TensorFlow :
A system for large-scale machine learning

kiho.hong (swear013@gmail.com)

TensorFlow: A system for large-scale machine learning

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean,
Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur,
Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker,
Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng

Google Brain

TensorFlow:
Large-Scale Machine Learning on Heterogeneous Distributed Systems
(Preliminary White Paper, November 9, 2015)

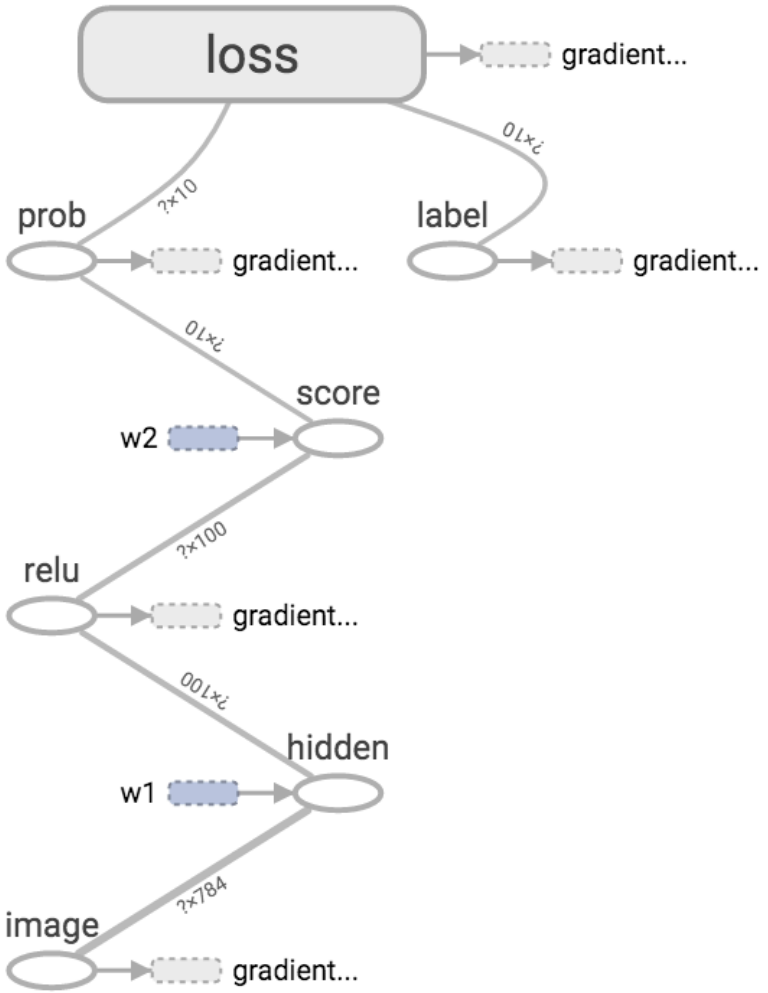
Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro,
Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow,
Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser,
Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray,
Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar,
Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals,
Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng
Google Research*

TensorFlow

- Operates at large scale & in heterogeneous system
- Dataflow model
- CPU, GPU, TPU(Tensor Processing Unit) supports
- Particularly strong support for deep neural net.

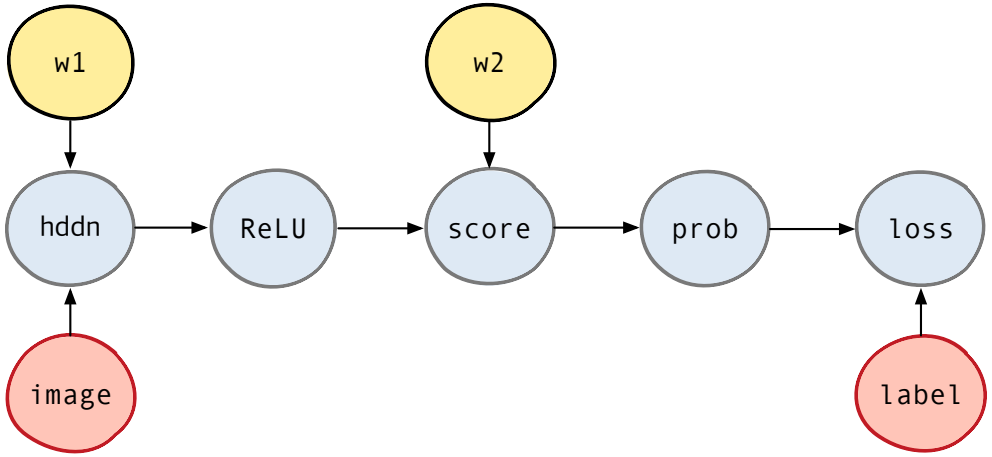
TensorFlow Impl.

```
1   mnist = data.read_data_sets('./data', one_hot=True)
2
3   image = tf.placeholder(tf.float32, [None, 28 * 28], name="image")
4   label = tf.placeholder(tf.float32, [None, 10], name="label")
5
6   w1 = tf.Variable(1e-3 * rd.randn(28 * 28, 100).astype(np.float32), name="w1")
7   w2 = tf.Variable(1e-3 * rd.randn(100, 10).astype(np.float32), name="w2")
8
9   hddn = tf.matmul(image, w1, name="hidden")
10  relu = tf.nn.relu(hddn, name="relu")
11
12  score = tf.matmul(relu, w2, name="score")
13  prob  = tf.nn.softmax(score, name="prob")
14
15  with tf.variable_scope("loss"):
16      loss  = -tf.reduce_sum(label * tf.log(prob), name="loss")
17
18  step = tf.train.GradientDescentOptimizer(1e-2, name="step").minimize(loss)
19
20  with tf.Session(target=str.encode('')) as sess:
21      init = tf.initialize_all_variables()
22      sess.run(init)
23
24      for i in range(200):
25          xs, ys= mnist.train.next_batch(100)
26          _, val = sess.run([step, loss], feed_dict = {image: xs, label: ys})
```

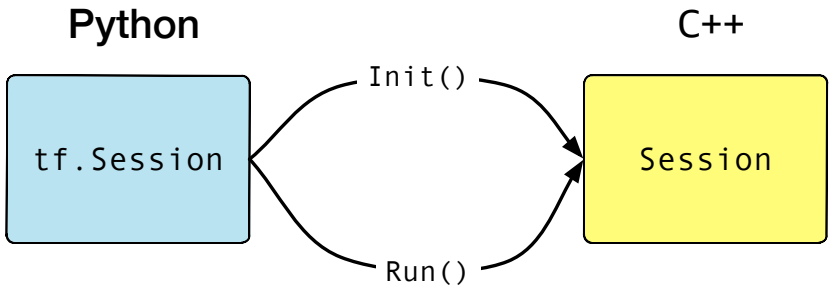


Python API

- Graph



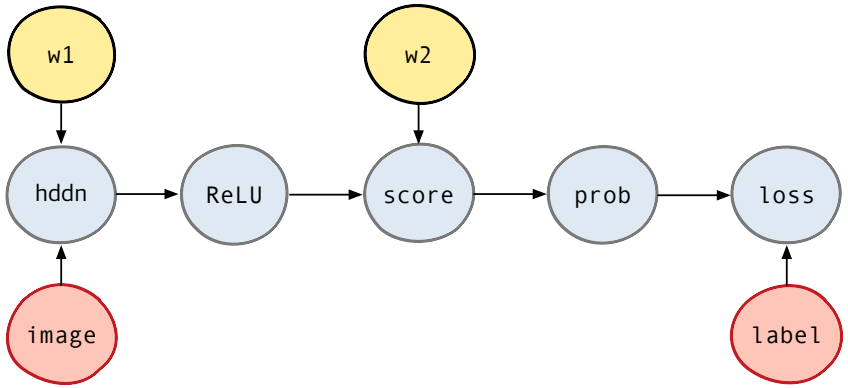
- Session



그래프 작성

- Python 구문을 이용하여 연산에 사용할 그래프를 구성.

```
image = tf.placeholder(tf.float32, [None, 28 * 28], name="image")
label = tf.placeholder(tf.float32, [None, 10], name="label")

w1 = tf.Variable(1e-3 * rd.randn(28 * 28, 100).astype(np.float32), name="w1")
w2 = tf.Variable(1e-3 * rd.randn(100, 10).astype(np.float32), name="w2")

hddn = tf.matmul(image, w1, name="hidden")
relu = tf.nn.relu(hddn, name="relu")

score = tf.matmul(relu, w2, name="score")
prob  = tf.nn.softmax(score, name="prob")
```



- 함수가 호출되는 즉시 내부 연산이 수행되는 것은 아니다.
  - 오로지 연산 그래프가 작성되는 과정만을 수행.

