ST446 Distributed Computing for Big Data

Lecture 10

Distributed dataflow graph computations

Milan Vojnovic

https://github.com/lse-st446/lectures2021

Goals of this lecture

- To introduce main concepts of dataflow graph computation for learning neural networks
- To learn main architectural principles of distributed systems for deep learning
- To learn about distributed computing principles of TensorFlow

Remarks:

- TensorFlow is used for discussion of distributed computing concepts and their implementation - many of these concepts are general
- The design of TensorFlow is discussed starting with a precursor system DistBelief, then TensorFlow 1.x and finally TensorFlow 2.x

Topics of this lecture

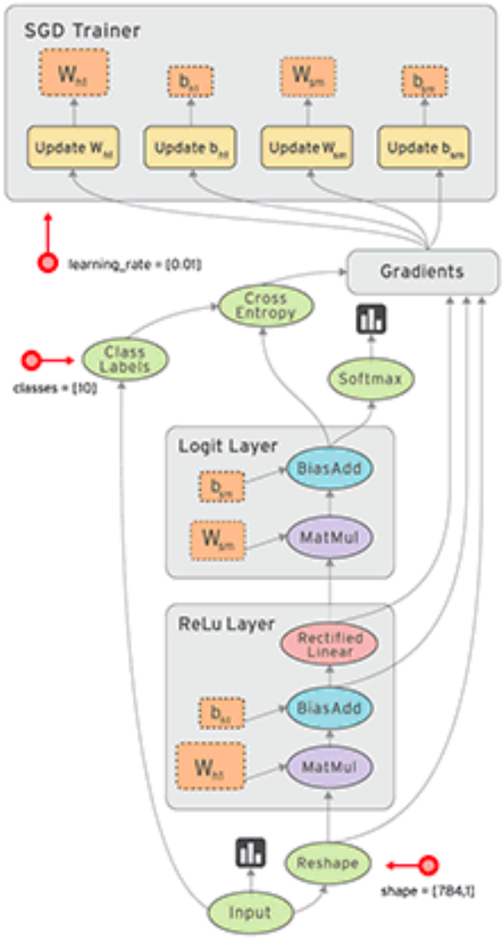
- Introduction to dataflow graph computations
- Graph, operations and tensors
- Computer system implementation
- Distributed computing strategies

Introduction to dataflow graph computations

Computation using dataflow graphs

- Dataflow graph: a graph representing computation, shared state and operations that mutate the state
- Distributed computing: mapping nodes of a dataflow graph across devices of a machine or machines of a cluster
  - Ex multicore CPUs, GPUs, TPUs
- Desiderata: a flexible system architecture
  - Allowing application developers to define dataflow graphs
  - Relaxing hard constraints of parameter server architecture
  - Enabling developers to experiment with new optimization algorithms
- Applications: general numerical computations, but primary focus on training and inference for deep neural networks
  - Support for both training (fitting parameters) and inference (making predictions)

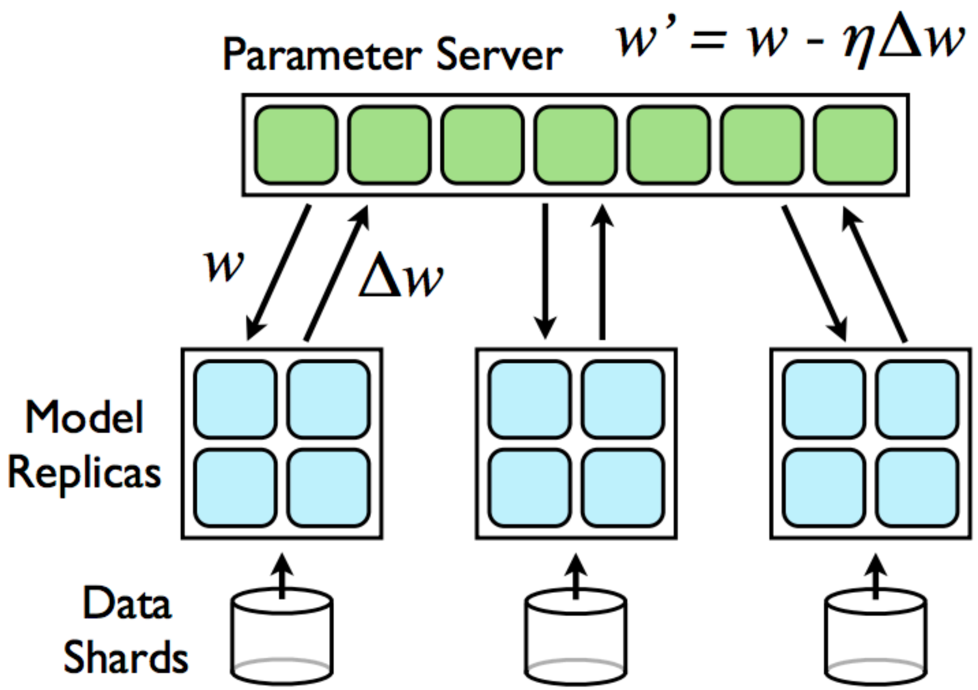
An illustration of a dataflow graph



- Source: https://www.tensorflow.org/guide/graphs

Early days: DistBelief

- DistBelief: a first-generation system
- Used in production by Google for 10+ years
- Based on parameter server architecture
- Described in [Dean et al, NIPS 2012]



Parameter server user input

- User input: a neural network definition and a loss function

- Neural network defined as a directed acyclic graph (DAG) of mathematical operators mapping an input variable to an output variable
  - Feedforward neural network architecture: operators partitioned in layers with a *connection between two operators allowed only if they reside in adjacent layers*

- For example, fully connected layer implements multiplication of the input with a weight matrix, addition of a bias term, and application of a non-linear activation function

- Activation function example ReLU $a(x) = \max\{x, 0\}$, and softmax $a_i(x) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$

- The weight matrix and bias terms are trainable parameters

Parameter server architecture: system concepts

- Each job consists of two different types of processes: parameter servers and worker nodes
- Parameter servers: maintain current values of model parameters
  - Stateful: maintaining state as the training goes through iterations of loss function minimization
- Worker nodes: perform the bulk of the work
  - Compute the gradient vector components of the training loss function
  - Stateless: no persistent state kept as the training goes through iterations of loss function minimization
  - DAG structure and knowledge of the semantics of layers used to compute gradients using the backpropagation algorithm
- Parameter updates are typically commutative and have weak consistency requirements, allowing worker nodes to compute gradients independently and write delta updates to parameter servers
- Parameter servers and worker nodes communicate using a simple interface
  - `put()` and `get()` functions

To probe further on backpropagation algorithm: [wikipedia]

TensorFlow vs DistBelief

- New requirements by application developers asked for new system architecture

- General requirement for more flexibility
  - Allowing experimentation with new optimization methods: hard to do in DistBelief as it requires modifying the parameter server implementation
  - Simple interface of DistBelief considered restrictive: ex. not allowing atomic updates of related parameters; not allowing offloading computation to a parameter server

- DistBelief was designed to follow a repetitive fixed execution pattern:
  - Worker nodes read a batch of input data, and parameter values from parameter servers
  - Compute loss function value (forward pass through the DAG)
  - Compute gradient components (backward pass through the DAG)
  - Write back the gradients to parameters servers

TensorFlow vs DistBelief (cont'd)

- The execution model of DistBelief suited only for *some* neural network architectures
  - Suitable for feedforward neural networks
  - Less suitable for recurrent neural networks, adversarial networks (two related network trained alternatively), and reinforcement learning models (loss function computed by some agent in a separate system)
- DistBelief was *primarily designed* for large clusters of multi-core servers
  - GPU acceleration was added only later (ex. used for efficient execution of convolutional kernels)
  - Difficult to scale down to other environments (ex. training first locally on a GPU equipped server before scaling the same code for training on a much larger dataset)
- TensorFlow provides a single programming model and runtime system
  - Combined provide flexibility and support for different environments

TensorFlow key design principles

- TensorFlow design inspired by
  - Dataflow systems: high-level programming model
  - Parameter server architecture: low-level system implementation and efficiency

- Traditional dataflow systems: DAG nodes represent functional computation on immutable data

- TensorFlow dataflow principles:
  - DAG nodes represent computations that own or update mutable state
  - Edges carry tensors (multi-dimensional arrays) between nodes

TensorFlow key design principles (cont'd)

- Simple dataflow-based programming abstraction
- Unified computation and state management in a single programming model
- Allows users to deploy applications on distributed clusters, local workstations, mobile devices and custom-designed accelerators
- High-level scripting interface wraps the construction of dataflow graphs
- Enables users to experiment with different model architectures and optimization algorithms without modifying the core system

