# TensorFlow - Masterpiece of Engineering

Hy Truong Son

STATT 37790 - Topics in Statistical Machine Learning:
High-Performance Machine Learning System Design

The University of Chicago

May 2019

# Reference

Reference:

1. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, Abadi et. al
2. `https://www.tensorflow.org/guide/extend/architecture`

To get this presentation slides:

`http://people.cs.uchicago.edu/~hytruongson/tensorflow.pdf`

THE UNIVERSITY OF CHICAGO

# What is TensorFlow?

## Definition

TensorFlow is an **interface** for expressing machine learning algorithms, and an implementation for executing such algorithms.

# What is TensorFlow?

## Definition

TensorFlow is an **interface** for expressing machine learning algorithms, and an implementation for executing such algorithms.

## Question

Why did they use the word **interface**? Why not **framework** like PyTorch framework?

# What is special about TensorFlow?

A computation expressed using TensorFlow can be executed with **little or no change on a wide variety of heterogeneous systems**, ranging from mobile devices such as phones and tablets up to large-scale distributed systems of hundreds of machines and thousands of computational devices such as GPU cards (update: TPU as well).

# What is special about TensorFlow?

A computation expressed using TensorFlow can be executed with **little or no change on a wide variety of heterogeneous systems**, ranging from mobile devices such as phones and tablets up to large-scale distributed systems of hundreds of machines and thousands of computational devices such as GPU cards (update: TPU as well).

**Reality**

The fact is: TensorFlow is designed **just** for Google infrastructure. Internal TF is always smooth and efficient in any scenario. But not really outside!

# History

The Google Brain project started in 2011 to explore the use of very-large-scale deep neural networks, both for research and for use in Google's products.

# History

The Google Brain project started in 2011 to explore the use of very-large-scale deep neural networks, both for research and for use in Google's products. As part of the early work in this project, we built **DistBelief**, our first-generation scalable distributed training and inference system, and **this system has served us well**.

# History

The Google Brain project started in 2011 to explore the use of very-large-scale deep neural networks, both for research and for use in Google's products. As part of the early work in this project, we built **DistBelief**, our first-generation scalable distributed training and inference system, and **this system has served us well**.

**Question 1**

Why did they name their first system as **DistBelief**?

# History

The Google Brain project started in 2011 to explore the use of very-large-scale deep neural networks, both for research and for use in Google's products. As part of the early work in this project, we built **DistBelief**, our first-generation scalable distributed training and inference system, and **this system has served us well**.

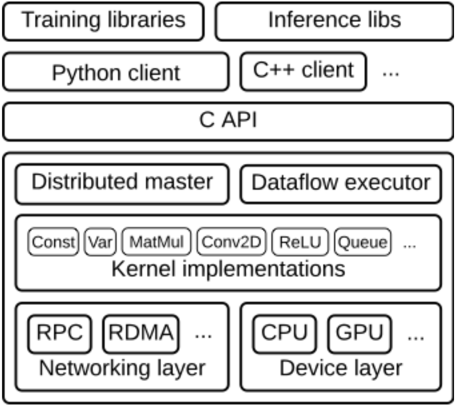
**Question 1**

Why did they name their first system as **DistBelief**?

**Question 2**

If **this system has served us well**, why did they have to make TensorFlow?

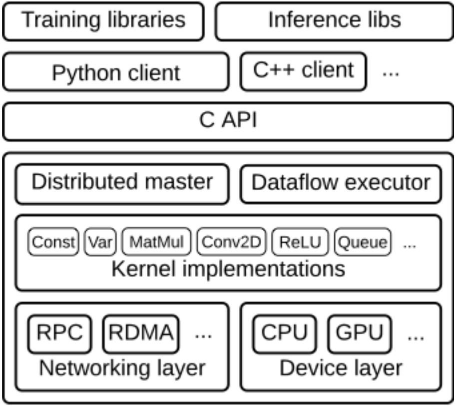
# Layers of TensorFlow - Client




**Client:**

- Defines the computation as a dataflow graph.
- Initiates **graph** execution using a **session**.