Session

- Session은 TensorFlow 그래프 연산을 수행하기 위한 사용자 인터페이스
  - Graph를 생성하고 Operation을 수행하며 Tensor를 평가.

```
with tf.Session(target=str.encode('')) as sess:
    init = tf.initialize_all_variables()
    sess.run(init)

    for i in range(200):
        xs, ys= mnist.train.next_batch(100)
        _, val = sess.run([step, loss], feed_dict = {image: xs, label: ys})
```

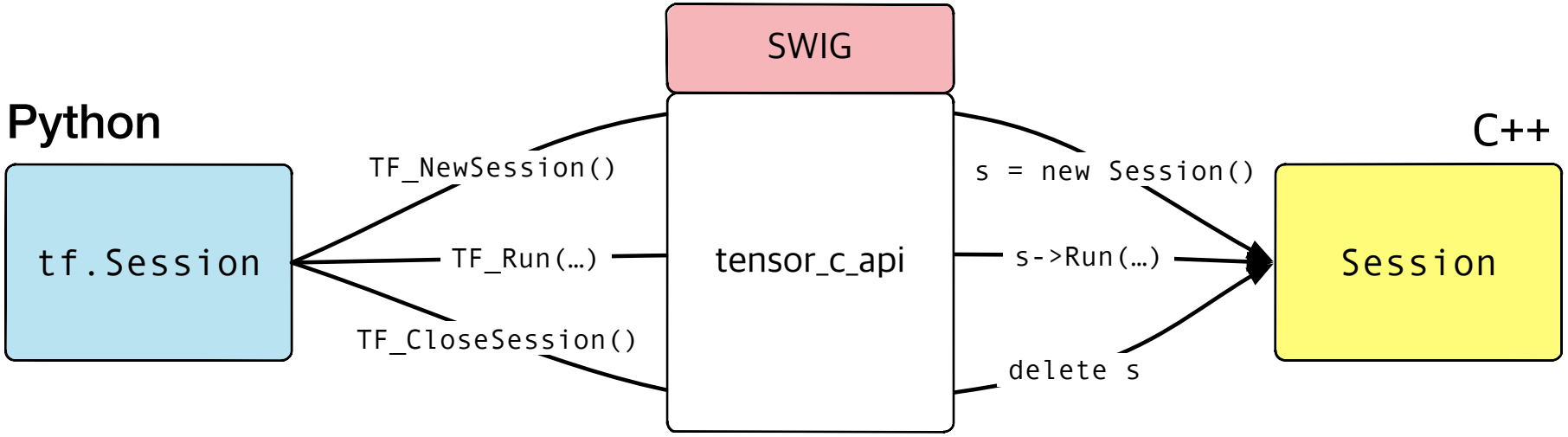
- tf.Session 은 반드시 한 개의 tf.Graph 객체를 포함.

```
g = tf.Graph()
```

- 특정 tf.Graph 를 명시하지 않은 경우 default graph 가 사용됨.

tf.Session

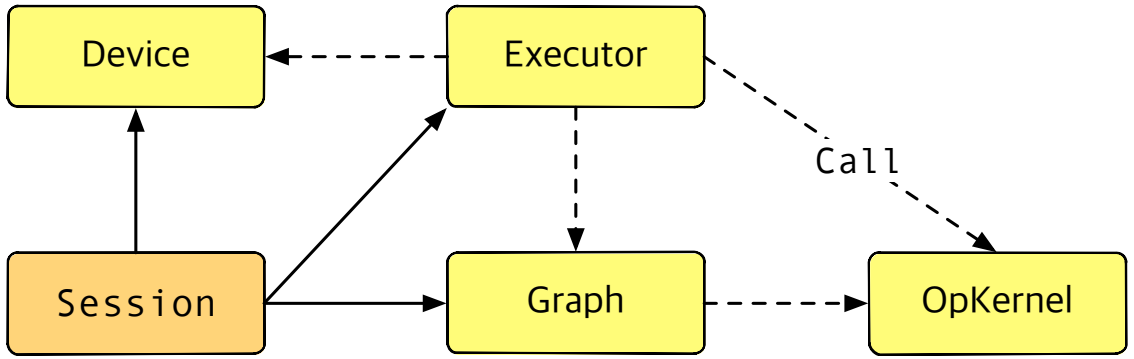
http://www.swig.org/




```
sess.run([step, loss], feed_dict = {image: xs, label: ys})
```

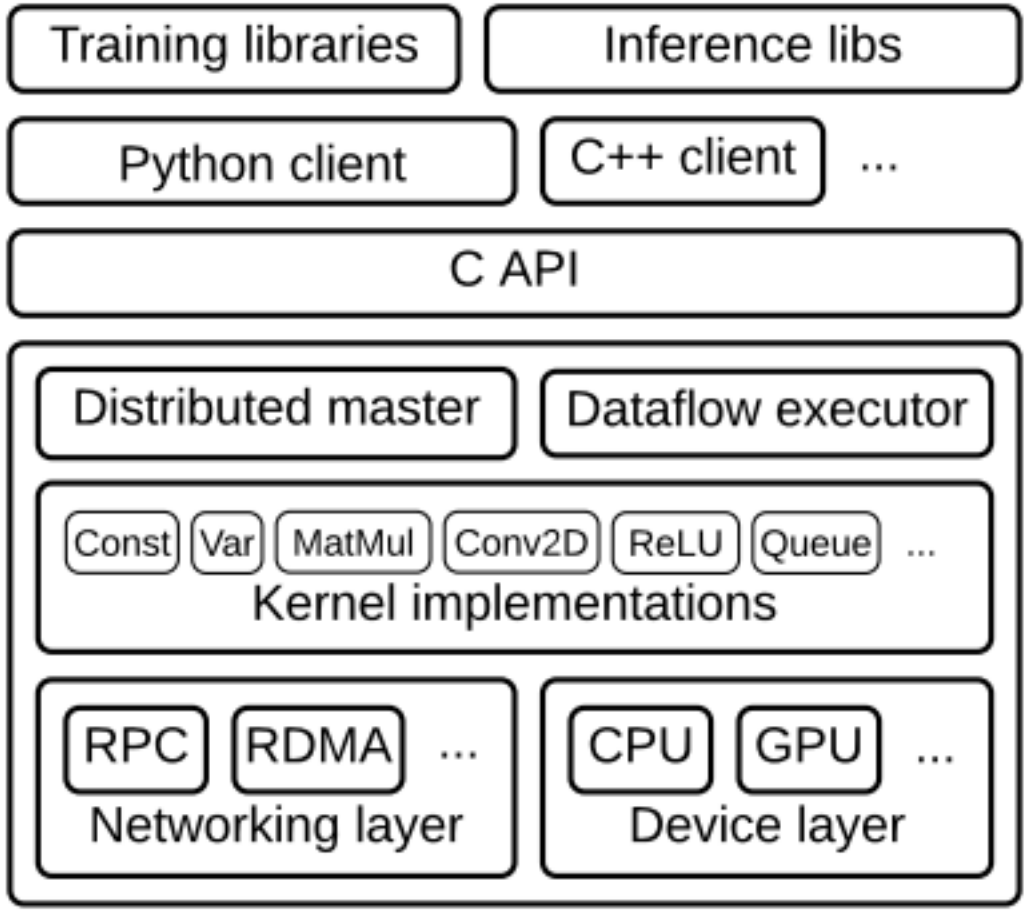
fetch feed

Overview (in C++)

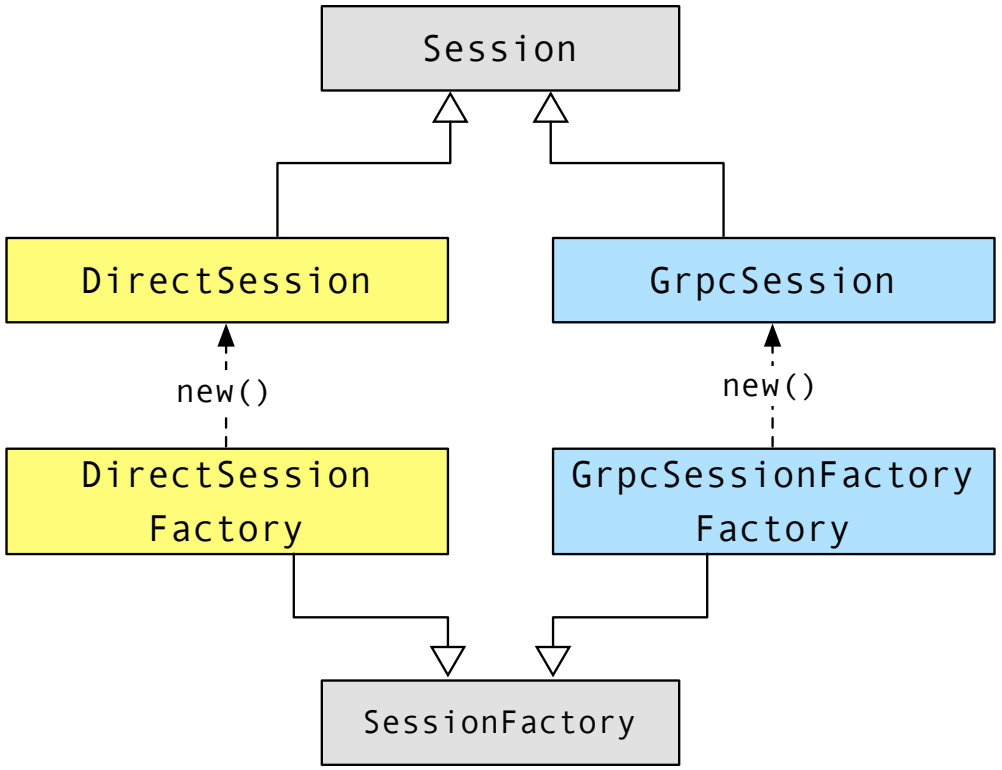


Session	그래프 연산 작업 관리
Device	디바이스 정보를 담고 있는 객체
Graph	연산(op)과 값(tensor)을 그래프 구조로 저장
Executor	그래프 연산을 실제로 수행
OpKernel	그래프 노드에 지정된 연산 (Functor)