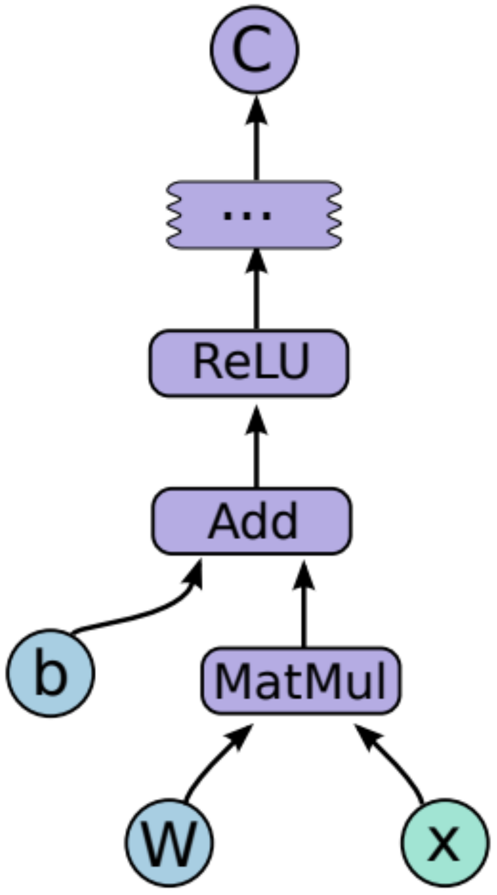
Graph, operations and tensors

MNIST character recognition example

```
import tensorflow as tf

mnist = mnist_data.read_data_sets("data", one_hot=True, reshape=False, validation_size=0)

X = tf.placeholder(tf.float32, [None, 28, 28, 1])
Y_ = tf.placeholder(tf.float32, [None, 10])

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

XX = tf.reshape(X, [-1, 784])

Y = tf.nn.softmax(tf.matmul(XX, W) + b)

cross_entropy = -tf.reduce_mean(Y_ * tf.log(Y)) * 1000.0

train_step = tf.train.GradientDescentOptimizer(0.005).minimize(cross_entropy)

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

for i in range(2000+1):
    batch_X, batch_Y = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={X: batch_X, Y_: batch_Y})
```

Dataflow graph representation

- Dataflow graph of primitive operators, represented as nodes
- Vertex (or node): represents a unit of local computation
- Edge: represents output from or input to a vertex
- Operation: computation performed by a vertex
- Primitive operators: mathematical operators like matrix multiplication, addition, function mapping



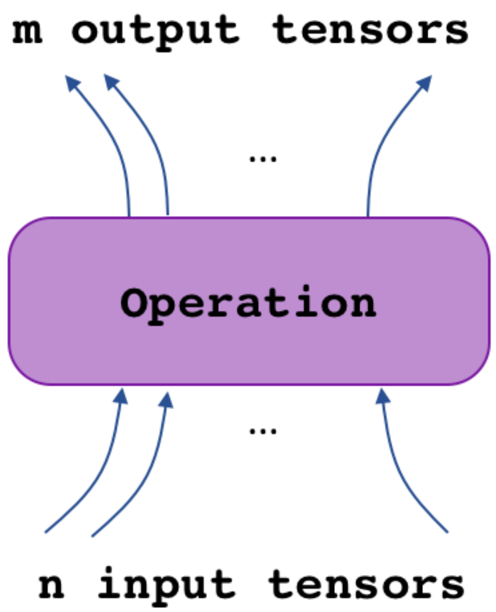
Operations

- Mathematical operations such as matrix multiplication `tf.matmul` and softmax function mapping `tf.softmax`

- Inputs and output of an operation are tensors (multi-dimensional arrays)

- Operations can be stateful
  - Ex. variables and queues

- More about operations here:
  - API guides: [math ops]
  - API guides: [matmul]



Stateful operations: variables

- An operation may contain mutable state that is read and/or written each time the operation is executed

- Variable operation: owns a mutable buffer that can be used to store shared parameters of a model as it is trained
  - Has no inputs, but a reference handle which acts as a typed capability for reading from or writing to the buffer

- Read operation: takes a reference handle `r` as input and outputs the value of the variable `State[r]` as a dense tensor

- Modify operation: takes a reference handle `r` and a tensor value `x` and then performs the state update
  - Ex. `AssignAdd: State[r] <- State[r] + x`

Stateful operations: queues

- Queue operation: allows adding and removing tensors from a queue in a specified order

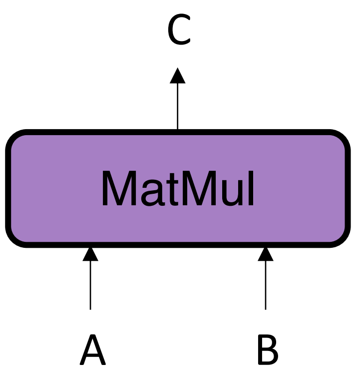
- Support different types of queues:
  - `FIFOQueue` (first-in-first-out order)
  - `RandomQueue` (random order)
  - `PriorityQueue` (priority index order)

- Support for standard queue operations: enqueue and dequeue

- Combining queues and dynamic control flow allows implementation of streaming computations between subgraphs

Tensors

- Tensors: multi-dimensional arrays of elements of primitive data types
- Primitive data types: `int32`, `float32`, `string` (arbitrary binary data)
- Example: matrix multiplication



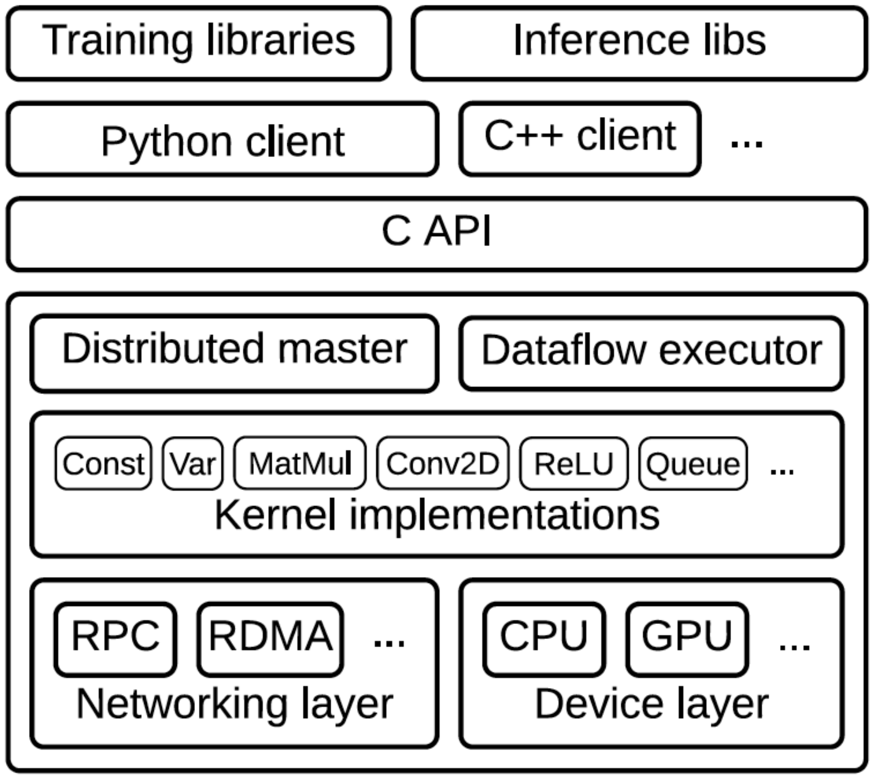
A, B, C are 2-dimensional tensors

- All data is represented by tensors in TensorFlow
- Tensors are either dense or sparse
- At the lowest level, tensors are dense
- Sparse tensors can be encoded as
  - Dense tensors by a variable-length string of elements
  - Using a tuple of dense tensors, ex. n-dimensional tensor with m non-zero elements encoded with a m-dimensional dense tensors of element coordinates and a m-dimensional dense tensor of element values

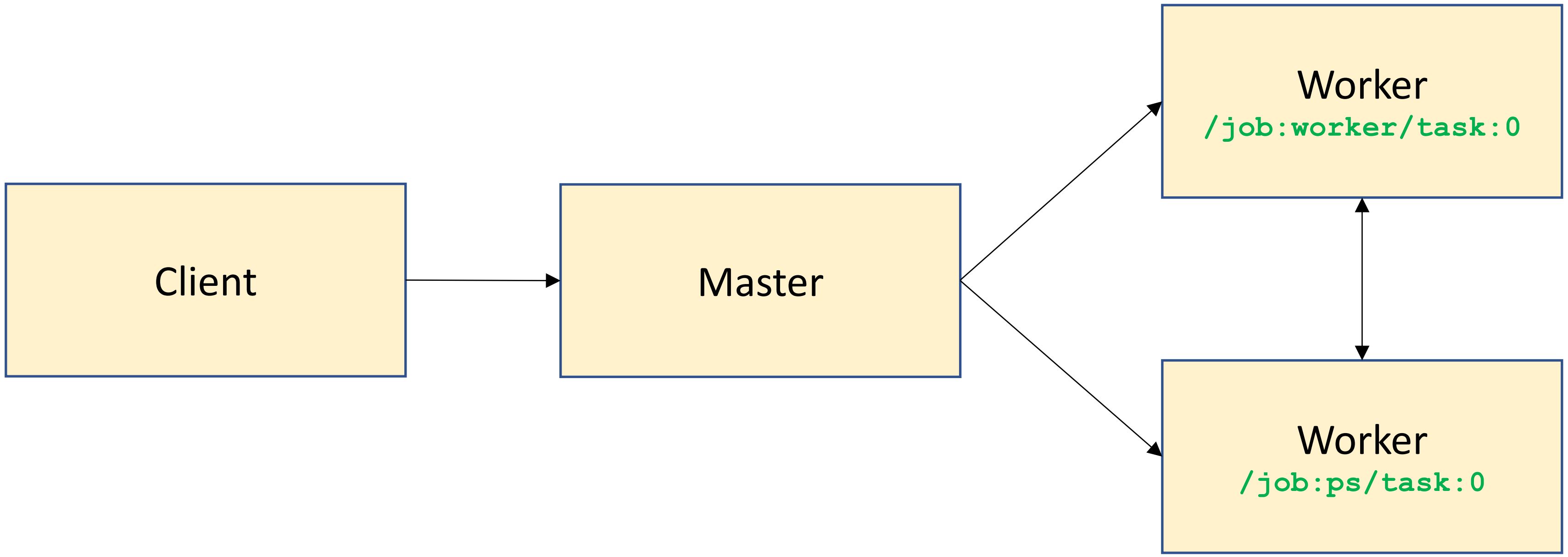
Computer system implementation

Implementation

- The core of the library is implemented in C++ for portability and performance
- Runs of different operations systems (Linux, Mac OS X, Windows, Android, iOS, …)
- The distributed master translates user requests into execution across a set of tasks
- The dataflow executor in each task handles requests from the master and schedules the execution of the kernels that comprise a local subgraph