# Layers of TensorFlow - Client




**Client:**

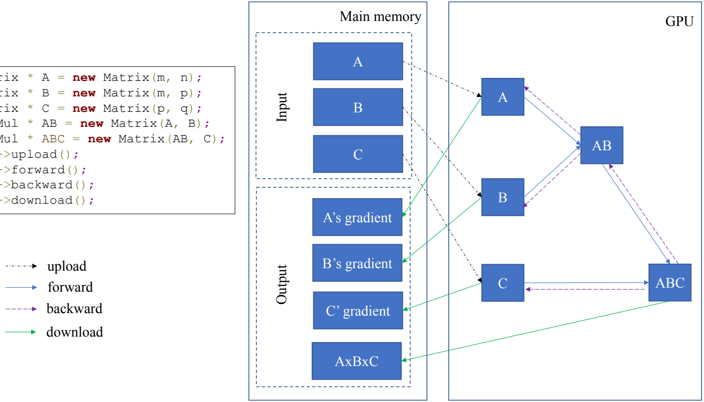
- Defines the computation as a dataflow graph.
- Initiates **graph** execution using a **session**.

**Question**

What is the **graph** here?

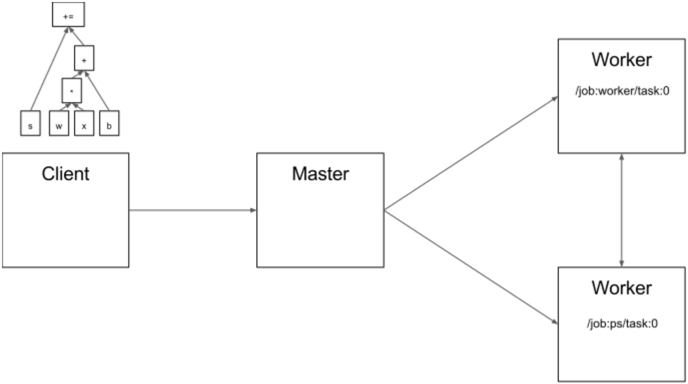
# Computation graph

```
Matrix * A = new Matrix(m, n);
Matrix * B = new Matrix(m, p);
Matrix * C = new Matrix(p, q);
MatMul * AB = new Matrix(A, B);
MalMul * ABC = new Matrix(AB, C);
ABC->upload();
ABC->forward();
ABC->backward();
ABC->download();
```

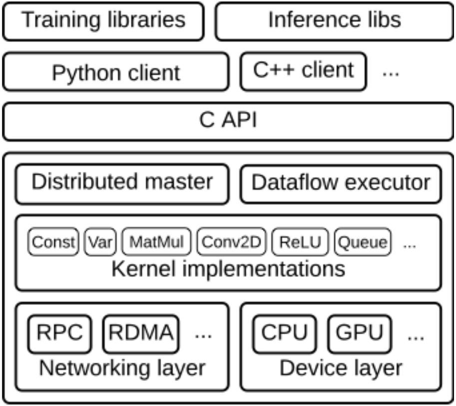



You can build your own computation graph in C++!

