn/api_guides/python/train#Gradient_Computation)）。要为新操作进行自动微分工作，必须注册一个梯度函数，该函数根据ops输出给定梯度，根据ops的输入计算梯度。

在数学上，如果一个操作计算 ÿ=F（X） 注册的梯度op转换渐变 ∂大号/∂ÿ 损失 大号 关于 ÿ 进入渐变 ∂大号/∂X 关于 X 通过链规则：

∂大号∂X=∂大号∂ÿ∂ÿ∂X=∂大号∂ÿ∂F∂X。

在这种情况下ZeroOut，输入中只有一个条目影响输出，因此相对于输入的梯度是稀疏的“一个热”张量。这表示如下：

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

有关注册梯度函数的详细信息 [tf.RegisterGradient](https://tensorflow.google.cn/api_docs/python/tf/RegisterGradient)：

* 对于具有一个输出运算，梯度函数将采取 以及 和建立新的OPS出张量 ， 和。有关任何attrs的信息可以通过 。[tf.Operation](https://tensorflow.google.cn/api_docs/python/tf/Operation) op[tf.Tensor](https://tensorflow.google.cn/api_docs/python/tf/Tensor)grad[op.inputs[i]](https://tensorflow.google.cn/api_docs/python/framework#Operation.inputs)[op.outputs[i]](https://tensorflow.google.cn/api_docs/python/framework#Operation.outputs)grad[tf.Operation.get\_attr](https://tensorflow.google.cn/api_docs/python/tf/Operation#get_attr)
* 如果op有多个输出，则梯度函数将采用op和 grads，其中grads是相对于每个输出的梯度列表。渐变函数的结果必须是Tensor表示相对于每个输入的渐变的对象列表。
* 如果某些输入没有明确定义的渐变，例如对于用作索引的整数输入，则应返回相应的返回渐变 None。例如，对于采用浮点张量x和整数索引的op i，渐变函数会return [x\_grad, None]。
* 如果op根本没有有意义的渐变，你通常不需要注册任何渐变，只要不需要op的渐变，你就可以了。在某些情况下，op没有明确定义的梯度，但可以参与梯度的计算。在这里，您可以使用 ops.NotDifferentiable自动向后传播零。

请注意，在调用梯度函数时，只有ops的数据流图可用，而不是张量数据本身。因此，必须使用其他tensorflow操作执行所有计算，以在图执行时间运行。

C ++中的Shape函数

TensorFlow API具有一个称为“形状推断”的功能，可以提供有关张量形状的信息，而无需执行图形。形状推断由在C ++ REGISTER\_OP声明中为每个op类型注册的“形状函数”支持，并执行两个角色：在图形构造期间断言输入的形状是兼容的，并指定输出的形状。

形状函数定义为shape\_inference::InferenceContext类的操作 。例如，在ZeroOut的形状函数中：

    .SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {  
      c->set\_output(0, c->input(0));  
      return Status::OK();  
    });

c->set\_output(0, c->input(0));声明第一个输出的形状应该设置为第一个输入的形状。如果输出是由其索引选择的，如上例所示，则第二个参数set\_output应该是一个ShapeHandle对象。您可以ShapeHandle通过其默认构造函数创建一个空对象。ShapeHandle具有索引的输入的对象idx可以通过获得c->input(idx)。

有许多常见的形状函数适用于许多操作，例如shape\_inference::UnchangedShape可以在[common\_shape\_fns.h中](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/common_shape_fns.h)找到并使用如下：

REGISTER\_OP("ZeroOut")  
    .Input("to\_zero: int32")  
    .Output("zeroed: int32")  
    .SetShapeFn(::tensorflow::shape\_inference::UnchangedShape);

形状函数还可以约束输入的形状。对于[ZeroOut具有矢量形状约束](https://tensorflow.google.cn/extend/adding_an_op#validation)的版本 ，形状函数将如下所示：

    .SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {  
      ::tensorflow::shape\_inference::ShapeHandle input;  
      TF\_RETURN\_IF\_ERROR(c->WithRank(c->input(0), 1, &input));  
      c->set\_output(0, input);  
      return Status::OK();  
    });

该WithRank调用验证输入形状c->input(0)具有恰好一维的形状（或者如果输入形状未知，则输出形状将是具有一个未知维度的矢量）。

如果您的op是[具有多个输入的多态](https://tensorflow.google.cn/extend/adding_an_op#polymorphism)，您可以使用成员InferenceContext来确定要检查的形状数，并 Merge验证形状是否全部兼容（或者，指示长度的访问属性InferenceContext::GetAttr，用于提供对属性的访问）的操作）。

    .SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {  
      ::tensorflow::shape\_inference::ShapeHandle input;  
      ::tensorflow::shape\_inference::ShapeHandle output;  
      for (size\_t i = 0; i < c->num\_inputs(); ++i) {  
        TF\_RETURN\_IF\_ERROR(c->WithRank(c->input(i), 2, &input));  
        TF\_RETURN\_IF\_ERROR(c->Merge(output, input, &output));  
      }  
      c->set\_output(0, output);  
      return Status::OK();  
    });

由于形状推断是可选特征，并且张量的形状可以动态变化，因此对于任何输入，形状函数必须对于不完整的形状信息是鲁棒的。该Merge方法[InferenceContext](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/shape_inference.h) 允许调用者断言两个形状是相同的，即使它们中的一个或两个都没有完整的信息。为所有核心TensorFlow操作定义了形状函数，并提供了许多不同的用法示例。

所述InferenceContext类具有许多可被用于定义形状的功能操作的功能。例如，您可以使用InferenceContext::Dim和 验证特定维度是否具有非常具体的值InferenceContext::WithValue; 可以指定的输出尺寸是使用两个输入尺寸之和/产品InferenceContext::Add和 InferenceContext::Multiply。查看InferenceContext您可以指定的所有各种形状操作的类。以下示例将第一个输出的形状设置为（n，3），其中第一个输入具有形状（n，...）

.SetShapeFn([](::tensorflow::shape\_inference::InferenceContext\* c) {  
    c->set\_output(0, c->Matrix(c->Dim(c->input(0), 0), 3));  
    return Status::OK();  
});

如果您具有复杂的形状函数，则应考虑添加测试以验证各种输入形状组合是否产生预期的输出形状组合。您可以在我们的一些[核心操作测试中](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/ops/array_ops_test.cc)看到如何编写这些测试的示例 。（语法INFER\_OK和INFER\_ERROR有点神秘，但尽量在代表测试输入和输出形状规格紧凑。现在，看到那些测试周围的意见，以获得形状串规范的意义上）。

添加自定义文件系统插件

背景

TensorFlow框架通常用于多进程和多机器环境，例如Google数据中心，Google Cloud Machine Learning，Amazon Web Services（AWS）和现场分布式群集。为了共享和保存由TensorFlow生成的某些类型的状态，该框架假定存在可靠的共享文件系统。这个共享文件系统有很多用途，例如：

* 状态检查点通常保存到分布式文件系统中，以实现可靠性和容错性。
* 培训过程通过将事件文件写入TensorBoard监视的目录来与TensorBoard进行通信。即使TensorBoard在不同的进程或机器中运行，共享文件系统也允许此通信工作。

在现实世界中有许多不同的共享或分布式文件系统实现，因此TensorFlow为用户提供了一种能够实现可以在TensorFlow运行时注册的自定义FileSystem插件的功能。当TensorFlow运行时尝试通过FileSystem 接口写入文件时，它使用路径名的一部分动态选择应该用于文件系统操作的实现。因此，添加对自定义文件系统的支持需要实现FileSystem 接口，构建包含该实现的共享对象，并在运行时加载该对象，无论哪个进程需要写入该文件系统。

请注意，TensorFlow已包含许多文件系统实现，例如：

* 标准POSIX文件系统

**注意：**NFS文件系统通常作为POSIX接口安装，因此标准TensorFlow可以在NFS安装的远程文件系统上运行。

* HDFS - Hadoop文件系统
* GCS - Google云端存储文件系统
* S3 - Amazon Simple Storage Service文件系统
* “内存映射文件”文件系统

本指南的其余部分介绍了如何实现自定义文件系统。

实现自定义文件系统插件

要实现自定义文件系统插件，您必须执行以下操作：

* 实现的子类RandomAccessFile，WriteableFile， AppendableFile，和ReadOnlyMemoryRegion。
* 将FileSystem接口实现为子类。
* FileSystem使用适当的前缀模式注册实现。
* 在要写入该文件系统的进程中加载​​文件系统插件。

FileSystem接口

该FileSystem接口是[file\_system.h中](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/platform/file_system.h)定义的抽象C ++接口 。FileSystem接口的实现应该实现接口定义的所有相关方法。实现该接口要求限定，例如，创建动作RandomAccessFile，WritableFile和实施标准的文件系统操作，例如FileExists，IsDirectory，GetMatchingPaths，DeleteFile，等。这些接口的实现通常涉及将函数的输入参数转换为委托给已经存在的库函数，以实现自定义文件系统中的等效功能。

例如，PosixFileSystem实现DeleteFile使用POSIX unlink()函数实现; CreateDir只是打电话mkdir(); GetFileSize 涉及调用stat()该文件，然后返回stat对象返回报告的文件大小。同样，对于HDFSFileSystem 实现中，这些调用直接委托给libHDFS类似的功能，如实现hdfsDelete了 [的DeleteFile](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/platform/hadoop/hadoop_file_system.cc#L386)。

我们建议查看这些代码示例，以了解不同的文件系统实现如何调用其现有库。例子包括：

* [POSIX插件](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/platform/posix/posix_file_system.h)
* [HDFS插件](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/platform/hadoop/hadoop_file_system.h)
* [GCS插件](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/platform/cloud/gcs_file_system.h)
* [S3插件](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/platform/s3/s3_file_system.h)

文件接口

除了允许您查询和操作文件系统中的文件和目录的操作之外，该FileSystem接口还要求您实现返回抽象对象（如[RandomAccessFile](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/platform/file_system.h#L223)）的实现的工厂 WritableFile，以便TensorFlow代码和读取和写入该FileSystem实现中的文件 。

要实现a RandomAccessFile，必须实现一个名为的接口 Read()，其中实现必须提供一种从命名文件中的偏移量读取的方法。

例如，下面是POSIX文件系统的RandomAccessFile的实现，它使用pread()随机访问POSIX函数来实现读取。请注意，特定实现必须知道如何从底层文件系统重试或传播错误。

    class PosixRandomAccessFile : public RandomAccessFile {  
     public:  
      PosixRandomAccessFile(const string& fname, int fd)  
          : filename\_(fname), fd\_(fd) {}  
      ~PosixRandomAccessFile() override { close(fd\_); }  
  
      Status Read(uint64 offset, size\_t n, StringPiece\* result,  
                  char\* scratch) const override {  
        Status s;  
        char\* dst = scratch;  
        while (n > 0 && s.ok()) {  
          ssize\_t r = pread(fd\_, dst, n, static\_cast<off\_t>(offset));  
          if (r > 0) {  
            dst += r;  
            n -= r;  
            offset += r;  
          } else if (r == 0) {  
            s = Status(error::OUT\_OF\_RANGE, "Read less bytes than requested");  
          } else if (errno == EINTR || errno == EAGAIN) {  
            // Retry  
          } else {  
            s = IOError(filename\_, errno);  
          }  
        }  
        \*result = StringPiece(scratch, dst - scratch);  
        return s;  
      }  
  
     private:  
      string filename\_;  
      int fd\_;  
    };

为了实现WritableFile顺序书写的抽象，一个必须实现的几个接口，如Append()，Flush()，Sync()，和Close()。

例如，下面是POSIX文件系统的WritableFile的实现，它FILE在其构造函数中获取一个对象，并在该对象上使用标准的posix函数来实现该接口。

    class PosixWritableFile : public WritableFile {  
     public:  
      PosixWritableFile(const string& fname, FILE\* f)  
          : filename\_(fname), file\_(f) {}  
  
      ~PosixWritableFile() override {  
        if (file\_ != NULL) {  
          fclose(file\_);  
        }  
      }  
  
      Status Append(const StringPiece& data) override {  
        size\_t r = fwrite(data.data(), 1, data.size(), file\_);  
        if (r != data.size()) {  
          return IOError(filename\_, errno);  
        }  
        return Status::OK();  
      }  
  
      Status Close() override {  
        Status result;  
        if (fclose(file\_) != 0) {  
          result = IOError(filename\_, errno);  
        }  
        file\_ = NULL;  
        return result;  
      }  
  
      Status Flush() override {  
        if (fflush(file\_) != 0) {  
          return IOError(filename\_, errno);  
        }  
        return Status::OK();  
      }  
  
      Status Sync() override {  
        Status s;  
        if (fflush(file\_) != 0) {  
          s = IOError(filename\_, errno);  
        }  
        return s;  
      }  
  
     private:  
      string filename\_;  
      FILE\* file\_;  
    };

有关更多详细信息，请参阅这些界面的文档，并查看示例实现的灵感。

注册并加载文件系统

一旦FileSystem为自定义文件系统实现了实现，就需要在“方案”下注册它，以便将以该方案为前缀的路径定向到您的实现。要做到这一点，你打电话 REGISTER\_FILE\_SYSTEM::

    REGISTER\_FILE\_SYSTEM("foobar", FooBarFileSystem);

当TensorFlow尝试对路径开头的文件进行操作时foobar://，它将使用该FooBarFileSystem实现。

    string filename = "foobar://path/to/file.txt";  
    std::unique\_ptr<WritableFile> file;  
  
    // Calls FooBarFileSystem::NewWritableFile to return  
    // a WritableFile class, which happens to be the FooBarFileSystem's  
    // WritableFile implementation.  
    TF\_RETURN\_IF\_ERROR(env->NewWritableFile(filename, &file));

接下来，您必须构建包含此实现的共享对象。cc\_binary可以在[此处](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/BUILD#L244)找到使用bazel 规则 执行此操作的示例，但您可以使用任何构建系统来执行此操作。有关类似说明，请参阅[构建op库](https://tensorflow.google.cn/extend/adding_an_op#build_the_op_library)的部分。

构建此目标的结果是.so共享对象文件。

最后，您必须在此过程中动态加载此实现。在Python中，您可以调用该tf.load\_file\_system\_library(file\_system\_library)函数，将路径传递给共享对象。在客户端程序中调用此方法会在进程中加载​​共享对象，从而将实现注册为可用于通过FileSystem接口的任何文件操作。您可以查看 [test\_file\_system.py](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/framework/file_system_test.py) 作为示例。

什么通过这个界面？

内几乎所有TensorFlow核心C ++文件操作使用FileSystem 界面，比如CheckpointWriter，在EventsWriter和许多其他工具。这意味着实现一个FileSystem实现允许大多数TensorFlow程序写入共享文件系统。

在Python中，gfile和file\_io类通过SWIG绑定到`FileSystem实现，这意味着一旦你加载了这个文件系统库，你就可以：

with gfile.Open("foobar://path/to/file.txt") as w:  
  
  w.write("hi")

执行此操作时，包含“hi”的文件将出现在共享文件系统的“/path/to/file.txt”中。

读取自定义文件和记录格式

先决条件：

* 熟悉C ++。
* 必须已经[下载了TensorFlow源](https://tensorflow.google.cn/install/source)，并且能够构建它。

我们将支持文件格式的任务分为两部分：

* 文件格式：我们使用阅读器[tf.data.Dataset](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset)从文件中读取*原始记录*（通常由标量字符串张量表示，但可以有更多结构）。
* 记录格式：我们使用解码器或解析操作将字符串记录转换为TensorFlow可用的张量。

例如，要重新实现[tf.contrib.data.make\_csv\_dataset](https://tensorflow.google.cn/api_docs/python/tf/contrib/data/make_csv_dataset)函数，我们可以使用[tf.data.TextLineDataset](https://tensorflow.google.cn/api_docs/python/tf/data/TextLineDataset)提取记录，然后使用[tf.data.Dataset.map](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset#map)和[tf.decode\_csv](https://tensorflow.google.cn/api_docs/python/tf/decode_csv)解析数据集中每行文本的CSV记录。

写Dataset一个文件格式

A [tf.data.Dataset](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset)表示*元素*序列，可以是文件中的各个记录。有几个已经内置在TensorFlow中的“阅读器”数据集示例：

* [tf.data.TFRecordDataset](https://tensorflow.google.cn/api_docs/python/tf/data/TFRecordDataset) （[来源kernels/data/reader\_dataset\_ops.cc](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/data/reader_dataset_ops.cc)）
* [tf.data.FixedLengthRecordDataset](https://tensorflow.google.cn/api_docs/python/tf/data/FixedLengthRecordDataset) （[来源kernels/data/reader\_dataset\_ops.cc](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/data/reader_dataset_ops.cc)）
* [tf.data.TextLineDataset](https://tensorflow.google.cn/api_docs/python/tf/data/TextLineDataset) （[来源kernels/data/reader\_dataset\_ops.cc](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/data/reader_dataset_ops.cc)）

这些实现中的每一个都包含三个相关类：

* 一个tensorflow::DatasetOpKernel子类（例如TextLineDatasetOp），它告诉TensorFlow如何在其MakeDataset()方法中从op的输入和attrs构造数据集对象。
* 甲tensorflow::GraphDatasetBase亚类（例如TextLineDatasetOp::Dataset），它代表了*不可变*的数据集本身的定义，并告诉TensorFlow如何构造迭代器对象在该数据集在其 MakeIteratorInternal()方法。
* 一个tensorflow::DatasetIterator<Dataset>子类（例如TextLineDatasetOp::Dataset::Iterator），它表示特定数据集上迭代器的*可变*状态，并告诉TensorFlow如何从迭代器中获取其GetNextInternal()方法中的下一个元素。

最重要的方法是GetNextInternal()方法，因为它定义了如何从文件中实际读取记录并将它们表示为一个或多个 Tensor对象。

要创建一个名为（例如）的新读者数据集MyReaderDataset，您需要：

1. 在C ++中，定义的子类tensorflow::DatasetOpKernel，tensorflow::GraphDatasetBase和tensorflow::DatasetIterator<Dataset> 实现该读逻辑。
2. 在C ++中，使用名称注册一个新的reader op和kernel "MyReaderDataset"。
3. 在Python中，定义[tf.data.Dataset](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset)被调用的子类MyReaderDataset。

您可以将所有C ++代码放在一个文件中，例如 my\_reader\_dataset\_op.cc。如果您熟悉[添加操作方法](https://tensorflow.google.cn/extend/adding_an_op)，将会有所帮助。以下框架可用作实施的起点：

#include "tensorflow/core/framework/common\_shape\_fns.h"  
#include "tensorflow/core/framework/dataset.h"  
#include "tensorflow/core/framework/op.h"  
#include "tensorflow/core/framework/shape\_inference.h"  
  
namespace myproject {  
namespace {  
  
using ::tensorflow::DT\_STRING;  
using ::tensorflow::PartialTensorShape;  
using ::tensorflow::Status;  
  
class MyReaderDatasetOp : public tensorflow::DatasetOpKernel {  
 public:  
  
  MyReaderDatasetOp(tensorflow::OpKernelConstruction\* ctx)  
      : DatasetOpKernel(ctx) {  
    // Parse and validate any attrs that define the dataset using  
    // `ctx->GetAttr()`, and store them in member variables.  
  }  
  
  void MakeDataset(tensorflow::OpKernelContext\* ctx,  
                   tensorflow::DatasetBase\*\* output) override {  
    // Parse and validate any input tensors that define the dataset using  
    // `ctx->input()` or the utility function  
    // `ParseScalarArgument<T>(ctx, &arg)`.  
  
    // Create the dataset object, passing any (already-validated) arguments from  
    // attrs or input tensors.  
    \*output = new Dataset(ctx);  
  }  
  
 private:  
  class Dataset : public tensorflow::GraphDatasetBase {  
   public:  
    Dataset(tensorflow::OpKernelContext\* ctx) : GraphDatasetBase(ctx) {}  
  
    std::unique\_ptr<tensorflow::IteratorBase> MakeIteratorInternal(  
        const string& prefix) const override {  
      return std::unique\_ptr<tensorflow::IteratorBase>(new Iterator(  
          {this, tensorflow::strings::StrCat(prefix, "::MyReader")}));  
    }  
  
    // Record structure: Each record is represented by a scalar string tensor.  
    //  
    // Dataset elements can have a fixed number of components of different  
    // types and shapes; replace the following two methods to customize this  
    // aspect of the dataset.  
    const tensorflow::DataTypeVector& output\_dtypes() const override {  
      static auto\* const dtypes = new tensorflow::DataTypeVector({DT\_STRING});  
      return \*dtypes;  
    }  
    const std::vector<PartialTensorShape>& output\_shapes() const override {  
      static std::vector<PartialTensorShape>\* shapes =  
          new std::vector<PartialTensorShape>({ {}});  
      return \*shapes;  
    }  
  
    string DebugString() const override { return "MyReaderDatasetOp::Dataset"; }  
  
   protected:  
    // Optional: Implementation of `GraphDef` serialization for this dataset.  
    //  
    // Implement this method if you want to be able to save and restore  
    // instances of this dataset (and any iterators over it).  
    Status AsGraphDefInternal(DatasetGraphDefBuilder\* b,  
                              tensorflow::Node\*\* output) const override {  
      // Construct nodes to represent any of the input tensors from this  
      // object's member variables using `b->AddScalar()` and `b->AddVector()`.  
      std::vector<tensorflow::Node\*> input\_tensors;  
      TF\_RETURN\_IF\_ERROR(b->AddDataset(this, input\_tensors, output));  
      return Status::OK();  
    }  
  
   private:  
    class Iterator : public tensorflow::DatasetIterator<Dataset> {  
     public:  
      explicit Iterator(const Params& params)  
          : DatasetIterator<Dataset>(params), i\_(0) {}  
  
      // Implementation of the reading logic.  
      //  
      // The example implementation in this file yields the string "MyReader!"  
      // ten times. In general there are three cases:  
      //  
      // 1. If an element is successfully read, store it as one or more tensors  
      //    in `\*out\_tensors`, set `\*end\_of\_sequence = false` and return  
      //    `Status::OK()`.  
      // 2. If the end of input is reached, set `\*end\_of\_sequence = true` and  
      //    return `Status::OK()`.  
      // 3. If an error occurs, return an error status using one of the helper  
      //    functions from "tensorflow/core/lib/core/errors.h".  
      Status GetNextInternal(tensorflow::IteratorContext\* ctx,  
                             std::vector<tensorflow::Tensor>\* out\_tensors,  
                             bool\* end\_of\_sequence) override {  
        // NOTE: `GetNextInternal()` may be called concurrently, so it is  
        // recommended that you protect the iterator state with a mutex.  
        tensorflow::mutex\_lock l(mu\_);  
        if (i\_ < 10) {  
          // Create a scalar string tensor and add it to the output.  
          tensorflow::Tensor record\_tensor(ctx->allocator({}), DT\_STRING, {});  
          record\_tensor.scalar<string>()() = "MyReader!";  
          out\_tensors->emplace\_back(std::move(record\_tensor));  
          ++i\_;  
          \*end\_of\_sequence = false;  
        } else {  
          \*end\_of\_sequence = true;  
        }  
        return Status::OK();  
      }  
  
     protected:  
      // Optional: Implementation of iterator state serialization for this  
      // iterator.  
      //  
      // Implement these two methods if you want to be able to save and restore  
      // instances of this iterator.  
      Status SaveInternal(tensorflow::IteratorStateWriter\* writer) override {  
        tensorflow::mutex\_lock l(mu\_);  
        TF\_RETURN\_IF\_ERROR(writer->WriteScalar(full\_name("i"), i\_));  
        return Status::OK();  
      }  
      Status RestoreInternal(tensorflow::IteratorContext\* ctx,  
                             tensorflow::IteratorStateReader\* reader) override {  
        tensorflow::mutex\_lock l(mu\_);  
        TF\_RETURN\_IF\_ERROR(reader->ReadScalar(full\_name("i"), &i\_));  
        return Status::OK();  
      }  
  
     private:  
      tensorflow::mutex mu\_;  
      int64 i\_ GUARDED\_BY(mu\_);  
    };  
  };  
};  
  
// Register the op definition for MyReaderDataset.  
//  
// Dataset ops always have a single output, of type `variant`, which represents  
// the constructed `Dataset` object.  
//  
// Add any attrs and input tensors that define the dataset here.  
REGISTER\_OP("MyReaderDataset")  
    .Output("handle: variant")  
    .SetIsStateful()  
    .SetShapeFn(tensorflow::shape\_inference::ScalarShape);  
  
