Create an op

Note: To guarantee that your C++ custom ops are ABI compatible with TensorFlow's official pip packages, please follow the guide at Custom op repository (https://github.com/tensorflow/custom-op). It has an end-to-end code example, as well as Docker images for building and distributing your custom ops.

If you'd like to create an op that isn't covered by the existing TensorFlow library, we recommend that you first try writing the op in Python as a composition of existing Python ops or functions. If that isn't possible, you can create a custom C++ op. There are several reasons why you might want to create a custom C++ op:

- It's not easy or possible to express your operation as a composition of existing ops.
- It's not efficient to express your operation as a composition of existing primitives.
- You want to hand-fuse a composition of primitives that a future compiler would find difficult fusing.

For example, imagine you want to implement something like "median pooling", similar to the "MaxPool" operator, but computing medians over sliding windows instead of maximum values. Doing this using a composition of operations may be possible (e.g., using ExtractImagePatches and TopK), but may not be as performance- or memory-efficient as a native operation where you can do something more clever in a single, fused operation. As always, it is typically first worth trying to express what you want using operator composition, only choosing to add a new operation if that proves to be difficult or inefficient.

To incorporate your custom op you'll need to:

1. Register the new op in a C++ file. Op registration defines an interface (specification) for the op's functionality, which is independent of the op's implementation. For example, op registration defines the op's name and the op's inputs and outputs. It also defines the shape function that is used for tensor shape inference.
2. Implement the op in C++. The implementation of an op is known as a kernel, and it is the concrete implementation of the specification you registered in Step 1. There can be multiple kernels for different input / output types or architectures (for example, CPUs, GPUs).

3. Create a Python wrapper (optional). This wrapper is the public API that's used to create the op in Python. A default wrapper is generated from the op registration, which can be used directly or added to.
4. Write a function to compute gradients for the op (optional).
5. Test the op. We usually do this in Python for convenience, but you can also test the op in C++. If you define gradients, you can verify them with the Python `tf.test.compute_gradient_error`. See `relu_op_test.py` (https://www.tensorflow.org/code/tensorflow/python/kernel_tests/relu_op_test.py) as an example that tests the forward functions of Relu-like operators and their gradients.

Prerequisites

- Some familiarity with C++.
- Must have installed the TensorFlow binary (https://www.tensorflow.org/install), or must have downloaded TensorFlow source (https://www.tensorflow.org/install/source), and be able to build it.

Define the op interface

You define the interface of an op by registering it with the TensorFlow system. In the registration, you specify the name of your op, its inputs (types and names) and outputs (types and names), as well as docstrings and any attrs (#attrs) the op might require.

To see how this works, suppose you'd like to create an op that takes a tensor of **`int32`**s and outputs a copy of the tensor, with all but the first element set to zero. To do this, create a file named **`zero_out.cc`**. Then add a call to the **`REGISTER_OP`** macro that defines the interface for your op:

```
#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();
    });
```

This **`ZeroOut`** op takes one tensor **`to_zero`** of 32-bit integers as input, and outputs a tensor **`zeroed`** of 32-bit integers. The op also uses a shape function to ensure that the output tensor is the same shape as the input tensor. For example, if the input is a tensor of shape [10, 20], then this shape function specifies that the output shape is also [10, 20].

Note: The op name must be in CamelCase and it must be unique among all other ops that are registered in the binary.

Implement the kernel for the op

After you define the interface, provide one or more implementations of the op. To create one of these kernels, create a class that extends **`OpKernel`** and overrides the **`Compute`** method. The **`Compute`** method provides one **`context`** argument of type **`OpKernelContext*`**, from which you can access useful things like the input and output tensors.

Add your kernel to the file you created above. The kernel might look something like this:

```
#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);
  }
};
```

After implementing your kernel, you register it with the TensorFlow system. In the registration, you specify different constraints under which this kernel will run. For example, you might have one kernel made for CPUs, and a separate one for GPUs.

To do this for the **`ZeroOut`** op, add the following to **`zero_out.cc`**:

```
REGISTER_KERNEL_BUILDER(Name("ZeroOut").Device(DEVICE_CPU), ZeroOutOp);
```

Important: Instances of your OpKernel may be accessed concurrently. Your **`Compute`** method must be thread-safe. Guard any access to class members with a mutex. Or better yet, don't share state via class members! Consider using a **`ResourceMgr`** (https://www.tensorflow.org/code/tensorflow/core/framework/resource_mgr.h) to keep track of op state.

Multi-threaded CPU kernels

To write a multi-threaded CPU kernel, the Shard function in **`work_sharder.h`** (https://www.tensorflow.org/code/tensorflow/core/util/work_sharder.h) can be used. This function shards a computation function across the threads configured to be used for intra-op threading (see intra_op_parallelism_threads in **`config.proto`** (https://www.tensorflow.org/code/tensorflow/core/protobuf/config.proto)).

GPU kernels

A GPU kernel is implemented in two parts: the OpKernel and the CUDA kernel and its launch code.

Sometimes the OpKernel implementation is common between a CPU and GPU kernel, such as around inspecting inputs and allocating outputs. In that case, a suggested implementation is to:

1. Define the OpKernel templated on the Device and the primitive type of the tensor.
2. To do the actual computation of the output, the Compute function calls a templated functor struct.
3. The specialization of that functor for the CPUDevice is defined in the same file, but the specialization for the GPUDevice is defined in a .cu.cc file, since it will be compiled with the CUDA compiler.