# Layers of TensorFlow - Client

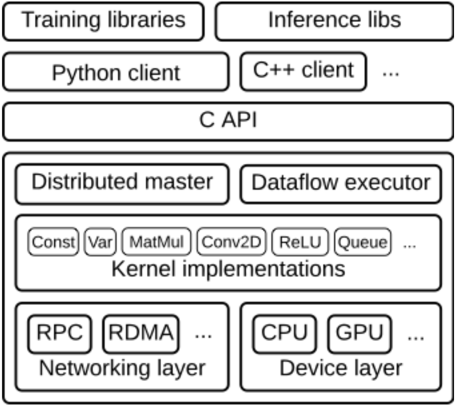


# Layers of TensorFlow - Distributed Master




**Distributed Master:**

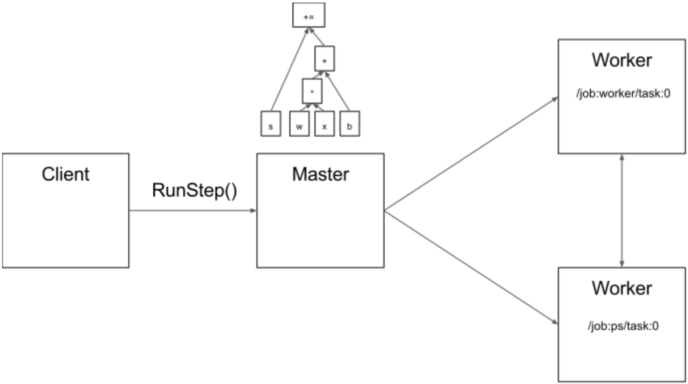
- Prunes a specific subgraph from the graph, as defined by the arguments to `Session.run()`.
- Partitions the subgraph into multiple pieces that run in different processes and devices.




**Distributed Master:**

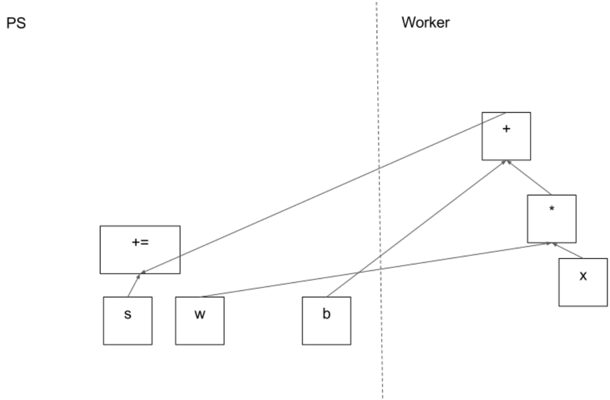
- Distributes the graph pieces to worker services.
- Initiates graph piece execution by worker services.

# Layers of TensorFlow - Distributed Master



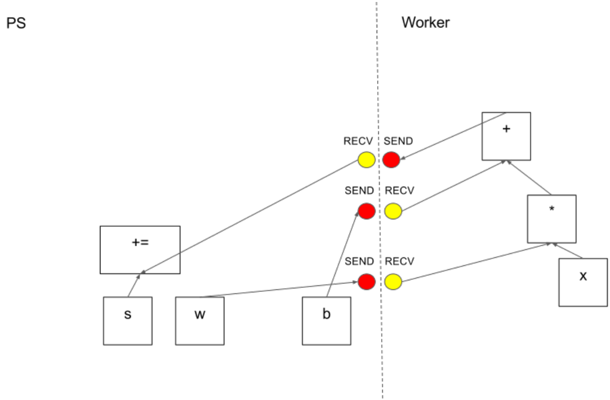
# Layers of TensorFlow - Distributed Master

The distributed master has grouped the model parameters in order to place them together on the parameter server.

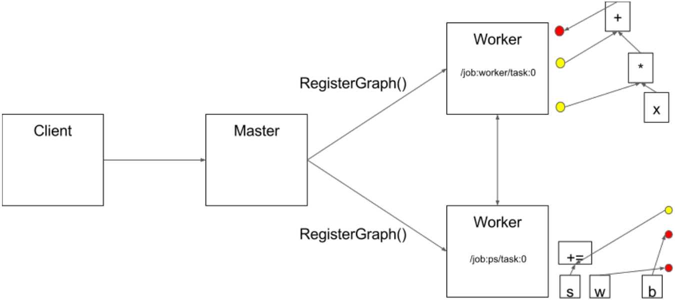


# Layers of TensorFlow - Distributed Master

Where graph edges are cut by the partition, the distributed master inserts send and receive nodes to pass information between the distributed tasks.