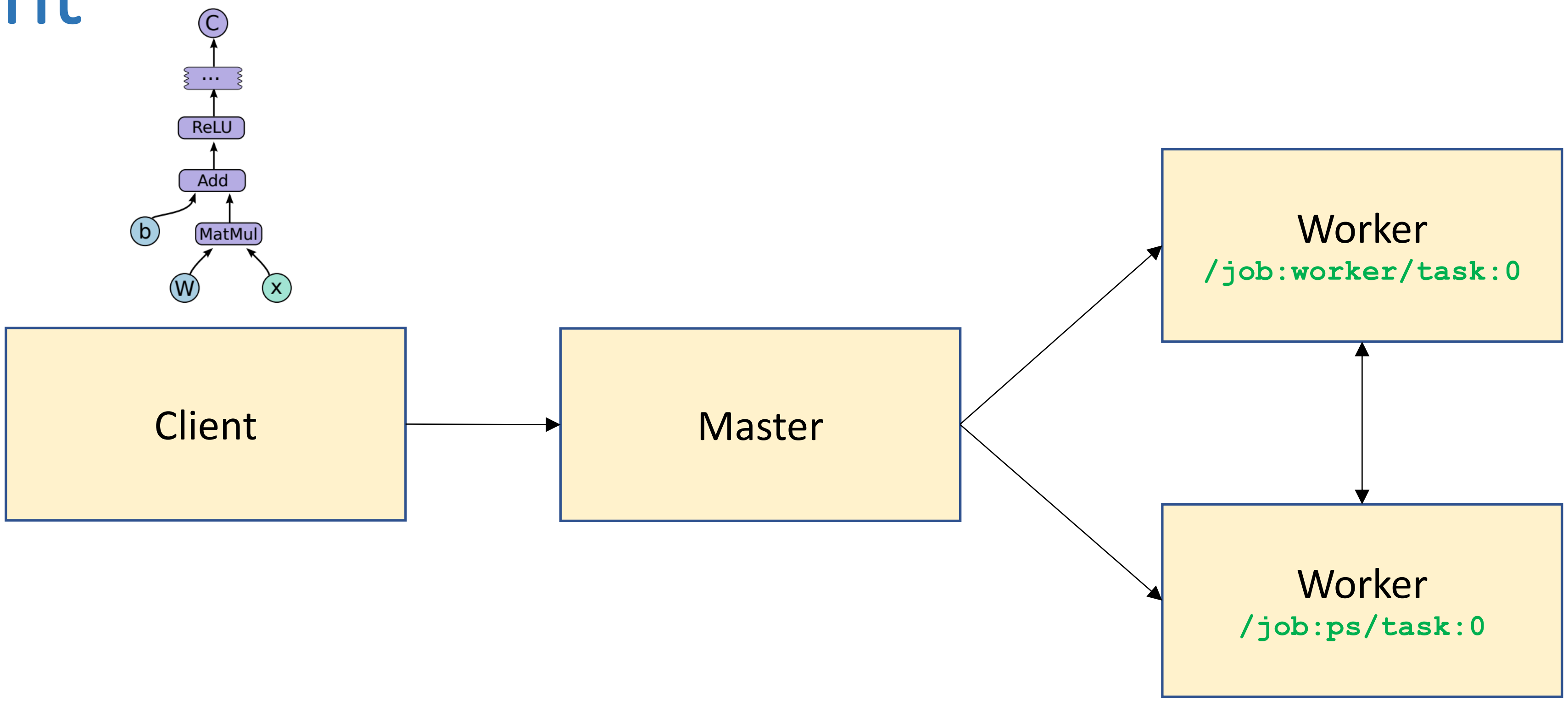


Client, master, worker nodes



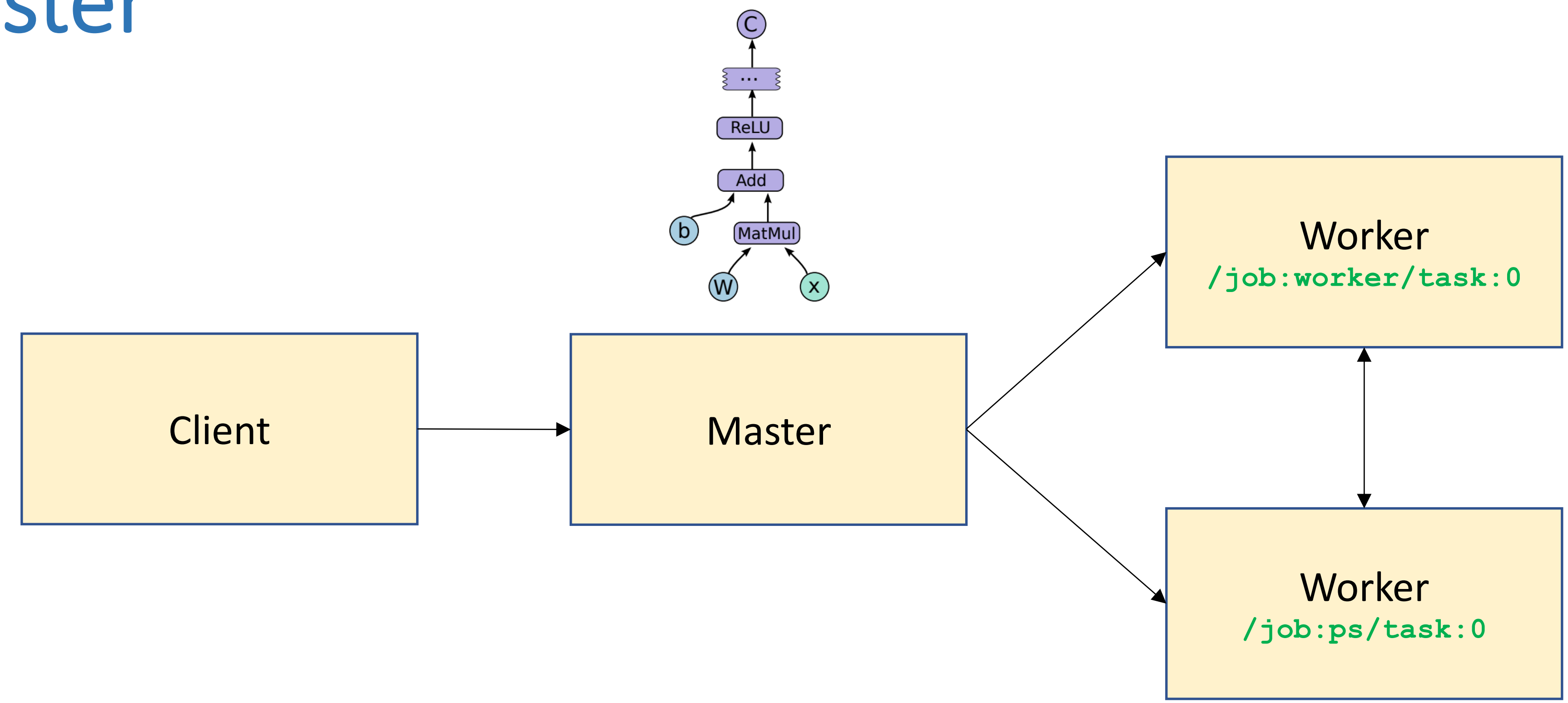
- `/job:worker/task:0` and `/job:ps/task:0` are both tasks with worker services
- `PS` parameter server: a task responsible for storing and updating model parameters
- Other tasks send updates to these parameters

Client



- User writes the client program that builds the computation graph
- Client creates a session, which sends the graph to master (using tf.GraphDef proto buffer)
- When the client evaluates a node in the graph, the evaluation triggers a call to the distributed master to initiate computation