// Register the kernel implementation for MyReaderDataset.  
REGISTER\_KERNEL\_BUILDER(Name("MyReaderDataset").Device(tensorflow::DEVICE\_CPU),  
                        MyReaderDatasetOp);  
  
}  // namespace  
}  // namespace myproject

最后一步是构建C ++代码并添加Python包装器。最简单的方法是[编译一个动态库](https://tensorflow.google.cn/extend/adding_an_op#build_the_op_library)（例如，调用"my\_reader\_dataset\_op.so"），并添加一个子类[tf.data.Dataset](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset)来包装它的Python类。这里给出了一个示例Python程序：

import tensorflow as tf  
  
# Assumes the file is in the current working directory.  
my\_reader\_dataset\_module = tf.load\_op\_library("./my\_reader\_dataset\_op.so")  
  
class MyReaderDataset(tf.data.Dataset):  
  
  def \_\_init\_\_(self):  
    super(MyReaderDataset, self).\_\_init\_\_()  
    # Create any input attrs or tensors as members of this class.  
  
  def \_as\_variant\_tensor(self):  
    # Actually construct the graph node for the dataset op.  
    #  
    # This method will be invoked when you create an iterator on this dataset  
    # or a dataset derived from it.  
    return my\_reader\_dataset\_module.my\_reader\_dataset()  
  
  # The following properties define the structure of each element: a scalar  
  # <a href="./../api\_docs/python/tf#string"><code>tf.string</code></a> tensor. Change these properties to match the `output\_dtypes()`  
  # and `output\_shapes()` methods of `MyReaderDataset::Dataset` if you modify  
  # the structure of each element.  
  @property  
  def output\_types(self):  
    return tf.string  
  
  @property  
  def output\_shapes(self):  
    return tf.TensorShape([])  
  
  @property  
  def output\_classes(self):  
    return tf.Tensor  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  # Create a MyReaderDataset and print its elements.  
  with tf.Session() as sess:  
    iterator = MyReaderDataset().make\_one\_shot\_iterator()  
    next\_element = iterator.get\_next()  
    try:  
      while True:  
        print(sess.run(next\_element))  # Prints "MyReader!" ten times.  
    except tf.errors.OutOfRangeError:  
      pass

你可以看到一些Dataset包装类的 例子[tensorflow/python/data/ops/dataset\_ops.py](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/python/data/ops/dataset_ops.py)。

为记录格式编写操作

通常，这是一个普通的op，它将标量字符串记录作为输入，因此请按照[说明添加Op](https://tensorflow.google.cn/extend/adding_an_op)。您可以选择将标量字符串键作为输入，并将其包含在报告格式不正确的数据的错误消息中。这样，用户可以更轻松地追踪坏数据的来源。

用于解码记录的Ops示例：

* [tf.parse\_single\_example](https://tensorflow.google.cn/api_docs/python/tf/parse_single_example)（和[tf.parse\_example](https://tensorflow.google.cn/api_docs/python/tf/parse_example)）
* [tf.decode\_csv](https://tensorflow.google.cn/api_docs/python/tf/decode_csv)
* [tf.decode\_raw](https://tensorflow.google.cn/api_docs/python/tf/io/decode_raw)

请注意，使用多个Ops来解码特定记录格式会很有用。例如，可能必须保存为一个字符串的图像 [一个](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/example/example.proto)[tf.train.Example](https://tensorflow.google.cn/api_docs/python/tf/train/Example)协议缓冲器。根据该图像的格式，你可能会采取相应的输出从[tf.parse\_single\_example](https://tensorflow.google.cn/api_docs/python/tf/parse_single_example)OP和呼叫[tf.image.decode\_jpeg](https://tensorflow.google.cn/api_docs/python/tf/image/decode_jpeg)， [tf.image.decode\_png](https://tensorflow.google.cn/api_docs/python/tf/image/decode_png)或[tf.decode\_raw](https://tensorflow.google.cn/api_docs/python/tf/io/decode_raw)。通常采用输出[tf.decode\_raw](https://tensorflow.google.cn/api_docs/python/tf/io/decode_raw)和使用[tf.slice](https://tensorflow.google.cn/api_docs/python/tf/slice)并[tf.reshape](https://tensorflow.google.cn/api_docs/python/tf/manip/reshape)提取碎片。

TensorFlow用其他语言

背景

本文档旨在为那些对在其他编程语言中创建或开发TensorFlow功能感兴趣的人提供指导。它描述了TensorFlow的功能以及推荐的步骤，以便在其他编程语言中提供相同的功能。

Python是TensorFlow支持的第一种客户端语言，目前支持最多的功能。越来越多的功能被移植到TensorFlow的核心（用C ++实现）并通过[C API](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/c/c_api.h)公开。客户端语言应使用该语言的[外部函数接口（FFI）](https://en.wikipedia.org/wiki/Foreign_function_interface)来调用此[C API](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/c/c_api.h)以提供TensorFlow功能。

概观

在编程语言中提供TensorFlow功能可以分为多个类别：

* *运行预定义图形*：给定GraphDef（或 MetaGraphDef）协议消息，能够创建会话，运行查询，并获得张量结果。这对于想要在预训练模型上运行推理的移动应用或服务器来说已足够。
* *图形构造*：每个定义的TensorFlow操作至少有一个函数，用于向图形添加操作。理想情况下，这些函数会自动生成，因此在修改op定义时它们会保持同步。
* *梯度（AKA自动微分）*：给定图形和输入和输出操作列表，向图形添加操作，计算输入相对于输出的偏导数（梯度）。允许为图形中的特定操作自定义渐变功能。
* *功能*：定义可在主体中的多个位置调用的子图GraphDef。FunctionDef在a 中FunctionDefLibrary 包含一个GraphDef。
* *控制流*：使用用户指定的子图构造“If”和“While”。理想情况下，这些工作与渐变（见上文）。
* *神经网络库*：一些组件，它们共同支持神经网络模型的创建和训练（可能在分布式环境中）。虽然以其他语言提供此功能会很方便，但目前还没有计划以Python以外的语言支持此功能。这些库通常是上述功能的包装器。

语言绑定至少应支持运行预定义的图形，但大多数还应支持图形构造。TensorFlow Python API提供了所有这些功能。

当前状态

应该在[C API](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/c/c_api.h)之上构建新的语言支持。但是，正如您在下表中看到的那样，并非所有功能在C中都可用。在[C API中](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/c/c_api.h)提供更多功能是一个正在进行的项目。

| 特征 | 蟒蛇 | C |
| --- | --- | --- |
| 运行预定义的图表 | [tf.import\_graph\_def](https://tensorflow.google.cn/api_docs/python/tf/import_graph_def)， [tf.Session](https://tensorflow.google.cn/api_docs/python/tf/Session) | TF\_GraphImportGraphDef，TF\_NewSession |
| 具有生成的op函数的图形构造 | 是 | 是（C API支持执行此操作的客户端语言） |
| 渐变 | [tf.gradients](https://tensorflow.google.cn/api_docs/python/tf/gradients) |  |
| 功能 | tf.python.framework.function.Defun |  |
| 控制流 | [tf.cond](https://tensorflow.google.cn/api_docs/python/tf/cond)， [tf.while\_loop](https://tensorflow.google.cn/api_docs/python/tf/while_loop) |  |
| 神经网络库 | [tf.train](https://tensorflow.google.cn/api_docs/python/tf/train)，[tf.nn](https://tensorflow.google.cn/api_docs/python/tf/nn)，[tf.contrib.layers](https://tensorflow.google.cn/api_docs/python/tf/contrib/layers)，[tf.contrib.slim](https://tensorflow.google.cn/api_docs/python/tf/contrib/slim) |  |

推荐方法

运行预定义的图表

语言绑定应该定义以下类：

* Graph：表示TensorFlow计算的图形。由操作组成（用Operations 表示的客户语言），对应TF\_Graph于C API中的a。主要用作创建新Operation对象和启动时的参数Session。还支持遍历graph（TF\_GraphNextOperation）中的操作，通过name（TF\_GraphOperationByName）查找操作，以及转换为GraphDef 协议消息（TF\_GraphToGraphDef以及TF\_GraphImportGraphDef在C API中）。
* Operation：表示图中的计算节点。对应 TF\_Operation于C API中的a。
* Output：表示图中操作的一个输出。有一个 DataType（最终形状）。可以作为输入参数传递给函数，用于向图形添加操作，或者 作为张量获取该输出Session的Run()方法。对应TF\_Output于C API中的a。
* Session：表示TensorFlow运行时的特定实例的客户端。它的主要工作是用一个Graph和一些选项构建，然后Run()对图形进行字段调用。对应TF\_Session于C API中的a。
* Tensor：表示元素全部相同的N维（矩形）数组DataType。获取数据流入和流出的Session的Run()电话。对应TF\_Tensor于C API中的a。
* DataType：具有TensorFlow支持的所有可能张量类型的枚举。对应TF\_DataType于C API，通常称为dtypePython API。

图形构造

TensorFlow有很多操作，列表不是静态的，所以我们建议生成用于将操作添加到图形中的函数，而不是手动单独编写它们（尽管手动编写一些是一个很好的方法来确定生成器应该是什么生成）。生成函数所需的信息包含在OpDef协议消息中。

有几种方法可以获得OpDef已注册操作的s 列表：

* TF\_GetAllOpList在C API中检索所有已注册的OpDef协议消息。这可以用于以客户端语言编写生成器。这要求客户端语言具有协议缓冲区支持以便解释OpDef消息。
* C ++函数OpRegistry::Global()->GetRegisteredOps()返回所有已注册的OpDefs（在中定义 [tensorflow/core/framework/op.h](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/op.h)）的相同列表。这可以用来用C ++编写生成器（对于没有协议缓冲支持的语言特别有用）。
* 该列表的ASCII序列化版本[tensorflow/core/ops/ops.pbtxt](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/ops/ops.pbtxt)由自动过程定期检入 。

该OpDef规定如下：

* CamelCase中op的名称。对于生成的函数，请遵循该语言的约定。例如，如果语言使用snake\_case，则使用该语言而不是CamelCase作为op的函数名称。
* 输入和输出列表。这些类型可以通过引用属性来实现多态，如[添加操作](https://tensorflow.google.cn/extend/adding_an_op)的输入和输出部分所述 。
* 属性列表及其默认值（如果有）。请注意，其中一些将被推断（如果它们由输入确定），一些将是可选的（如果它们具有默认值），并且一些将是必需的（无默认值）。
* 一般操作的文档以及输入，输出和非推断属性。
* 运行时使用的其他一些字段可以被代码生成器忽略。

OpDef可以将An 转换为函数的文本，该函数使用TF\_OperationDescriptionC API 将该op添加到图形中（包含在语言的FFI中）：

* 从TF\_NewOperation()创建开始TF\_OperationDescription\*。
* 每次输入调用TF\_AddInput()或TF\_AddInputList()一次（取决于输入是否具有列表类型）。
* 调用TF\_SetAttr\*()函数以设置非推断属性。如果您不想覆盖默认值，可以使用默认值跳过属性。
* 必要时设置可选字段：
  + TF\_SetDevice()：强制操作到特定设备。
  + TF\_AddControlInput()：在此操作开始运行之前添加另一个操作完成的要求
  + TF\_SetAttrString("\_kernel") 设置内核标签（很少使用）
  + TF\_ColocateWith() 将一个操作与另一个操作相互配合
* TF\_FinishOperation()完成后调用。这会将操作添加到图形中，之后无法修改。

现有示例将代码生成器作为构建过程的一部分运行（使用Bazel genrule）。或者，代码生成器可以由自动cron进程运行，可能检查结果。这会在生成的代码和OpDef检入存储库的s 之间产生分歧的风险，但对于预期代码如go getGo和cargo opsfor Rust 提前生成的语言非常有用。另一方面，对于某些语言，代码可以动态生成 [tensorflow/core/ops/ops.pbtxt](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/core/ops/ops.pbtxt)。

处理常量

如果用户可以为输入参数提供常量，则调用代码将更加简洁。生成的代码应该将这些常量转换为添加到图形中的操作，并用作实例化操作的输入。

可选参数

如果语言允许函数的可选参数（如Python中的默认值的关键字参数），则将它们用于可选属性，操作名称，设备，控制输入等。在某些语言中，可以使用动态范围设置这些可选参数（如Python中的“with”块）。如果没有这些功能，库可能会使用“构建器模式”，就像在TensorFlow API的C ++版本中所做的那样。

名称范围

使用某种范围层次结构支持命名图操作是一个好主意，特别是考虑到TensorBoard依赖于它以合理的方式显示大图的事实。现有的Python和C ++ API采用不同的方法：在Python中，名称的“目录”部分（最后一个“/”）来自with块。实际上，有一个线程局部堆栈，其范围定义了名称层次结构。名称的最后一个组件由用户显式提供（使用可选的name关键字参数），或者默认为要添加的操作类型的名称。在C ++中，名称的“目录”部分存储在显式 Scope对象中。该NewSubScope()方法附加到名称的该部分并返回一个新的Scope。名称的最后一个组件是使用该WithOpName()方法设置的 ，并且像Python一样默认为要添加的op类型的名称。Scope显式传递对象以指定上下文的名称。

包装

将生成的函数保留为某些操作的私有可能是有意义的，这样可以使用执行一些额外工作的包装函数。这也提供了一个逃生舱口，用于支持生成代码范围之外的功能。

包装器的一个用途是用于支持SparseTensor输入和输出。A SparseTensor是3个密集张量的元组：索引，值和形状。值是矢量大小[n]，形状是矢量大小[rank]，index是矩阵大小[n，rank]。有一些稀疏操作使用此三元组来表示单个稀疏张量。

使用包装器的另一个原因是用于保持状态的操作。有一些这样的操作（例如变量）有几个伴随操作在该状态下操作。Python API具有这些操作的类，其中构造函数创建op，并且该类上的方法向操作状态的图添加操作。

其他考虑因素

* 最好有一个关键字列表，用于重命名操作函数和与语言关键字冲突的参数（或其他会导致问题的符号，如生成的代码中引用的库函数或变量的名称）。
* Const向图形添加操作的功能通常是包装器，因为生成的函数通常具有冗余 DataType输入。

梯度，功能和控制流程

此时，支持渐变，函数和控制流操作（“if”和“while”）在Python以外的语言中不可用。当[C API](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/c/c_api.h)提供必要的支持时，这将更新。

TensorFlow模型文件的工具开发人员

大多数用户不需要关心TensorFlow如何在磁盘上存储数据的内部细节，但如果您是工具开发人员，则可能。例如，您可能希望分析模型，或在TensorFlow和其他格式之间来回转换。本指南试图解释如何使用包含模型数据的主文件的一些细节，以便更容易地开发这些工具。

协议缓冲区

TensorFlow的所有文件格式都基于 [协议缓冲区](https://developers.google.cn/protocol-buffers/?hl=en)，因此要开始熟悉它们的工作方式。摘要是您在文本文件中定义数据结构，protobuf工具生成C，Python和其他语言的类，可以友好的方式加载，保存和访问数据。我们经常将Protocol Buffers称为protobufs，我将在本指南中使用该约定。

GraphDef

TensorFlow中计算的基础是Graph对象。这保存了节点网络，每个节点代表一个操作，彼此连接作为输入和输出。创建Graph对象后，可以通过调用将其保存as\_graph\_def()，返回GraphDef对象。

GraphDef类是ProtoBuf库根据[tensorflow / core / framework / graph.proto中](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/graph.proto)的定义创建的对象。protobuf工具解析此文本文件，并生成用于加载，存储和操作图形定义的代码。如果您看到表示模型的独立TensorFlow文件，则可能包含GraphDefprotobuf代码保存的其中一个对象的序列化版本。

此生成的代码用于从磁盘保存和加载GraphDef文件。实际加载模型的代码如下所示：

graph\_def = graph\_pb2.GraphDef()

该行创建一个空GraphDef对象，该类是从graph.proto中的文本定义创建的。这是我们将使用文件中的数据填充的对象。

with open(FLAGS.graph, "rb") as f:

这里我们得到一个文件句柄，用于我们传递给脚本的路径

  if FLAGS.input\_binary:  
    graph\_def.ParseFromString(f.read())  
  else:  
    text\_format.Merge(f.read(), graph\_def)

文字还是二进制？

实际上有两种不同的格式可以保存ProtoBuf。TextFormat是一种人类可读的形式，这使得它很适合调试和编辑，但是当存在权重的数字数据时可以变大。你可以在[graph\_run\_run2.pbtxt中](https://github.com/tensorflow/tensorboard/blob/master/tensorboard/demo/data/graph_run_run2.pbtxt)看到一个小例子 。

二进制格式文件比它们的文本等价文件小很多，即使它们对我们来说不那么可读。在这个脚本中，我们要求用户提供一个标志，指示输入文件是二进制还是文本，因此我们知道要调用的正确函数。您可以在[inception\_v3存档中](https://storage.googleapis.com/download.tensorflow.org/models/inception_v3_2016_08_28_frozen.pb.tar.gz)找到大型二进制文件的示例 ，如inception\_v3\_2016\_08\_28\_frozen.pb。

API本身可能有点令人困惑 - 实际上是二进制调用 ParseFromString()，而您使用text\_format 模块中的实用程序函数来加载文本文件。

节点

将文件加载到graph\_def变量后，您现在可以访问其中的数据。对于大多数实际目的，重要部分是存储在节点成员中的节点列表。这是循环遍历这些的代码：

for node in graph\_def.node

每个节点都是一个NodeDef对象，在 [tensorflow / core / framework / node\_def.proto中定义](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/node_def.proto)。这些是TensorFlow图的基本构建块，每个图都定义了一个操作及其输入连接。以下是a的成员NodeDef，以及他们的意思。

name

每个节点都应该有一个唯一的标识符，该标识符不会被图中的任何其他节点使用。如果您在使用Python API构建图形时没有指定一个，那么将为您选择一个反映操作名称的图形，例如“MatMul”，与单调递增的数字连接，例如“5” 。在定义节点之间的连接时，以及在运行时为整个图形设置输入和输出时，将使用该名称。

op

这定义执行何种操作，例如"Add"，"MatMul"或 "Conv2D"。运行图形时，将在注册表中查找此操作名称以查找实现。注册表由对REGISTER\_OP()宏的调用填充 ，如 [tensorflow / core / ops / nn\_ops.cc中的调用](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/ops/nn_ops.cc)。

input

字符串列表，每个字符串都是另一个节点的名称，可选地后跟冒号和输出端口号。例如，具有两个输入的节点可能具有类似的列表["some\_node\_name", "another\_node\_name"]，它等同于["some\_node\_name:0", "another\_node\_name:0"]，并将节点的第一个输入定义为具有名称的节点的第一个输出 "some\_node\_name"，以及来自第一个输出的第二个输入。"another\_node\_name"

device

在大多数情况下，您可以忽略这一点，因为它定义了在分布式环境中运行节点的位置，或者您希望将操作强制到CPU或GPU上。

attr

这是一个包含节点所有属性的键/值存储。这些是节点的永久属性，在运行时不会更改的内容，例如卷积过滤器的大小或常量操作的值。因为可以有很多不同类型的属性值，从字符串，到整数，到张量值数组，还有一个单独的protobuf文件定义了包含它们的数据结构，在 [tensorflow / core / framework / attr\_value.proto中](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/attr_value.proto)。

每个属性都有一个唯一的名称字符串，并在定义操作时列出预期的属性。如果节点中不存在属性，但它在操作定义中列出了默认值，则在创建图形时将使用该默认值。

你可以通过调用访问所有这些成员node.name，node.op等在Python。存储在其中的节点列表GraphDef是模型体系结构的完整定义。

冷冻

关于这一点的一个令人困惑的部分是在训练期间权重通常不存储在文件格式内。相反，它们被保存在单独的检查点文件中，并且图中有Variable操作在初始化时加载最新值。在部署到生产环境时，使用单独的文件通常不是很方便，因此有一个 [freeze\_graph.py](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/tools/freeze_graph.py)脚本可以获取图形定义和一组检查点，并将它们一起冻结到一个文件中。

这样做是加载GraphDef，从最新的检查点文件中提取所有变量的值，然后将每个Variableop替换Const为具有存储在其属性中的权重的数值数据的值然后去除所有无关的节点用于前向推理，并将结果保存GraphDef到输出文件中。

重量格式

如果您正在处理代表神经网络的TensorFlow模型，最常见的问题之一是提取和解释权重值。存储它们的常用方法，例如在freeze\_graph脚本创建的图形中，作为Const包含权重的ops Tensors。它们在[tensorflow / core / framework / tensor.proto](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/framework/tensor.proto)中定义 ，包含有关数据大小和类型的信息，以及值本身。在Python中， 通过调用类似的东西，TensorProto从NodeDef表示Constop 的对象中获取对象some\_node\_def.attr['value'].tensor。

这将为您提供一个表示权重数据的对象。数据本身将存储在其中一个列表中，后缀为\_val，由对象类型指示，例如float\_val32位浮点数据类型。

在不同框架之间进行转换时，卷积权重值的排序通常很难处理。在TensorFlow中，操作的过滤器权重Conv2D存储在第二个输入上，并且预期按顺序排列[filter\_height, filter\_width, input\_depth, output\_depth]，其中filter\_count增加1意味着移动到内存中的相邻值。

希望这个纲要能让您更好地了解TensorFlow模型文件中的内容，并在您需要操作它们时帮助您。

绩效指南

本指南包含一组优化TensorFlow代码的最佳实践。该指南分为几个部分：

* [一般最佳实践](https://tensorflow.google.cn/performance/performance_guide#general_best_practices)涵盖了各种模型类型和硬件中常见的主题。
* [针对与GPU](https://tensorflow.google.cn/performance/performance_guide#optimizing_for_gpu)特别相关的[GPU](https://tensorflow.google.cn/performance/performance_guide#optimizing_for_gpu)详细信息提示进行[优化](https://tensorflow.google.cn/performance/performance_guide#optimizing_for_gpu)。
* [优化CPU](https://tensorflow.google.cn/performance/performance_guide#optimizing_for_cpu)详细信息CPU特定信息。

一般最佳实践

以下部分介绍了与各种硬件和型号相关的最佳实践。最佳实践部分分为以下几个部分：

* [输入管道优化](https://tensorflow.google.cn/performance/performance_guide#input_pipeline_optimization)
* [数据格式](https://tensorflow.google.cn/performance/performance_guide#data_formats)
* [常见的融合行动](https://tensorflow.google.cn/performance/performance_guide#common_fused_ops)
* [RNN性能](https://tensorflow.google.cn/performance/performance_guide#rnn_performance)
* [从源代码构建和安装](https://tensorflow.google.cn/performance/performance_guide#building_and_installing_from_source)

输入管道优化

典型模型从磁盘检索数据并在通过网络发送数据之前对其进行预处理。例如，处理JPEG图像的模型将遵循以下流程：从磁盘加载图像，将JPEG解码为张量，裁剪和填充，可能翻转和扭曲，然后批处理。该流程称为输入管道。随着GPU和其他硬件加速器变得更快，数据的预处理可能成为瓶颈。

确定输入管道是否是瓶颈可能很复杂。最简单的方法之一是在输入管道之后将模型简化为单个操作（普通模型）并每秒测量一些示例。如果完整模型和普通模型的每秒示例差异很小，那么输入管道可能是瓶颈。以下是一些识别问题的其他方法：

* 通过运行检查GPU是否未充分利用nvidia-smi -l 2。如果GPU利用率不接近80-100％，那么输入管道可能是瓶颈。
* 生成时间轴并查找大块空白区域（等待）。生成时间轴的示例作为[XLA JIT](https://tensorflow.google.cn/performance/xla/jit) 教程的一部分存在。
* 检查CPU使用情况。可以有一个优化的输入管道，并且没有CPU周期来处理管道。
* 估计所需的吞吐量并验证所使用的磁盘是否具有该吞吐量级别。某些云解决方案具有网络连接磁盘，启动速度低至50 MB /秒，比旋转磁盘（150 MB /秒），SATA SSD（500 MB /秒）和PCIe SSD（2,000+ MB /秒）慢。

在CPU上进行预处理

在CPU上放置输入管道操作可以显着提高性能。利用CPU作为输入管道可以让GPU专注于培训。要确保在CPU上进行预处理，请执行预处理操作，如下所示：

with tf.device('/cpu:0'):  
  # function to get and process images or data.  
  distorted\_inputs = load\_and\_distort\_images()

如果使用[tf.estimator.Estimator](https://tensorflow.google.cn/api_docs/python/tf/estimator/Estimator)输入功能自动放在CPU上。

使用tf.data API

该[tf.data API](https://tensorflow.google.cn/guide/datasets)正在取代queue\_runner作为构建输入管道的建议API。此 [ResNet示例](https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10_estimator/cifar10_main.py) （[arXiv：1512.03385](https://arxiv.org/abs/1512.03385)）培训CIFAR-10说明了[tf.data](https://tensorflow.google.cn/api_docs/python/tf/data)API的使用 [tf.estimator.Estimator](https://tensorflow.google.cn/api_docs/python/tf/estimator/Estimator)。

该[tf.data](https://tensorflow.google.cn/api_docs/python/tf/data)API使用C ++多线程，并且比基于Python的开销要低得多，queue\_runner后者受到Python多线程性能的限制。[tf.data](https://tensorflow.google.cn/api_docs/python/tf/data)可以在[此处](https://tensorflow.google.cn/performance/datasets_performance)找到API的 详细性能指南。

虽然使用a feed\_dict提供数据提供了高度的灵活性，但通常feed\_dict不提供可扩展的解决方案。如果仅使用单个GPU，则[tf.data](https://tensorflow.google.cn/api_docs/python/tf/data)API与feed\_dict性能之间的差异可以忽略不计。我们的建议是避免使用feed\_dict所有但非常简单的例子。特别是，避免使用feed\_dict大输入：

# feed\_dict often results in suboptimal performance when using large inputs.  
sess.run(train\_step, feed\_dict={x: batch\_xs, y\_: batch\_ys})

融合解码和裁剪

如果输入是需要裁剪的JPEG图像，请使用融合 [tf.image.decode\_and\_crop\_jpeg](https://tensorflow.google.cn/api_docs/python/tf/image/decode_and_crop_jpeg)来加速预处理。 [tf.image.decode\_and\_crop\_jpeg](https://tensorflow.google.cn/api_docs/python/tf/image/decode_and_crop_jpeg)仅解码裁剪窗口内的图像部分。如果裁剪窗口比完整图像小得多，这会显着加快处理速度。对于imagenet数据，这种方法可以将输入管道加速高达30％。

用法示例：

def \_image\_preprocess\_fn(image\_buffer):  
    # image\_buffer 1-D string Tensor representing the raw JPEG image buffer.  
  