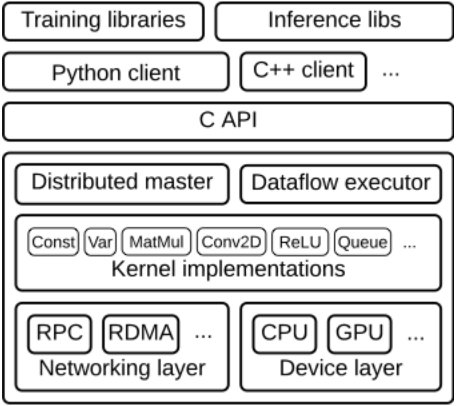


The distributed master then ships the graph pieces to the distributed tasks.




THE UNIVERSITY OF CHICAGO

# Layers of TensorFlow - Worker Services

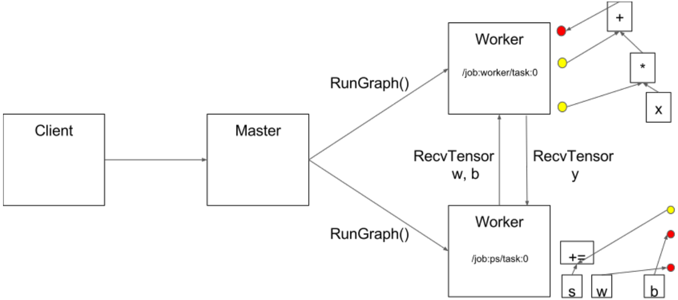



**Worker Services:**

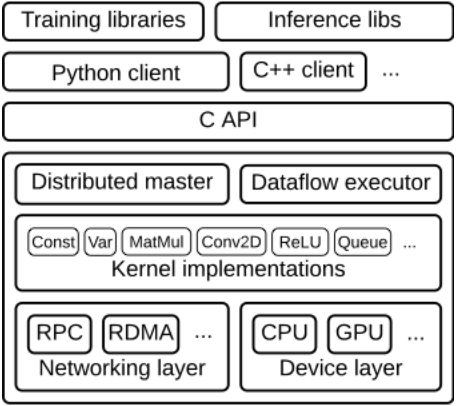
- Schedule the execution of graph operations using kernel implementations appropriate to the available hardware (CPUs, GPUs, TPUs, etc).
- Send and receive operation results to and from other worker services.

# Layers of TensorFlow - Worker Services

**gRPC over TCP**



# Layers of TensorFlow - Kernel Implementations




**Kernel Implementations:**

- Perform the computation for individual graph operations.

# Client - Master - Workers

The main components in a TensorFlow system are the **client**, which uses the **Session** interface to communicate with the master, and one or more **worker processes**, with each worker process responsible for arbitrating access to one or more computational **devices** (such as CPU cores or GPU cards) and for **executing graph nodes on those devices as instructed by the master**.

# Client - Master - Workers

The main components in a TensorFlow system are the **client**, which uses the **Session** interface to communicate with the master, and one or more **worker processes**, with each worker process responsible for arbitrating access to one or more computational **devices** (such as CPU cores or GPU cards) and for **executing graph nodes on those devices as instructed by the master**.

## Question

Why does TensorFlow's model of computation has only a single master?

# Client - Master - Workers

The main components in a TensorFlow system are the **client**, which uses the **Session** interface to communicate with the master, and one or more **worker processes**, with each worker process responsible for arbitrating access to one or more computational **devices** (such as CPU cores or GPU cards) and for **executing graph nodes on those devices as instructed by the master**.

## Question

Why does TensorFlow's model of computation has only a single master?

## Solution

Jeff Dean: MapReduce, GFS, etc.