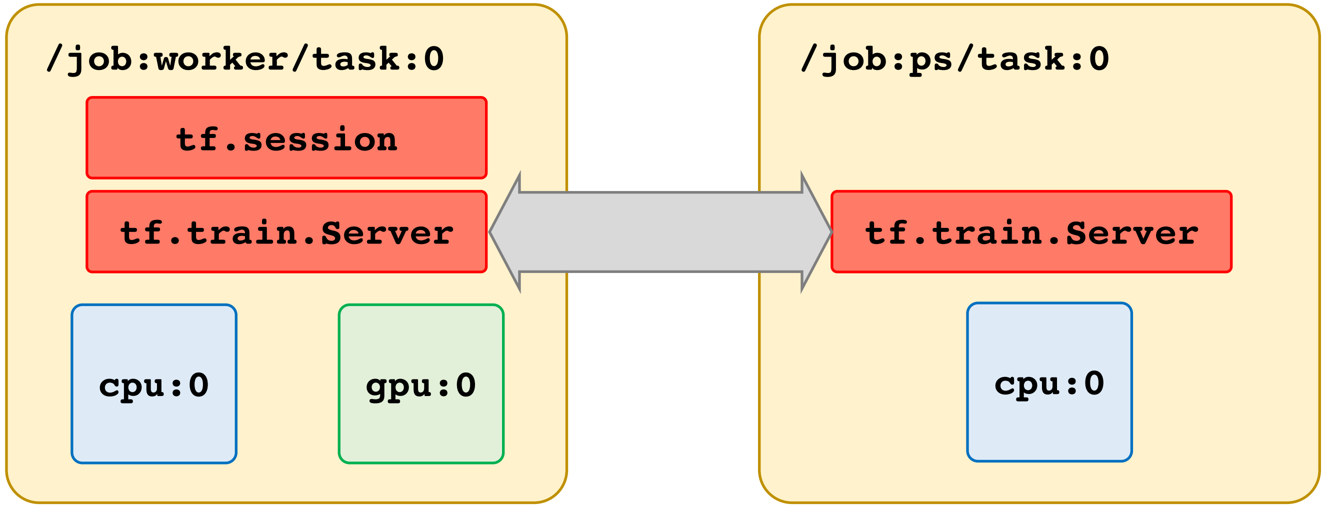
Master



- Prunes the graph to obtain the subgraph required to evaluate the nodes requested by the client
- Partitions the graph to obtain graph pieces for each device
- Caches these pieces so that they may be re-used in subsequent steps

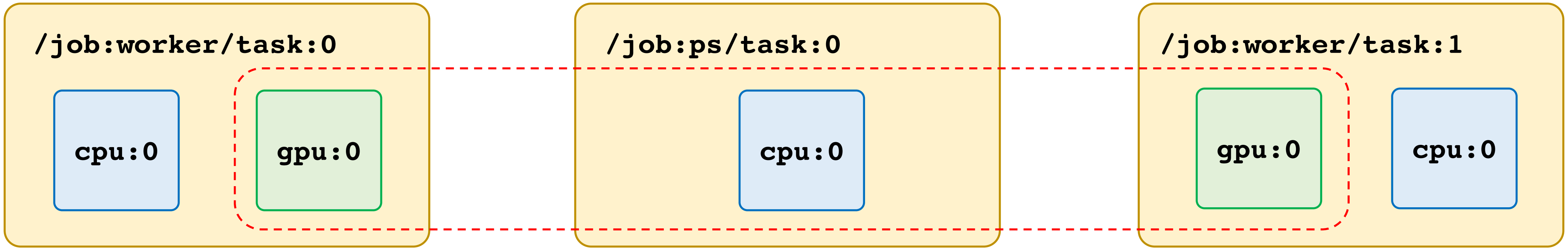
Sessions and servers

- A session runs one or more servers
- A server represents a task (either ps or worker)



In-graph replication

- Model is replicated over worker nodes

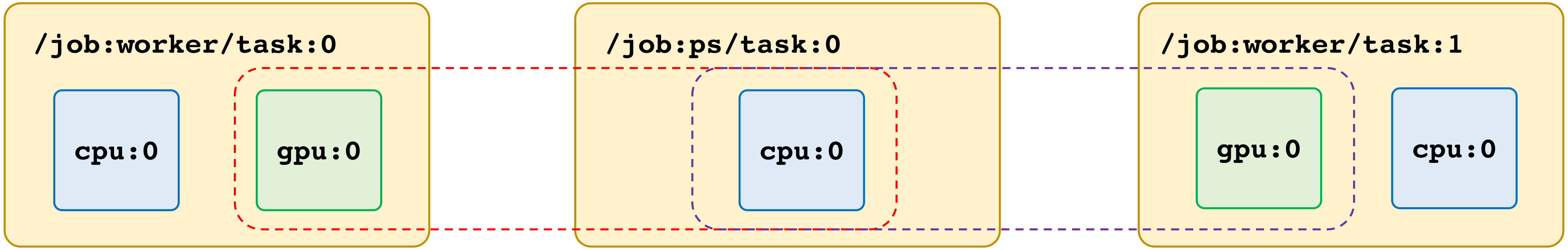



```python
with tf.device("/job:ps/task:0/cpu:0"):
    W = tf.Variable(...)
    b = tf.Variable(...)
    inputs = tf.split(0, num_workers, input)
    outputs = []
    for i in range(num_workers):
        with tf.device("/job:worker/task:%d/gpu:0", % i):
            outputs.append(tf.matmul(input[i], W) + b)
    loss = f(outputs)
```

- Limitation: client becomes a performance bottleneck when dealing with many model replicas

Between-graph replication

- Between graph replication: there is *a separate client* for each /job:worker task, typically in the same process as the worker task



Client 1:

```
with tf.device("/job:ps/task:0/cpu:0"):
    W = tf.Variable(...)
    b = tf.Variable(...)
    with tf.device("/job:worker/task:0/gpu:0"):
        output = tf.matmul(input,W) + b
        loss = func(output)
```

Client 2:

```
with tf.device("/job:ps/task:0/cpu:0"):
    W = tf.Variable(...)
    b = tf.Variable(...)
    with tf.device("/job:worker/task:1/gpu:0"):
        output = tf.matmul(input,W) + b
        loss = func(output)
```

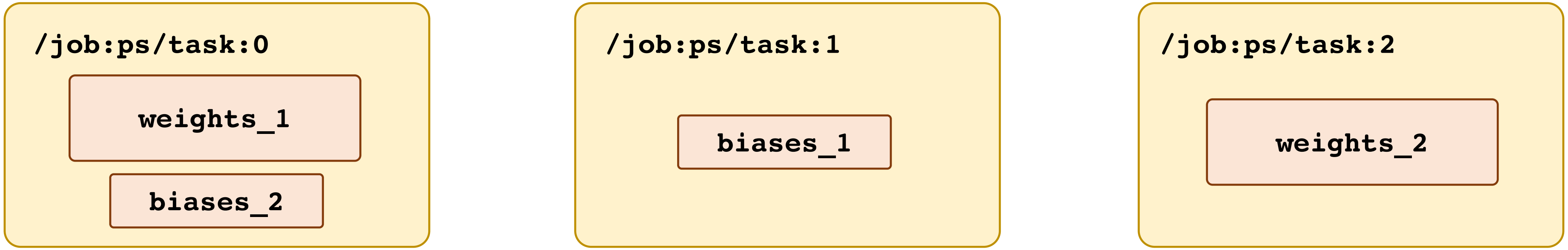
Variable placement

- Variables can be placed onto parameter servers
- Useful for distributing parameters across multiple parameters severs for models with many parameters
  - Ex. BERT 110M parameters
  - Ex. Turning-NLG 17B parameters
- Example:

```
with tf.device("/job:ps/task:0"):
    weights_1 = tf.get_variable("weights_1", [784,100])
    biases_1 = tf.get_variable("biases_1", [100])
    weights_2 = tf.get_variable("weights_2", [100,10])
    biases_2 = tf.get_variable("biases_2", [10])
```

Round-robin placement of variables

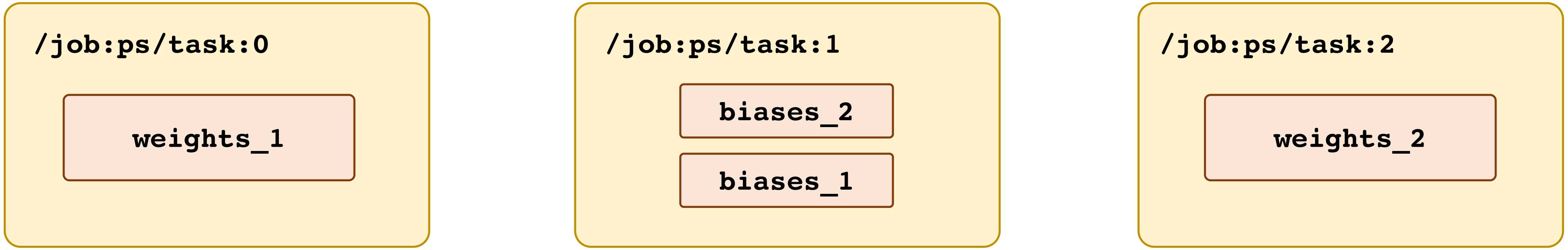
- Round-robin placement: assigning variables to tasks in a circular order




```
with tf.device("tf.train.replica_device_setter(ps_tasks=3)):"):
    weights_1 = tf.get_variable("weights_1", [784,100])
    biases_1 = tf.get_variable("biases_1", [100])
    weights_2 = tf.get_variable("weights_2", [100,10])
    biases_2 = tf.get_variable("biases_2", [10])
```

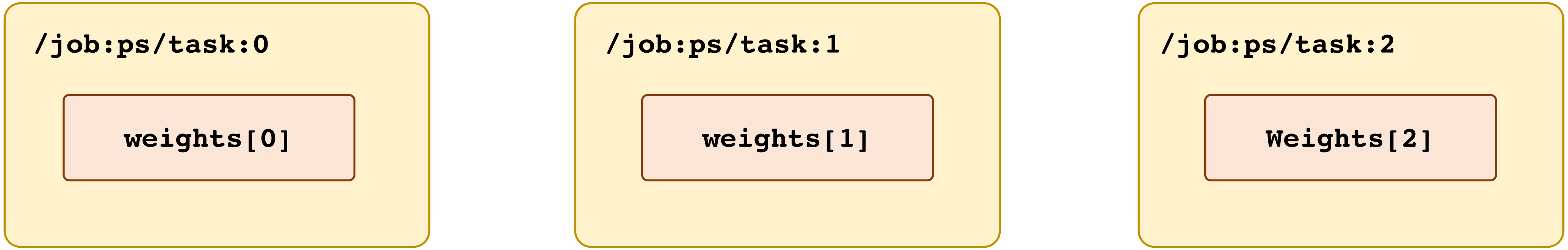
Load-balancing placement of variables

- Aim to minimize the maximum load across machines using some heuristics




```
greedy = tf.contrib.training.GreedyLoadBalancingStrategy(...)

with tf.device("tf.train.replica_device_setter(ps_tasks=3, ps_strategy=greedy)):"):
    weights_1 = tf.get_variable("weights_1", [784,100])
    biases_1 = tf.get_variable("biases_1", [100])
    weights_2 = tf.get_variable("weights_2", [100,10])
    biases_2 = tf.get_variable("biases_2", [10])
```