    # Extract image shape from raw JPEG image buffer.  
    image\_shape = tf.image.extract\_jpeg\_shape(image\_buffer)  
  
    # Get a crop window with distorted bounding box.  
    sample\_distorted\_bounding\_box = tf.image.sample\_distorted\_bounding\_box(  
      image\_shape, ...)  
    bbox\_begin, bbox\_size, distort\_bbox = sample\_distorted\_bounding\_box  
  
    # Decode and crop image.  
    offset\_y, offset\_x, \_ = tf.unstack(bbox\_begin)  
    target\_height, target\_width, \_ = tf.unstack(bbox\_size)  
    crop\_window = tf.stack([offset\_y, offset\_x, target\_height, target\_width])  
    cropped\_image = tf.image.decode\_and\_crop\_jpeg(image, crop\_window)

[tf.image.decode\_and\_crop\_jpeg](https://tensorflow.google.cn/api_docs/python/tf/image/decode_and_crop_jpeg)适用于所有平台。由于在其他平台上使用libjpegvs libjpeg-turbo，因此Windows上没有加速。

使用大文件

读取大量小文件会显着影响I / O性能。获得最大I / O吞吐量的一种方法是将输入数据预处理为更大（~100MB）的TFRecord文件。对于较小的数据集（200MB-1GB），最好的方法通常是将整个数据集加载到内存中。文档 [下载和转换为TFRecord格式](https://github.com/tensorflow/models/tree/master/research/slim#downloading-and-converting-to-tfrecord-format) 包括用于创建的信息和脚本TFRecords，此 [脚本](https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10_estimator/generate_cifar10_tfrecords.py) 将CIFAR-10数据集转换为TFRecords。

数据格式

数据格式是指传递给给定Op的Tensor的结构。下面的讨论具体是关于代表图像的4D张量。在TensorFlow中，4D张量的部分通常用以下字母表示：

* N表示批次中的图像数量。
* H指垂直（高度）维度中的像素数。
* W指水平（宽度）维度中的像素数。
* C指的是渠道。例如，1表示黑白或灰度，3表示RGB。

在TensorFlow中，有两种命名约定，代表两种最常见的数据格式：

* NCHW 要么 channels\_first
* NHWC 要么 channels\_last

NHWC是TensorFlow的默认值，NCHW是使用[cuDNN](https://developer.nvidia.com/cudnn)在NVIDIA GPU上进行培训时使用的最佳格式。

最佳实践是构建适用于两种数据格式的模型。这简化了GPU上的培训，然后在CPU上运行推理。如果使用[英特尔MKL](https://tensorflow.google.cn/performance/performance_guide#tensorflow_with_intel_mkl_dnn)优化编译TensorFlow ，则将优化和支持许多操作，尤其是与基于CNN的模型相关的操作NCHW。如果不使用MKL，则在使用时不支持CPU上的某些操作NCHW。

这两种格式的简要历史是TensorFlow开始使用， NHWC因为它在CPU上的速度要快一些。从长远来看，我们正在研究自动重写图形的工具，使格式之间的切换变得透明，并利用微优化，其中GPU Op可能NHWC比通常最高效的使用更快NCHW。

常见的融合行动

Fused Ops将多个操作组合到一个内核中以提高性能。TensorFlow中有许多融合的Ops，[XLA](https://tensorflow.google.cn/performance/xla/index)会在可能的情况下创建融合的Ops以自动提高性能。下面收集的是精选的融合操作，可以大大提高性能，可能会被忽视。

融合批量规范

融合批量规范将批量规范化所需的多个操作组合到单个内核中。批量规范是一个昂贵的过程，对于某些模型，占用了大部分操作时间。使用融合批量标准可以导致12％-30％的加速。

有两种常用的批量规范，都支持融合。核心 [tf.layers.batch\_normalization](https://tensorflow.google.cn/api_docs/python/tf/layers/batch_normalization)在TensorFlow 1.3中加入了融合。

bn = tf.layers.batch\_normalization(  
    input\_layer, fused=True, data\_format='NCHW')

[tf.contrib.layers.batch\_norm](https://tensorflow.google.cn/api_docs/python/tf/contrib/layers/batch_norm)在TensorFlow 1.0之前，contrib 方法已经融合为一个选项。

bn = tf.contrib.layers.batch\_norm(input\_layer, fused=True, data\_format='NCHW')

RNN性能

有许多方法可以在TensorFlow中指定RNN计算，并且它们在模型灵活性和性能方面具有折衷。本 [tf.nn.rnn\_cell.BasicLSTMCell](https://tensorflow.google.cn/api_docs/python/tf/nn/rnn_cell/BasicLSTMCell)应被视为一个参考实现，并仅作为最后的手段，当没有其他选择将工作。

当使用其中一个单元而不是完全融合的RNN层时，您可以选择是否使用[tf.nn.static\_rnn](https://tensorflow.google.cn/api_docs/python/tf/nn/static_rnn)或[tf.nn.dynamic\_rnn](https://tensorflow.google.cn/api_docs/python/tf/nn/dynamic_rnn)。在运行时通常不应存在性能差异，但是大量的展开量会增加图形的大小[tf.nn.static\_rnn](https://tensorflow.google.cn/api_docs/python/tf/nn/static_rnn)并导致长编译时间。另一个优点[tf.nn.dynamic\_rnn](https://tensorflow.google.cn/api_docs/python/tf/nn/dynamic_rnn)是它可以选择将内存从GPU交换到CPU，以便能够训练很长的序列。根据型号和硬件配置，这可能会带来性能损失。也可以并行运行多个迭代 [tf.nn.dynamic\_rnn](https://tensorflow.google.cn/api_docs/python/tf/nn/dynamic_rnn)和底层[tf.while\_loop](https://tensorflow.google.cn/api_docs/python/tf/while_loop)构造，尽管这对于RNN模型很少有用，因为它们本身是顺序的。

在NVIDIA GPU上，[tf.contrib.cudnn\_rnn](https://tensorflow.google.cn/api_docs/python/tf/contrib/cudnn_rnn)除非您需要不支持的图层规范化，否则应始终首选使用。它通常至少快一个数量级[tf.contrib.rnn.BasicLSTMCell](https://tensorflow.google.cn/api_docs/python/tf/nn/rnn_cell/BasicLSTMCell)，[tf.contrib.rnn.LSTMBlockCell](https://tensorflow.google.cn/api_docs/python/tf/contrib/rnn/LSTMBlockCell)并且使用的内存比内存少3-4倍[tf.contrib.rnn.BasicLSTMCell](https://tensorflow.google.cn/api_docs/python/tf/nn/rnn_cell/BasicLSTMCell)。

如果您需要一次运行RNN的一个步骤（如使用循环策略进行强化学习的情况），那么您应该[tf.contrib.rnn.LSTMBlockCell](https://tensorflow.google.cn/api_docs/python/tf/contrib/rnn/LSTMBlockCell)在[tf.while\_loop](https://tensorflow.google.cn/api_docs/python/tf/while_loop)构造中使用 自己的环境交互循环。可以一次运行RNN的一步并返回Python，但速度会慢一些。

在CPU，移动设备以及[tf.contrib.cudnn\_rnn](https://tensorflow.google.cn/api_docs/python/tf/contrib/cudnn_rnn)GPU上不可用的情况下，最快且最节省内存的选项是 [tf.contrib.rnn.LSTMBlockFusedCell](https://tensorflow.google.cn/api_docs/python/tf/contrib/rnn/LSTMBlockFusedCell)。

对于所有不常见的细胞类型，如[tf.contrib.rnn.NASCell](https://tensorflow.google.cn/api_docs/python/tf/contrib/rnn/NASCell)，[tf.contrib.rnn.PhasedLSTMCell](https://tensorflow.google.cn/api_docs/python/tf/contrib/rnn/PhasedLSTMCell)，[tf.contrib.rnn.UGRNNCell](https://tensorflow.google.cn/api_docs/python/tf/contrib/rnn/UGRNNCell)，[tf.contrib.rnn.GLSTMCell](https://tensorflow.google.cn/api_docs/python/tf/contrib/rnn/GLSTMCell)，[tf.contrib.rnn.Conv1DLSTMCell](https://tensorflow.google.cn/api_docs/python/tf/contrib/rnn/Conv1DLSTMCell)，[tf.contrib.rnn.Conv2DLSTMCell](https://tensorflow.google.cn/api_docs/python/tf/contrib/rnn/Conv2DLSTMCell)，[tf.contrib.rnn.LayerNormBasicLSTMCell](https://tensorflow.google.cn/api_docs/python/tf/contrib/rnn/LayerNormBasicLSTMCell)，等，应该意识到，他们在像图中被实现 [tf.contrib.rnn.BasicLSTMCell](https://tensorflow.google.cn/api_docs/python/tf/nn/rnn_cell/BasicLSTMCell)，因此会从相同的表现不佳以及高内存使用情况受到影响。在使用这些细胞之前，应该考虑这些权衡是否值得。例如，虽然层规范化可以加速收敛，但是因为cuDNN快20倍，所以通常在没有它的情况下获得最快的收敛时钟时间。

从源代码构建和安装

默认的TensorFlow二进制文件针对最广泛的硬件，使每个人都可以访问TensorFlow。如果使用CPU进行训练或推理，建议使用可用于所用CPU的所有优化来编译TensorFlow。下面在[比较编译器优化](https://tensorflow.google.cn/performance/performance_guide#comparing_compiler_optimizations)中记录了CPU训练和推理的加速比。

要安装最优化的TensorFlow [版本，请](https://tensorflow.google.cn/install/source)从源代码[构建和安装](https://tensorflow.google.cn/install/source)。如果需要在具有与目标不同的硬件的平台上构建TensorFlow，则使用针对目标平台的最高优化进行交叉编译。以下命令是bazel用于编译特定平台的示例：

# This command optimizes for Intel’s Broadwell processor  
bazel build -c opt --copt=-march="broadwell" --config=cuda //tensorflow/tools/pip\_package:build\_pip\_package

环境，构建和安装提示

* ./configure询问在构建中包含哪些计算能力。这不会影响整体性能，但会影响初始启动。运行TensorFlow一次后，编译的内核将被CUDA缓存。如果使用docker容器，则不会缓存数据，并且每次TensorFlow启动时都会支付罚金。最佳实践是包括 将使用的GPU 的[计算能力](http://developer.nvidia.com/cuda-gpus)，例如P100：6.0，Titan X（Pascal）：6.1，Titan X（Maxwell）：5.2和K80：3.7。
* 使用支持目标CPU的所有优化的gcc版本。推荐的最低gcc版本是4.8.3。在OS X上，升级到最新的Xcode版本并使用Xcode附带的clang版本。
* 安装TensorFlow支持的最新稳定CUDA平台和cuDNN库。

优化GPU

本节包含[一般性最佳实践](https://tensorflow.google.cn/performance/performance_guide#general_best_practices)中未涉及的特定于GPU的提示 。在多GPU上获得最佳性能是一项挑战。一种常见的方法是使用数据并行性。通过使用数据并行性进行扩展涉及制作模型的多个副本（称为“塔”），然后在每个GPU上放置一个塔。每个塔在不同的小批量数据上运行，然后更新需要在每个塔之间共享的变量（也称为参数）。每个塔如何获取更新的变量以及如何应用渐变会对模型的性能，缩放和收敛产生影响。本节的其余部分概述了可变放置和多个GPU上模型的高耸。 [高性能模型](https://tensorflow.google.cn/performance/performance_models)可以获得有关可用于在塔之间共享和更新变量的更复杂方法的更多详细信息。

处理变量更新的最佳方法取决于模型，硬件，甚至硬件的配置方式。例如，两个系统可以使用NVIDIA Tesla P100s构建，但其中一个可能使用PCIe和另一个[NVLink](http://www.nvidia.com/object/nvlink.html)。在那种情况下，每个系统的最佳解决方案可能不同。对于真实世界的示例，请阅读[基准](https://tensorflow.google.cn/performance/benchmarks)页面，其中详细说明了适用于各种平台的最佳设置。以下是从各种平台和配置基准测试中学到的内容摘要：

* **Tesla K80**：如果GPU位于同一PCI Express根复合体上并且能够使用[NVIDIA GPUDirect](https://developer.nvidia.com/gpudirect)Peer to Peer，那么将变量平均放置在用于训练的GPU上是最好的方法。如果GPU无法使用GPUDirect，则将变量放在CPU上是最佳选择。
* **Titan X（Maxwell和Pascal），M40，P100等**：对于像ResNet和InceptionV3这样的模型，在CPU上放置变量是最佳设置，但对于具有大量变量的模型，如AlexNet和VGG，使用GPU NCCL更好。

管理变量放置位置的常用方法是创建一种方法来确定每个Op的放置位置，并在调用时使用该方法代替特定的设备名称with tf.device():。考虑一种情况，即在2个GPU上训练模型，并将变量放在CPU上。将有一个循环用于在2个GPU中的每个GPU上创建和放置“塔”。自定义设备放置方法将被创建，对于类型的行动手表Variable，VariableV2以及VarHandleOp和表明它们将被放置在CPU上。所有其他Ops将被放置在目标GPU上。图表的构建将如下进行：

* 在第一个循环中，将创建模型的“塔” gpu:0。在放置Ops期间，自定义设备放置方法将指示要放置变量cpu:0和所有其他Ops gpu:0。
* 在第二个循环中，reuse设置为True指示要重用变量，然后创建“塔” gpu:1。在放置与“塔”相关联的Ops期间，cpu:0重复使用放置的变量，并创建并放置所有其他Opsgpu:1。

最终结果是所有变量都放在CPU上，每个GPU都具有与模型相关的所有计算Ops的副本。

下面的代码片段说明了两种不同的变量放置方法：一种是在CPU上放置变量; 另一种是在GPU中平均放置变量。

class GpuParamServerDeviceSetter(object):  
  """Used with tf.device() to place variables on the least loaded GPU.  
  
    A common use for this class is to pass a list of GPU devices, e.g. ['gpu:0',  
    'gpu:1','gpu:2'], as ps\_devices.  When each variable is placed, it will be  
    placed on the least loaded gpu. All other Ops, which will be the computation  
    Ops, will be placed on the worker\_device.  
  """  
  
  def \_\_init\_\_(self, worker\_device, ps\_devices):  
    """Initializer for GpuParamServerDeviceSetter.  
    Args:  
      worker\_device: the device to use for computation Ops.  
      ps\_devices: a list of devices to use for Variable Ops. Each variable is  
      assigned to the least loaded device.  
    """  
    self.ps\_devices = ps\_devices  
    self.worker\_device = worker\_device  
    self.ps\_sizes = [0] \* len(self.ps\_devices)  
  
  def \_\_call\_\_(self, op):  
    if op.device:  
      return op.device  
    if op.type not in ['Variable', 'VariableV2', 'VarHandleOp']:  
      return self.worker\_device  
  
    # Gets the least loaded ps\_device  
    device\_index, \_ = min(enumerate(self.ps\_sizes), key=operator.itemgetter(1))  
    device\_name = self.ps\_devices[device\_index]  
    var\_size = op.outputs[0].get\_shape().num\_elements()  
    self.ps\_sizes[device\_index] += var\_size  
  
    return device\_name  
  
def \_create\_device\_setter(is\_cpu\_ps, worker, num\_gpus):  
  """Create device setter object."""  
  if is\_cpu\_ps:  
    # tf.train.replica\_device\_setter supports placing variables on the CPU, all  
    # on one GPU, or on ps\_servers defined in a cluster\_spec.  
    return tf.train.replica\_device\_setter(  
        worker\_device=worker, ps\_device='/cpu:0', ps\_tasks=1)  
  else:  
    gpus = ['/gpu:%d' % i for i in range(num\_gpus)]  
    return ParamServerDeviceSetter(worker, gpus)  
  
# The method below is a modified snippet from the full example.  
def \_resnet\_model\_fn():  
    # When set to False, variables are placed on the least loaded GPU. If set  
    # to True, the variables will be placed on the CPU.  
    is\_cpu\_ps = False  
  
    # Loops over the number of GPUs and creates a copy ("tower") of the model on  
    # each GPU.  
    for i in range(num\_gpus):  
      worker = '/gpu:%d' % i  
      # Creates a device setter used to determine where Ops are to be placed.  
      device\_setter = \_create\_device\_setter(is\_cpu\_ps, worker, FLAGS.num\_gpus)  
      # Creates variables on the first loop.  On subsequent loops reuse is set  
      # to True, which results in the "towers" sharing variables.  
      with tf.variable\_scope('resnet', reuse=bool(i != 0)):  
        with tf.name\_scope('tower\_%d' % i) as name\_scope:  
          # tf.device calls the device\_setter for each Op that is created.  
          # device\_setter returns the device the Op is to be placed on.  
          with tf.device(device\_setter):  
            # Creates the "tower".  
            \_tower\_fn(is\_training, weight\_decay, tower\_features[i],  
                      tower\_labels[i], tower\_losses, tower\_gradvars,  
                      tower\_preds, False)

在不久的将来，上述代码仅用于说明目的，因为将有易于使用的高级方法来支持广泛的流行方法。 随着API的扩展和演进，此 [示例](https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10_estimator)将继续更新，以解决多GPU方案。

优化CPU

包含英特尔®至强融核™在内的CPU在使用目标CPU支持的所有指令[从源构建](https://tensorflow.google.cn/install/source) TensorFlow时可实现最佳性能。

除了使用最新的指令集，英特尔还增加了对TensorFlow的深度神经网络英特尔®数学核心库（英特尔®MKL-DNN）的支持。虽然名称不完全准确，但这些优化通常简称为“MKL”或“TensorFlow with MKL”。[带有英特尔®MKL-DNN的TensorFlow](https://tensorflow.google.cn/performance/performance_guide#tensorflow_with_intel_mkl_dnn)包含有关MKL优化的详细信息。

下面列出的两种配置用于通过调整线程池来优化CPU性能。

* intra\_op\_parallelism\_threads：可以使用多个线程并行化其执行的节点将计划各个部分进入此池。
* inter\_op\_parallelism\_threads：所有就绪节点都在此池中进行调度。

这些配置通过设置[tf.ConfigProto](https://tensorflow.google.cn/api_docs/python/tf/ConfigProto)并传递给[tf.Session](https://tensorflow.google.cn/api_docs/python/tf/Session) 在config属性作为显示在下面的代码段。对于这两个配置选项，如果未设置或设置为0，则默认为逻辑CPU核心数。测试表明，默认设置对于从具有4个核心的一个CPU到具有70个以上组合逻辑核心的多个CPU的系统是有效的。常见的替代优化是将两个池中的线程数设置为等于物理核心数而不是逻辑核心数。

  config = tf.ConfigProto()  
  config.intra\_op\_parallelism\_threads = 44  
  config.inter\_op\_parallelism\_threads = 44  
  tf.Session(config=config)

“ [比较编译器优化”](https://tensorflow.google.cn/performance/performance_guide#comparing_compiler_optimizations) 部分包含使用不同编译器优化的测试结果。

采用英特尔®MKLDNN的TensorFlow

英特尔®通过使用用于深度神经网络的英特尔®数学核心库（英特尔®MKL-DNN）优化基元，为英特尔®至强®和英特尔®至强融核™的TensorFlow添加了优化。优化还为消费者系列的处理器提供了加速，例如i5和i7 Intel处理器。英特尔发布的 [有关现代英特尔®架构的TensorFlow \*优化文章](https://software.intel.com/en-us/articles/tensorflow-optimizations-on-modern-intel-architecture) 包含有关实施的其他详细信息。

**注意：**MKL是从TensorFlow 1.2开始添加的，目前仅适用于Linux。它也在使用时也不起作用**--config=cuda**。

除了为训练基于CNN的模型提供显着的性能改进之外，使用MKL进行编译还可以创建针对AVX和AVX2优化的二进制文件。结果是单个二进制文件经过优化并与大多数现代（2011年后）处理器兼容。

TensorFlow可以使用以下命令使用MKL优化进行编译，这些命令取决于所使用的TensorFlow源的版本。

对于1.3.0之后的TensorFlow源版本：

./configure  
# Pick the desired options  
bazel build --config=mkl --config=opt //tensorflow/tools/pip\_package:build\_pip\_package

对于TensorFlow版本1.2.0到1.3.0：

./configure  
Do you wish to build TensorFlow with MKL support? [y/N] Y  
Do you wish to download MKL LIB from the web? [Y/n] Y  
# Select the defaults for the rest of the options.  
  
bazel build --config=mkl --copt="-DEIGEN\_USE\_VML" -c opt //tensorflow/tools/pip\_package:build\_pip\_package

调整MKL以获得最佳性能

本节详细介绍了可用于调整MKL以获得最佳性能的不同配置和环境变量。在调整各种环境变量之前，请确保模型使用NCHW（channels\_first） [数据格式](https://tensorflow.google.cn/performance/performance_guide#data_formats)。MKL经过优化NCHW，英特尔正在努力在使用时获得接近性能的平衡NHWC。

MKL使用以下环境变量来调整性能：

* KMP\_BLOCKTIME - 设置线程在完成并行区域的执行之后，在休眠之前应该等待的时间（以毫秒为单位）。
* KMP\_AFFINITY - 使运行时库能够将线程绑定到物理处理单元。
* KMP\_SETTINGS - 在程序执行期间启用（true）或禁用（false）打印OpenMP \*运行时库环境变量。
* OMP\_NUM\_THREADS - 指定要使用的线程数。

有关KMP变量的更多详细信息，请访问 [英特尔](https://software.intel.com/en-us/node/522775)网站和[gnu.org](https://gcc.gnu.org/onlinedocs/libgomp/Environment-Variables.html)上的OMP变量

虽然调整环境变量可以获得大量收益，这将在​​下面讨论，但简化的建议是将inter\_op\_parallelism\_threads物理CPU的数量设置为 等于并设置以下环境变量：

* KMP\_BLOCKTIME = 0
* KMP\_AFFINITY =粒度=细，冗长，紧凑，1,0

使用命令行参数设置MKL变量的示例：

KMP\_BLOCKTIME=0 KMP\_AFFINITY=granularity=fine,verbose,compact,1,0 \  
KMP\_SETTINGS=1 python your\_python\_script.py