Here is an example implementation.

```
// kernel_example.h
#ifndef KERNEL_EXAMPLE_H_
#define KERNEL_EXAMPLE_H_

#include <unsupported/Eigen/CXX11/Tensor>

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 T>
struct ExampleFunctor<Eigen::GpuDevice, T> {
  void operator()(const Eigen::GpuDevice& d, int size, const T* in, T* out)
};
#endif
```

```
#endif KERNEL_EXAMPLE_H_
```

```
// kernel_example.cc
#include "kernel_example.h"

#include "tensorflow/core/framework/op.h"
#include "tensorflow/core/framework/shape_inference.h"
#include "tensorflow/core/framework/op_kernel.h"

using namespace tensorflow;

using CPUDevice = Eigen::ThreadPoolDevice;
using GPUDevice = Eigen::GpuDevice;

REGISTER_OP("Example")
    .Attr("T: numbertype")
    .Input("input: T")
    .Output("input_times_two: T")
    .SetShapeFn([](::tensorflow::shape_inference::InferenceContext* c) {
      c->set_output(0, c->input(0));
      return Status::OK();
    });

// 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::kint32ma
                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 class 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 "kernel_example.h"
```

```
#include "tensorflow/core/util/gpu_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/gpu_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
```

Build the op library

Compile the op using your system compiler (TensorFlow binary installation)

You should be able to compile `zero_out.cc` with a `C++` compiler such as `g++` or `clang` available on your system. The binary PIP package installs the header files and the library that you need to compile your op in locations that are system specific. However, the TensorFlow python library provides the **`get_include`** function to get the header directory, and the **`get_lib`** directory has a shared object to link against. Here are the outputs of these functions on an Ubuntu machine.

```
$ python
>>> import tensorflow as tf
>>> tf.sysconfig.get_include()
'/usr/local/lib/python3.6/site-packages/tensorflow/include'
>>> tf.sysconfig.get_lib()
'/usr/local/lib/python3.6/site-packages/tensorflow'
```

Assuming you have **`g++`** installed, here is the sequence of commands you can use to compile your op into a dynamic library.

```
TF_CFLAGS=( $(python -c 'import tensorflow as tf; print(" ".join(tf.sysconf
TF_LFLAGS=( $(python -c 'import tensorflow as tf; print(" ".join(tf.sysconf
g++ -std=c++14 -shared zero_out.cc -o zero_out.so -fPIC ${TF_CFLAGS[@]} ${T
```

On macOS, the additional flag "-undefined dynamic_lookup" is required when building the **`.so`** file.

Note on **`gcc`** version **`>=5`**: gcc uses the new C++ ABI (https://gcc.gnu.org/gcc-5/changes.html#libstdcxx) since version **`5`**. The binary pip packages available on the TensorFlow website are built with **`gcc4`** that uses the older ABI. If you compile your op library with **`gcc>=5`**, add **`-D_GLIBCXX_USE_CXX11_ABI=0`** to the command line to make the library compatible with the older abi.

Compile the op using bazel (TensorFlow source installation)

If you have TensorFlow sources installed, you can make use of TensorFlow's build system to compile your op. Place a BUILD file with following Bazel build rule in the **`tensorflow/core/user_ops`** (https://www.tensorflow.org/code/tensorflow/core/user_ops/) directory.

```
load("//tensorflow:tensorflow.bzl", "tf_custom_op_library")

tf_custom_op_library(
    name = "zero_out.so",
    srcs = ["zero_out.cc"],
)
```

Run the following command to build **`zero_out.so`**.

```
$ bazel build --config opt //tensorflow/core/user_ops:zero_out.so
```

For compiling the **`Example`** operation, with the CUDA Kernel, you need to use the **`gpu_srcs`** parameter of **`tf_custom_op_library`**. Place a BUILD file with the following Bazel build rule in a new folder inside the **`tensorflow/core/user_ops`** (https://www.tensorflow.org/code/tensorflow/core/user_ops/) directory (e.g. "example_gpu").

```
load("//tensorflow:tensorflow.bzl", "tf_custom_op_library")

tf_custom_op_library(
    # kernel_example.cc  kernel_example.cu.cc  kernel_example.h
    name = "kernel_example.so",
    srcs = ["kernel_example.h", "kernel_example.cc"],
    gpu_srcs = ["kernel_example.cu.cc", "kernel_example.h"],
)
```

Run the following command to build **`kernel_example.so`**.

```
$ bazel build --config opt //tensorflow/core/user_ops/example_gpu:kernel_ex
```

Note: As explained above, if you are compiling with gcc>=5 add **`--cxxopt="-D_GLIBCXX_USE_CXX11_ABI=0"`** to the Bazel command line arguments.

Note: Although you can create a shared library (a **`.so`** file) with the standard **`cc_library`** rule, we strongly recommend that you use the **`tf_custom_op_library`** macro. It adds some required dependencies, and performs checks to ensure that the shared library is compatible with TensorFlow's plugin loading mechanism.