Partitioning variables

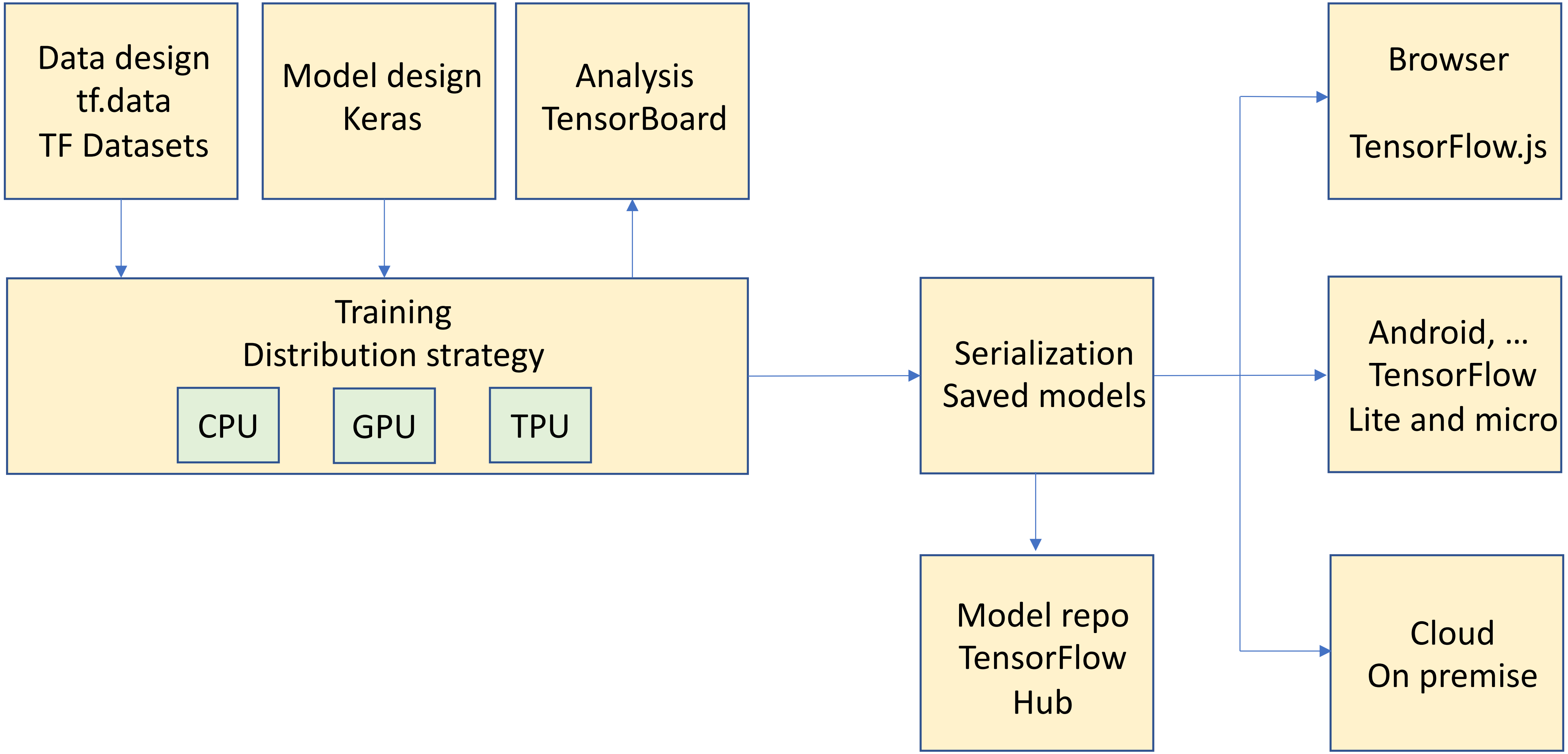



```
greedy = tf.contrib.training.GreedyLoadBalancingStrategy(...)

with tf.device("tf.train.replica_device_setter(ps_tasks=3, ps_strategy=greedy)):"):
    weights = tf.get_variable("weights", [10000000000000000,10], partitioner=tf.fixed_size_partitioner(3))
```

TensorFlow 2.x

- Several major changes have been introduced both for distributed training and API
- Introduced different strategies for distributed computation over devices (e.g. GPU, TPU) and worker machines: `tf.distribute.strategy`
- Parameter server architecture is just one strategy
- Several APIs allowing for varied degree of flexibility and easiness of use

Big picture



Data design
tf.data
TF Datasets
Model design
Keras
Analysis
TensorBoard
Training
Distribution strategy
CPU
GPU
TPU
Serialization
Saved models
Model repo
TensorFlow
Hub
Browser
TensorFlow.js
Android, …
TensorFlow
Lite and micro
Cloud
On premise

Several different APIs

- Sequential API
  - Level: new users, simple models
- Functional API + built-in layers
  - Level: engineers working on standard use cases
- Functional API + custom layers, metrics, and losses
  - Level: Engineers requiring increasing control
- Subclassing
  - Level: researcher
  - Writing everything yourself from scratch
  - Chainer / PyTorch style

Import TensorFlow and TensorFlow Datasets

```
# Import TensorFlow and TensorFlow Datasets
!pip install tf-nightly

import tensorflow_datasets as tfds
import tensorflow as tf
tfds.disable_progress_bar()

import os
```

```
[2] print(tf.__version__)

2.2.0-dev20200329
```

MNIST example:

```
datasets, info = tfds.load(name='mnist', with_info=True, as_supervised=True)

mnist_train, mnist_test = datasets['train'], datasets['test']
```

```
num_train_examples = info.splits['train'].num_examples
num_test_examples = info.splits['test'].num_examples

BUFFER_SIZE = 10000

BATCH_SIZE_PER_REPLICA = 64
BATCH_SIZE = BATCH_SIZE_PER_REPLICA * strategy.num_replicas_in_sync
```

```
train_dataset = mnist_train.map(scale).cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
eval_dataset = mnist_test.map(scale).batch(BATCH_SIZE)
```

Sequential API example

- Define the model

```
model = tf.keras.Sequential([
  tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
  tf.keras.layers.MaxPooling2D(),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(64, activation='relu'),
  tf.keras.layers.Dense(10)
])
```

- Compile the model

```
model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                optimizer=tf.keras.optimizers.Adam(),
                metrics=['accuracy'])
```

Sequential API example (cont'd)

- Define the callbacks

```
# Define the checkpoint directory to store the checkpoints

checkpoint_dir = './training_checkpoints'
# Name of the checkpoint files
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch}")

# Function for decaying the learning rate.
# You can define any decay function you need.
def decay(epoch):
  if epoch < 3:
    return 1e-3
  elif epoch >= 3 and epoch < 7:
    return 1e-4
  else:
    return 1e-5

# Callback for printing the LR at the end of each epoch.
class PrintLR(tf.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs=None):
    print('\nLearning rate for epoch {} is {}'.format(epoch + 1,
                                                      model.optimizer.lr.numpy()))

callbacks = [
    tf.keras.callbacks.TensorBoard(log_dir='./logs'),
    tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_prefix,
                                       save_weights_only=True),
    tf.keras.callbacks.LearningRateScheduler(decay),
    PrintLR()
]
```

- Fit the model

```
model.fit(train_dataset, epochs=12, callbacks=callbacks)

model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))

eval_loss, eval_acc = model.evaluate(eval_dataset)

print('Eval loss: {}, Eval Accuracy: {}'.format(eval_loss, eval_acc))
```