使用python设置MKL变量的示例os.environ：

os.environ["KMP\_BLOCKTIME"] = str(FLAGS.kmp\_blocktime)  
os.environ["KMP\_SETTINGS"] = str(FLAGS.kmp\_settings)  
os.environ["KMP\_AFFINITY"]= FLAGS.kmp\_affinity  
if FLAGS.num\_intra\_threads > 0:  
  os.environ["OMP\_NUM\_THREADS"]= str(FLAGS.num\_intra\_threads)

有些模型和硬件平台可以从不同的设置中受益。下面讨论影响性能的每个变量。

* **KMP\_BLOCKTIME**：MKL默认值为200ms，这在我们的测试中并不是最佳的。对于经过测试的CNN模型，0（0ms）是一个很好的默认值。AlexNex的最佳性能是在30ms时实现的，而GoogleNet和VGG11的最佳性能均为1ms。
* **KMP\_AFFINITY**：推荐设置为 granularity=fine,verbose,compact,1,0。
* **OMP\_NUM\_THREADS**：默认为物理核心数。某些型号使用英特尔®至强融合™（骑士登陆）时，调整此参数以匹配核心数量会产生影响。请参阅 [现代英特尔®架构上的TensorFlow \*优化，](https://software.intel.com/en-us/articles/tensorflow-optimizations-on-modern-intel-architecture) 以获得最佳设置。
* **intra\_op\_parallelism\_threads**：建议将此值设置为等于物理内核的数量。将值设置为0（默认值）会导致将值设置为逻辑核心数 - 这是尝试某些体系结构的备用选项。这个值OMP\_NUM\_THREADS 应该相等。
* **inter\_op\_parallelism\_threads**：建议将此值设置为等于套接字数。将值设置为0（默认值）会导致将值设置为逻辑核心数。

比较编译器优化

下面收集的是通过各种编译器优化在不同平台上运行培训和推断不同类型CPU的性能结果。使用的模型是ResNet-50（[arXiv：1512.03385](https://arxiv.org/abs/1512.03385)）和InceptionV3（[arXiv：1512.00567](https://arxiv.org/abs/1512.00567)）。

对于每个测试，当使用MKL优化时，环境变量KMP\_BLOCKTIME设置为0（0ms）并且KMP\_AFFINITY设置为 granularity=fine,verbose,compact,1,0。

推论InceptionV3

**环境**

* 实例类型：AWS EC2 m4.xlarge
* CPU：Intel（R）Xeon（R）CPU E5-2686 v4 @ 2.30GHz（Broadwell）
* 数据集：ImageNet
* TensorFlow版本：1.2.0 RC2
* 测试脚本：[tf\_cnn\_benchmarks.py](https://github.com/tensorflow/benchmarks/blob/mkl_experiment/scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py)

**批量大小：1**

为MKL测试执行的命令：

python tf\_cnn\_benchmarks.py --forward\_only=True --device=cpu --mkl=True \  
--kmp\_blocktime=0 --nodistortions --model=inception3 --data\_format=NCHW \  
--batch\_size=1 --num\_inter\_threads=1 --num\_intra\_threads=4 \  
--data\_dir=<path to ImageNet TFRecords>

| 优化 | 数据格式 | 图像/秒（步时） | 内线程 | Inter Threads |
| --- | --- | --- | --- | --- |
| AVX2 | NHWC | 7.0（142ms） | 4 | 0 |
| MKL | NCHW | 6.6（152ms） | 4 | 1 |
| AVX | NHWC | 5.0（202ms） | 4 | 0 |
| SSE3 | NHWC | 2.8（361ms） | 4 | 0 |

**批量大小：32**

为MKL测试执行的命令：

python tf\_cnn\_benchmarks.py --forward\_only=True --device=cpu --mkl=True \  
--kmp\_blocktime=0 --nodistortions --model=inception3 --data\_format=NCHW \  
--batch\_size=32 --num\_inter\_threads=1 --num\_intra\_threads=4 \  
--data\_dir=<path to ImageNet TFRecords>

| 优化 | 数据格式 | 图像/秒（步时） | 内线程 | Inter Threads |
| --- | --- | --- | --- | --- |
| MKL | NCHW | 10.3（3,104ms） | 4 | 1 |
| AVX2 | NHWC | 7.5（4,255ms） | 4 | 0 |
| AVX | NHWC | 5.1（6,275ms） | 4 | 0 |
| SSE3 | NHWC | 2.8（11,428ms） | 4 | 0 |

推论ResNet-50

**环境**

* 实例类型：AWS EC2 m4.xlarge
* CPU：Intel（R）Xeon（R）CPU E5-2686 v4 @ 2.30GHz（Broadwell）
* 数据集：ImageNet
* TensorFlow版本：1.2.0 RC2
* 测试脚本：[tf\_cnn\_benchmarks.py](https://github.com/tensorflow/benchmarks/blob/mkl_experiment/scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py)

**批量大小：1**

为MKL测试执行的命令：

python tf\_cnn\_benchmarks.py --forward\_only=True --device=cpu --mkl=True \  
--kmp\_blocktime=0 --nodistortions --model=resnet50 --data\_format=NCHW \  
--batch\_size=1 --num\_inter\_threads=1 --num\_intra\_threads=4 \  
--data\_dir=<path to ImageNet TFRecords>

| 优化 | 数据格式 | 图像/秒（步时） | 内线程 | Inter Threads |
| --- | --- | --- | --- | --- |
| AVX2 | NHWC | 8.8（113ms） | 4 | 0 |
| MKL | NCHW | 8.5（120ms） | 4 | 1 |
| AVX | NHWC | 6.4（157ms） | 4 | 0 |
| SSE3 | NHWC | 3.7（270ms） | 4 | 0 |

**批量大小：32**

为MKL测试执行的命令：

python tf\_cnn\_benchmarks.py --forward\_only=True --device=cpu --mkl=True \  
--kmp\_blocktime=0 --nodistortions --model=resnet50 --data\_format=NCHW \  
--batch\_size=32 --num\_inter\_threads=1 --num\_intra\_threads=4 \  
--data\_dir=<path to ImageNet TFRecords>

| 优化 | 数据格式 | 图像/秒（步时） | 内线程 | Inter Threads |
| --- | --- | --- | --- | --- |
| MKL | NCHW | 12.4（2,590ms） | 4 | 1 |
| AVX2 | NHWC | 10.4（3,079ms） | 4 | 0 |
| AVX | NHWC | 7.3（4,4416ms） | 4 | 0 |
| SSE3 | NHWC | 4.0（8,054ms） | 4 | 0 |

培训InceptionV3

**环境**

* 实例类型：专用AWS EC2 r4.16xlarge（Broadwell）
* CPU：Intel Xeon E5-2686 v4（Broadwell）处理器
* 数据集：ImageNet
* TensorFlow版本：1.2.0 RC2
* 测试脚本：[tf\_cnn\_benchmarks.py](https://github.com/tensorflow/benchmarks/blob/mkl_experiment/scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py)

为MKL测试执行的命令：

python tf\_cnn\_benchmarks.py --device=cpu --mkl=True --kmp\_blocktime=0 \  
--nodistortions --model=resnet50 --data\_format=NCHW --batch\_size=32 \  
--num\_inter\_threads=2 --num\_intra\_threads=36 \  
--data\_dir=<path to ImageNet TFRecords>

| 优化 | 数据格式 | 张/秒 | 内线程 | Inter Threads |
| --- | --- | --- | --- | --- |
| MKL | NCHW | 20.8 | 36 | 2 |
| AVX2 | NHWC | 6.2 | 36 | 0 |
| AVX | NHWC | 5.7 | 36 | 0 |
| SSE3 | NHWC | 4.3 | 36 | 0 |

ResNet和[AlexNet](http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf) 也以这种配置运行，但是以临时方式运行。没有足够的运行来发布连贯的结果表。不完整的结果强烈表明最终结果与上表类似，MKL比AVX2提供了显着的3倍+增益。

输入管道性能指南

GPU和TPU可以从根本上减少执行单个训练步骤所需的时间。实现最高性能需要高效的输入管道，在当前步骤完成之前为下一步提供数据。该 [tf.data](https://tensorflow.google.cn/api_docs/python/tf/data)API有助于建立灵活，高效的输入管道。本文档介绍了[tf.data](https://tensorflow.google.cn/api_docs/python/tf/data)API在各种模型和加速器上构建高性能TensorFlow输入管道的功能和最佳实践。

本指南执行以下操作：

* 说明TensorFlow输入管道本质上是一个 [ETL](https://en.wikipedia.org/wiki/Extract,_transform,_load)过程。
* 描述[tf.data](https://tensorflow.google.cn/api_docs/python/tf/data) API 上下文中的常见性能优化。
* 讨论应用转换的顺序的性能影响。
* 总结了设计高性能TensorFlow输入管道的最佳实践。

输入管道结构

典型的TensorFlow培训输入管道可以构建为ETL过程：

1. **提取**：从持久存储中读取数据 - 本地（例如HDD或SSD）或远程存储（例如[GCS](https://cloud.google.com/storage/)或[HDFS](https://en.wikipedia.org/wiki/Apache_Hadoop#Hadoop_distributed_file_system)）。
2. **转换**：使用CPU内核对数据进行解析和执行预处理操作，例如图像解压缩，数据扩充转换（例如随机裁剪，翻转和颜色失真），混洗和批处理。
3. **加载**：将转换后的数据加载到执行机器学习模型的加速器设备（例如，GPU或TPU）上。

这种模式有效地利用了CPU，同时保留了加速器，用于训练模型。此外，将输入管道视为ETL过程提供了便于性能优化应用的结构。

使用[tf.estimator.Estimator](https://tensorflow.google.cn/api_docs/python/tf/estimator/Estimator)API时，前两个阶段（Extract和Transform）将在input\_fn传递给的时候 捕获[tf.estimator.Estimator.train](https://tensorflow.google.cn/api_docs/python/tf/estimator/Estimator#train)。在代码中，这可能看起来像以下（幼稚，顺序）实现：

def parse\_fn(example):  
  "Parse TFExample records and perform simple data augmentation."  
  example\_fmt = {  
    "image": tf.FixedLengthFeature((), tf.string, ""),  
    "label": tf.FixedLengthFeature((), tf.int64, -1)  
  }  
  parsed = tf.parse\_single\_example(example, example\_fmt)  
  image = tf.image.decode\_image(parsed["image"])  
  image = \_augment\_helper(image)  # augments image using slice, reshape, resize\_bilinear  
  return image, parsed["label"]  
  
def input\_fn():  
  files = tf.data.Dataset.list\_files("/path/to/dataset/train-\*.tfrecord")  
  dataset = files.interleave(tf.data.TFRecordDataset)  
  dataset = dataset.shuffle(buffer\_size=FLAGS.shuffle\_buffer\_size)  
  dataset = dataset.map(map\_func=parse\_fn)  
  dataset = dataset.batch(batch\_size=FLAGS.batch\_size)  
  return dataset

下一部分将基于此输入管道构建，添加性能优化。

优化性能

随着新的计算设备（例如GPU和TPU）使得以越来越快的速度训练神经网络成为可能，CPU处理很容易成为瓶颈。所述[tf.data](https://tensorflow.google.cn/api_docs/python/tf/data)API为用户提供了构建块来设计有效地利用CPU，优化ETL过程的每个步骤输入管线。

流水线

要执行训练步骤，您必须首先提取并转换训练数据，然后将其提供给在加速器上运行的模型。但是，在一个简单的同步实现中，当CPU正在准备数据时，加速器处于空闲状态。相反，当加速器正在训练模型时，CPU处于空闲状态。因此，训练步骤时间是CPU预处理时间和加速器训练时间的总和。

**流水线操作**与训练步骤的预处理和模型执行重叠。当加速器正在执行训练步骤时N，CPU正在准备步骤的数据N+1。这样做可以将步骤时间缩短到训练的最大值（而不是总和）以及提取和转换数据所需的时间。

如果没有流水线操作，CPU和GPU / TPU大部分时间处于空闲状态：

使用流水线技术，空闲时间显着减少：

所述[tf.data](https://tensorflow.google.cn/api_docs/python/tf/data)API提供通过一个软件流水线机构 [tf.data.Dataset.prefetch](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset#prefetch)变换，其可被用于去耦数据从它被消耗时产生的时间。特别是，转换使用后台线程和内部缓冲区，以便在请求输入数据集之前从输入数据集中预取元素。因此，要实现上述流水线效果，您可以将prefetch(1)最终转换添加到数据集管道（或者prefetch(n)如果单个训练步骤消耗n个元素）。

要将此更改应用于我们的运行示例，请更改：

dataset = dataset.batch(batch\_size=FLAGS.batch\_size)  
return dataset

至：

dataset = dataset.batch(batch\_size=FLAGS.batch\_size)  
dataset = dataset.prefetch(buffer\_size=FLAGS.prefetch\_buffer\_size)  
return dataset

请注意，只要有机会将“生产者”的工作与“消费者”的工作重叠，预取转换就会产生效益。前面的建议只是最常见的应用程序。

并行化数据转换

准备批处理时，可能需要预处理输入元素。为此，[tf.data](https://tensorflow.google.cn/api_docs/python/tf/data)API提供[tf.data.Dataset.map](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset#map)转换，该转换将用户定义的函数（例如，parse\_fn从运行的示例）应用于输入数据集的每个元素。由于输入元素彼此独立，因此可以跨多个CPU内核并行化预处理。为了实现这一点，map转换提供了num\_parallel\_calls指定并行度的 参数。例如，下图说明设置的效果num\_parallel\_calls=2 的map变换：

为num\_parallel\_calls参数选择最佳值取决于您的硬件，训练数据的特征（例如其大小和形状），地图功能的成本以及CPU同时发生的其他处理; 一个简单的启发式方法是使用可用的CPU核心数。例如，如果执行上述示例的机器有4个内核，则设置效率会更高num\_parallel\_calls=4。另一方面，设置num\_parallel\_calls为远大于可用CPU数量的值可能导致调度效率低下，从而导致速度减慢。

要将此更改应用于我们的运行示例，请更改：

dataset = dataset.map(map\_func=parse\_fn)

至：

dataset = dataset.map(map\_func=parse\_fn, num\_parallel\_calls=FLAGS.num\_parallel\_calls)

此外，如果您的批量大小为数百或数千，您的管道可能还可以通过并行化批量创建而获益。为此，[tf.data](https://tensorflow.google.cn/api_docs/python/tf/data)API提供了[tf.contrib.data.map\_and\_batch](https://tensorflow.google.cn/api_docs/python/tf/contrib/data/map_and_batch) 转换，有效地“融合”了地图和批量转换。

要将此更改应用于我们的运行示例，请更改：

dataset = dataset.map(map\_func=parse\_fn, num\_parallel\_calls=FLAGS.num\_parallel\_calls)  
dataset = dataset.batch(batch\_size=FLAGS.batch\_size)

至：

dataset = dataset.apply(tf.contrib.data.map\_and\_batch(  
    map\_func=parse\_fn, batch\_size=FLAGS.batch\_size))

并行化数据提取

在实际环境中，输入数据可能被远程存储（例如，GCS或HDFS），因为输入数据不适合本地，或者因为训练是分布式的，因此复制输入数据是没有意义的。每台机器。在本地读取数据时运行良好的数据集管道在远程读取数据时可能会成为I / O的瓶颈，因为本地存储和远程存储之间存在以下差异：

* **从第一个字节开始：**从远程存储器读取文件的第一个字节可能比从本地存储器中读取的时间长几个数量级。
* **读取吞吐量：**虽然远程存储通常提供较大的聚合带宽，但读取单个文件可能只能利用此带宽的一小部分。

此外，一旦将原始字节读入存储器，也可能需要对数据进行反序列化或解密（例如，[protobuf](https://developers.google.cn/protocol-buffers/)），这会增加额外的开销。无论数据是本地存储还是远程存储，都存在这种开销，但如果数据未被有效预取，则在远程情况下可能更糟。

为了减轻各种数据提取开销的影响，[tf.data](https://tensorflow.google.cn/api_docs/python/tf/data) API提供了[tf.contrib.data.parallel\_interleave](https://tensorflow.google.cn/api_docs/python/tf/contrib/data/parallel_interleave)转换。使用此转换可以并行化其他数据集（例如数据文件读取器）的内容的执行和交错。可以通过cycle\_length参数指定要重叠的数据集的数量。

下图说明了cycle\_length=2为parallel\_interleave转换提供的效果：

要将此更改应用于我们的运行示例，请更改：

dataset = files.interleave(tf.data.TFRecordDataset)

至：

dataset = files.apply(tf.contrib.data.parallel\_interleave(  
    tf.data.TFRecordDataset, cycle\_length=FLAGS.num\_parallel\_readers))

由于负载或网络事件，远程存储系统的吞吐量可能会随时间而变化。为了解释这种差异，parallel\_interleave 转换可以选择使用预取。（详见[tf.contrib.data.parallel\_interleave](https://tensorflow.google.cn/api_docs/python/tf/contrib/data/parallel_interleave)）。

默认情况下，parallel\_interleave转换提供元素的确定性排序以帮助重现性。作为预取的替代方法（在某些情况下可能无效），parallel\_interleave 转换还提供了一种选项，可以以订购保证为代价提高性能。特别是，如果sloppy参数设置为true，则转换可能会偏离其确定性排序，暂时跳过请求下一个元素时元素不可用的文件。

性能注意事项

该[tf.data](https://tensorflow.google.cn/api_docs/python/tf/data)API的设计是围绕组合的变换来为其用户提供具有灵活性。虽然这些转换中的许多都是可交换的，但某些转换的顺序具有性能影响。

地图和批处理

调用传递给map转换的用户定义函数会产生与调度和执行用户定义函数相关的开销。通常，与函数执行的计算量相比，这种开销很小。但是，如果map不起作用，这种开销可能会占据总成本。在这种情况下，我们建议对矢量化用户定义的函数（即，有它在一次工作在批输入）和应用 batch改造*之前*的map改造。

地图和缓存

该[tf.data.Dataset.cache](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset#cache)改造可以缓存的数据集，在内存或本地存储。如果传递给map 转换的用户定义函数很昂贵，只要生成的数据集仍然适合内存或本地存储，就可以在映射转换后应用缓存转换。如果用户定义的函数增加了存储数据集所需的空间超出缓存容量，请考虑在训练作业之前预处理数据以减少资源使用。

Map和Interleave / Prefetch / Shuffle

许多转换，包括interleave，prefetch和shuffle维护元素的内部缓冲区。如果传递给map转换的用户定义函数改变了元素的大小，那么映射转换的顺序和缓冲元素的转换会影响内存使用。通常，我们建议选择导致较低内存占用的顺序，除非性能需要不同的顺序（例如，启用映射和批转换的融合）。

重复和随机播放

的[tf.data.Dataset.repeat](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset#repeat)变换重复的输入数据的次数有限（或无限的）号码; 每次重复数据通常称为*时期*。该[tf.data.Dataset.shuffle](https://tensorflow.google.cn/api_docs/python/tf/data/Dataset#shuffle)改造的随机化数据集的例子的顺序。

如果在repeat变换之前应用shuffle变换，则时间边界模糊。也就是说，某些元素可以在其他元素出现之前重复一次。另一方面，如果在shuffle 重复变换之前应用变换，则性能可能在与shuffle变换的内部状态的初始化相关的每个时期的开始时减慢。换句话说，前者（repeat之前shuffle）提供了更好的性能，而后者（shuffle之前repeat）提供了更强的排序保证。

如果可能，我们建议使用融合 [tf.contrib.data.shuffle\_and\_repeat](https://tensorflow.google.cn/api_docs/python/tf/contrib/data/shuffle_and_repeat)转换，它结合了两全其美（良好的性能和强大的订购保证）。否则，我们建议在重复之前进行改组。

最佳实践摘要

以下是设计输入管道的最佳实践摘要：

* 使用prefetch转换来重叠生产者和消费者的工作。特别是，我们建议将prefetch（n）（其中n是训练步骤消耗的元素/批次数）添加到输入管道的末尾，以便在CPU上执行的转换与加速器上的训练重叠。
* map通过设置num\_parallel\_calls 参数来并行化转换。我们建议使用可用CPU核心数作为其值。
* 如果您使用batch 转换将预处理元素组合成批处理，我们建议使用融合map\_and\_batch转换; 特别是如果您使用大批量。
* 如果您正在处理远程存储的数据和/或需要反序列化，我们建议使用parallel\_interleave 转换来重叠来自不同文件的数据的读取（和反序列化）。
* 向量化传递给map转换的廉价用户定义函数，以分摊与调度和执行函数相关的开销。
* 如果您的数据可以适合内存，请使用cache转换在第一个纪元期间将其缓存在内存中，以便后续纪元可以避免与读取，解析和转换相关的开销。
* 如果您的前处理增加了你的数据的大小，我们建议应用interleave，prefetch以及shuffle第一（如果可能的话），以减少内存使用。
* 建议应用shuffle改造*之前*的repeat 改造，最理想的是采用融合的shuffle\_and\_repeat转型。

# TensorFlow.js中的核心概念

**TensorFlow.js**是一个用于机器智能的开源WebGL加速JavaScript库。它为您的指尖带来了高性能的机器学习构建块，允许您在浏览器中训练神经网络或在推理模式下运行预先训练的模型。有关安装/配置TensorFlow.js的指南，请参阅“ [入门”](https://js.tensorflow.org/index.html#getting-started)。

TensorFlow.js提供用于机器学习的低级构建块以及用于构建神经网络的高级Keras启发式API。我们来看看该库的一些核心组件。

## 张量

TensorFlow.js中的中心数据单位是张量：一组数值，形状为一个或多个维度的数组。甲[Tensor](https://js.tensorflow.org/api/latest/index.html#class:Tensor)实例有一个shape定义该阵列形状属性（即，有多少个值是在所述阵列的每一维）。

主要Tensor构造函数是[tf.tensor](https://js.tensorflow.org/api/latest/index.html#tensor)函数：

*// 2x3 Tensor*

const shape = [2, 3]; *// 2 rows, 3 columns*

const a = tf.tensor([1.0, 2.0, 3.0, 10.0, 20.0, 30.0], shape);

a.print(); *// print Tensor values*

*// Output: [[1 , 2 , 3 ],*

*// [10, 20, 30]]*

*// The shape can also be inferred:*

const b = tf.tensor([[1.0, 2.0, 3.0], [10.0, 20.0, 30.0]]);

b.print();

*// Output: [[1 , 2 , 3 ],*

*// [10, 20, 30]]*

然而，构建低秩张量，我们建议您使用以下功能来提高代码的可读性：[tf.scalar](https://js.tensorflow.org/api/latest/index.html#scalar)，[tf.tensor1d](https://js.tensorflow.org/api/latest/index.html#tensor1d)，[tf.tensor2d](https://js.tensorflow.org/api/latest/index.html#tensor2d)，[tf.tensor3d](https://js.tensorflow.org/api/latest/index.html#tensor3d)和[tf.tensor4d](https://js.tensorflow.org/api/latest/index.html#tensor4d)。

以下示例使用以下内容创建与上面相同的张量tf.tensor2d：

const c = tf.tensor2d([[1.0, 2.0, 3.0], [10.0, 20.0, 30.0]]);

c.print();

*// Output: [[1 , 2 , 3 ],*

*// [10, 20, 30]]*

TensorFlow.js还提供了方便的功能，用于创建张量，所有值设置为0（[tf.zeros](https://js.tensorflow.org/api/latest/index.html#zeros)）或所有值设置为1（[tf.ones](https://js.tensorflow.org/api/latest/index.html#ones)）：

*// 3x5 Tensor with all values set to 0*

const zeros = tf.zeros([3, 5]);

*// Output: [[0, 0, 0, 0, 0],*

*// [0, 0, 0, 0, 0],*

*// [0, 0, 0, 0, 0]]*

在TensorFlow.js中，张量是不可变的; 一旦创建，您就无法更改其值。而是对它们执行生成新张量的操作。

## 变量

[Variables](https://js.tensorflow.org/api/latest/index.html#class:Variable)用值张量初始化。Tensor然而，与s 不同，它们的值是可变的。您可以使用以下assign方法为现有变量指定新的张量：

const initialValues = tf.zeros([5]);

const biases = tf.variable(initialValues); *// initialize biases*

biases.print(); *// output: [0, 0, 0, 0, 0]*

const updatedValues = tf.tensor1d([0, 1, 0, 1, 0]);

biases.assign(updatedValues); *// update values of biases*

biases.print(); *// output: [0, 1, 0, 1, 0]*

变量主要用于在模型训练期间存储然后更新值。

## 运营（行动）

虽然张量允许您存储数据，但操作（操作）允许您操作该数据。TensorFlow.js提供了多种适用于线性代数和机器学习的运算，可以在张量上执行。因为张量是不可变的，所以这些操作不会改变它们的值; 相反，ops返回新的张量。