THE UNIVERSITY OF CHICAGO

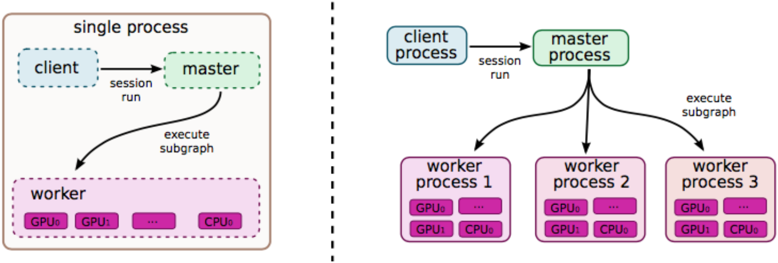
# Local vs Distributed (1)

There are 2 implementations of TensorFlow:

1. **Local**: The local implementation is used when the client, the master, and the worker all run on a single machine in the context of a **single operating system process** (possibly with multiple devices, e.g. GPU cards in one machine).
2. **Distributed**: The distributed implementation shares most of the code with the local implementation, but extends it with support for an environment where the client, the master, and the workers can all be in **different processes on different machines**.

THE UNIVERSITY OF CHICAGO

# Local vs Distributed (2)



# Single-Device Execution

This is the simplest execution scenario: a single worker process with a single device. The nodes of the graph are executed in **an order that respects the dependencies between nodes**.

# Single-Device Execution

This is the simplest execution scenario: a single worker process with a single device. The nodes of the graph are executed in **an order that respects the dependencies between nodes**.

**Question**

How to find the **order that respects the dependencies between nodes**?

# Single-Device Execution

This is the simplest execution scenario: a single worker process with a single device. The nodes of the graph are executed in **an order that respects the dependencies between nodes**.

## Question

How to find the **order that respects the dependencies between nodes**?

## Solution

- We keep track of a count per node of the number of dependencies of that node that have not yet been executed.
- Once this count drops to zero, the node is eligible for execution and is added to a ready queue.
- The ready queue is processed in some unspecified order, delegating execution of the kernel for a node to the device object.
- When a node has finished executing, the counts of all nodes that depend on the completed node are decremented.

# Multi-Device Execution

Once a system has multiple devices, there are two main complications:

1. Deciding which device to place the computation for each node in the graph ⇒ **Node Placement Algorithm**.
2. Managing the required communication of data across device boundaries implied by these placement decisions ⇒ **Cross-Device Communication**.

# Node Placement Algorithm (1)

**Task**

Given a computation graph, one of the main responsibilities of the TensorFlow implementation is to map the computation onto the set of available devices.

# Node Placement Algorithm (1)

## Task

Given a computation graph, one of the main responsibilities of the TensorFlow implementation is to map the computation onto the set of available devices.

## Input

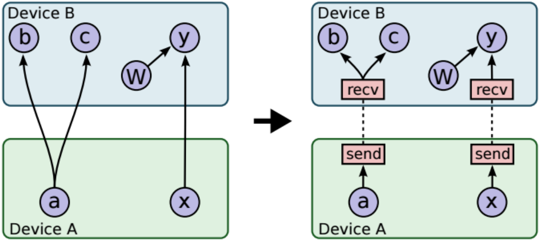
One input to the placement algorithm is a **cost model**, which contains estimates of the sizes (in bytes) of the input and output tensors for each graph node, along with estimates of the computation time required for each node when presented with its input tensors.

# Node Placement Algorithm (2)

## Solution

1. The placement algorithm runs a **simulated execution** of the graph.
2. It starts with the sources of the computation graph, and simulates the activity on each device in the system as it progresses.
3. For nodes with multiple devices, the placement algorithm uses a **greedy heuristic** that examines the effects on the completion time of the node of placing the node on each possible device.
4. The node to device placement generated by this simulation is also used as the placement for the real execution.

# Cross-Device Communication



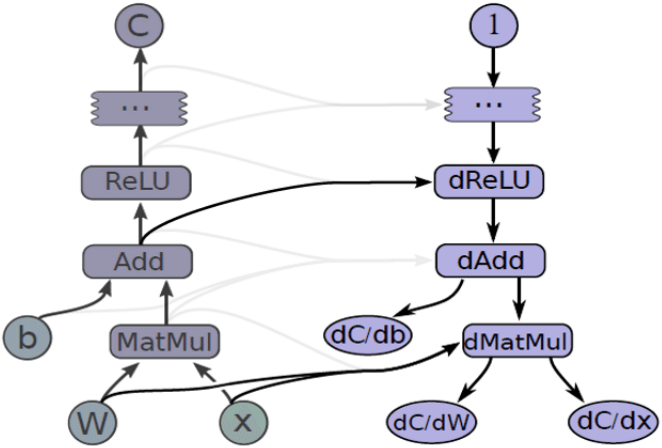

Decentralization philosophy: allow the scheduling of individual nodes of the graph on different devices to be decentralized into the workers.