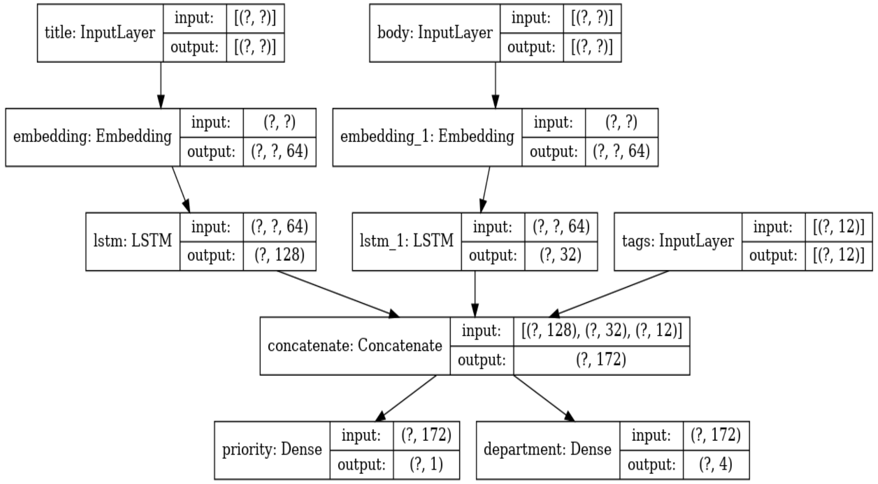
```
Epoch 1/12
936/938 [============================>.] - ETA: 0s - accuracy: 0.9442 - loss: 0.1924
Learning rate for epoch 1 is 0.0010000000474974513
938/938 [==============================] - 31s 33ms/step - accuracy: 0.9442 - loss: 0.1921 - lr: 0.0010
Epoch 2/12
936/938 [============================>.] - ETA: 0s - accuracy: 0.9800 - loss: 0.0657
Learning rate for epoch 2 is 0.0010000000474974513
938/938 [==============================] - 25s 27ms/step - accuracy: 0.9800 - loss: 0.0657 - lr: 0.0010
Epoch 3/12
936/938 [============================>.] - ETA: 0s - accuracy: 0.9863 - loss: 0.0447
Learning rate for epoch 3 is 0.0010000000474974513
938/938 [==============================] - 25s 27ms/step - accuracy: 0.9863 - loss: 0.0447 - lr: 0.0010
Epoch 4/12
937/938 [============================>.] - ETA: 0s - accuracy: 0.9934 - loss: 0.0240
Learning rate for epoch 4 is 9.999999747378752e-05
938/938 [==============================] - 25s 27ms/step - accuracy: 0.9934 - loss: 0.0240 - lr: 1.0000e-04
Epoch 5/12
937/938 [============================>.] - ETA: 0s - accuracy: 0.9947 - loss: 0.0208
Learning rate for epoch 5 is 9.999999747378752e-05
```

- To probe further: writing your own callbacks

Functional API example

- A way to create models that is more flexible than the `tf.keras.Sequential` API
- Allows for models with ”non-linear” topologies, models with sharded layers, and models with multiple inputs or outputs
- Example: ranking custom issue tickets by priority and routing them to a department

```python
num_tags = 12  # Number of unique issue tags
num_words = 10000  # Size of vocabulary obtained when preprocessing text data
num_departments = 4  # Number of departments for predictions

title_input = keras.Input(shape=(None,), name='title')  # Variable-length sequence of ints
body_input = keras.Input(shape=(None,), name='body')  # Variable-length sequence of ints
tags_input = keras.Input(shape=(num_tags,), name='tags')  # Binary vectors of size `num_tags`

# Embed each word in the title into a 64-dimensional vector
title_features = layers.Embedding(num_words, 64)(title_input)
# Embed each word in the text into a 64-dimensional vector
body_features = layers.Embedding(num_words, 64)(body_input)

# Reduce sequence of embedded words in the title into a single 128-dimensional vector
title_features = layers.LSTM(128)(title_features)
# Reduce sequence of embedded words in the body into a single 32-dimensional vector
body_features = layers.LSTM(32)(body_features)

# Merge all available features into a single large vector via concatenation
x = layers.concatenate([title_features, body_features, tags_input])

# Stick a logistic regression for priority prediction on top of the features
priority_pred = layers.Dense(1, name='priority')(x)
# Stick a department classifier on top of the features
department_pred = layers.Dense(num_departments, name='department')(x)

# Instantiate an end-to-end model predicting both priority and department
model = keras.Model(inputs=[title_input, body_input, tags_input],
                    outputs=[priority_pred, department_pred])
```



- To probe further: https://www.tensorflow.org/guide/keras/functional

Custom training loops

```python
# Instantiate an optimizer.
optimizer = keras.optimizers.SGD(learning_rate=1e-3)
# Instantiate a loss function.
loss_fn = keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

```python
epochs = 2
for epoch in range(epochs):
    print("\nStart of epoch %d" % (epoch,))

    # Iterate over the batches of the dataset.
    for step, (x_batch_train, y_batch_train) in enumerate(train_dataset):

        # Open a GradientTape to record the operations run
        # during the forward pass, which enables auto-differentiation.
        with tf.GradientTape() as tape:

            # Run the forward pass of the layer.
            # The operations that the layer applies
            # to its inputs are going to be recorded
            # on the GradientTape.
            logits = model(x_batch_train, training=True)  # Logits for this minibatch

            # Compute the loss value for this minibatch.
            loss_value = loss_fn(y_batch_train, logits)

# Use the gradient tape to automatically retrieve
# the gradients of the trainable variables with respect to the loss.
grads = tape.gradient(loss_value, model.trainable_weights)

# Run one step of gradient descent by updating
# the value of the variables to minimize the loss.
optimizer.apply_gradients(zip(grads, model.trainable_weights))
```

- To probe further: TensorFlow tutorial [custom training]

Distributed computing strategies

Distributed computing strategies

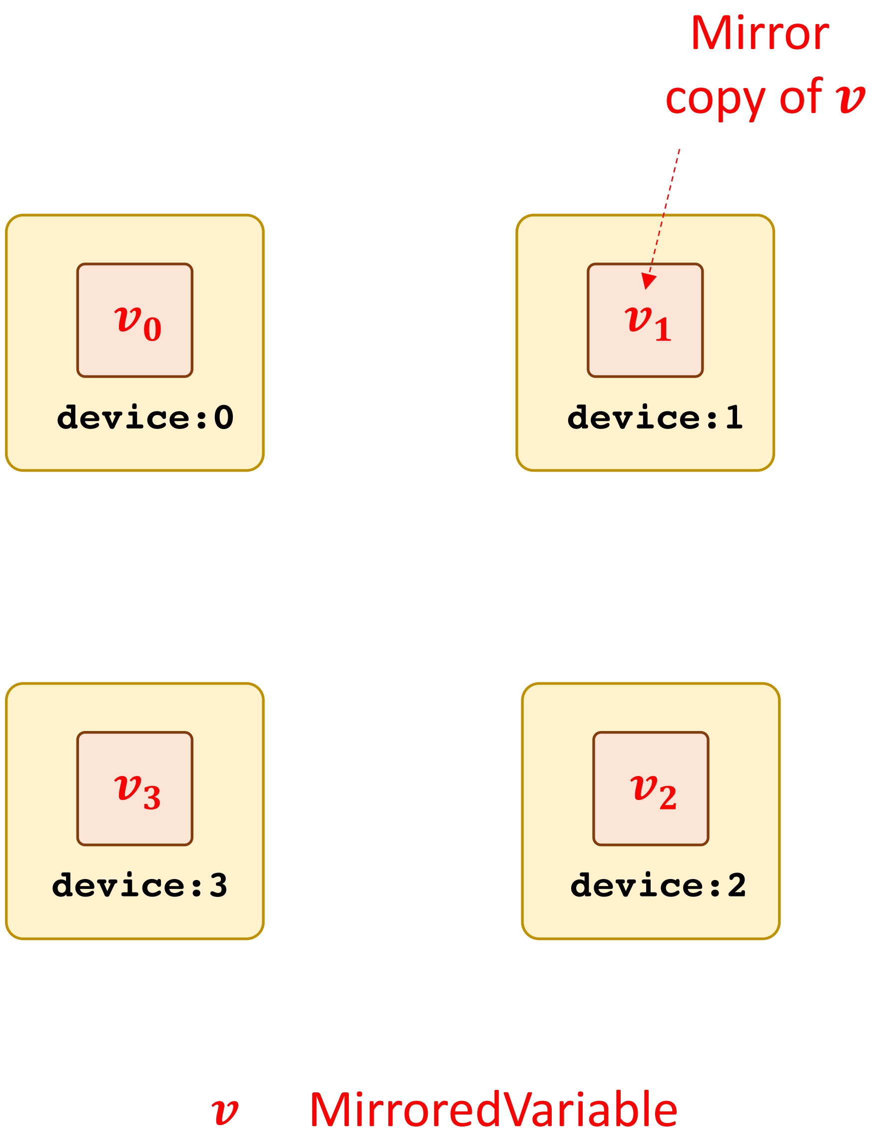
- `tf.distribute.Strategy`: API that provides an abstraction for distributing training across multiple processing units (devices like GPUs or cluster machines)

- Types of strategies:
    - MirroredStrategy
    - MultiWorkerMirroredStrategy
    - CentralStorageStrategy
    - ParameterServerStrategy
    - OneDeviceStrategy
    - TPUStrategy

- Currently, this API supports only data parallel computation model
- Current solution for model parallel computation model is Mesh-TensorFlow

Mirrored strategy

- Supports synchronous distributed training on multiple GPUs on one machine
- Creates one replica per GPU device
- Each variable in the model is mirrored across all the replicas
  - These variables form a single conceptual variable called MirroredVariable
  - Kept in sync with each other by applying identical updates
- All-reduce algorithms used to communicate variable updates across the devices
  - Aggregates tensors across all the devices by adding them up, and makes them available on each device



Code example