可用的操作包括一元操作，例如[square](https://js.tensorflow.org/api/latest/index.html#square)：

const d = tf.tensor2d([[1.0, 2.0], [3.0, 4.0]]);

const d\_squared = d.square();

d\_squared.print();

*// Output: [[1, 4 ],*

*// [9, 16]]*

而如二进制OPS [add](https://js.tensorflow.org/api/latest/index.html#add)，[sub](https://js.tensorflow.org/api/latest/index.html#sub)以及[mul](https://js.tensorflow.org/api/latest/index.html#mul)：

const e = tf.tensor2d([[1.0, 2.0], [3.0, 4.0]]);

const f = tf.tensor2d([[5.0, 6.0], [7.0, 8.0]]);

const e\_plus\_f = e.add(f);

e\_plus\_f.print();

*// Output: [[6 , 8 ],*

*// [10, 12]]*

TensorFlow.js有一个可链接的API; 你可以在ops的结果上调用ops：

const sq\_sum = e.add(f).square();

sq\_sum.print();

*// Output: [[36 , 64 ],*

*// [100, 144]]*

*// All operations are also exposed as functions in the main namespace,*

*// so you could also do the following:*

const sq\_sum = tf.square(tf.add(e, f));

## 模型和图层

从概念上讲，模型是一种函数，给定一些输入将产生一些所需的输出。

在TensorFlow.js中，有两种方法可以创建模型。您可以直接使用ops来表示模型所做的工作。例如：

*// Define function*

function predict(input) {

*// y = a \* x ^ 2 + b \* x + c*

*// More on tf.tidy in the next section*

return tf.tidy(() => {

const x = tf.scalar(input);

const ax2 = a.mul(x.square());

const bx = b.mul(x);

const y = ax2.add(bx).add(c);

return y;

});

}

*// Define constants: y = 2x^2 + 4x + 8*

const a = tf.scalar(2);

const b = tf.scalar(4);

const c = tf.scalar(8);

*// Predict output for input of 2*

const result = predict(2);

result.print() *// Output: 24*

您还可以使用高级API [tf.model](https://js.tensorflow.org/api/latest/index.html#model)来构建层模型，这是深度学习中的流行抽象。以下代码构造了一个[tf.sequential](https://js.tensorflow.org/api/latest/index.html#sequential)模型：

const model = tf.sequential();

model.add(

tf.layers.simpleRNN({

units: 20,

recurrentInitializer: 'GlorotNormal',

inputShape: [80, 4]

})

);

const optimizer = tf.train.sgd(LEARNING\_RATE);

model.compile({optimizer, loss: 'categoricalCrossentropy'});

model.fit({x: data, y: labels});

TensorFlow.js中有许多不同类型的层。举几个例子包括[tf.layers.simpleRNN](https://js.tensorflow.org/api/latest/index.html#layers.simpleRNN)，[tf.layers.gru](https://js.tensorflow.org/api/latest/index.html#layers.gru)，和[tf.layers.lstm](https://js.tensorflow.org/api/latest/index.html#layers.lstm)。

## 内存管理：处理和tf.tidy

由于TensorFlow.js使用GPU来加速数学运算，因此在使用张量和变量时需要管理GPU内存。

TensorFlow.js提供了两个函数来帮助解决这个问题：dispose和[tf.tidy](https://js.tensorflow.org/api/latest/index.html#tidy)。

### 部署

您可以调用dispose张量或变量来清除它并释放其GPU内存：

const x = tf.tensor2d([[0.0, 2.0], [4.0, 6.0]]);

const x\_squared = x.square();

x.dispose();

x\_squared.dispose();

### tf.tidy

dispose在进行大量张量操作时使用可能很麻烦。TensorFlow.js提供了另一个函数，tf.tidy它在JavaScript中扮演与常规作用域类似的角色，但对于GPU支持的张量。

tf.tidy执行一个函数并清除所有创建的中间张量，释放它们的GPU内存。它不会清除内部函数的返回值。

*// tf.tidy takes a function to tidy up after*

const average = tf.tidy(() => {

*// tf.tidy will clean up all the GPU memory used by tensors inside*

*// this function, other than the tensor that is returned.*

*//*

*// Even in a short sequence of operations like the one below, a number*

*// of intermediate tensors get created. So it is a good practice to*

*// put your math ops in a tidy!*

const y = tf.tensor1d([1.0, 2.0, 3.0, 4.0]);

const z = tf.ones([4]);

return y.sub(z).square().mean();

});

average.print() *// Output: 3.5*

使用tf.tidy将有助于防止应用程序中的内存泄漏。它还可以用于更加谨慎地控制何时回收内存。

#### 两个重要的注释

* 传递给的函数tf.tidy应该是同步的，也不会返回Promise。我们建议保留更新UI或在远程请求之外的代码tf.tidy。
* tf.tidy 不会清理变量。变量通常持续到机器学习模型的整个生命周期，因此TensorFlow.js即使它们是在一个中创建的，也不会清理它们tidy。但是，您可以dispose手动调用它们。

训练第一步：拟合曲线到合成数据

在本教程中，我们将使用TensorFlow.js将曲线拟合到合成数据集。给定一些使用添加了一些噪声的多项式函数生成的数据，我们将训练模型以发现用于生成数据的系数。

先决条件

本教程假设您熟悉TensorFlow.js中[Core Concepts中](https://js.tensorflow.org/tutorials/core-concepts.html)引入的TensorFlow.js的基本构建块：张量，变量和操作。我们建议在完成本教程之前完成Core Concepts。

运行代码

本教程重点介绍用于构建模型和学习其系数的TensorFlow.js代码。可以在[此处](https://github.com/tensorflow/tfjs-examples/tree/master/polynomial-regression-core)找到本教程的完整代码（包括数据生成和图表绘图代码）。

要在本地运行代码，您需要安装以下依赖项：

* [Node.js](https://nodejs.org/)版本8.9或更高版本
* [纱线](https://yarnpkg.com/en/)或[NPM CLI](https://docs.npmjs.com/cli/npm)

这些说明使用Yarn，但是如果您熟悉NPM CLI并且更喜欢使用它，那么它仍然可以使用。

$ git clone https://github.com/tensorflow/tfjs-examples

$ cd tfjs-examples/polynomial-regression-core

$ yarn

$ yarn watch

上面的[tfjs-examples / polynomial-regression-core](https://github.com/tensorflow/tfjs-examples/tree/master/polynomial-regression-core)目录是完全独立的，因此您可以复制它以启动您自己的项目。

输入数据

我们的合成数据集由x坐标和y坐标组成，在笛卡尔平面上绘制时如下所示：

使用格式*y* = *a* x 3 + *b* x 2 + *c* x + *d*的三次函数生成该数据。

我们的任务是学习该函数的*系数*：最适合数据的*a*，*b*，*c*和*d*的值。让我们看一下如何使用TensorFlow.js操作学习这些值。

第1步：设置变量

首先，让我们创建一些变量来保持我们在模型训练的每个步骤中对这些值的当前最佳估计。首先，我们将为每个变量分配一个随机数：

const a = tf.variable(tf.scalar(Math.random()));

const b = tf.variable(tf.scalar(Math.random()));

const c = tf.variable(tf.scalar(Math.random()));

const d = tf.variable(tf.scalar(Math.random()));

第2步：构建模型

我们可以通过链接一系列数学运算来表示我们在TensorFlow.js中的多项式函数*y* = *a* x 3 + *b* x 2 + *c* x + *d*：addition（[add](https://js.tensorflow.org/api/latest/index.html#add)），multiplication（[mul](https://js.tensorflow.org/api/latest/index.html#mul)）和exponentiation（[pow](https://js.tensorflow.org/api/latest/index.html#pow)和[square](https://js.tensorflow.org/api/latest/index.html#square)）。

以下代码构造一个作为输入并返回的predict函数：xy

function predict(x) {

*// y = a \* x ^ 3 + b \* x ^ 2 + c \* x + d*

return tf.tidy(() => {

return a.mul(x.pow(tf.scalar(3))) *// a \* x^3*

.add(b.mul(x.square())) *// + b \* x ^ 2*

.add(c.mul(x)) *// + c \* x*

.add(d); *// + d*

});

}

让我们继续使用我们在步骤1中设置的*a*，*b*，*c*和*d*的随机值绘制我们的多项式函数。我们的图可能看起来像这样：

因为我们从随机值开始，所以我们的函数很可能不适合数据集。该模型尚未学习更好的系数值。

第3步：训练模型

我们的最后一步是训练模型以学习系数的良好值。为了训练我们的模型，我们需要定义三件事：

* 一个*丧失功能*，是衡量一个给定的多项式吻合程度的数据。损耗值越低，多项式拟合数据越好。
* 一个*优化器*，它实现的算法用于基于所述损失函数的输出修改我们的系数值。优化器的目标是最小化损失函数的输出值。
* 一个*训练循环*，这将反复运行优化，以尽量减少损失。

定义损失函数

对于本教程，我们将使用[均方误差（MSE）](https://developers.google.com/machine-learning/crash-course/glossary/#MSE)作为我们的损失函数。通过对我们数据集中每个*x*值的实际*y*值和预测*y*值之间的差值求平方，然后取所有结果项的均值来计算MSE 。

我们可以在TensorFlow.js中定义一个MSE丢失函数，如下所示：

function loss(predictions, labels) {

*// Subtract our labels (actual values) from predictions, square the results,*

*// and take the mean.*

const meanSquareError = predictions.sub(labels).square().mean();

return meanSquareError;

}

定义优化程序

对于我们的优化器，我们将使用[Stochastic Gradient Descent](https://developers.google.com/machine-learning/crash-course/glossary#SGD)（SGD）。SGD通过获取数据集中随机点的[梯度](https://developers.google.com/machine-learning/crash-course/glossary#gradient)并使用其值来通知是否增加或减少模型系数的值来工作。

TensorFlow.js为执行SGD提供了便利功能，因此您不必担心自己执行所有这些数学运算。[tf.train.sgd](https://js.tensorflow.org/api/latest/index.html#train.sgd)将所需的学习速率作为输入，并返回一个SGDOptimizer对象，可以调用该对象以优化损失函数的值。

该*学习速率*控制有多大改善其预测当模特的调整会。低学习率将使学习过程运行得更慢（学习好系数需要更多的训练迭代），而高学习率将加速学习，但可能导致模型围绕正确值振荡，总是过度校正。

以下代码构造了一个学习率为0.5的SGD优化器：

const learningRate = 0.5;

const optimizer = tf.train.sgd(learningRate);

定义训练循环

现在我们已经定义了损失函数和优化器，我们可以构建一个训练循环，迭代地执行SGD来优化我们的模型系数以最小化损失（MSE）。这是我们的循环：

function train(xs, ys, numIterations = 75) {

const learningRate = 0.5;

const optimizer = tf.train.sgd(learningRate);

for (let iter = 0; iter < numIterations; iter++) {

optimizer.minimize(() => {

const predsYs = predict(xs);

return loss(predsYs, ys);

});

}

}

让我们一步一步地仔细研究代码。首先，我们定义训练函数，将数据集的*x*和*y*值以及指定的迭代次数作为输入：

function train(xs, ys, numIterations) {

...

}

接下来，我们定义学习速率和SGD优化器，如上一节所述：

const learningRate = 0.5;

const optimizer = tf.train.sgd(learningRate);

最后，我们建立了一个for运行numIterations训练迭代的循环。在每次迭代中，我们调用[minimize](https://js.tensorflow.org/api/latest/index.html#class:train.Optimizer)优化器，这是魔术发生的地方：

for (let iter = 0; iter < numIterations; iter++) {

optimizer.minimize(() => {

const predsYs = predict(xs);

return loss(predsYs, ys);

});

}

minimize 采取做两件事的功能：

1. 它使用我们之前在步骤2中定义的模型函数预测所有*x*值的*y*值（predYs）。predict
2. 它使用我们之前在**定义损失函数**中**定义的损失函数**返回那些预测的均方误差损失。

minimize然后自动调整任何Variable通过该功能使用S（这里，系数a，b，c，和d），以尽量减少返回值（我们的损失）。

运行我们的训练循环后a，b，c，和d将包含SGD后75次迭代由模型学到的系数值。

查看结果！

一旦程序完成运行，我们可以采取的最终值我们的变量a，b，c，和d，并利用它们来绘制曲线：

结果比我们最初使用系数的随机值绘制的曲线好得多。

图像训练：用卷积神经网络识别手写数字

在本教程中，我们将构建一个TensorFlow.js模型，用卷积神经网络对手写数字进行分类。首先，我们将通过“查看”成千上万的手写数字图像及其标签来训练分类器。然后我们将使用模型从未见过的测试数据来评估分类器的准确性。

先决条件

本教程假设您熟悉TensorFlow.js（张量，变量和操作）的基本构建块，以及优化和丢失的概念。有关这些主题的更多背景知识，我们建议您在本教程之前完成以下教程：

* [TensorFlow.js中的核心概念](https://js.tensorflow.org/tutorials/core-concepts.html)
* [训练第一步：拟合曲线到合成数据](https://js.tensorflow.org/tutorials/fit-curve.html)

运行代码

可以在[TensorFlow.js示例存储库](https://github.com/tensorflow/tfjs-examples/tree/master/mnist)的[tfjs-examples / mnist](https://github.com/tensorflow/tfjs-examples/tree/master/mnist)目录中找到本教程的完整代码。

要在本地运行代码，您需要安装以下依赖项：

* [Node.js](https://nodejs.org/)版本8.9或更高版本
* [纱线](https://yarnpkg.com/en/)或[NPM CLI](https://docs.npmjs.com/cli/npm)

这些说明使用Yarn，但是如果您熟悉NPM CLI并且更喜欢使用它，那么它仍然可以使用。

您可以通过克隆repo并构建演示来运行示例的代码：

$ git clone https://github.com/tensorflow/tfjs-examples

$ cd tfjs-examples/mnist

$ yarn

$ yarn watch

上面的[tfjs-examples / mnist](https://github.com/tensorflow/tfjs-examples/tree/master/mnist) 目录是完全独立的，因此您可以复制它以启动您自己的项目。

**注意：**本教程的代码与[tfjs-examples / mnist-core](https://github.com/tensorflow/tfjs-examples/tree/master/mnist-core)示例之间的区别在于，我们使用TensorFlow.js的更高级API（Model，Layers）来构建模型，而[mnist-core](https://github.com/tensorflow/tfjs-examples/tree/master/mnist-core)使用更低级别的线性代数操作建立一个神经网络。

数据

我们将在本教程中使用[MNIST手写数据集](http://yann.lecun.com/exdb/mnist/)。我们将学习分类的手写MNIST数字如下所示：

为了预处理我们的数据，我们编写了[data.js](https://github.com/tensorflow/tfjs-examples/blob/master/mnist-core/data.js)，其中包含MnistData从我们提供的MNIST数据集的托管版本中提取随机批次的MNIST图像的类。

MnistData将整个数据集拆分为训练数据和测试数据。当我们训练模型时，分类器将仅看到训练集。当我们评估模型时，我们将仅使用模型尚未看到的测试集中的数据来查看模型的预测对于全新数据的推广程度。

MnistData 有两种公共方法：

* nextTrainBatch(batchSize)：从训练集中返回随机批次的图像及其标签
* nextTestBatch(batchSize)：从测试集中返回一批图像及其标签

**注意：**在训练MNIST分类器时，随机重排数据非常重要，因此模型的预测不受我们提供图像的顺序的影响。例如，如果我们首先将模型全部输入*1个*数字，在此阶段的训练期间，模型可能会学会简单地预测*1*（因为这会使损失最小化）。如果我们然后仅将模型馈送*2*秒，它可能只是切换到仅预测*2*并且从不预测*1*（因为，再次，这将最小化新图像集的损失）。该模型永远不会学习对有代表性的数字样本进行准确预测。

构建模型

在本节中，我们将构建一个卷积图像分类器模型。为此，我们将使用一个Sequential模型（最简单的模型），其中张量从一个层连续传递到下一个层。

首先，让我们Sequential用tf.sequential以下方法实例化我们的模型：

const model = tf.sequential();

现在我们已经创建了一个模型，让我们为它添加图层。

添加第一层

我们要添加的第一层是二维卷积层。Convolutions在图像上滑动滤镜窗口以学习空间不变的变换（即，图像的不同部分中的图案或对象将以相同的方式处理）。有关卷积的更多信息，请参阅[此文章](http://colah.github.io/posts/2014-07-Understanding-Convolutions/)。

我们可以使用创建二维卷积层[tf.layers.conv2d](https://js.tensorflow.org/api/latest/index.html#layers.conv2d)，它接受一个定义图层结构的配置对象：

model.add(tf.layers.conv2d({

inputShape: [28, 28, 1],

kernelSize: 5,

filters: 8,

strides: 1,

activation: 'relu',

kernelInitializer: 'VarianceScaling'

}));

让我们分解配置对象中的每个参数：

* inputShape。将流入模型第一层的数据形状。在这种情况下，我们的MNIST示例是28x28像素的黑白图像。图像数据的规范格式是[row, column, depth]，所以这里我们要为[28, 28, 1]每个维度中的像素数配置-28行和列的形状，深度为1，因为我们的图像只有1个颜色通道：
* kernelSize。要应用于输入数据的滑动卷积滤波器窗口的大小。在这里，我们设置kernelSize的5，它指定一个正方形，5x5的卷积窗口。
* filters。kernelSize要应用于输入数据的大小的过滤器窗口数。在这里，我们将对数据应用8个过滤器。
* strides。滑动窗口的“步长” - 即每次在图像上移动时滤波器将移动多少像素。在这里，我们指定步幅为1，这意味着滤镜将以1像素的步长滑过图像。
* activation。卷积完成后应用于数据的[激活功能](https://developers.google.com/machine-learning/glossary/#activation_function)。在这种情况下，我们正在应用[整流线性单元（ReLU）](https://developers.google.com/machine-learning/glossary/#ReLU)功能，这是ML模型中非常常见的激活功能。
* kernelInitializer。用于随机初始化模型权重的方法，这对训练动力学非常重要。我们不会在这里详细介绍初始化，但是VarianceScaling（这里使用）通常是一个很好的初始化器选择。

添加第二层

让我们在模型中添加第二层：最大池层，我们将使用它创建[tf.layers.maxPooling2d](https://js.tensorflow.org/api/latest/index.html#layers.maxPooling2d)。该层将通过计算每个滑动窗口的最大值来从卷积中对结果（也称为激活）进行下采样：

model.add(tf.layers.maxPooling2d({

poolSize: [2, 2],

strides: [2, 2]

}));

让我们打破这些论点：

* poolSize。要应用于输入数据的滑动池窗口的大小。在这里，我们设置poolSize的[2,2]，这意味着汇集层将应用2×2窗口的输入数据。
* strides。滑动池窗口的“步长” - 即每次窗口在输入数据上移动时窗口将移动多少像素。在这里，我们指定步幅[2, 2]，这意味着滤镜将在水平和垂直方向上以2像素的步长滑过图像。

**注：**由于这两个poolSize和strides是2×2的集中窗口将完全不重叠。这意味着池化层将激活前一层的激活大小减半。

添加剩余的图层

重复层结构是神经网络中的常见模式。让我们添加第二个卷积层，然后添加另一个池模型到我们的模型。请注意，在我们的第二个卷积层中，我们将过滤器的数量从8增加到16.还要注意我们没有指定inputShape，因为它可以从前一层的输出形状推断：

model.add(tf.layers.conv2d({

kernelSize: 5,

filters: 16,

strides: 1,

activation: 'relu',

kernelInitializer: 'VarianceScaling'

}));

model.add(tf.layers.maxPooling2d({

poolSize: [2, 2],

strides: [2, 2]

}));

接下来，让我们添加一个[flatten](https://js.tensorflow.org/api/latest/index.html#layers.flatten)图层，将前一层的输出展平为矢量：

model.add(tf.layers.flatten());

最后，让我们添加一个[dense](https://js.tensorflow.org/api/latest/index.html#layers.dense)层（也称为完全连接层），它将执行最终分类。在密集层之前展平卷积+池层对的输出是神经网络中的另一种常见模式：

model.add(tf.layers.dense({

units: 10,

kernelInitializer: 'VarianceScaling',

activation: 'softmax'

}));

让我们分解传递给dense图层的参数。

* units。输出激活的大小。由于这是最后一层，我们正在进行10级分类任务（数字0-9），我们在这里使用10个单位。（有时单位被称为*神经元*的数量，但我们将避免使用该术语。）
* kernelInitializer。我们将对VarianceScaling用于卷积层的密集层使用相同的初始化策略。
* activation。分类任务的最后一层的激活功能通常是[softmax](https://developers.google.com/machine-learning/glossary/#softmax)。Softmax将我们的10维输出向量归一化为概率分布，因此我们有10个类别中每个类别的概率。

培训模型

为了实际驱动模型的训练，我们需要构造一个优化器并定义一个损失函数。我们还将定义评估指标，以衡量我们的模型对数据的执行情况。

**注意：**要深入了解TensorFlow.js中的优化器和损失函数，请参阅“ [培训第一步](https://js.tensorflow.org/tutorials/fit-curve.html) ”教程。

定义优化程序

对于我们的卷积神经网络模型，我们将使用学习率为0.15 的[随机梯度下降（SGD）优化器](https://developers.google.com/machine-learning/glossary/#SGD)：

const LEARNING\_RATE = 0.15;

const optimizer = tf.train.sgd(LEARNING\_RATE);

定义损失

对于我们的损失函数，我们将使用cross-entropy（categoricalCrossentropy），它通常用于优化分类任务。categoricalCrossentropy测量由模型的最后一层生成的概率分布与我们的标签给出的概率分布之间的误差，该分布将是在正确的类标签中具有1（100％）的分布。例如，给定数字*7*的示例的以下标签和预测值：

| **类** | **0** | **1** | **2** | **3** | **4** | **五** | **6** | **7** | **8** | **9** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 标签 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 预测 | 0.1 | 0.01 | 0.01 | 0.01 | 0.20 | 0.01 | 0.01 | 0.60 | 0.03 | 0.02 |

categoricalCrossentropy给出了一个较低的损耗值，如果预测是高概率位是*7*，和一个更高的损耗值，如果预测是低概率*7*。在训练期间，模型将更新其内部参数以最小化categoricalCrossentropy整个数据集。

定义评估指标

对于我们的评估指标，我们将使用准确度，该准确度衡量所有预测中正确预测的百分比。

编译模型

为了编译模型，我们使用我们的优化器，损失函数和评估指标列表（仅此处'accuracy'）传递一个配置对象：

model.compile({

optimizer: optimizer,

loss: 'categoricalCrossentropy',

metrics: ['accuracy'],

});

配置批量大小

在开始培训之前，我们需要定义一些与批量大小相关的参数：

*// How many examples the model should "see" before making a parameter update.*

const BATCH\_SIZE = 64;

*// How many batches to train the model for.*

const TRAIN\_BATCHES = 100;

*// Every TEST\_ITERATION\_FREQUENCY batches, test accuracy over TEST\_BATCH\_SIZE examples.*

*// Ideally, we'd compute accuracy over the whole test set, but for performance*

*// reasons we'll use a subset.*

const TEST\_BATCH\_SIZE = 1000;

const TEST\_ITERATION\_FREQUENCY = 5;

**有关批处理和批处理大小的更多信息**

为了充分利用GPU并行计算的能力，我们希望将多个输入一起批处理，并使用单个前馈调用通过网络提供它们。

我们对计算进行批处理的另一个原因是，在优化期间，我们仅在对几个示例的梯度进行平均后才更新内部参数（采取步骤）。这有助于我们避免因错误的方向而向前迈出一步（例如，错误标记的数字）。

在对输入数据进行批处理时，我们引入了秩*D + 1*的张量，其中*D*是单个输入的维数。

如前所述，我们的MNIST数据集中单个图像的维度是[28, 28, 1]。当我们设置BATCH\_SIZE64时，我们一次批量处理64个图像，这意味着我们数据的实际形状是[64, 28, 28, 1]（批处理总是最外层的维度）。

*注意：* \*回想一下，inputShape我们的第一个配置中conv2d没有指定批量大小（64）。配置编写为批量大小不可知，因此它们能够接受任意大小的批量。

编码训练循环

以下是训练循环的代码：

for (let i = 0; i < TRAIN\_BATCHES; i++) {

const batch = data.nextTrainBatch(BATCH\_SIZE);

let testBatch;

let validationData;