# Gradient Computation (1)

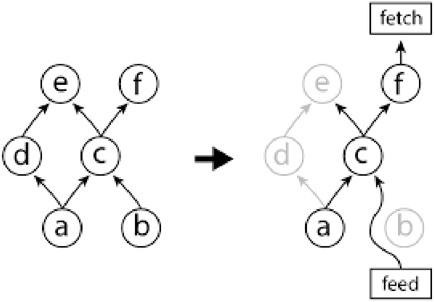
Consider the following Python code:

```
tensorflow as tf
b = tf.Variable(tf.zeros([1000]))
W = tf.Variable(tf.random_uniform([784, 1000], -1, 1))
x = tf.placeholder(name = "x")
relu = tf.nn.relu(tf.matmul(W, x) + b)
C = [...]
s = tf.Session()
for step in range(0, 100):
 input = ...
 result = s.run(C, feed_dict = x:  input)
 print step, result
```

# Gradient Computation (2)

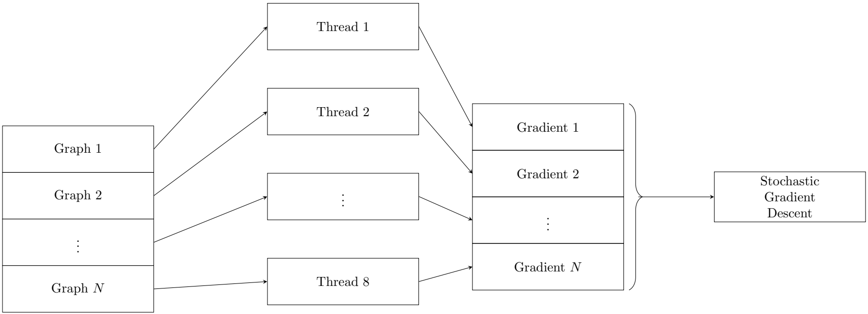


# Partial Execution



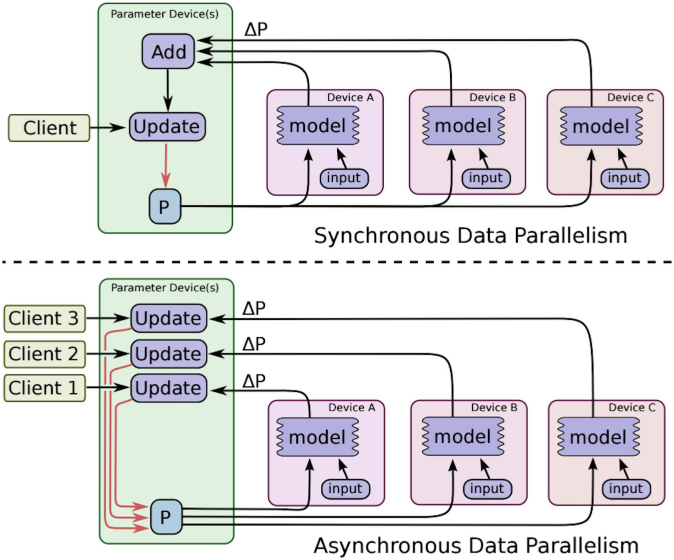

Often a client wants to execute just a subgraph of the entire execution graph. For example: Only route $f \leftarrow c \leftarrow a$ is needed.