Architecture

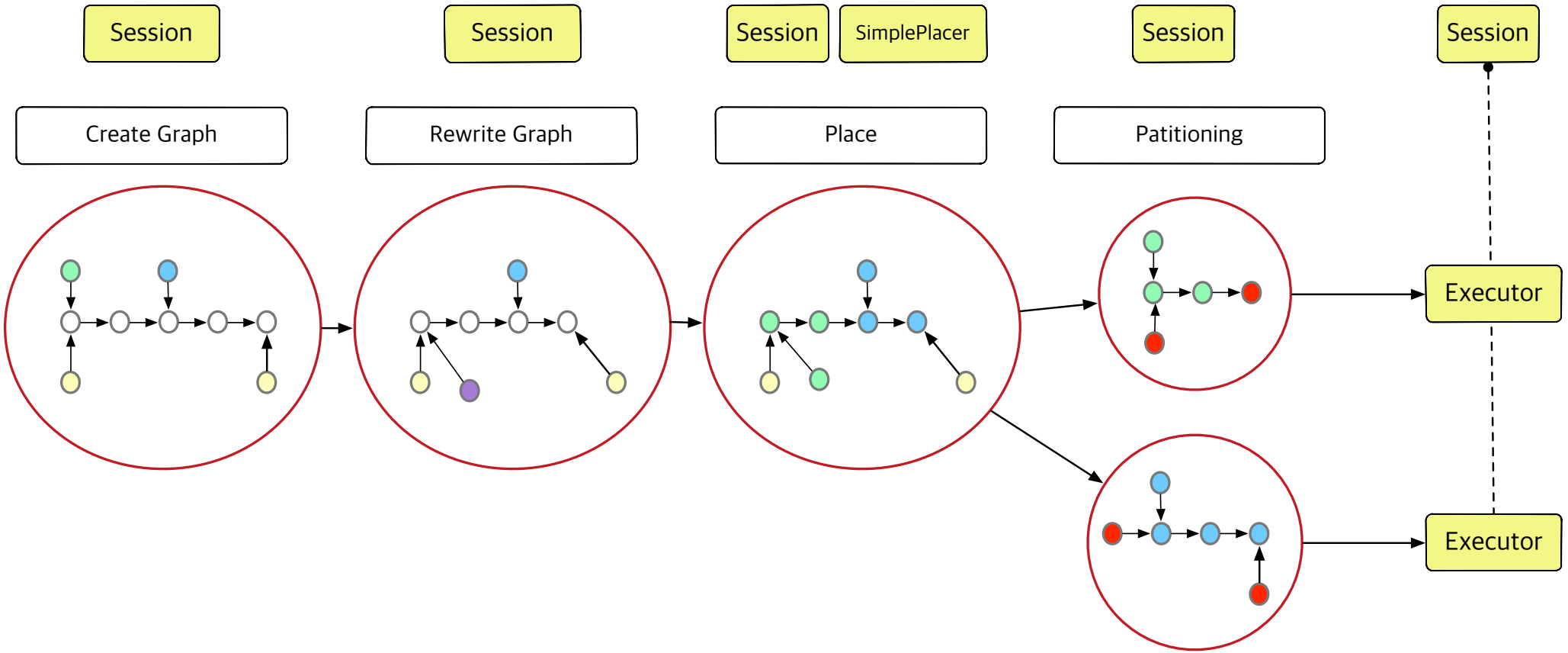


Session



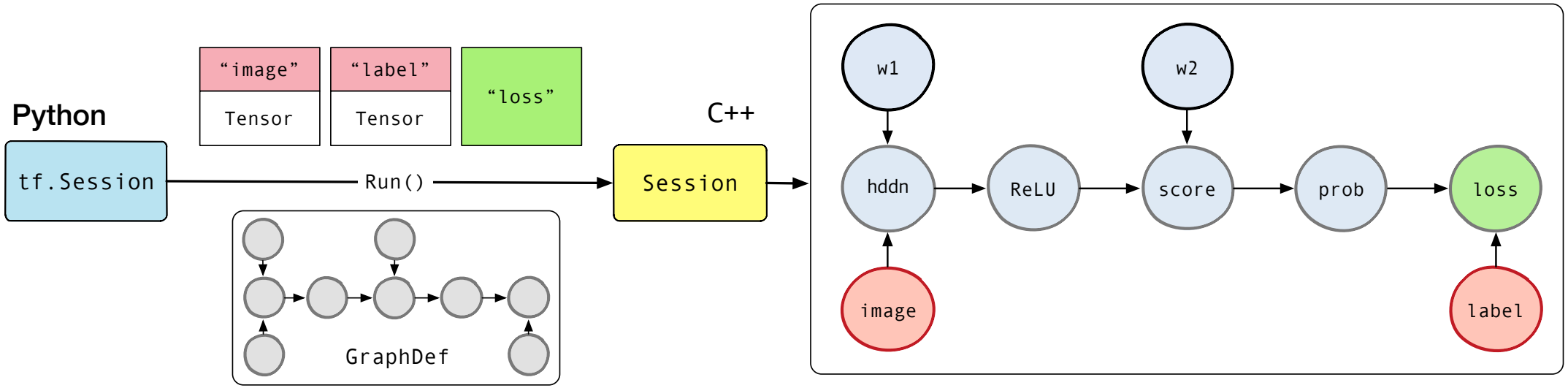
- 사용자가 요청한그래프 연산을 실제로 수행하는 객체.
- 현재 2가지 Session 제공.
  - 로컬 환경 : DirectSession
  - 분산 환경 : GrpcSession

그래프 변환


Session
Session
Session
SimplePlacer
Session
Session
Create Graph
Rewrite Graph
Place
Patitioning
Executor
Executor

Session : 그래프 생성

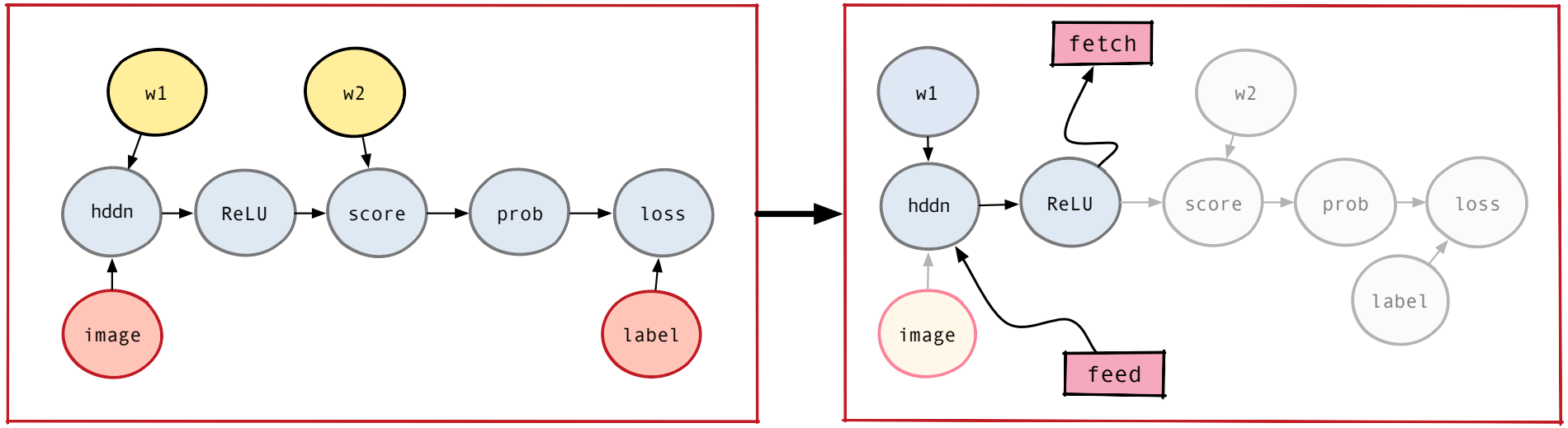
- Session::Run() 호출시 가장 먼저 GraphDef 를 Graph 객체로 변환.



Rewrite Graph

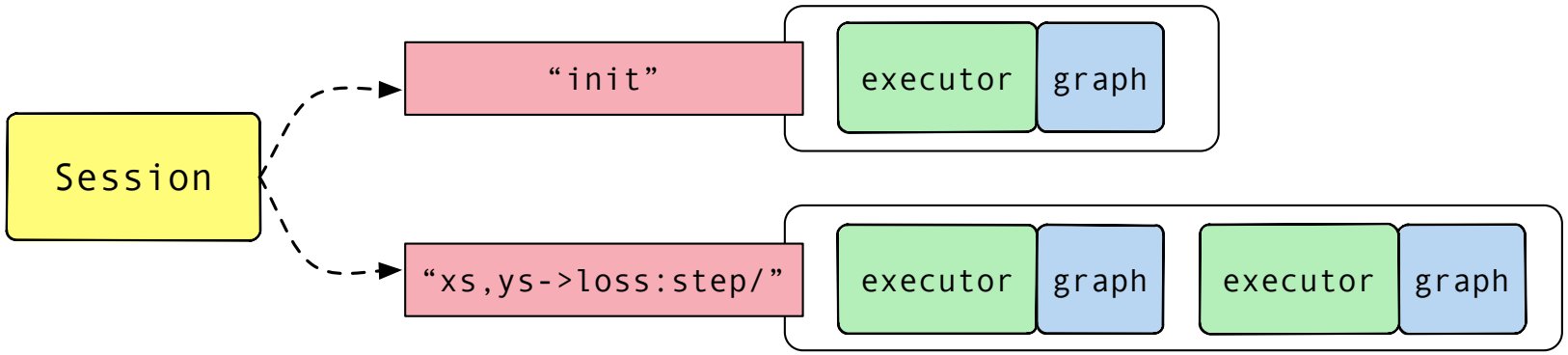
```
sess.Run(["relu"], {image:xs})
```

fetch feed



Rewrite Graph (cont'd)

```
with tf.Session(target=str.encode('')) as sess:
    init = tf.initialize_all_variables()
    sess.run(init)

    for i in range(200):
        xs, ys= mnist.train.next_batch(100)
        _, val = sess.run([step, loss], feed_dict = {image: xs, label: ys})
```



Node에 Device 설정

- TensorFlow with GPU?
  - 일반적인 딥러닝 라이브러리에서는 **Process** 단위 CPU/GPU 모드를 제공.
  - 하지만 TensorFlow 에서는 **Node** 단위 Device를 선택 가능.

- Node 단위 실행을 위해 Graph 객체는 **분할**될 수 있다.