*// Every few batches test the accuracy of the mode.*

if (i % TEST\_ITERATION\_FREQUENCY === 0) {

testBatch = data.nextTestBatch(TEST\_BATCH\_SIZE);

validationData = [

testBatch.xs.reshape([TEST\_BATCH\_SIZE, 28, 28, 1]), testBatch.labels

];

}

*// The entire dataset doesn't fit into memory so we call fit repeatedly*

*// with batches.*

const history = await model.fit(

batch.xs.reshape([BATCH\_SIZE, 28, 28, 1]),

batch.labels,

{

batchSize: BATCH\_SIZE,

validationData,

epochs: 1

});

const loss = history.history.loss[0];

const accuracy = history.history.acc[0];

*// ... plotting code ...*

}

让我们打破代码吧。首先，我们获取一批培训示例。回想一下，我们批量示例利用GPU并行化并在进行参数更新之前平均来自许多示例的证据：

const batch = data.nextTrainBatch(BATCH\_SIZE);

每5个步骤（TEST\_ITERATION\_FREQUENCY我们构建validationData一个包含来自测试集的一批MNIST图像的两个元素的数组及其相应的标签。我们将使用此数据来评估模型的准确性：

if (i % TEST\_ITERATION\_FREQUENCY === 0) {

testBatch = data.nextTestBatch(TEST\_BATCH\_SIZE);

validationData = [

testBatch.xs.reshape([TEST\_BATCH\_SIZE, 28, 28, 1]),

testBatch.labels

];

}

model.fit 是训练模型和参数实际更新的地方。

**注意：**model.fit()在整个数据集上调用一次将导致将整个数据集上载到GPU，这可能会冻结应用程序。为避免将过多数据上传到GPU，我们建议model.fit()在for循环内调用，一次传递一批数据，如下所示：

*// The entire dataset doesn't fit into memory so we call fit repeatedly*

*// with batches.*

const history = await model.fit(

batch.xs.reshape([BATCH\_SIZE, 28, 28, 1]), batch.labels,

{batchSize: BATCH\_SIZE, validationData: validationData, epochs: 1});

让我们再次打破这些争论：

* x。我们的输入图像数据。请记住，我们正在批量提供示例，因此我们必须告诉 fit函数我们的批次有多大。MnistData.nextTrainBatch返回具有形状[BATCH\_SIZE, 784]的图像 - 所有图像的数据在长度为784（28 \* 28）的1-D向量中。但是，我们的模型期望图像数据在形状中[BATCH\_SIZE, 28, 28, 1]，因此我们[reshape](https://js.tensorflow.org/api/latest/index.html#reshape)相应地。
* y。我们的标签; 每个图像的正确数字分类。
* batchSize。每个培训批次中包含多少图像。之前我们BATCH\_SIZE在这里设置了64个。
* validationData。我们构建每个TEST\_ITERATION\_FREQUENCY（此处，5个）批次的验证集。这个数据是形状[TEST\_BATCH\_SIZE, 28, 28, 1]。之前，我们设置了TEST\_BATCH\_SIZE1000.我们将在此数据集上计算我们的评估指标（准确度）。
* epochs。批量执行的训练次数。由于我们正在迭代地为批次提供批次fit，我们只希望它一次性从该批次进行培训。

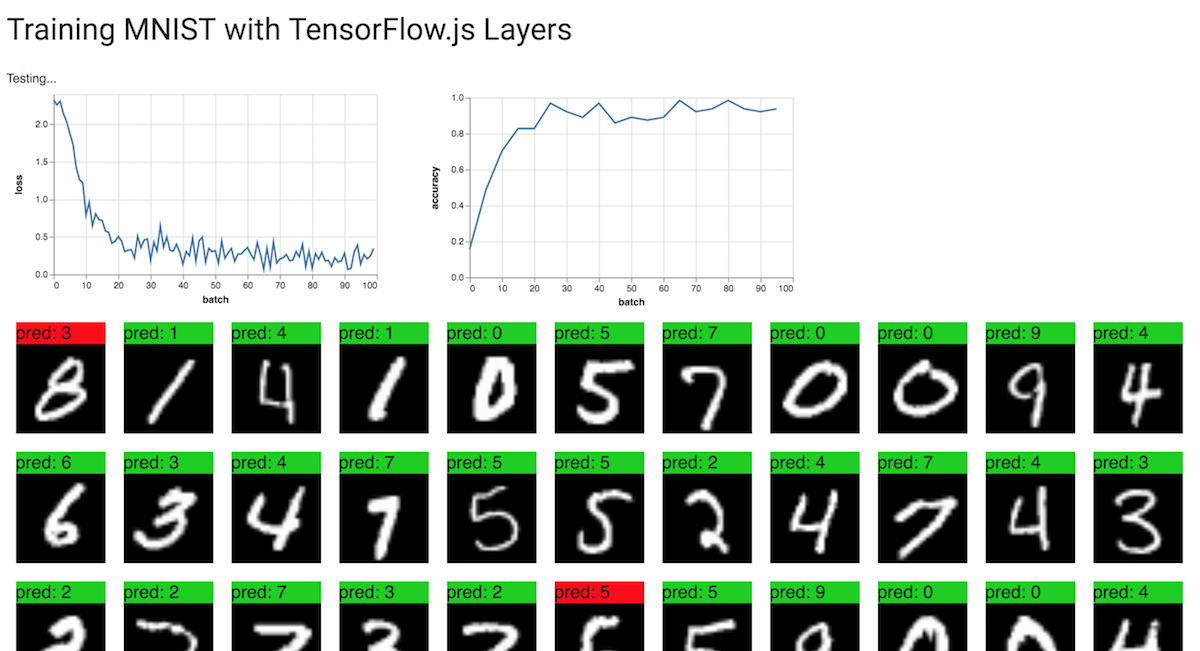
每次调用时fit，它都会返回一个富对象，其中包含我们存储的指标日志history。我们提取每次训练迭代的损失和准确性，因此我们可以在图表上绘制它们：

const loss = history.history.loss[0];

const accuracy = history.history.acc[0];

查看结果！

如果您运行完整代码，您应该看到如下输出：



看起来模型正在预测大多数图像的正确数字。做得好！

转移学习 - 训练神经网络以预测网络摄像头数据

在开始之前，我们强烈建议您使用该演示。 [试试吧！](https://storage.googleapis.com/tfjs-examples/webcam-transfer-learning/dist/index.html)

在[核心概念教程中](https://js.tensorflow.org/tutorials/core-concepts.html)，我们学习了如何使用张量和运算来执行基本线性代数。

在[卷积图像分类器教程中](https://js.tensorflow.org/tutorials/mnist.html)，我们学习了如何构建卷积图像分类器以识别来自MNIST数据集的手写数字。

在[Importing a Keras模型教程中，](https://js.tensorflow.org/tutorials/import-keras.html)我们学习了如何将预训练的Keras模型移植到浏览器中进行推理。

在本教程中，我们将使用转移学习从网络摄像头数据（姿势，对象，面部表情等）预测用户定义的类，并通过将每个姿势分配给“向上”，“向下”，“向左”来玩Pacman，和“正确”。

关于游戏

游戏分为三个阶段。

1. **数据收集：**播放器将来自网络摄像头的图像与上，下，左，右四个类中的每一个相关联。
2. **训练：**训练神经网络从输入图像预测类。
3. **推理/播放：**使用我们训练的模型从网络摄像头数据进行上，下，左，右预测，并将其输入Pacman游戏！

关于模型

要了解在合理的时间量从网络摄像头不同的类分类，我们将*重新训练*，或*微调*，一个预训练的 [MobileNet](https://github.com/tensorflow/models/blob/master/research/slim/nets/mobilenet_v1.md) 模型，使用内部激活（从MobileNet的内部层的输出）作为输入到我们的新模型。

为此，我们实际上在页面上有两个模型。

一个模型将是预先训练的MobileNet模型，该模型被截断以输出内部激活。我们称之为“截断的MobileNet模型”。加载到浏览器后，此模型未经过培训。

第二个模型将截断的MobileNet模型的内部激活的输出作为输入，并将预测4个输出类（上，下，左和右）中的每一个的概率。这是我们实际在浏览器中训练的模型。

通过使用MobileNet的内部激活，我们可以重用MobileNet已经学习的功能，通过相对较少的再训练来预测1000类ImageNet。

关于本教程

要在本地运行代码，您需要安装以下依赖项：

* [Node.js](https://nodejs.org/)版本8.9或更高版本
* [纱线](https://yarnpkg.com/en/)或[NPM CLI](https://docs.npmjs.com/cli/npm)

这些说明使用Yarn，但是如果您熟悉NPM CLI并且更喜欢使用它，那么它仍然可以使用。

您可以通过克隆repo并构建演示来运行示例的代码：

git clone https://github.com/tensorflow/tfjs-examples

cd tfjs-examples/webcam-transfer-learning

yarn

yarn watch

上面的[tfjs-examples / webcam-transfer-learning](https://github.com/tensorflow/tfjs-examples/tree/master/webcam-transfer-learning) 目录是完全独立的，因此您可以将其复制以启动您自己的项目。

*注意：这种方法与*[*Teachable Machine*](https://teachablemachine.withgoogle.com/)*采用的方法不同 。可训练的机器使用K-最近邻（KNN）对来自预训练的SqueezeNet模型的预测进行分类，而这种方法使用从MobileNet的内部激活训练的第二神经网络。KNN图像分类器在较少量的数据下工作得更好，但具有传递学习的神经网络更好地概括。与两个演示一起玩，探索两种不同的网络摄像头预测方式有何不同！*

数据

在我们训练模型之前，我们需要一种Tensor从网络摄像头中获取s 的方法。

我们提供了一个webcam.js被调用的类，Webcam它从<video>标签中读取图像作为TensorFlow.js Tensor。

我们来看一下capture方法吧Webcam。

capture() {

return tf.tidy(() => {

const webcamImage = tf.fromPixels(this.webcamElement);

const croppedImage = this.cropImage(webcamImage);

const batchedImage = croppedImage.expandDims(0);

return batchedImage.toFloat().div(oneTwentySeven).sub(one);

});

}

让我们分解这些界限。

const webcamImage = tf.fromPixels(this.webcamElement);

此行从webcam <video>元素读取单个帧并返回 Tensor形状[height, width, 3]。最内层尺寸3对应于三个通道RGB。

有关支持的输入HTML元素类型，请参阅[tf.fromPixels](https://js.tensorflow.org/api/latest/index.html#fromPixels)的文档。

const croppedImage = this.cropImage(webcamImage);

设置方形网络摄像头元素时，网络摄像头源的自然宽高比为矩形（浏览器会在矩形图像周围放置空白区域以使其成为方形）。

但是，MobileNet模型需要方形输入图像。此行[224, 224]从网络摄像头元素中裁剪出一个方形中心块。请注意，有更多的代码Webcam增加了视频元素的大小，因此我们可以裁剪一个[224, 224]方块而不会获得白色填充。

const batchedImage = croppedImage.expandDims(0);

expandDims创建一个大小为1的新外部尺寸。在这种情况下，我们从网络摄像头读取的裁剪图像具有形状[224, 224, 3]。调用 expandDims(0)将此张量重新整形为[1, 224, 224, 3]，表示一批单个图像。MobileNet期望批量输入。

batchedImage.toFloat().div(tf.scalar(127)).sub(tf.scalar(1));

在这一行中，我们将图像转换为浮点并在-1和1之间将其标准化（这就是模型的训练方式）。我们知道默认情况下图像中的值介于0到255之间，因此为了在-1和1之间进行归一化，我们除以127并减去1。

return tf.tidy(() => {

...

});

通过调用tf.tidy()，我们告诉TensorFlow.js破坏Tensor我们在内部分配的中间存储器的内存capture()。有关内存管理和更多信息，请参阅 [核心概念教程](https://js.tensorflow.org/tutorials/core-concepts.html)tf.tidy()

正在加载mobilenet

在我们建立模型之前，我们需要将预先训练的MobileNet加载到网页中。从这个模型中，我们将构建一个新模型，从MobileNet输出内部激活。

这是执行此操作的代码：

async function loadMobilenet() {

const mobilenet = await tf.loadModel(

'https://storage.googleapis.com/tfjs-models/tfjs/mobilenet\_v1\_0.25\_224/model.json');

*// Return a model that outputs an internal activation.*

const layer = mobilenet.getLayer('conv\_pw\_13\_relu');

return tf.model({inputs: mobilenet.inputs, outputs: layer.output});

});

通过调用getLayer('conv\_pw\_13\_relu')，我们将进入预训练的MobileNet模型的内部层，并构建一个新模型，其中输入是MobileNet的相同输入，但输出的层是MobileNet的中间层，名为conv\_pw\_13\_relu。

*注意：我们根据经验选择了这一层 - 它适用于我们的任务。一般而言，面向预训练模型末尾的层将在传递学习任务中表现更好，因为它包含输入的更高级语义特征。尝试选择另一个图层，看看它如何影响模型质量！您可以使用它model.layers来打印模型的图层。*

*注意：有关如何将Keras模型移植到TensorFlow.js的详细信息，请查看*[*导入Keras模型*](https://js.tensorflow.org/tutorials/import-keras.html)*教程。*

阶段1：收集数据

游戏的第一阶段是数据收集阶段。用户将从网络摄像头保存帧并将它们与4个类中的每一个相关联：向上，向下，向左和向右。

当我们从网络摄像头收集帧时，我们将立即通过截断的MobileNet模型提供它们并保存激活张量我们不需要保存从网络摄像头捕获的原始图像，因为我们将使用的模型火车只需要这些激活作为输入。之后，当我们从网络摄像头进行预测以实际玩游戏时，我们将首先通过截断的MobileNet模型提供帧，然后通过我们的第二个模型提供截断的Mobilenet模型的输出。

我们提供了一个ControllerDataset保存这些激活的课程，以便在培训阶段使用它们。ControllerDataset有一个方法，addExample。这将通过Tensor我们截断的MobileNet 的激活调用，并label作为一个关联number。

添加新示例时，我们将保留两个Tensor代表整个数据集的内容，xs以及ys。这些将用作我们将要训练的模型的输入。

xs表示截断的MobileNet中针对所有收集的数据的所有激活，并将所有收集的数据ys的标签表示为“一个热”表示。当我们训练我们的模型时，我们将为它提供xs和的整个数据集ys。

*有关单热编码的更多详细信息，请查看*[*MLCC词汇表*](https://developers.google.com/machine-learning/crash-course/glossary#o)*。*

我们来看看实现。

addExample(example, label) {

const y = tf.tidy(() => tf.oneHot(tf.tensor1d([label]), this.numClasses));

if (this.xs == null) {

this.xs = tf.keep(example);

this.ys = tf.keep(y);

} else {

const oldX = this.xs;

this.xs = tf.keep(oldX.concat(example, 0));

const oldY = this.ys;

this.ys = tf.keep(oldY.concat(y, 0));

oldX.dispose();

oldY.dispose();

y.dispose();

}

}

让我们打破这个功能。

const y = tf.tidy(() => tf.oneHot(tf.tensor1d([label]), this.numClasses));

该行将对应于标签的整数转换为该标签的单热表示。

例如，如果label = 1对应于“左”类，则单热表示将是[0, 1, 0, 0]。我们进行这种转换，以便这代表概率分布，在类1中具有100％概率，“左”

if (this.xs == null) {

this.xs = tf.keep(example);

this.ys = tf.keep(y);

}

当我们将第一个示例添加到数据集中时，我们将只保留给定的值。

我们调用tf.keep()输入Tensors，这样它们就不会被任何tf.tidy()可以包含调用的东西处理掉addExample。有关内存管理的更多信息，请参阅[Core Concepts](https://js.tensorflow.org/tutorials/core-concepts.html)。

} else {

const oldX = this.xs;

this.xs = tf.keep(oldX.concat(example, 0));

const oldY = this.ys;

this.ys = tf.keep(oldY.concat(y, 0));

oldX.dispose();

oldY.dispose();

y.dispose();

}

当我们已经添加了一个例子，我们的数据，我们会通过调用串连新的例子来设定的现有的例子concat，用axis 的参数组来0。这会不断将我们的输入激活堆叠到xs我们的标签中ys。然后我们将处理（）任何旧的值xs和ys。

例如，如果我们的第一个标签（1）看起来像：

[[0, 1, 0, 0]]

然后第二个电话后addExample有label = 2，ys会是这样的：

[[0, 1, 0, 0],

[0, 0, 1, 0]]

xs将具有相似的形状，但具有更高的维度，因为我们正在使用3D激活（制作xs4D，其中最外层的维度是收集的示例的数量）。

现在，回到index.js定义核心逻辑的地方，我们已经定义：

ui.setExampleHandler(label => {

tf.tidy(() => {

const img = webcam.capture();

controllerDataset.addExample(mobilenet.predict(img), label);

*// ...*

});

});

在这个块中，我们正在使用UI注册处理程序，以便在按下向上，向下，向左或向右按​​钮之一时进行处理，其中label对应于类索引：0,1,2或3。

在这个处理程序中，我们只是从网络摄像头捕获一个帧，通过截断的MobileNet提供它，生成内部激活，然后将其保存在我们的ControllerDataset对象中。

阶段2：训练模型

一旦用户从相关类的网络摄像头数据中收集了所有示例，我们就应该训练我们的模型！

首先，让我们设置模型的拓扑。我们将创建一个2层密集（完全连接）模型，relu在第一个密集层之后具有激活功能。

model = tf.sequential({

layers: [

*// Flattens the input to a vector so we can use it in a dense layer. While*

*// technically a layer, this only performs a reshape (and has no training*

*// parameters).*

tf.layers.flatten({inputShape: [7, 7, 256]}),

tf.layers.dense({

units: ui.getDenseUnits(),

activation: 'relu',

kernelInitializer: 'varianceScaling',

useBias: true

}),

*// The number of units of the last layer should correspond*

*// to the number of classes we want to predict.*

tf.layers.dense({

units: NUM\_CLASSES,

kernelInitializer: 'varianceScaling',

useBias: false,

activation: 'softmax'

})

]

});

您会注意到模型的第一层实际上是一个flatten图层。我们需要将输入展平为向量，以便我们可以在密集层中使用它们。inputShape展平图层的 参数对应于我们截断的MobileNet的激活形状。

我们要添加的下一层是一个密集层。我们将使用用户从UI中选择的单元对其进行初始化，使用relu 激活函数，使用varianceScaling内核初始化程序，我们将添加偏差。

我们要添加的最后一层是另一个密集层。我们将使用与我们想要预测的类数相对应的单位数来初始化它。我们将使用softmax激活函数，这意味着我们将最后一层的输出解释为可能类的概率分布。

*查看*[*API参考*](https://js.tensorflow.org/api/latest/index.html#layers.dense)*以获取有关层构造函数参数的详细信息，或查看*[*卷积MNIST教程*](https://js.tensorflow.org/tutorials/mnist.html)*。*

const optimizer = tf.train.adam(ui.getLearningRate());

model.compile({optimizer: optimizer, loss: 'categoricalCrossentropy'});

这是我们构建优化器，定义损失函数，编译模型以准备进行训练的地方。

我们在Adam这里使用的是优化器，它可以很好地完成这项任务。我们的损失函数categoricalCrossentropy将测量我们4个类的预测概率分布与真实标签（单热编码标签）之间的误差。

const batchSize =

Math.floor(controllerDataset.xs.shape[0] \* ui.getBatchSizeFraction());

由于我们的数据集是动态的（用户定义要收集的数据集的大小），因此我们相应地调整批量大小。用户可能不会收集数千个示例，因此我们的批量大小可能不会太大。

现在让我们训练模型吧！

model.fit(controllerDataset.xs, controllerDataset.ys, {

batchSize,

epochs: ui.getEpochs(),

callbacks: {

onBatchEnd: async (batch, logs) => {

*// Log the cost for every batch that is fed.*

ui.trainStatus('Cost: ' + logs.loss.toFixed(5));

await tf.nextFrame();

}

}

});

model.fit可以将整个数据集作为xs和ys我们从控制器数据集传递的数据集。

我们epochs从UI 设置，允许用户定义训练模型的时间。

我们还注册了一个onBatchEnd回调fit函数，该函数在完成培训批处理的内部培训循环后调用，允许我们在模型训练时向用户显示中间成本值。我们await tf.nextFrame() 允许UI在培训期间更新。

*有关此损失函数的更多详细信息，请参阅*[*卷积MNIST教程*](https://js.tensorflow.org/tutorials/mnist.html)*。*

第3阶段：玩Pacman

一旦我们的模型经过培训，并且我们的成本价值已经下降，我们就可以通过网络摄像头进行预测！

这是预测循环：

while (isPredicting) {

const predictedClass = tf.tidy(() => {

const img = webcam.capture();

const activation = mobilenet.predict(img);

const predictions = model.predict(activation);

return predictions.as1D().argMax();

});

const classId = (await predictedClass.data())[0];

predictedClass.dispose();

ui.predictClass(classId);

await tf.nextFrame();

}

让我们分清界限：

const img = webcam.capture();

正如我们之前看到的，它从网络摄像头捕获一帧作为Tensor。

const activation = mobilenet.predict(img);

现在，通过我们截断的MobileNet模型提供网络摄像头帧，以获得内部MobileNet激活。

const predictions = model.predict(act);

现在，通过我们训练的模型提供激活，以获得一组预测。这是输出类别上的概率分布（该预测向量中的4个值中的每一个代表该类的概率）。

predictions.as1D().argMax();

最后，压平输出，然后调用argMax。这将返回具有最高值（概率）的索引。这对应于预测的类别。

const classId = (await predictedClass.data())[0];

predictedClass.dispose();

ui.predictClass(classId);

现在我们有一个Tensor预测标量，下载并在UI中显示！（注意我们需要Tensor在获取其值后手动处理此处，因为我们处于无法包装的异步上下文中 tf.tidy()。）

包起来

而已！您现在已经学会了如何训练神经网络以从一组用户定义的类中进行预测。图像永远不会离开浏览器！

如果您将此演示分叉以进行修改，则可能必须更改模型参数才能使其适用于您的任务。

# 将Keras模型导入TensorFlow.js

Keras模型（通常通过Python API创建）可以以[多种格式之一](https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model)保存。“整个模型”格式可以转换为TensorFlow.js图层格式，可以直接加载到TensorFlow.js进行推理或进一步培训。

目标TensorFlow.js图层格式是一个包含model.json文件和一组二进制格式的分片权重文件的目录。该model.json文件包含模型拓扑（又名“架构”或“图形”：层的描述及其连接方式）和权重文件的清单。

## 要求

转换过程需要Python环境; 你可能想用[pipenv](https://github.com/pypa/pipenv)或[virtualenv](https://virtualenv.pypa.io/)保持一个孤立的。要安装转换器，请使用pip install tensorflowjs。

将Keras模型导入TensorFlow.js分为两步。首先，将现有的Keras模型转换为TF.js图层格式，然后将其加载到TensorFlow.js中。

## 步骤1.将现有的Keras模型转换为TF.js图层格式

Keras模型通常通过via保存model.save(filepath)，生成一个包含模型拓扑和权重的HDF5（.h5）文件。要将此类文件转换为TF.js图层格式，请运行以下命令，其中*path/to/my\_model.h5*是源Keras .h5文件，并且*path/to/tfjs\_target\_dir*是TF.js文件的目标输出目录：

*# bash*

tensorflowjs\_converter --input\_format keras \

path/to/my\_model.h5 \

path/to/tfjs\_target\_dir

## 替代方法：使用Python API直接导出到TF.js图层格式

如果您在Python中使用Keras模型，则可以将其直接导出为TensorFlow.js图层格式，如下所示：

*# Python*

import tensorflowjs as tfjs

def train(...):

model = keras.models.Sequential() *# for example*

...

model.compile(...)

model.fit(...)

tfjs.converters.save\_keras\_model(model, tfjs\_target\_dir)

## 第2步：将模型加载到TensorFlow.js中

使用Web服务器为您在步骤1中生成的转换后的模型文件提供服务。请注意，您可能需要将服务器配置为[允许跨源资源共享（CORS）](https://enable-cors.org/)，以便允许使用JavaScript获取文件。

然后通过提供model.json文件的URL将模型加载到TensorFlow.js中：

*// JavaScript*

import \* as tf from '@tensorflow/tfjs';

const model = await tf.loadModel('https://foo.bar/tfjs\_artifacts/model.json');