Use the op in Python

TensorFlow Python API provides the **`tf.load_op_library`** (https://www.tensorflow.org/api_docs/python/tf/load_op_library) function to load the dynamic library and register the op with the TensorFlow framework. **`load_op_library`** returns a Python module that contains the Python wrappers for the op and the kernel. Thus, once you have built the op, you can do the following to run it from Python:

```
import tensorflow as tf
zero_out_module = tf.load_op_library('./zero_out.so')
print(zero_out_module.zero_out([[1, 2], [3, 4]]).numpy())

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

Keep in mind, the generated function will be given a snake_case name (to comply with PEP8 (https://www.python.org/dev/peps/pep-0008/)). So, if your op is named **`ZeroOut`** in the C++ files, the python function will be called **`zero_out`**.

To make the op available as a regular function **`import`**-able from a Python module, it maybe useful to have the **`load_op_library`** call in a Python source file as follows:

```
import tensorflow as tf

zero_out_module = tf.load_op_library('./zero_out.so')
zero_out = zero_out_module.zero_out
```

Verify that the op works

A good way to verify that you've successfully implemented your op is to write a test for it. Create the file **`zero_out_op_test.py`** with the contents:

```
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()
```

Then run your test (assuming you have tensorflow installed):

```
$ python zero_out_op_test.py
```

Build advanced features into your op

Now that you know how to build a basic (and somewhat restricted) op and implementation, we'll look at some of the more complicated things you will typically need to build into your op. This includes:

- Conditional checks and validation (#conditional_checks_and_validation)
- Op registration (#op_registration)
    - Attrs (#attrs)
    - Attr types (#attr_types)
    - Polymorphism (#polymorphism)
    - Inputs and outputs (#inputs_and_outputs)

Conditional checks and validation

The example above assumed that the op applied to a tensor of any shape. What if it only applied to vectors? That means adding a check to the above OpKernel implementation.

```
  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."));
    // ...
  }
```

This asserts that the input is a vector, and returns having set the `InvalidArgument` status if it isn't. The `OP_REQUIRES` macro (https://www.tensorflow.org/code/tensorflow/core/platform/errors.h) takes three arguments:

- The `context`, which can either be an `OpKernelContext` or `OpKernelConstruction` pointer (see `tensorflow/core/framework/op_kernel.h` (https://www.tensorflow.org/code/tensorflow/core/framework/op_kernel.h)), for its `SetStatus()` method.
- The condition. For example, there are functions for validating the shape of a tensor in `tensorflow/core/framework/tensor_shape.h` (https://www.tensorflow.org/code/tensorflow/core/framework/tensor_shape.h)
- The error itself, which is represented by a `Status` object, see `tensorflow/core/platform/status.h` (https://www.tensorflow.org/code/tensorflow/core/platform/status.h). A `Status` has both a type (frequently `InvalidArgument`, but see the list of types) and a message. Functions

for constructing an error may be found in **`tensorflow/core/platform/errors.h`** (https://www.tensorflow.org/code/tensorflow/core/platform/errors.h).

Alternatively, if you want to test whether a **`Status`** object returned from some function is an error, and if so return it, use **`OP_REQUIRES_OK`** (https://www.tensorflow.org/code/tensorflow/core/platform/errors.h). Both of these macros return from the function on error.

Op registration

Attrs

Ops can have attrs, whose values are set when the op is added to a graph. These are used to configure the op, and their values can be accessed both within the kernel implementation and in the types of inputs and outputs in the op registration. Prefer using an input instead of an attr when possible, since inputs are more flexible. This is because attrs are constants and must be defined at graph construction time. In contrast, inputs are Tensors whose values can be dynamic; that is, inputs can change every step, be set using a feed, etc. Attrs are used for things that can't be done with inputs: any configuration that affects the signature (number or type of inputs or outputs) or that can't change from step-to-step.

You define an attr when you register the op, by specifying its name and type using the **`Attr`** method, which expects a spec of the form:

```
<name>: <attr-type-expr>
```

where **`<name>`** begins with a letter and can be composed of alphanumeric characters and underscores, and **`<attr-type-expr>`** is a type expression of the form described below (#attr_types).

For example, if you'd like the **`ZeroOut`** op to preserve a user-specified index, instead of only the 0th element, you can register the op like so:

```
REGISTER_OP("ZeroOut")
    .Attr("preserve_index: int")
    .Input("to_zero: int32")
    .Output("zeroed: int32");
```

(Note that the set of attribute types (#attr_types) is different from the **`tf.DType`** (https://www.tensorflow.org/api_docs/python/tf/dtypes/DType) used for inputs and outputs.)

Your kernel can then access this attr in its constructor via the **`context`** parameter:

```
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_;
};
```

which can then be used in the **`Compute`** method:

```
  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 types

The following types are supported in an attr:

- `string`: Any sequence of bytes (not required to be UTF8).
- `int`: A signed integer.
- `float`: A floating point number.
- `bool`: True or false.
- `type`: One of the (non-ref) values of `DataType` (https://www.tensorflow.org/code/tensorflow/core/framework/types.cc).
- `shape`: A `TensorShapeProto` (https://www.tensorflow.org/code/tensorflow/core/framework/tensor_shape.proto).
- `list(<type>)`: A list of `<type>`, where `<type>` is one of the above types. Note that `list(list(<type>))` is invalid.

See also: `op_def_builder.cc:FinalizeAttr` (https://www.tensorflow.org/code/tensorflow/core/framework/op_def_builder.cc) for a definitive list.