```
strategy = tf.distribute.MirroredStrategy()
```

```
print('Number of devices: {}'.format(strategy.num_replicas_in_sync))
```

```
INFO:tensorflow:Using MirroredStrategy with devices ('/job:localhost/replica:0/task:0/device:GPU:0',)
INFO:tensorflow:Using MirroredStrategy with devices ('/job:localhost/replica:0/task:0/device:GPU:0',)
Number of devices: 1
```

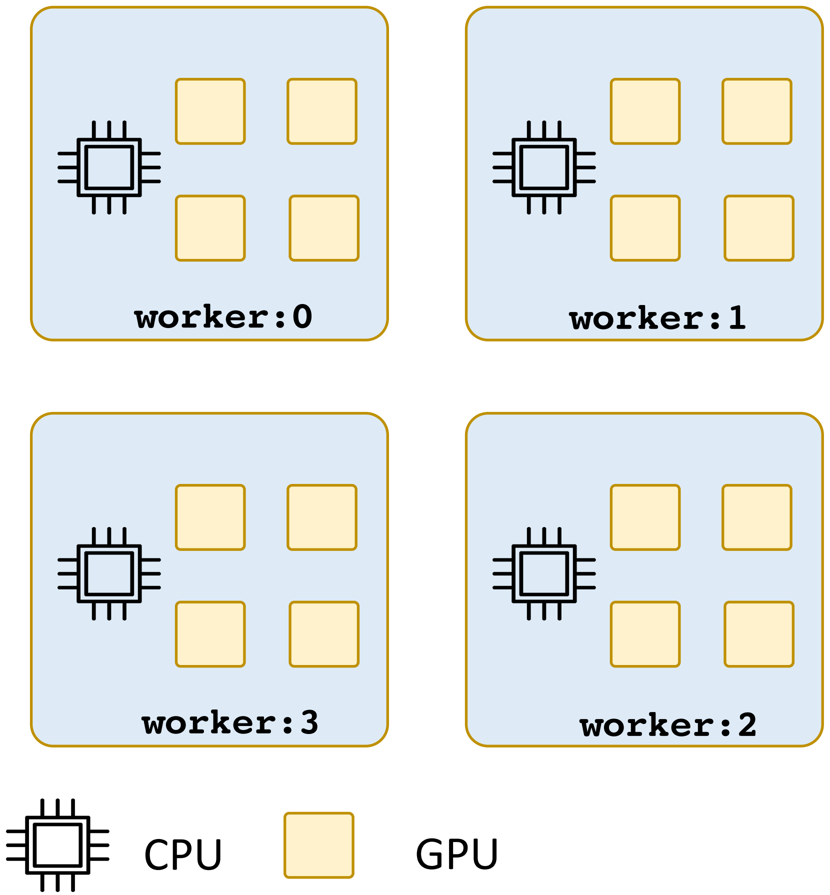
- Mirrored strategy applied with Sequential API

```
with strategy.scope():
  model = tf.keras.Sequential([
      tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
      tf.keras.layers.MaxPooling2D(),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(64, activation='relu'),
      tf.keras.layers.Dense(10)
  ])
```

Multi-worker mirrored strategy

```
multiworker_strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()
```

- Synchronous distributed training across multiple workers, each with potentially multiple devices (GPUs)
- Creates copies of all variables in the model on each device across all workers
- Uses CollectiveOps as the multi-worker all-reduce communication method to keep variables in sync
- CollectiveOps: operators implementing distributed reduction, all gather, broadcast send, and broadcast recv
- More on CollectiveOps: [collective_ops.py]()



Code example: Multi-worker training with Keras

```python
def build_and_compile_cnn_model():
  model = tf.keras.Sequential([
      tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape=(28, 28, 1)),
      tf.keras.layers.MaxPooling2D(),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(64, activation='relu'),
      tf.keras.layers.Dense(10)
  ])
  model.compile(
      loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
      optimizer=tf.keras.optimizers.SGD(learning_rate=0.001),
      metrics=['accuracy'])
  return model
```

```python
os.environ['TF_CONFIG'] = json.dumps({
    'cluster': {
        'worker': ["localhost:12345", "localhost:23456"]
    },
    'task': {'type': 'worker', 'index': 0}
})
```

```python
NUM_WORKERS = 2
# Here the batch size scales up by number of workers since
# `tf.data.Dataset.batch` expects the global batch size. Previously we used 64,
# and now this becomes 128.
GLOBAL_BATCH_SIZE = 64 * NUM_WORKERS

# Creation of dataset needs to be after MultiWorkerMirroredStrategy object
# is instantiated.
train_datasets = make_datasets_unbatched().batch(GLOBAL_BATCH_SIZE)
with strategy.scope():
  # Model building/compiling need to be within `strategy.scope()`.
  multi_worker_model = build_and_compile_cnn_model()

# Keras' `model.fit()` trains the model with specified number of epochs and
# number of steps per epoch. Note that the numbers here are for demonstration
# purposes only and may not sufficiently produce a model with good quality.
multi_worker_model.fit(x=train_datasets, epochs=3, steps_per_epoch=5)
```

- To probe further: TensorFlow tutorial [multi-worker with Keras]

Parameter server strategy

```
ps_strategy = tf.distribute.experimental.ParameterServerStrategy()
```

- Supports parameter server training on multiple machines
  - Some machines are designated as workers and some as parameter servers
  - Each variable of the model is placed on one parameter server
  - Computation is replicated across all devices (GPUs) of the workers
  - Between-graph replication
- For parameter server training, TF_CONFIG needs to specify the configuration of parameter servers and workers in the cluster

Example TF_CONFIG:

```
os.environ["TF_CONFIG"] = json.dumps({
    "cluster": {
        "worker": ["host1:port", "host2:port", "host3:port"],
        "ps": ["host4:port", "host5:port"]
    },
   "task": {"type": "worker", "index": 1}
})
```

- To probe more: https://www.tensorflow.org/guide/distributed_training#TF_CONFIG

Central storage strategy

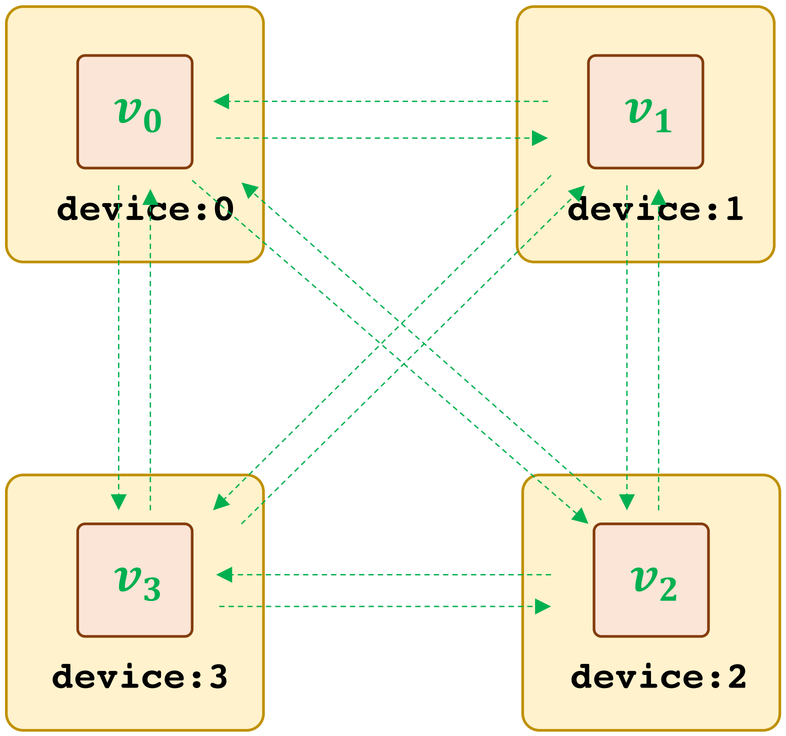
```
central_storage_strategy = tf.distribute.experimental.CentralStorageStrategy()
```

- Synchronous training
- Variables are not mirrored
- Variables are placed on the CPU and operations are replicated across all local GPUs
- In-graph replication

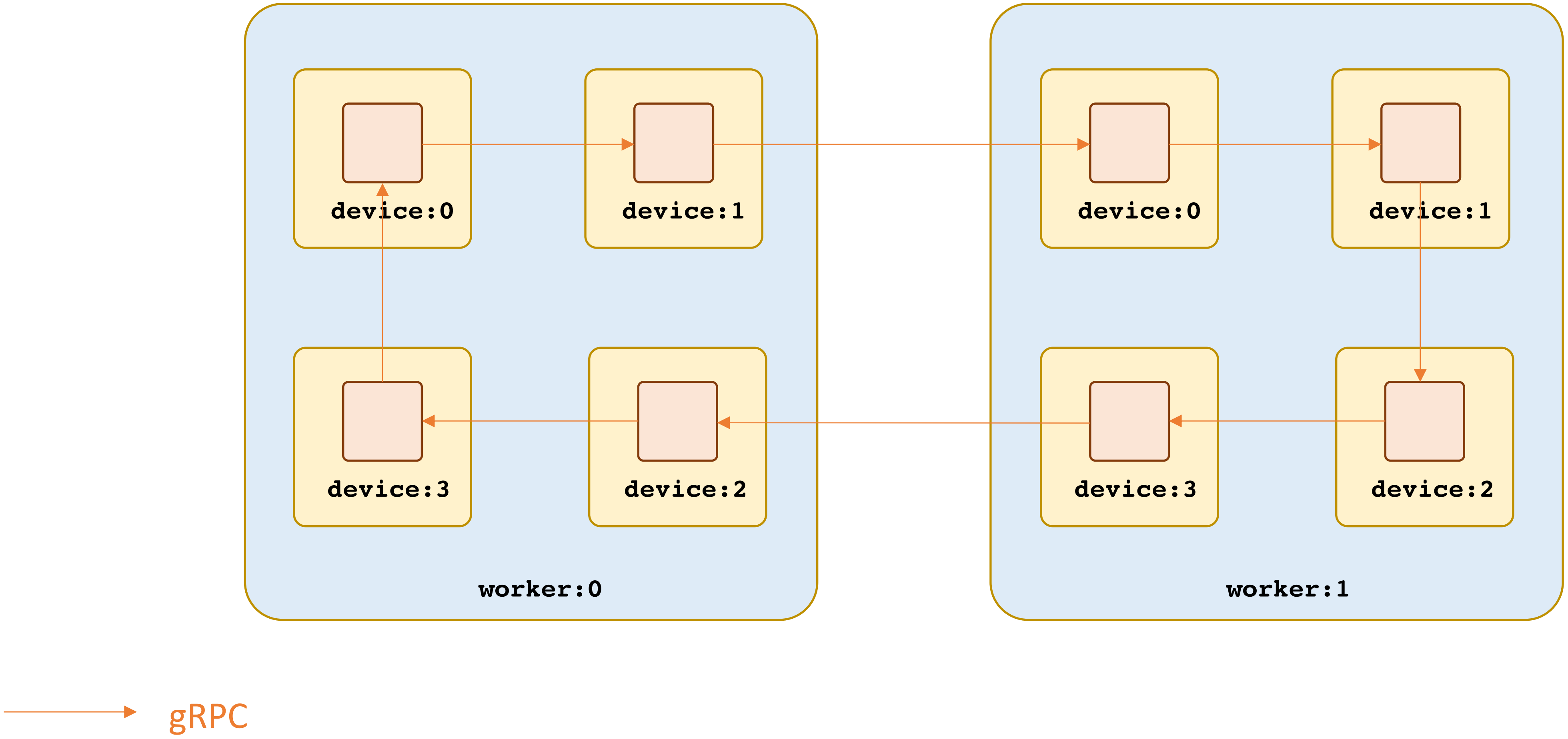
All-reduce

- All-reduce algorithm
  - All: from every device to every device
  - Reduce: sum or min (or max, min)
- Network efficient
- Requires synchronization between devices
- All-reduce algorithms for distributed TensorFlow implemented in [cross_device_ops.py]()