# Data Parallel Training - Single CPU



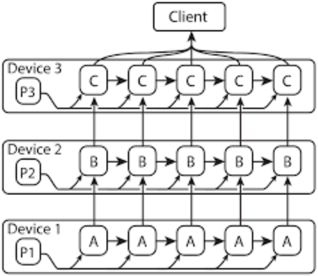

Mini-Batch

# Synchronous vs Asynchronous




THE UNIVERSITY OF CHICAGO

# Model Parallel Training




Model parallel training, where different portions of the model computation are done on different computational devices simultaneously for the same batch of examples. For example: A recurrent, deep LSTM model used for sequence to sequence learning, parallelized across three different devices.

# Customized Kernel Implementation - PyTorch (1)

**so3vector_product.cpp**

```
std::vector<torch::Tensor> product_forward(
    const std::vector<torch::Tensor> &v1,
    const std::vector<torch::Tensor> &v2,
    const int L
) {
    ...
}

std::vector<torch::Tensor> product_backward(
    const std::vector<torch::Tensor> &product_grad,
    const std::vector<torch::Tensor> &v1,
    const std::vector<torch::Tensor> &v2
) {
    ...
}
```

THE UNIVERSITY OF CHICAGO

# Customized Kernel Implementation - PyTorch (2)

**so3vector_product.cpp**

```
PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
    m.def(
        "product_forward",
        &product_forward,
        "Tensor product operation - Forward pass");

    m.def(
        "backward_forward",
        &backward_forward,
        "Tensor product operation - Backward pass");
}
```

# Customized Kernel Implementation - PyTorch (3)

**_so3vector_product.py**

```
import so3vector_product

class SO3vector_Product(torch.autograd.Function):
  @staticmethod
  def forward(ctx, v1, v2, L):
    product = so3vector_product.product_forward(v1, v2, L)
    variables = v1 + v2
    ctx.save_for_backward(*variables)
    ctx.L1 = len(v1)
    return tuple(product)

  @staticmethod
  def backward(ctx, product_grad):
    v = ctx.saved_variables
    v1 = v[0:ctx.L1+1]
    v2 = v[ctx.L1+1:]
    grads = so3vector_product.product_backward(product_grad, v1, v2)
    v1_grad = grads[0:ctx.L1+1]
    v2_grad = grads[ctx.L1+1:]
    return tuple(v1_grad), tuple(v2_grad)
```

# Customized Kernel Implementation - TensorFlow (1)

**CPU_API.cc**

```
typedef float TYPE;
class TensorProductOp : public OpKernel {
public:
    explicit TensorProductOp(OpKernelConstruction *context) : OpKernel(context) {
        OP_REQUIRES_OK(context, context -> GetAttr("L", &L));
    }
    void Compute(OpKernelContext *context) override {
        ...
    }
private:
    int L;
};


REGISTER_OP("TensorProduct")
.Attr("T: list(type)")
.Attr("L: int")
.Input("in: T")
.Output("out: T")
;

REGISTER_KERNEL_BUILDER(Name("TensorProduct").Device(DEVICE_CPU),
    TensorProductOp);
```

# Customized Kernel Implementation - TensorFlow (2)

**CPU_API.cc**

```
class TensorProductGradOp : public OpKernel {
public:
    explicit TensorProductGradOp(OpKernelConstruction *context) : OpKernel(context)
        OP_REQUIRES_OK(context, context -> GetAttr("L", &L));
    }
    void Compute(OpKernelContext *context) override {
        ...
    }
private:
    int L;
};

REGISTER_OP("TensorProductGrad")
.Attr("P: list(type)")
.Attr("T: list(type)")
.Input("product_grad: P")
.Input("v: T")
.Output("v_grads: T")
;

REGISTER_KERNEL_BUILDER(Name("TensorProductGrad").Device(DEVICE_CPU),
    TensorProductGradOp);
```

# Customized Kernel Implementation - TensorFlow (3)

**CPU_API_grad.py**

```
CPU_API = tf.load_op_library("CPU_API/CPU_API.so")

@ops.RegisterGradient("TensorProduct")
def tensor_product_grad(op, *grad):
    L = op.get_attr("L")
    return CPU_API.tensor_product_grad(grad, op.inputs, L = L)
```