Default values and constraints

Attrs may have default values, and some types of attrs can have constraints. To define an attr with constraints, you can use the following `<attr-type-expr>`s:

`{'<string1>', '<string2>'}`: The value must be a string that has either the value `<string1>` or `<string2>`. The name of the type, `string`, is implied when you use this syntax. This emulates an enum:

```
REGISTER_OP("EnumExample")
    .Attr("e: {'apple', 'orange'}");
```

`{<type1>, <type2>}`: The value is of type **`type`**, and must be one of **`<type1>`** or **`<type2>`**, where **`<type1>`** and **`<type2>`** are supported **`tf.DType`**. You don't specify that the type of the attr is **`type`**. This is implied when you have a list of types in **`{...}`**. For example, in this case the attr **`t`** is a type that must be an **`int32`**, a **`float`**, or a **`bool`**:

```
REGISTER_OP("RestrictedTypeExample")
    .Attr("t: {int32, float, bool}");
```

There are shortcuts for common type constraints:

- **`numbertype`**: Type **`type`** restricted to the numeric (non-string and non-bool) types.
- **`realnumbertype`**: Like **`numbertype`** without complex types.
- **`quantizedtype`**: Like **`numbertype`** but just the quantized number types.

The specific lists of types allowed by these are defined by the functions (like **`NumberTypes()`**) in **`tensorflow/core/framework/types.h`** (https://www.tensorflow.org/code/tensorflow/core/framework/types.h). In this example the attr **`t`** must be one of the numeric types:

```
REGISTER_OP("NumberType")
    .Attr("t: numbertype");
```

For this op:

```
tf.number_type(t=tf.int32)  # Valid
tf.number_type(t=tf.bool)   # Invalid
```

Lists can be combined with other lists and single types. The following op allows attr **`t`** to be any of the numeric types, or the bool type:

```
REGISTER_OP("NumberOrBooleanType")
    .Attr("t: {numbertype, bool}");
```

For this op:

```
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>`: The value must be an int whose value is greater than or equal to **`<n>`**, where **`<n>`** is a natural number. For example, the following op registration specifies that the attr **`a`** must have a value that is at least **`2`**:

```
REGISTER_OP("MinIntExample")
    .Attr("a: int >= 2");
```

`list(<type>) >= <n>`: A list of type **`<type>`** whose length is greater than or equal to **`<n>`**. For example, the following op registration specifies that the attr **`a`** is a list of types (either **`int32`** or **`float`**), and that there must be at least 3 of them:

```
REGISTER_OP("TypeListExample")
    .Attr("a: list({int32, float}) >= 3");
```

To set a default value for an attr (making it optional in the generated code), add **`= <default>`** to the end, as in:

```
REGISTER_OP("AttrDefaultExample")
    .Attr("i: int = 0");
```

Additionally, both a constraint and a default value can be specified:

```
REGISTER_OP("AttrConstraintAndDefaultExample")
    .Attr("i: int >= 1 = 1");
```

The supported syntax of the default value is what would be used in the proto representation of the resulting GraphDef definition.

Here are examples for how to specify a default for all types:

```
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]");
```

Note in particular that the values of type `type` use `tf.DType` (https://www.tensorflow.org/api_docs/python/tf/dtypes/DType).

Polymorphism

Type polymorphism

For ops that can take different types as input or produce different output types, you can specify an attr (#attrs) in an input or output type (#inputs_and_outputs) in the op registration. Typically you would then register an `OpKernel` for each supported type.

For instance, if you'd like the `ZeroOut` op to work on `float`s in addition to `int32`s, your op registration might look like:

```
REGISTER_OP("ZeroOut")
    .Attr("T: {float, int32}")
    .Input("to_zero: T")
    .Output("zeroed: T");
```

Your op registration now specifies that the input's type must be `float`, or `int32`, and that its output will be the same type, since both have type `T`.

Naming

Inputs, outputs, and attrs generally should be given snake_case names. The one exception is attrs that are used as the type of an input or in the type of an output. Those attrs can be inferred when the op is added to the graph and so don't appear in the op's function. For example, this last definition of ZeroOut will generate a Python function that looks like:

```
def zero_out(to_zero, name=None):
  """...
  Args:
    to_zero: A `Tensor`. Must be one of the following types:
        `float32`, `int32`.
    name: A name for the operation (optional).

  Returns:
    A `Tensor`. Has the same type as `to_zero`.
  """
```

If **`to_zero`** is passed an **`int32`** tensor, then **`T`** is automatically set to **`int32`** (well, actually **`DT_INT32`**). Those inferred attrs are given Capitalized or CamelCase names.

Compare this with an op that has a type attr that determines the output type:

```
REGISTER_OP("StringToNumber")
    .Input("string_tensor: string")
    .Output("output: out_type")
    .Attr("out_type: {float, int32} = DT_FLOAT");
    .Doc(R"doc(
Converts each string in the input Tensor to the specified numeric type.
)doc");
```

In this case, the user has to specify the output type, as in the generated Python:

```
def string_to_number(string_tensor, out_type=None, name=None):
  """Converts each string in the input Tensor to the specified numeric type

  Args:
    string_tensor: A `Tensor` of type `string`.
    out_type: An optional `tf.DType` from: `tf.float32, tf.int32`.
```

```
      Defaults to `tf.float32`.
    name: A name for the operation (optional).