```python
class MultiWorkerAllReduce(AllReduceCrossDeviceOps):
  """All-reduce algorithms for distributed TensorFlow."""

  def __init__(self,
               worker_devices,
               num_gpus_per_worker,
               all_reduce_spec=("pscpu/pscpu", 2, -1),
               num_packs=0,
               agg_small_grads_max_bytes=0,
               agg_small_grads_max_group=10):
    """Initialize the all-reduce algorithm.

```

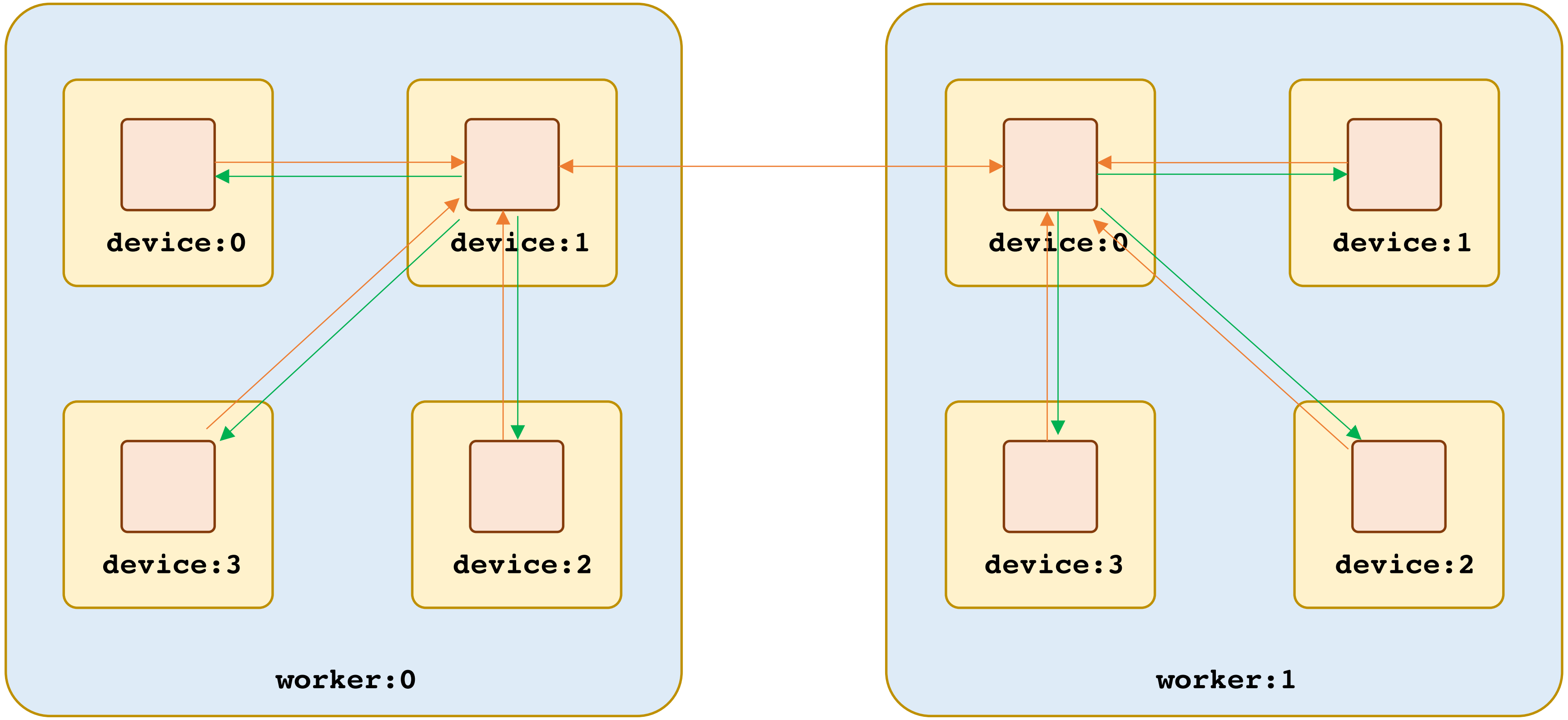


Ring all-reduce



- gRPC: a high-performance, open source universal RPC framework https://grpc.io/
- RPC: remote procedure call [wikipedia]()

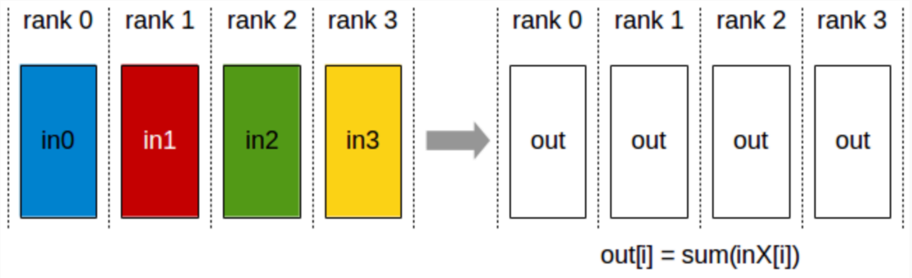
Hierarchical all-reduce



NVIDA NCCL (“nickel”)

- NVIDIA NCCL: NVIDIA Collective Communications Library implements multi-GPU and multi-node collective communication primitives optimized for NVIDIA GPUs
- Provides routines such as all-gather, all-reduce, broadcast, reduce, reduce-scatter optimized to achieve high bandwidth over PCIs and NVLink high-speed interconnect

AllReduce starts with independent arrays Vk of N values on each of K ranks and ends with identical arrays S of N values, where S[i] = V0[i]+V1[i]+...+Vk-1[i], for each rank k.



All-Reduce operation: each rank receives the reduction of input values across ranks.

- To probe further: https://developer.nvidia.com/nccl
- PCI: Peripheral Component Interconnect (a local computer bus for attaching hardware devices in computer)
- NVLink high-speed interconnect: developed by NVIDIA, uses mesh network to communicate instead of a central hub

References

- Dean et al, Large-Scale Distributed Deep Networks, NIPS 2012
- Li et al, Scaling Distributed Machine Learning with the Parameter Server, OSDI 2014
- Abadi et al, TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, whitepaper 2015
- Abadi et al, TensorFlow: A system for large-scale machine learning, OSDI 2016
- Dean, Large-Scale Deep Learning with Tensorflow, ScaledML 2016
- Dean, Machine Learning for Systems and Systems for Machine Learning, NIPS 2017

References (cont'd)

TensorFlow

- tensorflow.org: Tutorials, Programmer's Guide, API docs
- Gordon, Introduction to TensorFlow 2.0: Easier for beginners and more powerful for experts, TF World 2019

Distributed TensorFlow

- TensorFlow guide, distributed training with TensorFlow
- TensorFlow example, Distributed TensorFlow
- Levenberg, Inside TensorFlow: tf.distribute.Strategy, 2019
- Murray, Distributed TensorFlow, TensorFlow Dev Summit 2017
- Dowling, Distributed TensorFlow, O'Reilly Ideas, 2017

References (cont'd)

Mesh-TensorFlow

- Shazeer et al, Mesh-TensorFlow: Deep Learning for Supercomputers, NIPS 2018
- GitHub repo Mesh TensorFlow: Model Parallelism Made Easier

Other frameworks for neural networks

- Chainer
- PyTorch
- Microsoft Cognitive Tookit (CNTK)

Seminar 10

- Distributed training with TensorFlow
  - Sequential API
  - Functional API
  - Custom training loops
  - MirroredStrategy
  - MultiWorkerMirroredStrategy

https://github.com/lse-st446/lectures2021/blob/master/Week11/class/README.md