Node에 device 설정하기 (Cont'd)

```
with tf.device('/gpu:0'):
    w1 = tf.Variable(1e-3 * rd.randn(28 * 28, 100).astype(np.float32), name="w1")

with tf.device('/gpu:1'):
    w2 = tf.Variable(1e-3 * rd.randn(100, 10).astype(np.float32), name="w2")
```

- 사용자가 명시적으로 코드 상에서 device를 지정.
  - 이런 구문을 사용하면 해당 Node에 device 정보가 기록됨.
  - Node에 device 정보가 없는 경우 TensorFlow가 알아서 device를 지정. (Placer)

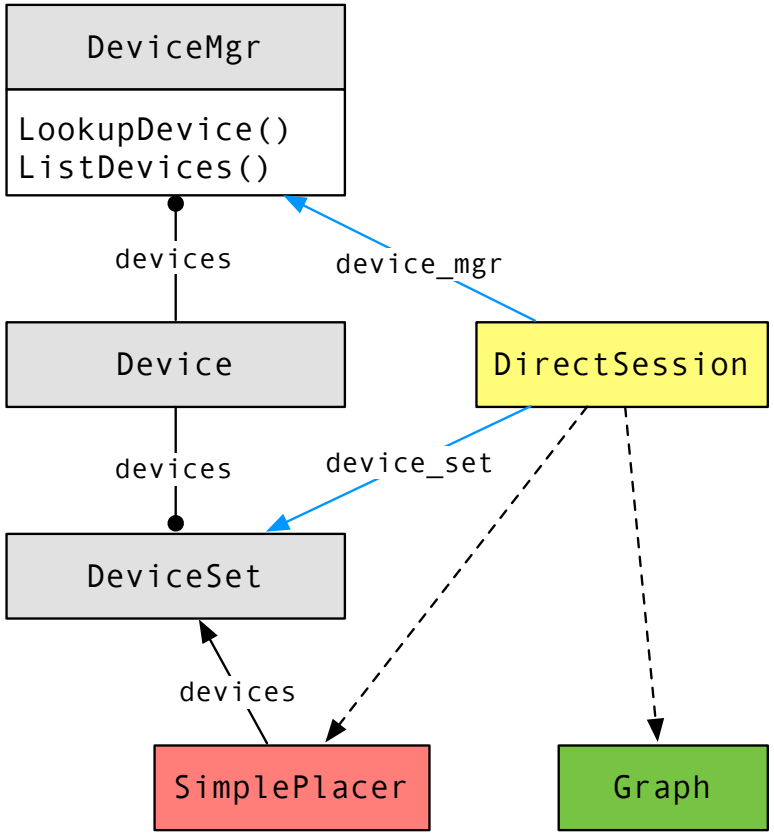
Device 할당 제약

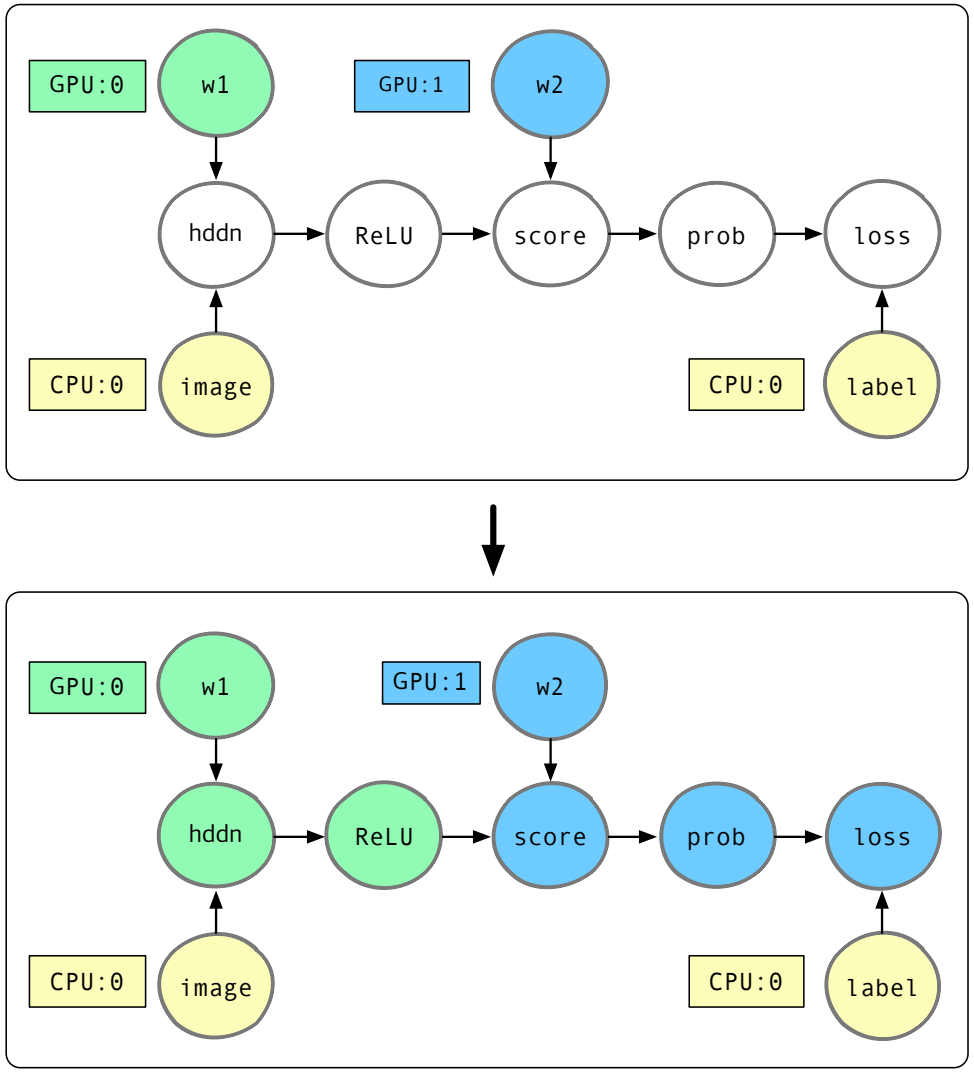
1. 사용자가 코드 상에 명시적으로 device 를 기술한 경우
   - 반드시 해당 device 로 노드를 할당.

2. 어떤 노드가 다른 노드의 참조 타입으로 생성된 경우
   - 이 경우 두 노드는 동일한 device 로 할당.

3. A와 B노드가 주어지고 B노드에 **@A**와 같은 colocation이 사용된 경우
   - 이 경우도 A와 B 노드는 동일한 device 로 할당.

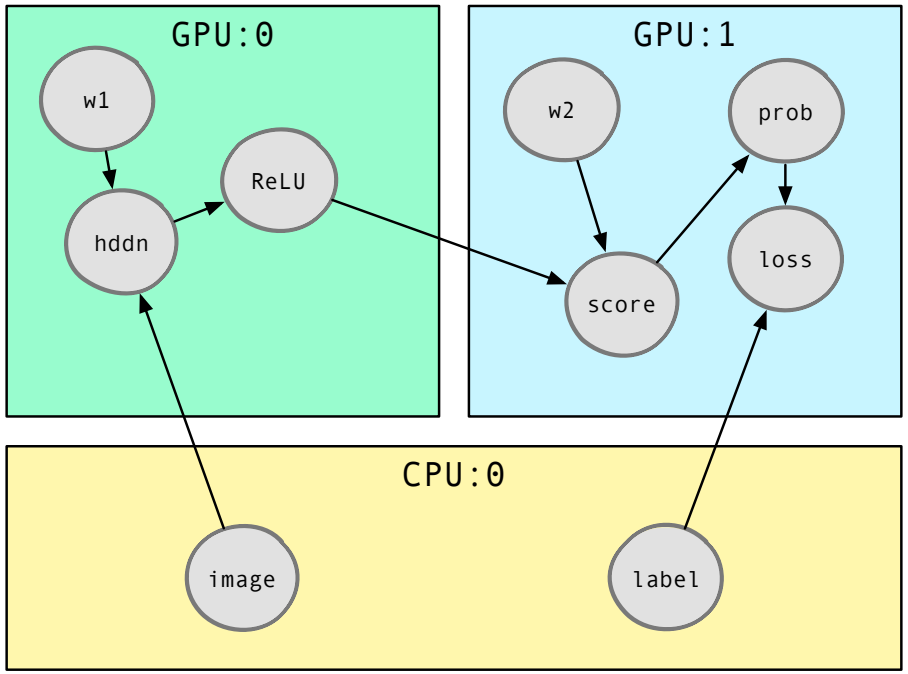
SimplePlacer





SimplePlacer (Cont'd)

- TensorFow White Paper 에 따르면,
- 할당시 Cost 기반 할당을 수행한다고 되어있다.