现在，该模型已准备好进行推理，评估或重新培训。例如，加载的模型可以立即用于进行预测：

*// JavaScript*

const example = tf.fromPixels(webcamElement); *// for example*

const prediction = model.predict(example);

许多[TensorFlow.js示例](https://github.com/tensorflow/tfjs-examples)采用此方法，使用已在Google云端存储上转换和托管的预训练模型。

请注意，您使用model.json文件名引用整个模型。 loadModel(...)获取model.json，然后发出额外的HTTP（S）请求以获取权model.json重清单中引用的分片权重文件。这种方法允许所有这些文件由浏览器缓存（也可能由互联网上的其他缓存服务器缓存），因为model.json和权重分片都小于典型的缓存文件大小限制。因此，模型很可能在随后的场合加载更快。

## 支持的功能

TensorFlow.js图层目前仅支持使用标准Keras构造的Keras模型。使用不受支持的操作或图层的模型（例如自定义图层，Lambda图层，自定义损失或自定义指标）无法自动导入，因为它们依赖于无法可靠地转换为JavaScript的Python代码。

保存并加载tf.Model

本教程介绍如何在TensorFlow.js中保存和加载模型。保存和加载模型是一项重要的功能。例如，如何保存仅由浏览器中可用的数据（例如，来自附加传感器的图像和音频数据）微调的模型的权重，以便在用户加载时模型将处于已调整的状态页面又来了？还要考虑Layers API允许您[tf.Model](https://js.tensorflow.org/api/latest/#class:Model)在浏览器中从头开始创建名为s的模型这一事实 。你如何保存以这种方式创建的模型？自版本0.11.1起，TensorFlow.js中提供的save / load API解决了这些问题。

注意：本文档是关于保存和加载tf.Models（即tfjs-layers API中的Keras样式模型）。tf.FrozenModel目前还不支持保存和加载（即从TensoFlow SavedModels 加载的模型），并且正在积极开展工作。

保存tf.Model

让我们从最基本，最轻松的方式开始，将tf.Model：保存到Web浏览器的本地存储。本地存储是标准的客户端数据存储。保存在那里的数据可以在同一页面的多个负载中持续存在。您可以在此[MDN页面上](https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage)了解有关它的更多信息。

假设你有一个tf.Model名为的对象model。无论是从头开始使用Layers API还是从预训练的Keras模型加载/微调，您都可以使用一行代码将其保存到本地存储：

const saveResult = await model.save('localstorage://my-model-1');

有些事情值得指出：

* 该save方法采用类似于URL的字符串参数，该参数以**方案**开头。在这种情况下，我们使用该localstorage://方案指定将模型保存到本地存储。
* 该计划之后是一条**路径**。在保存到本地存储的情况下，路径只是一个任意字符串，用于唯一标识要保存的模型。例如，当您从本地存储加载模型时，将使用它。
* 该save方法是异步的，因此您需要使用then或者await如果其完成形成其他操作的前提条件。
* 返回值model.save是一个JSON对象，它携带一些可能有用的信息，例如模型拓扑和权重的字节大小。
* 任何tf.Model，无论它是否由 [tf.sequential](https://js.tensorflow.org/api/latest/#sequential) 构成，它包含哪些类型的层，都可以这种方式保存。

下表列出了所有当前支持的保存模型目的地及其相应的方案和示例。

| **保存目的地** | **方案字符串** | **代码示例** |
| --- | --- | --- |
| 本地存储（浏览器） | localstorage:// | await model.save('localstorage://my-model-1'); |
| IndexedDB（浏览器） | indexeddb:// | await model.save('indexeddb://my-model-1'); |
| 触发文件下载（浏览器） | downloads:// | await model.save('downloads://my-model-1'); |
| HTTP请求（浏览器） | http:// 要么 https:// | await model.save('http://model-server.domain/upload'); |
| 文件系统（Node.js） | file:// | await model.save('file:///tmp/my-model-1'); |

我们将在以下部分中扩展一些保存路线。

IndexedDB的

[IndexedDB](https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API) 是大多数主流Web浏览器支持的另一个客户端数据存储。与本地存储不同，它更好地支持存储大型二进制数据（BLOB）和更大的配额。因此，tf.Model与本地存储相比，保存到IndexedDB通常可以提供更好的存储效率和更大的大小限制。

文件下载

该downloads://方案后面的字符串是将要下载的文件名称的前缀。例如，该行将 model.save('downloads://my-model-1')导致浏览器下载两个共享相同文件名前缀的文件：

1. 一个名为的文本JSON文件my-model-1.json，它在其modelTopology字段中包含模型的拓扑，并在其字段中显示权重清单 weightsManifest。
2. 一个二进制文件，带有名为的权重值my-model-1.weights.bin。

这些文件的格式与[tensorflowjs转换](https://pypi.org/project/tensorflowjs/)器从[Keras](https://pypi.org/project/tensorflowjs/) HDF5文件转换的工件格式相同。

注意：某些浏览器要求用户在同时下载多个文件之前授予权限。

HTTP请求

如果tf.Model.save使用HTTP / HTTPS URL调用，则模型的拓扑和权重将通过[POST](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST)请求发送到指定的HTTP服务器。POST请求的主体具有一个名为的格式 multipart/form-data。它是用于将文件上载到服务器的标准MIME格式。正文由两个文件组成，文件名model.json和文件名 model.weights.bin。文件格式与downloads://方案触发的下载文件格式相同（参见上文）。此 [文档字符串](https://js.tensorflow.org/api/latest/#tf.io.browserHTTPRequest) 包含一个Python代码片段，演示了如何使用[烧瓶](http://flask.pocoo.org/) Web框架以及Keras和TensorFlow来处理源自save请求的有效负载并将其重新构建为服务器内存中的Keras Model对象。

通常，您的HTTP服务器对请求有特殊约束和要求，例如HTTP方法，标头和身份验证凭据。您可以save通过将URL字符串参数替换为调用来获得对请求的这些方面的细粒度控制tf.io.browserHTTPRequest。它是一个更详细的API，但它在控制由此产生的HTTP请求时提供了更大的灵活性save。例如：

await model.save(tf.io.browserHTTPRequest(

'http://model-server.domain/upload',

{method: 'PUT', headers: {'header\_key\_1': 'header\_value\_1'}}));

本机文件系统

TensorFlow.js可以在Node.js中使用。有关更多详细信息，请参阅 [tfjs-node项目](https://github.com/caisq/tfjs-node)。与Web浏览器不同，Node.js可以直接访问本地文件系统。因此，您可以将tf.Models 保存到文件系统，这与在Keras中将模型保存到磁盘的方式非常相似。为此，首先确保已导入@tensorflow/tfjs-node包，例如，使用Node.js的require语法：

require('@tensorflow/tfjs-node');

导入后，file://URL方案可用于模型保存和加载。对于模型保存，该方案后面是要保存模型工件的目录的路径，例如：

await model.save('file:///tmp/my-model-1');

上面的命令将在目录中生成一个model.json文件和一个weights.bin文件/tmp/my-model-1。这两个文件的格式与上面的“文件下载”和“HTTP请求”部分中描述的文件格式相同。保存模型后，可以将其加载回运行TensorFlow.js的Node.js程序，或者为TensorFlow.js的浏览器版本提供服务。要实现前者，请tf.loadModel()使用model.json文件路径调用：

const model = await tf.loadModel('file:///tmp/my-model-1/model.json');

要实现后者，请将保存的文件作为Web服务器的静态文件提供。

正在加载tf.Model

tf.Model如果之后无法加载模型，则保存s的功能将无用。通过tf.loadModel使用基于方案的类似URL的字符串参数调用来完成模型加载。tf.Model.save在大多数情况下，字符串参数是对称的 。下表给出了支持的加载路径的摘要：

| **装货路线** | **方案字符串** | **例** |
| --- | --- | --- |
| 本地存储（浏览器） | localstorage:// | await tf.loadModel('localstorage://my-model-1'); |
| IndexedDB（浏览器） | indexeddb:// | await tf.loadModel('indexeddb://my-model-1'); |
| 用户上传的文件（浏览器） | N / A | await tf.loadModel(tf.io.browserFiles([modelJSONFile, weightsFile])); |
| HTTP请求（浏览器） | http:// 要么https:// | await tf.loadModel('http://model-server.domain/download/model.json'); |
| 文件系统（Node.js） | file:// | await tf.loadModel('file:///tmp/my-model-1/model.json'); |

在所有加载路由中， 如果加载成功则tf.loadModel返回一个（Promiseof）tf.Model对象，Error如果失败则抛出一个。

从本地存储或IndexedDB加载与保存完全对称。但是，从用户上传的文件加载与从浏览器下载文件完全不对称。特别是，用户上传的文件不表示为类似URL的字符串。相反，他们被指定为Array的 [文件](https://developer.mozilla.org/en-US/docs/Web/API/File)对象。典型的工作流程是让用户使用HTML [文件输入](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/file) 元素从本地文件系统中选择文件

<input name="json-upload" type="file" />

<input name="weights-upload" type="file" />

这些将在浏览器中显示为两个“选择文件”按钮，用户可以使用这些按钮来选择文件。一旦用户分别在两个文件输入中选择了model.json文件和权重文件，文件对象将在相应的HTML元素下可用，并且它们可用于加载tf.Model 如下：

const jsonUpload = document.getElementById('json-upload');

const weightsUpload = document.getElementById('weights-upload');

const model = await tf.loadModel(

tf.io.browserFiles([jsonUpload.files[0], weightsUpload.files[0]]));

从HTTP请求加载模型对于通过HTTP请求保存模式也略微不对称。特别是，tf.loadModel将URL或路径作为model.json文件，如上表中的示例所示。这是自TensorFlow.js初始发布以来一直存在的API。

管理存储在浏览器Local Storage和IndexedDB中的模型

正如你在上面学到，你可以存储tf.Model的拓扑结构和权在用户的客户端浏览器的数据存储，包括本地存储和索引资料库，通过使用代码，如 model.save('localstorage://my-model')和model.save('indexeddb://my-model')。但到目前为止，您如何找到存储的模型？这可以通过使用tf.ioAPI 附带的模型管理方法来实现 ：

*// List models in Local Storage.*

console.log(await tf.io.listModels());

方法的返回值listModels不仅包括存储模型的路径，还包括一些关于它们的简短元数据，例如拓扑和权重的字节大小。

管理API还允许您复制，移动或删除现有模型。例如：

*// Copy model from existing path to a new path.*

*// Copying between Local Storage and IndexedDB is supported.*

tf.io.copyModel('localstorage://my-model', 'indexeddb://cloned-model');

*// Move model from a path to another.*

*// Moving between Local Storage and IndexedDB is supported.*

tf.io.moveModel('localstorage://my-model', 'indexeddb://cloned-model');

*// Remove model.*

tf.io.removeModel('indexeddb://cloned-model');

将保存的tf.Model转换为Keras格式

如上所述，有两种方法可以保存tf.Model as文件：

* 通过文件从Web浏览器下载，使用该downloads://方案
* 使用该file://方案将模型直接写入Node.js中的本机文件系统 。使用[tensorflowjs转换器](https://pypi.org/project/tensorflowjs/)，您可以将这些文件转换为HDF5格式，然后可以将其加载到Python中的Keras中。例如：

*# Suppose you have downloaded `my-model-1.json`, accompanied by a weights file.*

pip install tensorflowjs

tensorflowjs\_converter \

--input\_format tensorflowjs --output\_format keras \

./my-model-1.json /tmp/my-model-1.h5

将基于TensorFlow GraphDef的模型导入TensorFlow.js

基于TensorFlow GraphDef的模型（通常通过Python API创建）可以采用以下格式之一保存：

1. TensorFlow [SavedModel](https://www.tensorflow.org/programmers_guide/saved_model#overview_of_saving_and_restoring_models)
2. [冷冻模型](https://www.tensorflow.org/mobile/prepare_models#how_do_you_get_a_model_you_can_use_on_mobile)
3. [会话包](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/session_bundle/README.md)
4. [Tensorflow Hub模块](https://www.tensorflow.org/hub/)

所有上述格式都可以通过[TensorFlow.js转换器转换](https://github.com/tensorflow/tfjs-converter)为TensorFlow.js Web友好格式，可以直接加载到TensorFlow.js进行推理。

（注意：TensorFlow已弃用会话包格式，请将模型迁移到SavedModel格式。）

要求

转换过程需要Python环境; 你可能想用[pipenv](https://github.com/pypa/pipenv)或[virtualenv](https://virtualenv.pypa.io/)保持一个孤立的。要安装转换器，请运行以下命令：

pip install tensorflowjs

将TensorFlow模型导入TensorFlow.js需要两个步骤。首先，将现有模型转换为TensorFlow.js Web格式，然后将其加载到TensorFlow.js中。

步骤1.将现有TensorFlow模型转换为TensorFlow.js Web格式

运行pip包提供的转换器脚本：

用法：SavedModel示例：

tensorflowjs\_converter \

--input\_format=tf\_saved\_model \

--output\_node\_names='MobilenetV1/Predictions/Reshape\_1' \

--saved\_model\_tags=serve \

/mobilenet/saved\_model \

/mobilenet/web\_model

冷冻模型示例：

tensorflowjs\_converter \

--input\_format=tf\_frozen\_model \

--output\_node\_names='MobilenetV1/Predictions/Reshape\_1' \

/mobilenet/frozen\_model.pb \

/mobilenet/web\_model

会话包模型示例：

tensorflowjs\_converter \

--input\_format=tf\_session\_bundle \

--output\_node\_names='MobilenetV1/Predictions/Reshape\_1' \

/mobilenet/session\_bundle \

/mobilenet/web\_model

Tensorflow Hub模块示例：

tensorflowjs\_converter \

--input\_format=tf\_hub \

'https://tfhub.dev/google/imagenet/mobilenet\_v1\_100\_224/classification/1' \

/mobilenet/web\_model

| **位置参数** | **描述** |
| --- | --- |
| input\_path | 保存的模型目录，会话包目录，冻结模型文件或TensorFlow Hub模块句柄或路径的完整路径。 |
| output\_path | 所有输出工件的路径。 |

| **选项** | **描述** |
| --- | --- |
| --input\_format | 输入模型的格式，使用tf\_saved\_model表示SavedModel，tf\_frozen\_model表示冻结模型，tf\_session\_bundle表示会话束，tf\_hub表示TensorFlow Hub模块，keras表示Keras HDF5。 |
| --output\_node\_names | 输出节点的名称，以逗号分隔。 |
| --saved\_model\_tags | 仅适用于SavedModel转换，以逗号分隔格式加载MetaGraphDef的标签。默认为serve。 |
| --signature\_name | 仅适用于TensorFlow Hub模块转换，签名加载。默认为default。请参阅https://www.tensorflow.org/hub/common\_signatures/。 |

使用以下命令获取详细帮助消息：

tensorflowjs\_converter --help

转换器生成的文件

上面的转换脚本生成3种类型的文件：

* web\_model.pb （数据流图）
* weights\_manifest.json （重量清单文件）
* group1-shard\\*of\\* （二进制权重文件的集合）

例如，以下是在以下位置转换并提供的MobileNet模型：

https://storage.cloud.google.com/tfjs-models/savedmodel/mobilenet\_v1\_1.0\_224/optimized\_model.pb

https://storage.cloud.google.com/tfjs-models/savedmodel/mobilenet\_v1\_1.0\_224/weights\_manifest.json

https://storage.cloud.google.com/tfjs-models/savedmodel/mobilenet\_v1\_1.0\_224/group1-shard1of5

...

https://storage.cloud.google.com/tfjs-models/savedmodel/mobilenet\_v1\_1.0\_224/group1-shard5of5

第2步：在浏览器中加载并运行

1. 安装tfjs-converter npm包

yarn add @tensorflow/tfjs 要么 npm install @tensorflow/tfjs

1. 实例化[FrozenModel类](https://github.com/tensorflow/tfjs-converter/src/executor/frozen_model.ts)并运行推理。

import \* as tf from '@tensorflow/tfjs';

import {loadFrozenModel} from '@tensorflow/tfjs-converter';

const MODEL\_URL = 'https://.../mobilenet/web\_model.pb';

const WEIGHTS\_URL = 'https://.../mobilenet/weights\_manifest.json';

const model = await loadFrozenModel(MODEL\_URL, WEIGHTS\_URL);

const cat = document.getElementById('cat');

model.execute({input: tf.fromPixels(cat)});

查看我们的[MobileNet演示版](https://github.com/tensorflow/tfjs-converter/demo/mobilenet/README.md)。

如果您的服务器请求访问模型文件的凭据，您可以提供可选的RequestOption参数，该参数将直接传递给fetch函数调用。

const model = await loadFrozenModel(MODEL\_URL, WEIGHTS\_URL,

{credentials: 'include'});

有关详细信息，请参阅[fetch（）文档](https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/fetch)。

支持的操作

目前TensorFlow.js仅支持一组有限的TensorFlow Ops。查看 [完整列表](https://github.com/tensorflow/tfjs-converter/docs/supported_ops.md)。如果您的模型使用任何不受支持的操作，则tensorflowjs\_converter脚本将失败并生成模型中不受支持的操作的列表。请提交[问题](https://github.com/tensorflow/tfjs/issues)，告诉我们您需要支持的操作。

仅加载重量

如果您只想加载权重，可以使用以下代码段。

import \* as tf from '@tensorflow/tfjs';

const weightManifestUrl = "https://example.org/model/weights\_manifest.json";

const manifest = await fetch(weightManifestUrl);

this.weightManifest = await manifest.json();

const weightMap = await tf.io.loadWeights(

this.weightManifest, "https://example.org/model");

创建自定义WebGL操作

要定义自定义WebGL操作，我们所要做的就是创建一个实现的对象tf.webgl.GPGPUProgram。

该接口定义为：

interface GPGPUProgram {

variableNames: string[];

outputShape: number[];

userCode: string;

supportsBroadcasting?: boolean;

}

对于一个人为的例子，让我们实现一个计算的操作f(x) = x \* x + x。

这个的GLSL代码是：

void main() {

float x = getXAtOutCoords();

float value = x \* x + x;

setOutput(value);

}

其中getXAtOutCoords和setOutput是[由Tensorflow.js提供](https://js.tensorflow.org/tutorials/custom-webgl-op.html#stdlib)到着色器。请注意，为输出张量中的每个值调用main函数。

完整的GPGPUProgram定义是：

const squareAndAddKernel = inputShape => ({

variableNames: ['X'],

outputShape: inputShape.slice(),

userCode: `

void main() {

float x = getXAtOutCoords();

float value = x \* x + x;

setOutput(value);

}

`

})

要运行此操作，您可以使用tf.ENV.backend.compileAndRun(program: GPGPUProgram, inputs: tf.Tensor[]): tf.Tensor。请注意，如果后端不是webgl后端，则这将是未定义的。

const x = tf.tensor([1, 2, 3, 4]);

const program = squareAndAddKernel(x.shape);

const result = tf.ENV.backend.compileAndRun(program, [x]);

但是，我们可能还想为此op定义渐变，以便渐变可以通过它反向传播。

为此，我们使用[tf.customGrad](https://js.tensorflow.org/api/latest/#customGrad)。

const squareAndAddBackpropKernel = inputShape => ({

variableNames: ['X'],

outputShape: inputShape.slice(),

userCode: `

void main() {

float x = getXAtOutCoords();

float value = 2.0 \* x + 1.0;

setOutput(value);

}

`

});

const squareAndAdd = tf.customGrad(x => {

const backend = tf.ENV.backend;

const program = squareAndAddKernel(x.shape);

const backpropProgram = squareAndAddBackpropKernel(x.shape);

const value = backend.compileAndRun(program, [x]);

const gradFunc = dy =>

[backend.compileAndRun(backpropProgram, [x]).mul(dy)];

return {value, gradFunc}

});

然后我们可以使用它：

const x = tf.tensor([1, 2, 3, 4]);

const value = squareAndAdd(x);

const grads = tf.grad(x => squareAndAdd(x));

const dx = grads(input);

*// value == [3, 6, 12, 20]*

*// dx == [3, 5, 7, 9]*

或者更简洁：

const {value, grad} = tf.valueAndGrad(squareAndAdd)(x);

由Tensorflow.js生成的GLSL函数

Tensorflow.js生成可用于从输入张量读取并写入输出张量的函数，以及其他数字实用程序函数。这些由[着色器编译器](https://github.com/tensorflow/tfjs-core/blob/master/src/kernels/webgl/shader_compiler.ts)预先添加到您的代码中。

* void setOutput(float value)
  + 设置运行片段着色器的坐标的输出值（相当于gl\_FragCoord = vec4(value, 0.0, 0.0, 0.0)）。
* indexType getOutputCoords()
  + 其中indexType一个int | ivec2 | ivec3 | ivec4 | ivec5 | ivec6。
  + 返回int如果输出张量为rank-0或rank-1，否则返回ivecNN == rank。这是此程序将写入的输出张量中的单元格的坐标。
* Tensorflow.js生成GLSL函数以从输入张量进行采样。这些形式如下：
* float get{VarName}AtOutCoords()
* float get{VarName}() *// rank-0 input*
* float get{VarName}(int x) *// rank-1 input*
* float get{VarName}(int x, int y) *// rank-2 input*
* float get{VarName}(int x, int y, int z) *// rank-3 input*
* float get{VarName}(int x, int y, int z, int w) *// rank-4 input*
* *// continue as above for rank-5 & rank-6*
* *// For example, for rank-2 Tensor named x:*

*// float getX(int x, int y)*

哪里VarName是为中定义的变量名variableNames的数组GPGPUProgram中**与captialised的第一个字母**。这意味着对于名为的变量matrix，TF.js将生成getMatrix。

其中许多函数都取决于输入张量的等级，所以在你的GPGPUProgram经常想要根据inputShapes 的行列发出不同的代码。例如，如果get{VarName}AtOutCoords()不存在，我们可能写成squareAndAddKernel：

const squareAndAddKernel = inputShape => ({

const variableNames = ['X']

const outputShape = inputShape.slice()

const rank = outputShape.length

const coordSnippets = ['',

'coords',

'coords.x, coords.y',

'coords.x, coords.y, coords.z',

'coords.x, coords.y, coords.z, coords.w']

const coordType = rank < 2 ? 'int' : `ivec${rank}`

const userCode = `

void main() {

${coordType} coords = getOutputCoords();

float x = getX(${coordSnippets[rank]});

setOutput(x \* x + x);

}`

return {variableNames, outputShape, userCode}

})

* + bool isNaN(float val)
    - true如果val是a NaN，否则为false。
  + int round(float value)
    - 舍value入到最接近的整数。
  + int imod(int x, int y)
    - 与float mod(float x, float y)int 相同，因为GLSL不提供给我们一个。
  + float random(float seed)
    - 返回一个伪随机数，基于Dav Hoskins的论坛，网址为https://www.shadertoy.com/view/4djSRW。

TensorFlow Lite简介

TensorFlow Lite是TensorFlow针对移动和嵌入式设备的轻量级解决方案。它支持设备内机器学习推理，具有低延迟和小二进制大小。TensorFlow Lite还支持[Android神经​​网络API的](https://developer.android.google.cn/ndk/guides/neuralnetworks/index.html)硬件加速。

TensorFlow Lite使用许多技术来实现低延迟，例如优化移动应用程序的内核，预融合激活以及允许更小和更快（定点数学）模型的量化内核。

我们的大多数TensorFlow Lite文档 暂时都[在GitHub](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/lite)上。

TensorFlow Lite包含什么？

TensorFlow Lite支持一组核心运算符，包括量化和浮点运算，它们已针对移动平台进行了调整。它们结合了预融合激活和偏置，以进一步提高性能和量化精度。此外，TensorFlow Lite还支持在模型中使用自定义操作。

TensorFlow Lite基于[FlatBuffers](https://google.github.io/flatbuffers/)定义了一种新的模型文件格式 。FlatBuffers是一个高效的开源跨平台序列化库。它类似于 [协议缓冲区](https://developers.google.cn/protocol-buffers/?hl=en)，但主要区别在于FlatBuffers在访问数据之前不需要对辅助表示进行解析/解包步骤，通常与每个对象的内存分配相结合。此外，FlatBuffers的代码占用空间比协议缓冲区小一个数量级。

TensorFlow Lite拥有一个新的移动优化解释器，其主要目标是保持应用程序的精简和快速。解释器使用静态图形排序和自定义（动态性较小）内存分配器来确保最小的负载，初始化和执行延迟。

TensorFlow Lite提供了一个利用硬件加速的接口（如果在设备上可用）。它通过 [Android神经​​网络API实现](https://developer.android.google.cn/ndk/guides/neuralnetworks/index.html)，可在Android 8.1（API级别27）及更高版本上使用。