  Returns:
    A `Tensor` of type `out_type`.
  """
```

Type polymorphism example

```
#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(), &outpu
    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"),
    ZeroOutInt32Op);
REGISTER_KERNEL_BUILDER(
    Name("ZeroOut")
    .Device(DEVICE_CPU)
    .TypeConstraint<float>("T"),
    ZeroOutFloatOp);
```

To preserve backwards compatibility (#backwards_compatibility), you should specify a default value (#default_values_and_constraints) when adding an attr to an existing op:

```
REGISTER_OP("ZeroOut")
  .Attr("T: {float, int32} = DT_INT32")
  .Input("to_zero: T")
  .Output("zeroed: T")
```

Let's say you wanted to add more types, say **`double`**:

```
REGISTER_OP("ZeroOut")
    .Attr("T: {float, double, int32}")
    .Input("to_zero: T")
    .Output("zeroed: T");
```

Instead of writing another **`OpKernel`** with redundant code as above, often you will be able to use a C++ template instead. You will still have one kernel registration (**`REGISTER_KERNEL_BUILDER`** call) per overload.

```
template <typename T>
class ZeroOutOp : public OpKernel {
 public:
  explicit ZeroOutOp(OpKernelConstruction* context) : OpKernel(context) {}
  void Compute(OpKernelContext* context) override {
    // Grab the input tensor
    const Tensor& input_tensor = context->input(0);
    auto input = input_tensor.flat<T>();

    // Create an output tensor
    Tensor* output = NULL;
    OP_REQUIRES_OK(context,
                   context->allocate_output(0, input_tensor.shape(), &outpu
    auto output_flat = output->template flat<T>();

    // Set all the elements of the output tensor to 0
    const int N = input.size();
    for (int i = 0; i < N; i++) {
      output_flat(i) = 0;
    }

    // Preserve the first input value
    if (N > 0) output_flat(0) = input(0);
  }
};

// Note that TypeConstraint<int32>("T") means that attr "T" (defined
// in the op registration above) must be "int32" to use this template
// instantiation.
REGISTER_KERNEL_BUILDER(
    Name("ZeroOut")
    .Device(DEVICE_CPU)
    .TypeConstraint<int32>("T"),
    ZeroOutOp<int32>);
REGISTER_KERNEL_BUILDER(
    Name("ZeroOut")
    .Device(DEVICE_CPU)
    .TypeConstraint<float>("T"),
    ZeroOutOp<float>);
REGISTER_KERNEL_BUILDER(
    Name("ZeroOut")
    .Device(DEVICE_CPU)
    .TypeConstraint<double>("T"),
    ZeroOutOp<double>);
```

If you have more than a couple overloads, you can put the registration in a macro.

```
#include "tensorflow/core/framework/op_kernel.h"

#define REGISTER_KERNEL(type)                                       \
  REGISTER_KERNEL_BUILDER(                                          \
      Name("ZeroOut").Device(DEVICE_CPU).TypeConstraint<type>("T"), \
      ZeroOutOp<type>)

REGISTER_KERNEL(int32);
REGISTER_KERNEL(float);
REGISTER_KERNEL(double);

#undef REGISTER_KERNEL
```

Depending on the list of types you are registering the kernel for, you may be able to use a macro provided by **`tensorflow/core/framework/register_types.h`** (https://www.tensorflow.org/code/tensorflow/core/framework/register_types.h):

```
#include "tensorflow/core/framework/op_kernel.h"
#include "tensorflow/core/framework/register_types.h"

REGISTER_OP("ZeroOut")
    .Attr("T: realnumbertype")
    .Input("to_zero: T")
    .Output("zeroed: T");

template <typename T>
class ZeroOutOp : public OpKernel { ... };

#define REGISTER_KERNEL(type)                                       \
  REGISTER_KERNEL_BUILDER(                                          \
      Name("ZeroOut").Device(DEVICE_CPU).TypeConstraint<type>("T"), \
      ZeroOutOp<type>)

TF_CALL_REAL_NUMBER_TYPES(REGISTER_KERNEL);

#undef REGISTER_KERNEL
```

List inputs and outputs

In addition to being able to accept or produce different types, ops can consume or produce a variable number of tensors.

In the next example, the attr `T` holds a *list* of types, and is used as the type of both the input `in` and the output `out`. The input and output are lists of tensors of that type (and the number and types of tensors in the output are the same as the input, since both have type `T`).

```
REGISTER_OP("PolymorphicListExample")
    .Attr("T: list(type)")
    .Input("in: T")
    .Output("out: T");
```

You can also place restrictions on what types can be specified in the list. In this next case, the input is a list of `float` and `double` tensors. The op accepts, for example, input types `(float, double, float)` and in that case the output type would also be `(float, double, float)`.

```
REGISTER_OP("ListTypeRestrictionExample")
    .Attr("T: list({float, double})")
    .Input("in: T")
    .Output("out: T");
```

If you want all the tensors in a list to be of the same type, you might do something like:

```
REGISTER_OP("IntListInputExample")
    .Attr("N: int")
    .Input("in: N * int32")
    .Output("out: int32");
```

This accepts a list of `int32` tensors, and uses an `int` attr `N` to specify the length of the list.

This can be made type polymorphic (#type_polymorphism) as well. In the next example, the input is a list of tensors (with length `"N"`) of the same (but unspecified) type (`"T"`), and the output is a single tensor of matching type:

```
REGISTER_OP("SameListInputExample")
    .Attr("N: int")
    .Attr("T: type")
    .Input("in: N * T")
    .Output("out: T");
```

By default, tensor lists have a minimum length of 1. You can change that default using a `">="` constraint on the corresponding attr (#default_values_and_constraints). In this next example, the input is a list of at least 2 **`int32`** tensors:

```
REGISTER_OP("MinLengthIntListExample")
    .Attr("N: int >= 2")
    .Input("in: N * int32")
    .Output("out: int32");
```

The same syntax works with "**`list(type)`**" attrs:

```
REGISTER_OP("MinimumLengthPolymorphicListExample")
    .Attr("T: list(type) >= 3")
    .Input("in: T")
    .Output("out: T");
```

Inputs and outputs

To summarize the above, an op registration can have multiple inputs and outputs:

```
REGISTER_OP("MultipleInsAndOuts")
    .Input("y: int32")
    .Input("z: float")
    .Output("a: string")
    .Output("b: int32");
```

Each input or output spec is of the form:

```
<name>: <io-type-expr>
```

where **`<name>`** begins with a letter and can be composed of alphanumeric characters and underscores. **`<io-type-expr>`** is one of the following type expressions:

- **`<type>`**, where **`<type>`** is a supported input type (e.g. **`float`**, **`int32`**, **`string`**). This specifies a single tensor of the given type.

  See **`tf.DType`** (https://www.tensorflow.org/api_docs/python/tf/dtypes/DType).

  ```
  REGISTER_OP("BuiltInTypesExample")
      .Input("integers: int32")
      .Input("complex_numbers: complex64");
  ```

- **`<attr-type>`**, where **`<attr-type>`** is the name of an Attr (#attrs) with type **`type`** or **`list(type)`** (with a possible type restriction). This syntax allows for polymorphic ops (#polymorphism).

  ```
  REGISTER_OP("PolymorphicSingleInput")
      .Attr("T: type")
      .Input("in: T");

  REGISTER_OP("RestrictedPolymorphicSingleInput")
      .Attr("T: {int32, int64}")
      .Input("in: T");
  ```

  Referencing an attr of type **`list(type)`** allows you to accept a sequence of tensors.

  ```
  REGISTER_OP("ArbitraryTensorSequenceExample")
      .Attr("T: list(type)")
      .Input("in: T")
      .Output("out: T");

  REGISTER_OP("RestrictedTensorSequenceExample")
      .Attr("T: list({int32, int64})")
      .Input("in: T")
      .Output("out: T");
  ```

Note that the number and types of tensors in the output `out` is the same as in the input `in`, since both are of type `T`.

- For a sequence of tensors with the same type: `<number> * <type>`, where `<number>` is the name of an Attr (#attrs) with type `int`. The `<type>` can either be a `tf.DType` (https://www.tensorflow.org/api_docs/python/tf/dtypes/DType), or the name of an attr with type `type`. As an example of the first, this op accepts a list of `int32` tensors:

```
REGISTER_OP("Int32SequenceExample")
    .Attr("NumTensors: int")
    .Input("in: NumTensors * int32")
```

  Whereas this op accepts a list of tensors of any type, as long as they are all the same:

```
REGISTER_OP("SameTypeSequenceExample")
    .Attr("NumTensors: int")
    .Attr("T: type")
    .Input("in: NumTensors * T")
```

- For a reference to a tensor: `Ref(<type>)`, where `<type>` is one of the previous types.

Any attr used in the type of an input will be inferred. By convention those inferred attrs use capital names (like `T` or `N`). Otherwise inputs, outputs, and attrs have names like function parameters (e.g. `num_outputs`). For more details, see the earlier section on naming (#naming).

For more details, see `tensorflow/core/framework/op_def_builder.h` (https://www.tensorflow.org/code/tensorflow/core/framework/op_def_builder.h).

Backwards compatibility

Let's assume you have written a nice, custom op and shared it with others, so you have happy customers using your operation. However, you'd like to make changes to the op in some way.

In general, changes to existing, checked-in specifications must be backwards-compatible: changing the specification of an op must not break prior serialized `GraphDef` protocol buffers constructed from older specifications. The details of `GraphDef` compatibility are described here (https://www.tensorflow.org/guide/versions#compatibility_of_graphs_and_checkpoints).

There are several ways to preserve backwards-compatibility.

1. Any new attrs added to an operation must have default values defined, and with that default value the op must have the original behavior. To change an operation from not polymorphic to polymorphic, you *must* give a default value to the new type attr to preserve the original signature by default. For example, if your operation was:

   ```
   REGISTER_OP("MyGeneralUnaryOp")
       .Input("in: float")
       .Output("out: float");
   ```

   you can make it polymorphic in a backwards-compatible way using:

   ```
   REGISTER_OP("MyGeneralUnaryOp")
       .Input("in: T")
       .Output("out: T")
       .Attr("T: numerictype = DT_FLOAT");
   ```

2. You can safely make a constraint on an attr less restrictive. For example, you can change from `{int32, int64}` to `{int32, int64, float}` or `type`. Or you may change from `{"apple", "orange"}` to `{"apple", "banana", "orange"}` or `string`.

3. You can change single inputs / outputs into list inputs / outputs, as long as the default for the list type matches the old signature.

4. You can add a new list input / output, if it defaults to empty.

5. Namespace any new ops you create, by prefixing the op names with something unique to your project. This avoids having your op colliding with any ops that might be included in future versions of TensorFlow.

6. Plan ahead! Try to anticipate future uses for the op. Some signature changes can't be done in a compatible way (for example, making a list of the same type into a list of varying types).

The full list of safe and unsafe changes can be found in `tensorflow/core/framework/op_compatibility_test.cc` (https://www.tensorflow.org/code/tensorflow/core/framework/op_compatibility_test.cc). If you cannot make your change to an operation backwards compatible, then create a new operation with a new name with the new semantics.

Also note that while these changes can maintain `GraphDef` compatibility, the generated Python code may change in a way that isn't compatible with old callers. The Python API may be kept compatible by careful changes in a hand-written Python wrapper, by keeping the old signature except possibly adding new optional arguments to the end. Generally incompatible changes may only be made when TensorFlow changes major versions, and must conform to the `GraphDef` version semantics (https://www.tensorflow.org/guide/versions#compatibility_of_graphs_and_checkpoints).

GPU support

You can implement different OpKernels and register one for CPU and another for GPU, just like you can register kernels for different types (#polymorphism). There are several examples of kernels with GPU support in `tensorflow/core/kernels/` (https://www.tensorflow.org/code/tensorflow/core/kernels/). Notice some kernels have a CPU version in a `.cc` file, a GPU version in a file ending in `_gpu.cu.cc`, and some code shared in common in a `.h` file.

For example, the `tf.pad` (https://www.tensorflow.org/api_docs/python/tf/pad) has everything but the GPU kernel in `tensorflow/core/kernels/pad_op.cc` (https://www.tensorflow.org/code/tensorflow/core/kernels/pad_op.cc). The GPU kernel is in `tensorflow/core/kernels/pad_op_gpu.cu.cc` (https://www.tensorflow.org/code/tensorflow/core/kernels/pad_op_gpu.cu.cc), and the shared code is a templated class defined in `tensorflow/core/kernels/pad_op.h` (https://www.tensorflow.org/code/tensorflow/core/kernels/pad_op.h). We organize the code this way for two reasons: it allows you to share common code among the CPU and GPU implementations, and it puts the GPU implementation into a separate file so that it can be compiled only by the GPU compiler.

One thing to note, even when the GPU kernel version of `pad` is used, it still needs its `"paddings"` input in CPU memory. To mark that inputs or outputs are kept on the CPU, add a `HostMemory()` call to the kernel registration, e.g.:

```
#define REGISTER_GPU_KERNEL(T)                         \
  REGISTER_KERNEL_BUILDER(Name("Pad")                  \
                              .Device(DEVICE_GPU)      \
                              .TypeConstraint<T>("T")  \
                              .HostMemory("paddings"), \
                          PadOp<GPUDevice, T>)
```

Compiling the kernel for the GPU device

Look at cuda_op_kernel.cu.cc (https://www.tensorflow.org/code/tensorflow/examples/adding_an_op/cuda_op_kernel.cu.cc) for an example that uses a CUDA kernel to implement an op. The **tf_custom_op_library** accepts a **gpu_srcs** argument in which the list of source files containing the CUDA kernels (***.cu.cc** files) can be specified. For use with a binary installation of TensorFlow, the CUDA kernels have to be compiled with NVIDIA's **nvcc** compiler. Here is the sequence of commands you can use to compile the cuda_op_kernel.cu.cc (https://www.tensorflow.org/code/tensorflow/examples/adding_an_op/cuda_op_kernel.cu.cc) and cuda_op_kernel.cc (https://www.tensorflow.org/code/tensorflow/examples/adding_an_op/cuda_op_kernel.cc) into a single dynamically loadable library:

```
nvcc -std=c++14 -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++14 -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 produced above can be loaded as usual in Python, using the **tf.load_op_library** (https://www.tensorflow.org/api_docs/python/tf/load_op_library) function.

Note that if your CUDA libraries are not installed in **/usr/local/lib64**, you'll need to specify the path explicitly in the second (g++) command above. For example, add **-L /usr/local/cuda-8.0/lib64/** if your CUDA is installed in **/usr/local/cuda-8.0**.

Note: In some Linux settings, additional options to **nvcc** compiling step are needed. Add **-D_MWAITXINTRIN_H_INCLUDED** to the **nvcc** command line to avoid errors from **mwaitxintrin.h**.

Implement the gradient in Python

Given a graph of ops, TensorFlow uses automatic differentiation (backpropagation) to add new ops representing gradients with respect to the existing ops. To make automatic differentiation work for new ops, you must register a gradient function which computes gradients with respect to the ops' inputs given gradients with respect to the ops' outputs.

Mathematically, if an op computes $y = f(x)$ the registered gradient op converts gradients $\partial L/\partial y$ of loss L with respect to y into gradients $\partial L/\partial x$ with respect to x via the chain rule:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial f}{\partial x}.$$

In the case of `ZeroOut`, only one entry in the input affects the output, so the gradient with respect to the input is a sparse "one hot" tensor. This is expressed as follows:

```
from tensorflow.python.framework import ops
from tensorflow.python.ops import array_ops
from tensorflow.python.ops import sparse_ops

@ops.RegisterGradient("ZeroOut")
def _zero_out_grad(op, grad):
  """The gradients for `zero_out`.

  Args:
    op: The `zero_out` `Operation` that we are differentiating, which we ca
      to find the inputs and outputs of the original op.
    grad: Gradient with respect to the output of the `zero_out` op.

  Returns:
    Gradients with respect to the input of `zero_out`.
  """
  to_zero = op.inputs[0]
  shape = array_ops.shape(to_zero)
  index = array_ops.zeros_like(shape)
  first_grad = array_ops.reshape(grad, [-1])[0]
  to_zero_grad = sparse_ops.sparse_to_dense([index], shape, first_grad, 0)
  return [to_zero_grad]  # List of one Tensor, since we have one input
```