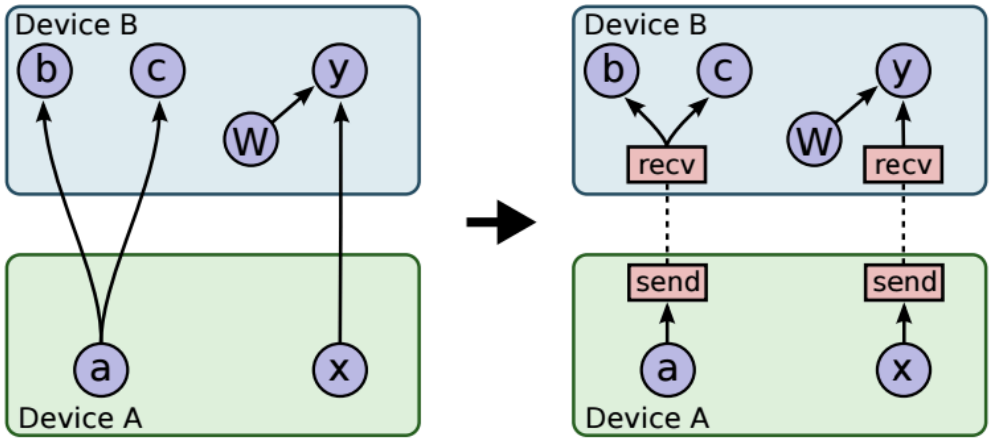


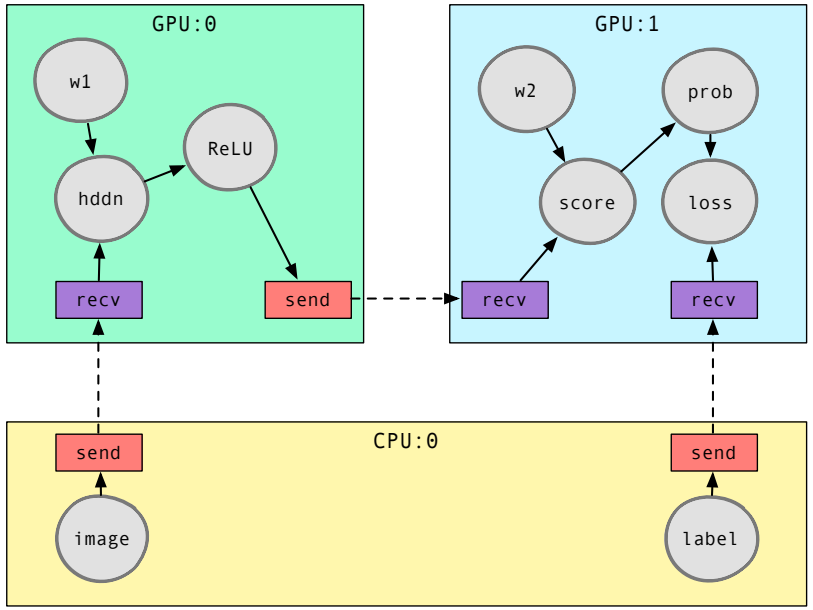
- 하지만 현실은 ?
  - 일단 device를 할당할 노드는 부모 노드의 device 정보를 얻어 자신의 device 정보로 사용.
  - 이 때 부모 노드의 device가 지정되어 있지 않은 경우, 사전 고려된 순서에 따라 할당.
  - 따라서 GPU 장비가 설치되어 있는 장비의 경우 대개 GPU:0 이 할당.

Graph Partitioning

- Device 단위로 파티셔닝을 수행.

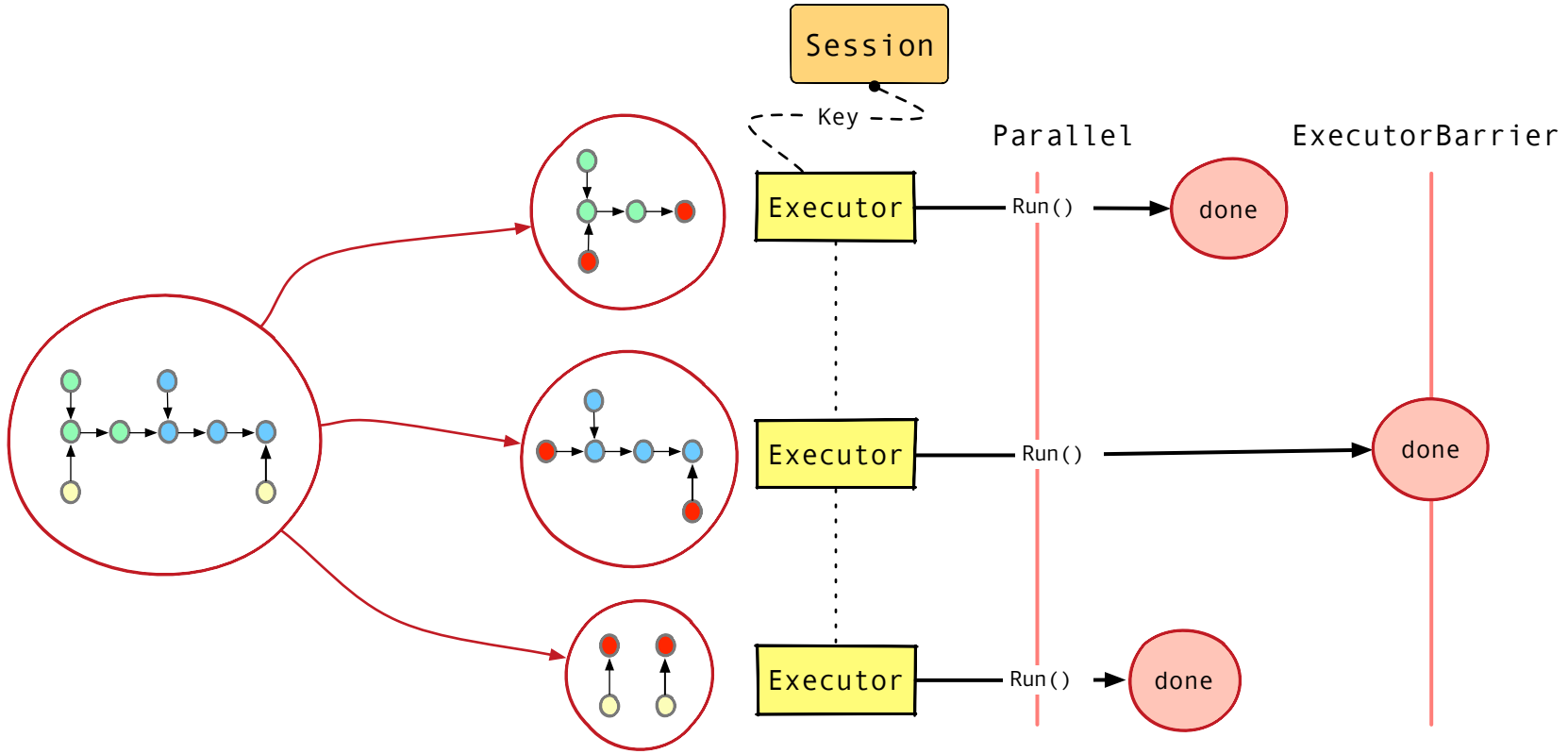
- 그래프 분할을 완료하기 위해서는 추가 작업이 필요하다.
  - 접점이 되는 노드에 recv, send 노드를 추가하여 그래프 분할을 완성.
  - 이 때 서브그래프에서 recv가 여러 노드에서 사용된다면 하나로 치환.





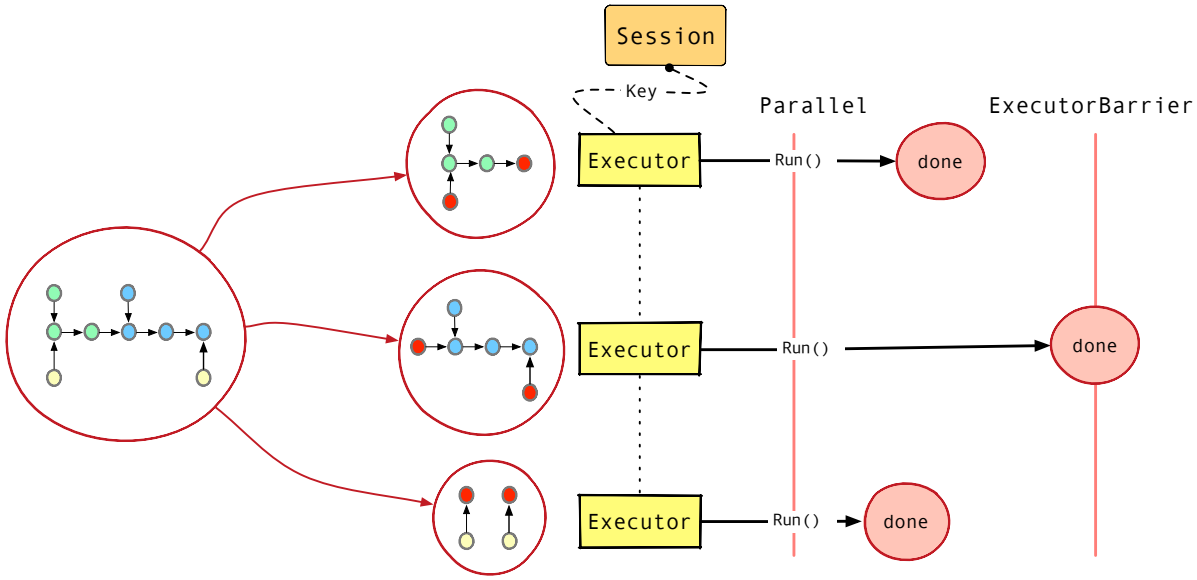
Executor

- 분할된 그래프들을 실행하는 주체는 Executor 객체.

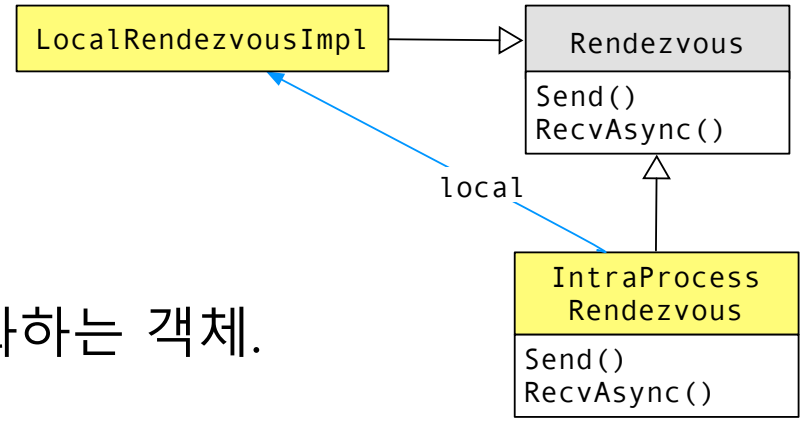


Executor

- Executor는 서브-그래프 하나마다 생성.
- Executor는 개별적인 Thread 작업으로 구성.
- Executor는 서브-그래프의 시작 노드를 찾아 그래프 연산을 수행.
- 모든 Executor의 작업이 끝날 때까지 ExecutorBarrier가 작업을 제어.
- Executor가 하는 일은 좀 더 복잡하지만 넘어가자.