为什么我们需要一个新的移动专用库？

机器学习正在改变计算范式，我们看到了移动和嵌入式设备上新用例的新趋势。在相机和语音交互模型的推动下，消费者的期望也趋向于与其设备进行自然的，类似人的交互。

有几个因素引起了这个领域的兴趣：

* 硅层的创新为硬件加速提供了新的可能性，而Android神经​​网络API等框架可以轻松利用这些功能。
* 最近在实时计算机视觉和口语理解方面取得的进展使得移动优化的基准模型成为开源的（例如MobileNets，SqueezeNet）。
* 广泛可用的智能设备为设备智能创造了新的可能性。
* 对用户数据不需要离开移动设备的更强大的用户数据隐私范例感兴趣。
* 能够提供“离线”用例，其中设备不需要连接到网络。

我们相信下一波机器学习应用程序将在移动和嵌入式设备上进行大量处理。

TensorFlow Lite亮点

TensorFlow Lite提供：

* 一组核心运营商，包括量化和浮动，其中许多已经针对移动平台进行了调整。这些可用于创建和运行自定义模型。开发人员还可以编写自己的自定义运算符并在模型中使用它们。
* 一种新的基于[FlatBuffers](https://google.github.io/flatbuffers/)的模型文件格式。
* 具有内核优化的设备上解释器，可在移动设备上更快地执行。
* TensorFlow转换器将TensorFlow训练的模型转换为TensorFlow Lite格式。
* 尺寸较小：当所有支持的运营商链接时，TensorFlow Lite小于300KB，当仅使用支持InceptionV3和Mobilenet所需的运营商时，小于200KB。
* **预先测试的型号：**

以下所有型号均可保证开箱即用：

* + 初始V3，一种用于检测图像中存在的主要对象的流行模型。
  + [MobileNets](https://github.com/tensorflow/models/blob/master/research/slim/nets/mobilenet_v1.md)，一系列移动优先计算机视觉模型，旨在有效地最大限度地提高准确性，同时注意到设备或嵌入式应用程序的受限资源。它们是小型，低延迟，低功耗模型，参数化以满足各种用例的资源限制。它们可以用于分类，检测，嵌入和分割。MobileNet模型 比Inception V3 更小但[精度更低](https://research.googleblog.com/2017/06/mobilenets-open-source-models-for.html)。
  + 在设备智能回复上，一种设备上模型，通过建议与上下文相关的消息，为传入的文本消息提供一键式回复。该模型专为内存受限设备（如手表和手机）而构建，并已成功用于将[Android Wear](https://research.googleblog.com/2017/02/on-device-machine-intelligence.html)上的 [智能回复](https://research.googleblog.com/2017/02/on-device-machine-intelligence.html) 呈现给所有第一方和第三方应用。

另请参阅[TensorFlow Lite支持的模型](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/lite/g3doc/models.md)的完整列表 ，包括型号，性能编号和可下载的模型文件。

* MobileNet模型的量化版本，其运行速度比CPU上的非量化（浮点）版本快。
* 新的Android演示应用程序，用于说明使用TensorFlow Lite和量化的MobileNet模型进行对象分类。
* Java和C ++ API支持

入门

我们建议您使用上面指出的预测试模型试用TensorFlow Lite。如果您有现有型号，则需要测试您的型号是否与转换器和支持的操作员集兼容。要测试您的模型，请参阅[GitHub上](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/lite)的 [文档](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/lite)。

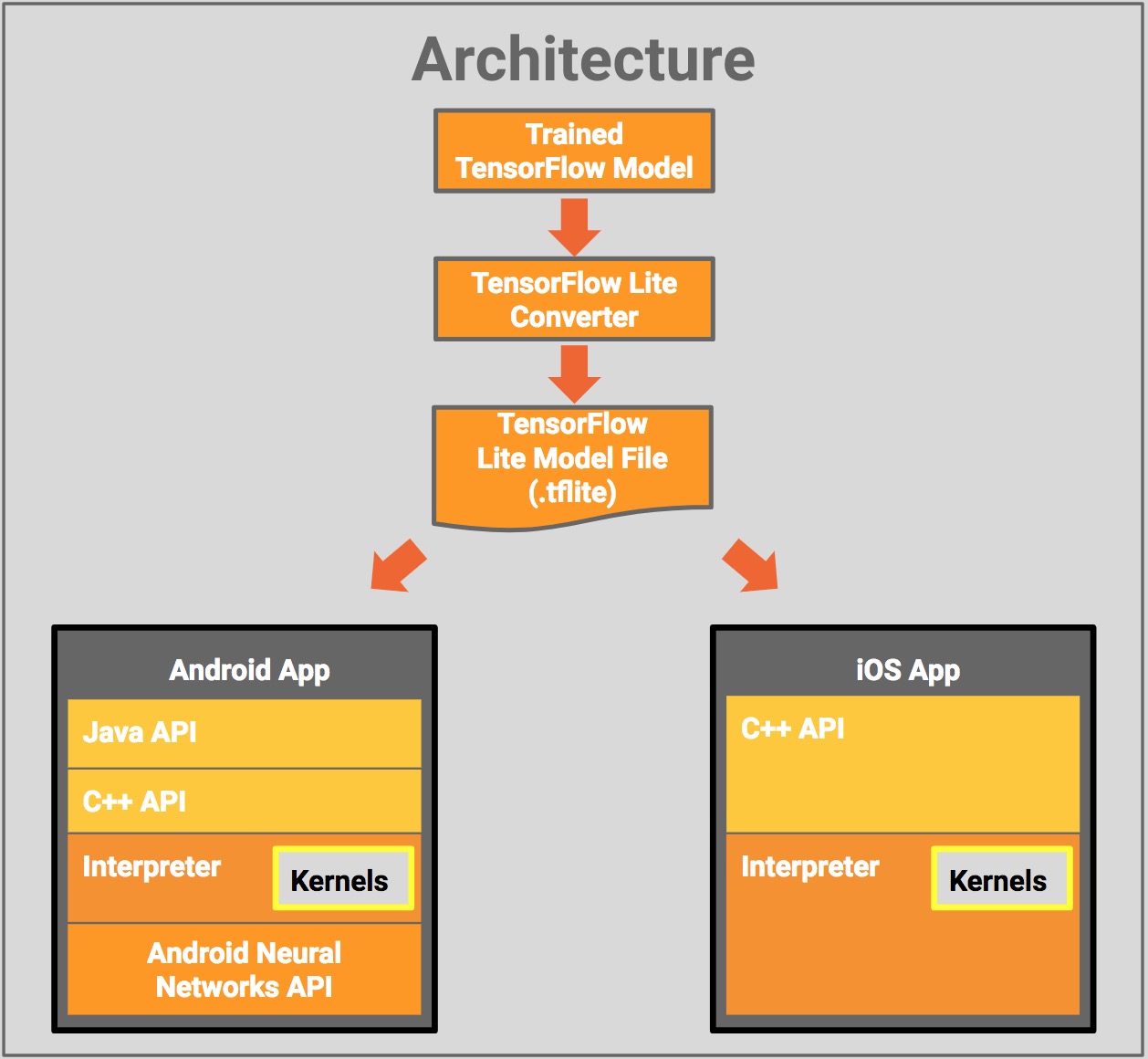
为自定义数据集重新调整Inception-V3或MobileNet

上面提到的预训练模型已经在ImageNet数据集上进行了训练，该数据集由1000个预定义类组成。如果这些类对您的用例不相关或有用，则需要重新训练这些模型。这种技术称为转移学习，它从已经训练过一个问题的模型开始，然后再对类似的问题进行再训练。从头开始深度学习可能需要数天时间，但转移学习可以相当快地完成。为此，您需要生成标有相关类的自定义数据集。

在[为诗人TensorFlow](https://codelabs.developers.google.com/codelabs/tensorflow-for-poets/) 代码实验室通过这个过程走一步一步的。重新训练代码支持对浮点和量化推理进行重新训练。

TensorFlow Lite架构

下图显示了TensorFlow Lite的架构设计：



从磁盘上经过训练的TensorFlow模型开始，您将.tflite使用TensorFlow Lite转换器将该模型转换为TensorFlow Lite文件格式（）。然后，您可以在移动应用程序中使用该转换后的文件。

部署TensorFlow Lite模型文件使用：

* Java API：围绕Android上的C ++ API的便利包装器。
* C ++ API：加载TensorFlow Lite模型文件并调用Interpreter。Android和iOS都提供相同的库。
* 解释器：使用一组内核执行模型。解释器支持选择性内核加载; 没有内核它只有100KB，加载了所有内核300KB。这比TensorFlow Mobile要求的1.5M显着降低。
* 在部分Android设备上，Interpreter将使用Android神经​​网络API进行硬件加速，如果没有，则默认为CPU执行。

您还可以使用可由Interpreter使用的C ++ API实现自定义内核。

未来的工作

在未来的版本中，TensorFlow Lite将支持更多模型和内置运算符，包括定点和浮点模型的性能改进，工具的改进，以便更轻松地开发工作流程以及支持其他更小的设备等。在我们继续开发的过程中，我们希望TensorFlow Lite能够大大简化针对小型设备模型的开发人员体验。

未来的计划包括使用专门的机器学习硬件来为特定设备上的特定模型获得最佳性能。

TF Lite开发人员指南

在移动应用中使用TensorFlow Lite模型需要多个注意事项：您必须选择预先训练的或自定义模型，将模型转换为TensorFLow Lite格式，最后将模型集成到您的应用中。

1.选择一个型号

根据用例，您可以选择一种流行的开源模型，例如*InceptionV3*或*MobileNets*，并使用自定义数据集重新训练这些模型，甚至可以构建您自己的自定义模型。

使用预先训练的模型

[MobileNets](https://research.googleblog.com/2017/06/mobilenets-open-source-models-for.html) 是TensorFlow的移动优先计算机视觉模型系列，旨在有效地最大限度地提高准确性，同时考虑到设备或嵌入式应用程序的受限资源。MobileNets是小型，低延迟，低功耗模型，参数化以满足各种用途的资源限制。它们可用于分类，检测，嵌入和分割 - 类似于其他流行的大型模型，例如 [Inception](https://arxiv.org/pdf/1602.07261.pdf)。Google为[MobileNets](http://www.image-net.org/challenges/LSVRC/)提供了16个经过预先培训的 [ImageNet](http://www.image-net.org/challenges/LSVRC/)分类检查点，可用于各种规模的移动项目。

[Inception-v3](https://arxiv.org/abs/1512.00567)是一种图像识别模型，可以实现相当高的准确度，可以识别1000个类别的一般对象，例如“斑马”，“达尔马提亚”和“洗碗机”。该模型使用卷积神经网络从输入图像中提取一般特征，并基于具有完全连接和softmax层的那些特征对它们进行分类。

[On Device Smart Reply](https://research.googleblog.com/2017/02/on-device-machine-intelligence.html) 是一种设备上模型，通过建议与上下文相关的消息，为传入的文本消息提供一键式回复。该模型专为内存受限设备（如手表和手机）而构建，并已成功用于Android Wear上的Smart Replies。目前，此模型是特定于Android的。

这些预先训练的模型[可供下载](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/lite/g3doc/models.md)

为自定义数据集重新训练Inception-V3或MobileNet

这些预先训练的模型在*ImageNet*数据集上进行训练，该数据集包含1000个预定义的类。如果这些类不足以满足您的使用需求，则需要重新训练模型。这种技术称为 *转移学习*，从已经训练过问题的模型开始，然后在类似的问题上重新训练模型。从头开始深度学习可能需要数天时间，但转移学习相当快。为此，您需要生成标有相关类的自定义数据集。

在[为诗人TensorFlow](https://codelabs.developers.google.com/codelabs/tensorflow-for-poets/) 代码实验室通过再培训过程中走一步一步的。该代码支持浮点和量化推理。

训练自定义模型

开发人员可以选择使用Tensorflow训练自定义模型（有关构建和训练模型的示例，请参阅[TensorFlow教程](https://tensorflow.google.cn/tutorials/)）。如果您已经编写了模型，则第一步是将其导出到tf.GraphDef文件中。这是必需的，因为某些格式不会将模型结构存储在代码之外，我们必须与框架的其他部分进行通信。请参阅 [导出推理图](https://github.com/tensorflow/models/blob/master/research/slim/README.md) 以为自定义模型创建.pb文件。

TensorFlow Lite目前支持TensorFlow运算符的子集。 有关支持的运算符及其用法，请参阅[TensorFlow Lite和TensorFlow兼容性指南](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/lite/g3doc/tf_ops_compatibility.md)。这套运营商将在未来的Tensorflow Lite版本中继续增长。

2.转换模型格式

在上一步中生成（或下载）的模型是*标准的* Tensorflow模型，您现在应该有.pb或.pbtxt tf.GraphDef文件。必须转换使用传输学习（重新训练）或自定义模型生成的模型 - 但是，我们必须先冻结图形以将模型转换为Tensorflow Lite格式。此过程使用多种模型格式：

* tf.GraphDef（.pb） - 表示TensorFlow训练或计算图的protobuf。它包含运算符，张量和变量定义。
* *CheckPoint*（.ckpt） - 来自TensorFlow图的转化变量。由于这不包含图形结构，因此无法自行解释。
* FrozenGraphDef- 它的子类GraphDef不包含变量。A GraphDef可以FrozenGraphDef通过获取CheckPoint和a 转换为a GraphDef，并使用从CheckPoint检索的值将每个变量转换为常量。
* SavedModel-A GraphDef和CheckPoint带有标记，用于标记模型的输入和输出参数。GraphDef可以从a中提取A 和CheckPoint SavedModel。
* *TensorFlow Lite模型*（.tflite） - 一个序列化的 [FlatBuffer](https://google.github.io/flatbuffers/)，包含用于TensorFlow Lite解释器的TensorFlow Lite运算符和张量，类似于 FrozenGraphDef。

冻结图

要将GraphDef.pb文件与TensorFlow Lite一起使用，您必须具有包含经过训练的权重参数的检查点。.pb文件仅包含图形的结构。将检查点值与图结构合并的过程称为*冻结图*。

您应该有一个检查点文件夹或下载它们以获得预先训练的模型（例如， [MobileNets](https://github.com/tensorflow/models/blob/master/research/slim/nets/mobilenet_v1.md)）。

要冻结图形，请使用以下命令（更改参数）：

freeze\_graph --input\_graph=/tmp/mobilenet\_v1\_224.pb \  
  --input\_checkpoint=/tmp/checkpoints/mobilenet-10202.ckpt \  
  --input\_binary=true \  
  --output\_graph=/tmp/frozen\_mobilenet\_v1\_224.pb \  
  --output\_node\_names=MobileNetV1/Predictions/Reshape\_1

input\_binary必须启用该标志，以便以二进制格式读取和写入protobuf。设置input\_graph和input\_checkpoint文件。

在output\_node\_names构建模型的代码之外可能并不明显。找到它们的最简单方法是使用[TensorBoard](https://codelabs.developers.google.com/codelabs/tensorflow-for-poets-2/#3) 或者可视化图形 graphviz。

冻结GraphDef现在可以转换为FlatBuffer格式（.tflite），以便在Android或iOS设备上使用。对于Android，Tensorflow Optimizing Converter工具支持浮点和量化模型。要将冻结转换为GraphDef.tflite格式：

toco --input\_file=$(pwd)/mobilenet\_v1\_1.0\_224/frozen\_graph.pb \  
  --input\_format=TENSORFLOW\_GRAPHDEF \  
  --output\_format=TFLITE \  
  --output\_file=/tmp/mobilenet\_v1\_1.0\_224.tflite \  
  --inference\_type=FLOAT \  
  --input\_type=FLOAT \  
  --input\_arrays=input \  
  --output\_arrays=MobilenetV1/Predictions/Reshape\_1 \  
  --input\_shapes=1,224,224,3

该input\_file参数应该引用冻结GraphDef包含模型架构文件。 这里使用的[frozen\_graph.pb](https://storage.googleapis.com/download.tensorflow.org/models/mobilenet_v1_1.0_224_frozen.tgz)文件可供下载。output\_file是生成TensorFlow Lite模型的地方。在input\_type和inference\_type 参数应设置为FLOAT，除非转换 [量化模型](https://tensorflow.google.cn/performance/quantization)。设置input\_array，output\_array和input\_shape论据并不那么简单。找到这些值的最简单方法是使用Tensorboard探索图形。重用参数以在freeze\_graph步骤中指定推理的输出节点。

也可以将Tensorflow Optimizing Converter与来自Python或命令行的protobuf一起使用（请参阅[toco\_from\_protos.py](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/lite/toco/python/toco_from_protos.py) 示例）。这允许您将转换步骤集成到模型设计工作流程中，确保模型可以轻松转换为移动推理图。例如：

import tensorflow as tf  
  
img = tf.placeholder(name="img", dtype=tf.float32, shape=(1, 64, 64, 3))  
val = img + tf.constant([1., 2., 3.]) + tf.constant([1., 4., 4.])  
out = tf.identity(val, name="out")  
  
with tf.Session() as sess:  
  tflite\_model = tf.contrib.lite.toco\_convert(sess.graph\_def, [img], [out])  
  open("converteds\_model.tflite", "wb").write(tflite\_model)

有关用法，请参阅Tensorflow Optimizing Converter [命令行示例](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/lite/toco/g3doc/cmdline_examples.md)。

有关故障排除帮助，请参阅 [Ops兼容性指南](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/lite/g3doc/tf_ops_compatibility.md)，如果这样做无效，请 [提出问题](https://github.com/tensorflow/tensorflow/issues)。

在[发展回购](https://github.com/tensorflow/tensorflow)包含转换后的可视化TensorFlow精简版机型的工具。要构建 [visualize.py](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/lite/tools/visualize.py) 工具：

bazel run tensorflow/contrib/lite/tools:visualize -- model.tflite model\_viz.html

这将生成一个交互式HTML页面，其中列出了子图，操作和图形可视化。

3.使用TensorFlow Lite模型在移动应用程序中进行推理

完成上述步骤后，您现在应该有一个.tflite模型文件。

Android的

由于Android应用程序是用Java编写的，而核心TensorFlow库是用C ++编写的，因此提供了一个JNI库作为接口。这仅用于推理 - 它提供加载图形，设置输入和运行模型以计算输出的能力。

开源Android演示应用程序使用JNI接口，可 [在GitHub上使用](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/lite/java/demo/app)。您还可以下载 [预建的APK](http://download.tensorflow.org/deps/tflite/TfLiteCameraDemo.apk)。有关详细信息，请参阅[Android演示](https://tensorflow.google.cn/lite/demo_android)指南。

在[Android手机](https://tensorflow.google.cn/lite/android_build)引导有Android上安装TensorFlow和设置说明bazel和Android工作室。

iOS版

要在iOS应用程序中集成TensorFlow模型，请参阅 [TensorFlow Lite for iOS](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/lite/g3doc/ios.md) 指南和[iOS演示](https://tensorflow.google.cn/lite/demo_ios)指南。

核心ML支持

Core ML是Apple产品中使用的机器学习框架。除了直接在您的应用程序中使用Tensorflow Lite模型外，您还可以将经过培训的Tensorflow模型转换为 [CoreML](https://developer.apple.com/machine-learning/)格式，以便在Apple设备上使用。要使用转换器，请参阅 [Tensorflow-CoreML转换器文档](https://github.com/tf-coreml/tf-coreml)。

树莓派

按照[RPi构建说明](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/lite/g3doc/rpi.md)为Raspberry Pi编译Tensorflow Lite 这将编译.a用于构建应用程序的静态库文件（）。有Python绑定和演示应用程序的计划。

Android演示应用

[GitHub上](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/lite/java/demo)提供[了](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/lite/java/demo)使用TensorFLow Lite的示例Android应用程序 。该演示是一个示例相机应用程序，使用量化的Mobilenet模型或浮点Inception-v3模型连续分类图像。要运行演示，需要运行Android 5.0（API 21）或更高版本的设备。

在演示应用程序中，使用TensorFlow Lite Java API进行推理。该演示应用程序实时对帧进行分类，显示最可能的分类。它还显示检测对象所用的时间。

将演示应用程序添加到您的设备有三种方法：

* 下载[预建的二进制APK](http://download.tensorflow.org/deps/tflite/TfLiteCameraDemo.apk)。
* 使用Android Studio构建应用程序。
* 下载TensorFlow Lite的源代码和演示，并使用bazel构建它。

下载预先构建的二进制文件

尝试演示的最简单方法是下载 [预先构建的二进制APK](https://storage.googleapis.com/download.tensorflow.org/deps/tflite/TfLiteCameraDemo.apk)

安装APK后，单击应用程序图标以启动该程序。第一次打开应用程序时，它会要求运行时权限以访问设备相机。演示应用程序打开设备的后置摄像头，识别摄像机视野中的物体。在图像的底部（如果设备处于横向模式，则位于图像的左侧），它显示分类的前三个对象和分类延迟。

使用JCenter的TensorFlow Lite AAR在Android Studio中构建

使用Android Studio尝试更改项目代码并编译演示应用程序：

* 安装最新版本的 [Android Studio](https://developer.android.google.cn/studio/index.html)。
* 确保Android SDK版本大于26且NDK版本大于14（在Android Studio设置中）。
* 将tensorflow/contrib/lite/java/demo目录导入为新的Android Studio项目。
* 安装它请求的所有Gradle扩展。

现在您可以构建并运行演示应用程序。

构建过程下载量化的[Mobilenet TensorFlow Lite模型](https://storage.googleapis.com/download.tensorflow.org/models/tflite/mobilenet_v1_224_android_quant_2017_11_08.zip)，并将其解压缩到assets目录中：tensorflow/contrib/lite/java/demo/app/src/main/assets/。

[TF Lite Android App页面](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/lite/java/demo/README.md)上提供了一些其他详细信息 。

使用其他型号

要使用其他模型：\*下载浮点[Inception-v3模型](https://storage.googleapis.com/download.tensorflow.org/models/tflite/inception_v3_slim_2016_android_2017_11_10.zip)。\*解压缩并复制inceptionv3\_non\_slim\_2015.tflite到assets目录。\*将[Camera2BasicFragment.java中](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/lite/java/demo/app/src/main/java/com/example/android/tflitecamerademo/Camera2BasicFragment.java)选择的分类[器](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/lite/java/demo/app/src/main/java/com/example/android/tflitecamerademo/Camera2BasicFragment.java)  
从：更改classifier = new ImageClassifierQuantizedMobileNet(getActivity());  
为：classifier = new ImageClassifierFloatInception(getActivity());。

从源代码构建TensorFlow Lite和演示应用程序

克隆TensorFlow回购

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

安装Bazel

如果bazel系统上未安装，请参阅 [安装Bazel](https://bazel.build/versions/master/docs/install.html)。

**注意：**Bazel目前不支持Windows上的Android版本。Windows用户应该下载 [预构建的二进制文件](https://storage.googleapis.com/download.tensorflow.org/deps/tflite/TfLiteCameraDemo.apk)。

安装Android NDK和SDK

Android NDK是构建本机（C / C ++）TensorFlow Lite代码所必需的。当前推荐的版本是*14b*，可以在[NDK Archives](https://developer.android.google.cn/ndk/downloads/older_releases.html#ndk-14b-downloads) 页面上找到 。

Android SDK和构建工具可以单独[下载，](https://developer.android.google.cn/tools/revisions/build-tools.html)也可以 作为[Android Studio的](https://developer.android.google.cn/studio/index.html)一部分使用 。要构建TensorFlow Lite Android演示，构建工具需要API> = 23（但它将在API> = 21的设备上运行）。

在TensorFlow存储库的根目录中，使用SDK和NDK的位置更新WORKSPACE文件 api\_level。如果您使用Android Studio安装它，则可以在SDK管理器中找到SDK路径。默认的NDK路径是：{SDK path}/ndk-bundle.例如：

android\_sdk\_repository (  
    name = "androidsdk",  
    api\_level = 23,  
    build\_tools\_version = "23.0.2",  
    path = "/home/xxxx/android-sdk-linux/",  
)  
  
android\_ndk\_repository(  
    name = "androidndk",  
    path = "/home/xxxx/android-ndk-r10e/",  
    api\_level = 19,  
)

[TF Lite Android App页面](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/lite/java/demo/README.md)上提供了一些其他详细信息 。

构建源代码

要构建演示应用程序，请运行bazel：

bazel build --cxxopt=--std=c++11 //tensorflow/contrib/lite/java/demo/app/src/main:TfLiteCameraDemo