Details about registering gradient functions with **`tf.RegisterGradient`** (https://www.tensorflow.org/api_docs/python/tf/RegisterGradient):

- For an op with one output, the gradient function will take an **tf.Operation** (https://www.tensorflow.org/api_docs/python/tf/Operation), `op`, and a **tf.Tensor** (https://www.tensorflow.org/api_docs/python/tf/Tensor) `grad` and build new ops out of the tensors `op.inputs[i]`, `op.outputs[i]`, and `grad`. Information about any attrs can be found via **tf.Operation.get_attr** (https://www.tensorflow.org/api_docs/python/tf/Operation#get_attr).
- If the op has multiple outputs, the gradient function will take `op` and `grads`, where `grads` is a list of gradients with respect to each output. The result of the gradient function must be a list of `Tensor` objects representing the gradients with respect to each input.
- If there is no well-defined gradient for some input, such as for integer inputs used as indices, the corresponding returned gradient should be `None`. For example, for an op taking a floating point tensor `x` and an integer index `i`, the gradient function would `return [x_grad, None]`.
- If there is no meaningful gradient for the op at all, you often will not have to register any gradient, and as long as the op's gradient is never needed, you will be fine. In some cases, an op has no well-defined gradient but can be involved in the computation of the gradient. Here you can use `ops.NotDifferentiable` to automatically propagate zeros backwards.

Note that at the time the gradient function is called, only the data flow graph of ops is available, not the tensor data itself. Thus, all computation must be performed using other tensorflow ops, to be run at graph execution time.

Shape functions in C++

The TensorFlow API has a feature called "shape inference" that provides information about the shapes of tensors without having to execute the graph. Shape inference is supported by "shape functions" that are registered for each op type in the C++ `REGISTER_OP` declaration, and perform two roles: asserting that the shapes of the inputs are compatible during graph construction, and specifying the shapes for the outputs.

Shape functions are defined as operations on the `shape_inference::InferenceContext` class. For example, in the shape function for ZeroOut:

```
.SetShapeFn([](::tensorflow::shape_inference::InferenceContext* c) {
  c->set_output(0, c->input(0));
```

```
      return Status::OK();
    });
```

`c->set_output(0, c->input(0));` declares that the first output's shape should be set to the first input's shape. If the output is selected by its index as in the above example, the second parameter of `set_output` should be a `ShapeHandle` object. You can create an empty `ShapeHandle` object by its default constructor. The `ShapeHandle` object for an input with index `idx` can be obtained by `c->input(idx)`.

There are a number of common shape functions that apply to many ops, such as `shape_inference::UnchangedShape` which can be found in common_shape_fns.h (https://www.tensorflow.org/code/tensorflow/core/framework/common_shape_fns.h) and used as follows:

```
REGISTER_OP("ZeroOut")
    .Input("to_zero: int32")
    .Output("zeroed: int32")
    .SetShapeFn(::tensorflow::shape_inference::UnchangedShape);
```

A shape function can also constrain the shape of an input. For the version of `ZeroOut` with a vector shape constraint (#conditional_checks_and_validation), the shape function would be as follows:

```
    .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();
    });
```

The `WithRank` call validates that the input shape `c->input(0)` has a shape with exactly one dimension (or if the input shape is unknown, the output shape will be a vector with one unknown dimension).

If your op is polymorphic with multiple inputs (#polymorphism), you can use members of `InferenceContext` to determine the number of shapes to check, and `Merge` to validate that the shapes are all compatible (alternatively, access attributes that indicate the lengths, with **`InferenceContext::GetAttr`**, which provides access to the attributes of the op).

```
    .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();
    });
```

Since shape inference is an optional feature, and the shapes of tensors may vary dynamically, shape functions must be robust to incomplete shape information for any of the inputs. The **`Merge`** method in **`InferenceContext`** (https://www.tensorflow.org/code/tensorflow/core/framework/shape_inference.h) allows the caller to assert that two shapes are the same, even if either or both of them do not have complete information. Shape functions are defined for all of the core TensorFlow ops and provide many different usage examples.

The **`InferenceContext`** class has a number of functions that can be used to define shape function manipulations. For example, you can validate that a particular dimension has a very specific value using **`InferenceContext::Dim`** and **`InferenceContext::WithValue`**; you can specify that an output dimension is the sum / product of two input dimensions using **`InferenceContext::Add`** and **`InferenceContext::Multiply`**. See the **`InferenceContext`** class for all of the various shape manipulations you can specify. The following example sets shape of the first output to (n, 3), where first input has shape (n, ...)

```
.SetShapeFn([](::tensorflow::shape_inference::InferenceContext* c) {
    c->set_output(0, c->Matrix(c->Dim(c->input(0), 0), 3));
    return Status::OK();
});
```

If you have a complicated shape function, you should consider adding a test for validating that various input shape combinations produce the expected output shape combinations. You can see examples of how to write these tests in some our core ops tests (https://www.tensorflow.org/code/tensorflow/core/ops/array_ops_test.cc). (The syntax of `INFER_OK` and `INFER_ERROR` are a little cryptic, but try to be compact in representing input and output shape specifications in tests. For now, see the surrounding comments in those tests to get a sense of the shape string specification).

Build a pip package for your custom op

To build a `pip` package for your op, see the tensorflow/custom-op (https://github.com/tensorflow/custom-op) example. This guide shows how to build custom ops from the TensorFlow pip package instead of building TensorFlow from source.