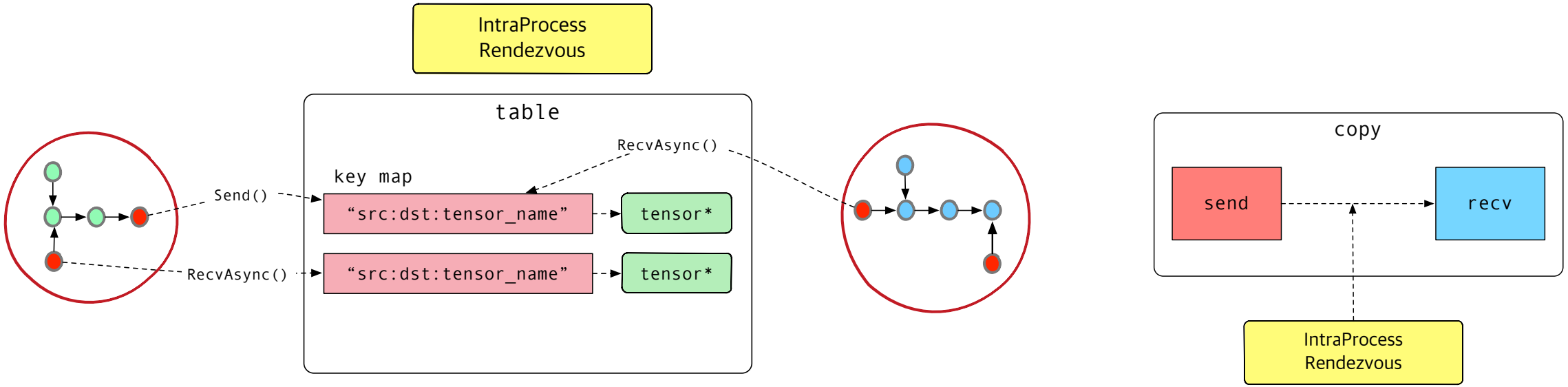


랑데뷰 (Rendezvous)



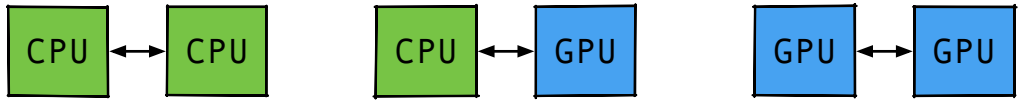
- 서브-그래프 사이의 send와 recv 노드 데이터를 동기화하는 객체.
- 각 Executor 들은 자신의 서브 그래프 내의 send와 recv 객체를 Rendezvous 객체에 등록.
  - Send()
    - send 노드에 속한 Tensor 데이터를 다른 그래프로 보내줄 때 사용.
    - Send() 호출시 이미 다른 Executor 가 해당 노드의 데이터를 요구한 사실이 있다면 바로 데이터를 전달.
  - RecvAsync()
    - recv 노드에 속한 Tensor 데이터를 다른 그래프로부터 얻어올 때 사용.
    - RecvAsync() 호출시 이미 다른 Executor 가 해당 노드를 등록해 놓았다면 바로 데이터를 수신.

Rendezvous Table



Device 사이의 데이터 전달

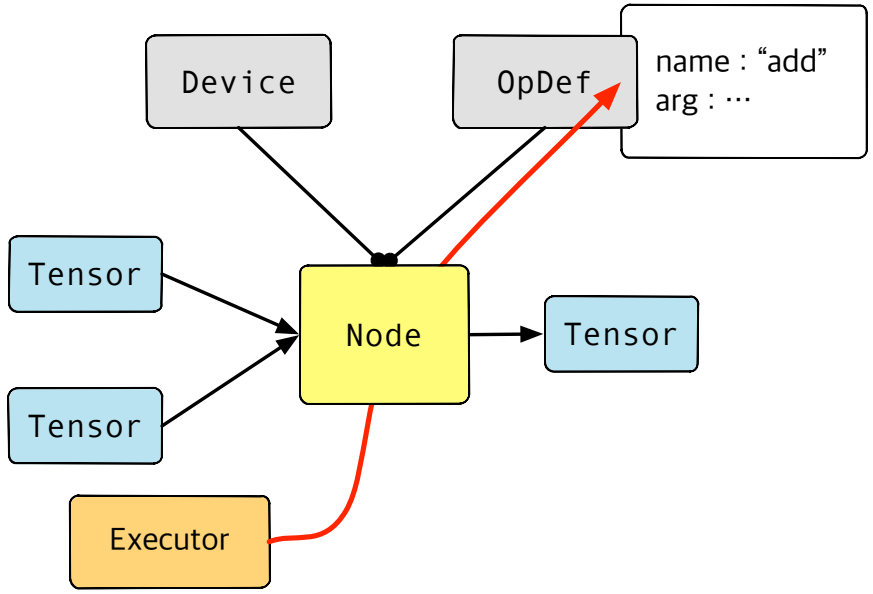
- 앞서 살펴보았지만 각각의 그래프는 device 단위로 분할.
- 그래프가 서로 주고 받는 데이터는 send와 recv 노드에 속한 Tensor 자료구조.
  - 여기서 Tensor란 차원(dimension) 정보를 가진 메모리 데이터 블록을 의미.
- 서로 다른 device 사이의 데이터 복사는 어떻게 ?

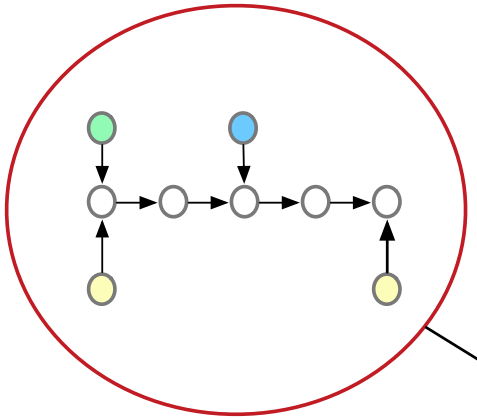


- Device 라는 객체에는 자신의 Device 타입과 메모리 할당용 Allocator를 포함하고 있음.
- InterProcessRendezvous 객체에서 Device 타입을 조회하여 적절한 Copy 함수를 호출.

서브 그래프의 실행

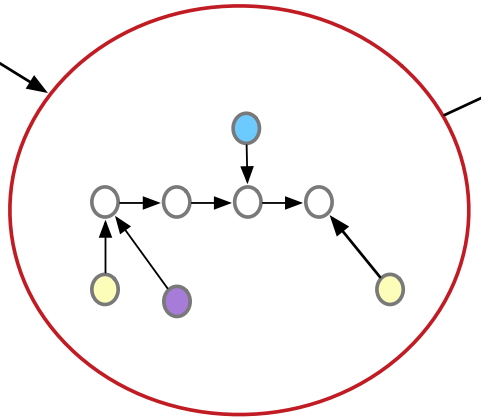
- Executor가 Graph를 실행한다고 했는데 도대체 뭘 실행하는 것일까?
  - 하나의 노드는, 하나의 Operation을 의미.
  - 그리고 이 Operation에는 적절한 연산들이 미리 정의.
  - 입출력 데이터는 Operation에 설정된 Tensor 를 사용.
  - 동일한 연산이라도 Device 타입에 따라 구현이 다름.
    - 예로 CPU 와 GPU 환경에서 add 연산의 구현이 다르다.

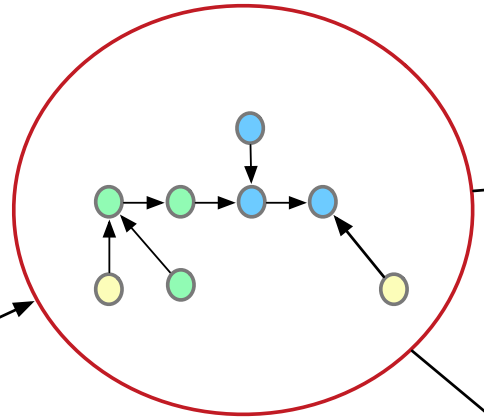




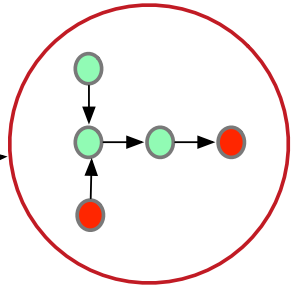
그래프를 만들고,

정제한뒤,

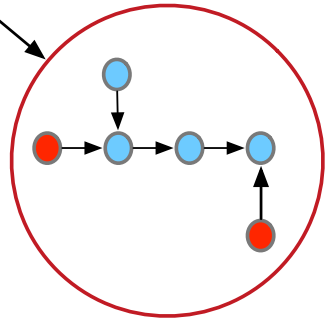




디바이스에 할당하고,



나누어



실행

Data Flow

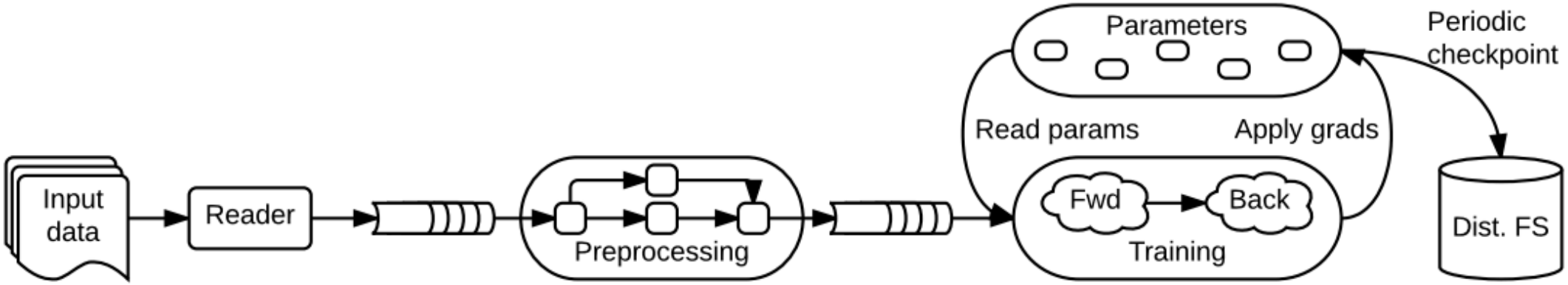
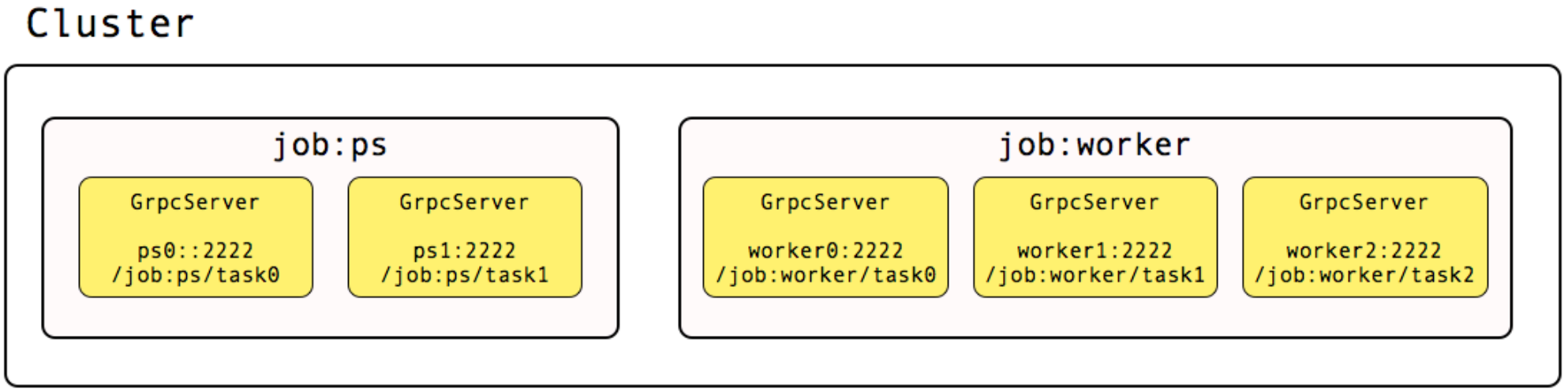


Figure 1: A schematic TensorFlow dataflow graph for a training pipeline contains subgraphs for reading input data, preprocessing, training, and checkpointing state.

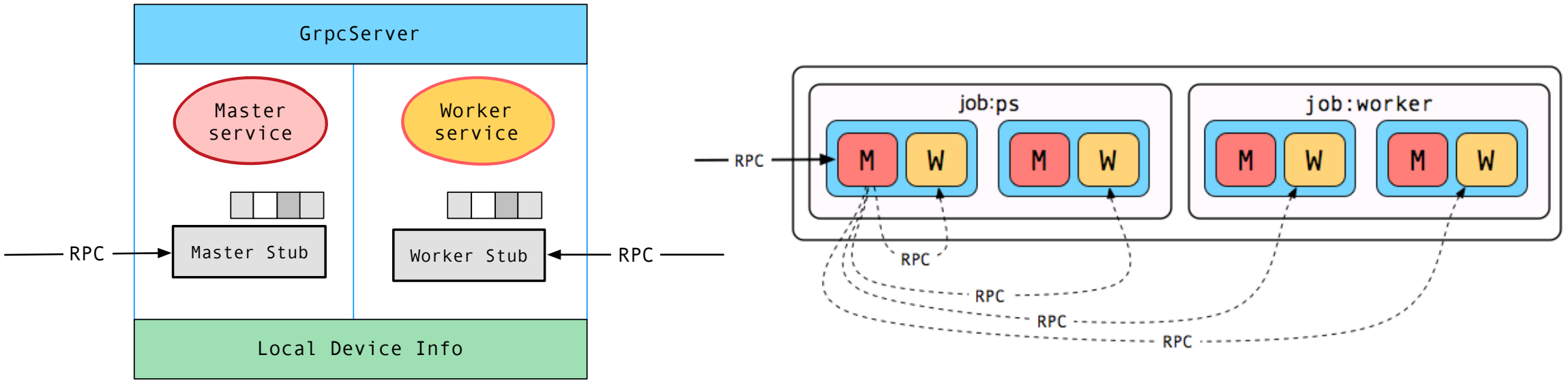
Distributed (a.k.a Cluster) Concept

```
1   tf.make_cluster_def({
2     'worker': ['worker0:2222', 'worker1:2222', 'worker2:2222'],
3     'ps': ['ps0:2222', 'ps1:2222']
4   })
```

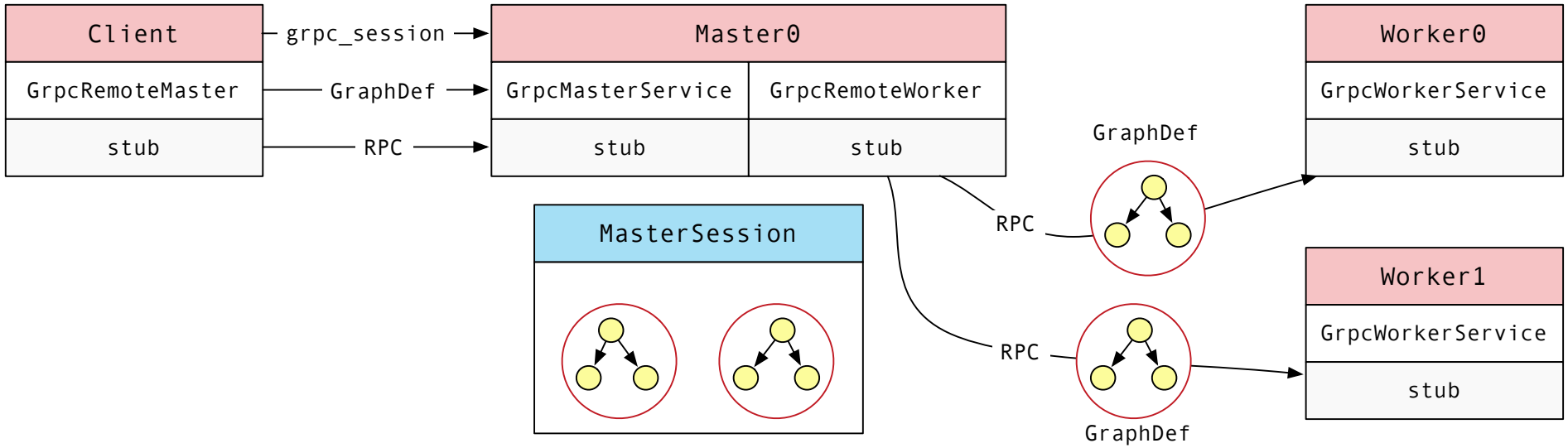


GrpcServer

- GrpcServer는 그래프 연산 Process 시작 전에 이미 각 장비에서 구동되어 있어야 한다.
  - Binary 명령어로 실행할수도 있고, python 코드로도 구동 가능.

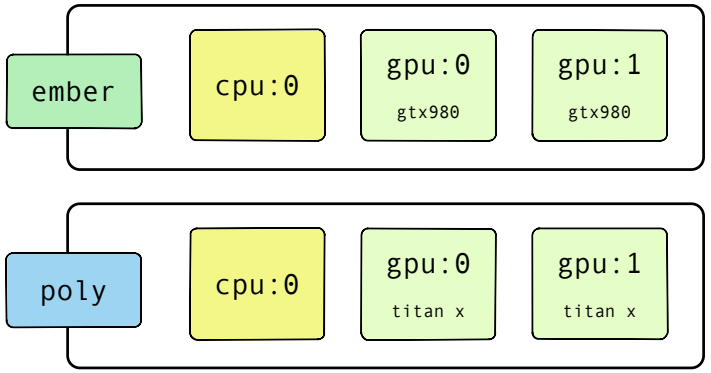


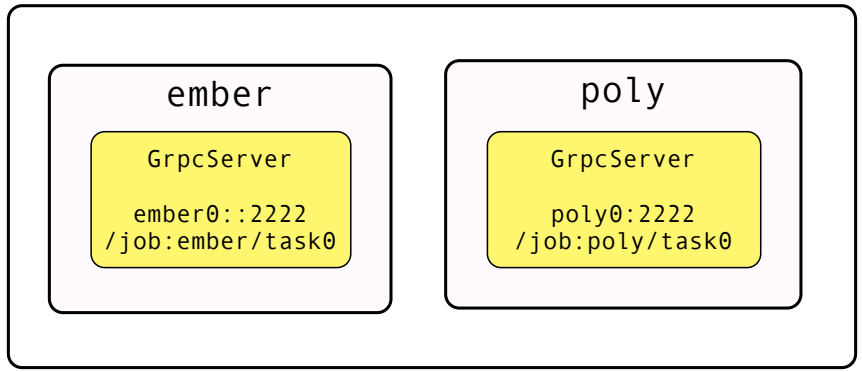
Distributed Computation Graph



- Client : Master로 GraphDef 객체를 전송.
- Master : Graph 객체를 구성하고 분할하여 Worker에게 전달.
- Worker : 로컬 머신에서 처리한 것과 동일하게 Graph 를 연산.

Cluster example

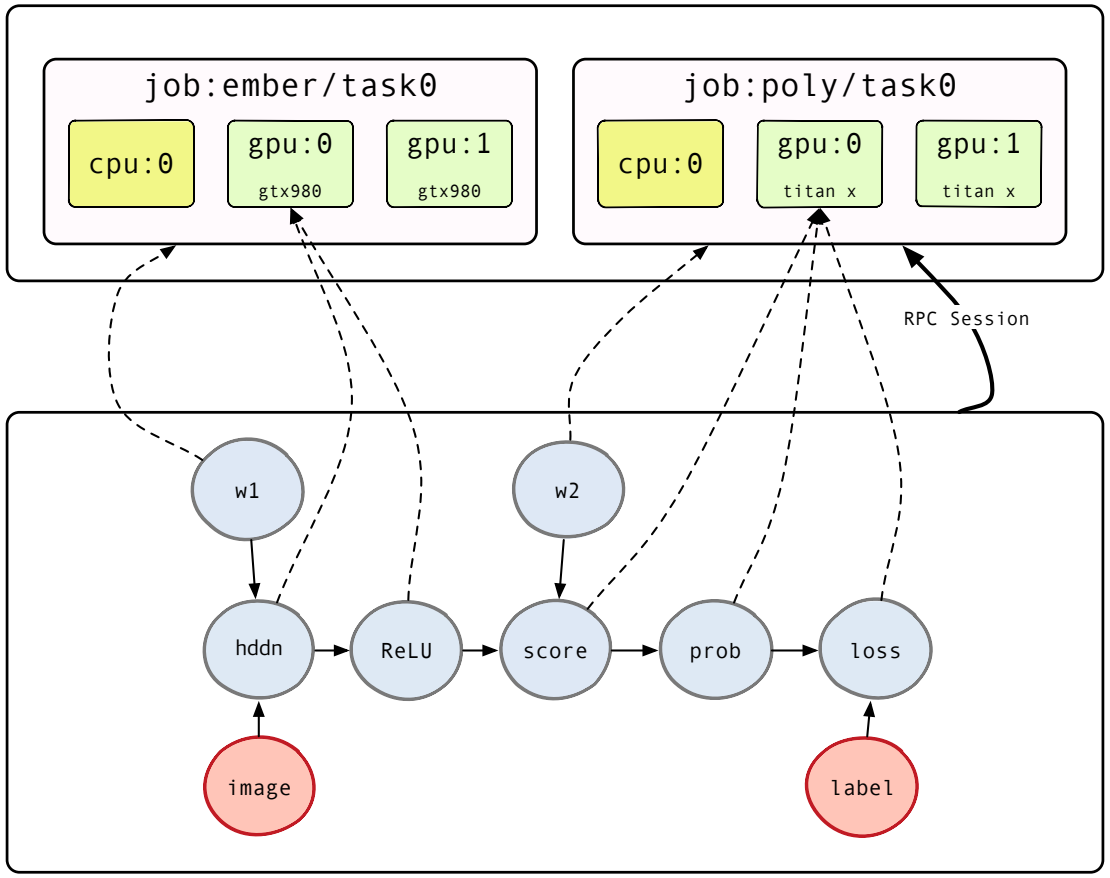




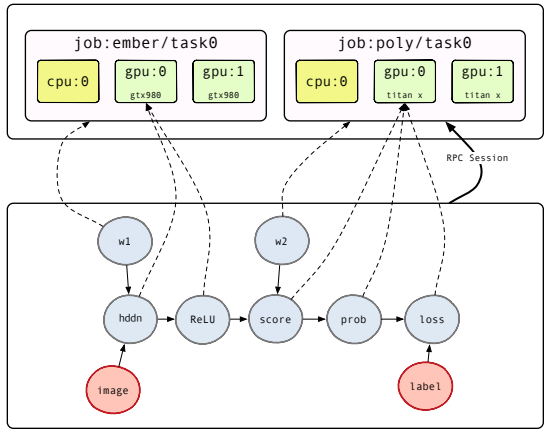

```
flags = tf.app.flags
FLAGS= flags.FLAGS
flags.DEFINE_string('job_name', 'ember', 'jobname')
flags.DEFINE_integer('task_index', 0, 'task_index')

cdef = tf.make_cluster_def({"poly": ["poly.[redacted]:50000"], "ember" : ["ember.[redacted]:60000"]})

s2def = tf.ServerDef(cluster=cdef, job_name=FLAGS.job_name, task_index=FLAGS.task_index, protocol="grpc")
s2 = tf.GrpcServer(s2def)
```

분산 코드 구현

```python
mnist = data.read_data_sets('./data', one_hot=True)

image = tf.placeholder(tf.float32, [None, 28 * 28], name="image")
label = tf.placeholder(tf.float32, [None, 10], name="label")

with tf.device("/job:ember/replica:0/task:0"):
    w1 = tf.Variable(1e-3 * rd.randn(28 * 28, 100).astype(np.float32), name="w1")

with tf.device("/job:poly/replica:0/task:0"):
    w2 = tf.Variable(1e-3 * rd.randn(100, 10).astype(np.float32), name="w2")

with tf.device("/job:ember/replica:0/task:0/gpu:0"):
    hddn = tf.matmul(image, w1, name="hidden")
    relu = tf.nn.relu(hddn, name="relu")

with tf.device("/job:poly/replica:0/task:0/gpu:0"):
    score = tf.matmul(relu, w2, name="score")
    prob  = tf.nn.softmax(score, name="prob")

    with tf.variable_scope("loss"):
        loss  = -tf.reduce_sum(label * tf.log(prob), name="loss")

    with tf.device("/job:poly/replica:0/task:0/gpu:1"):
        step = tf.train.GradientDescentOptimizer(1e-2, name="step").minimize(loss)

with tf.Session("grpc://localhost:50000") as sess:
    init = tf.initialize_all_variables()
    sess.run(init)

    for i in range(200):
        xs, ys = mnist.train.next_batch(100)
        _, val = sess.run([step, loss], feed_dict = {image: xs, label: ys})
```



그래프 전달

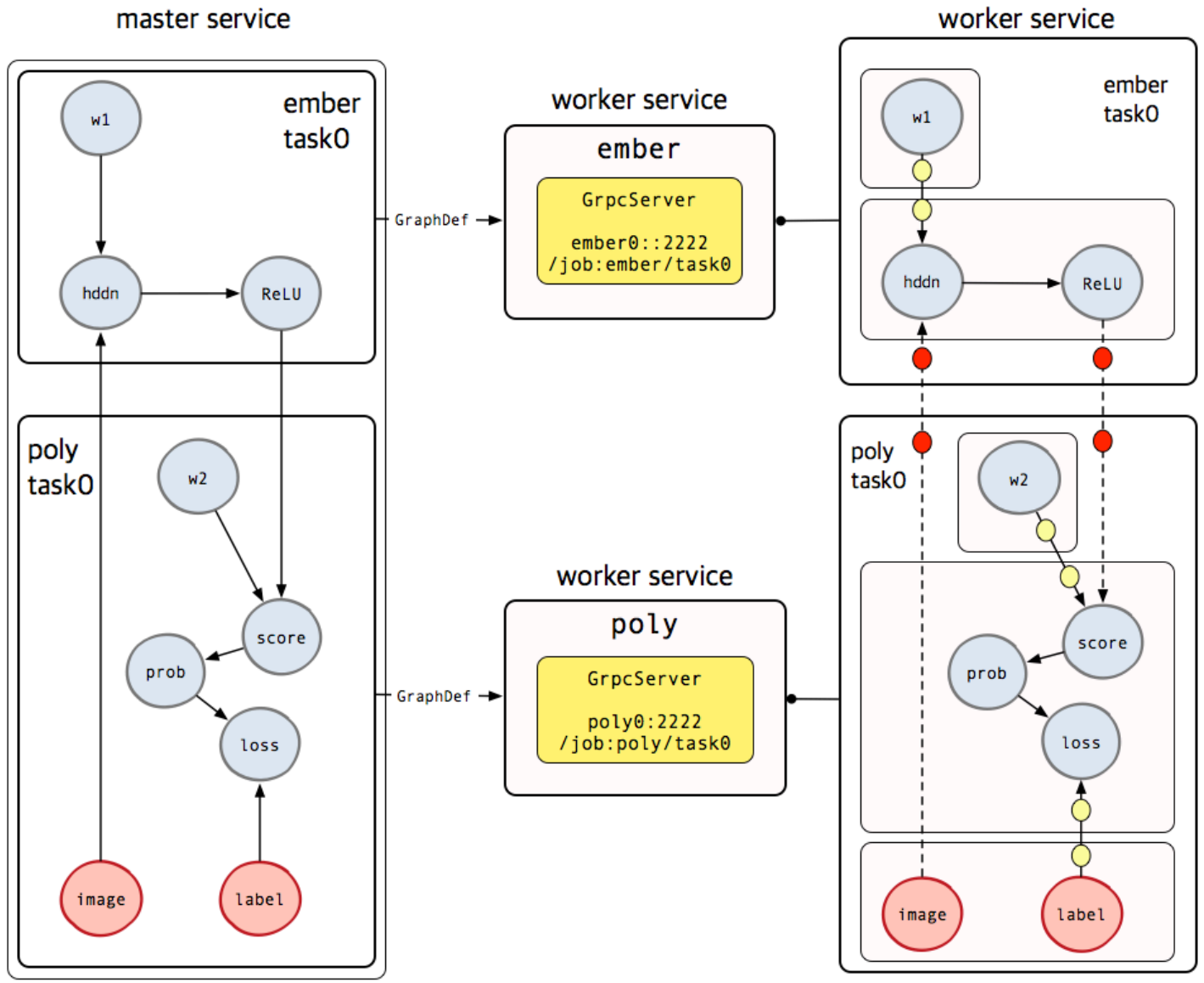


[마스터 서버]

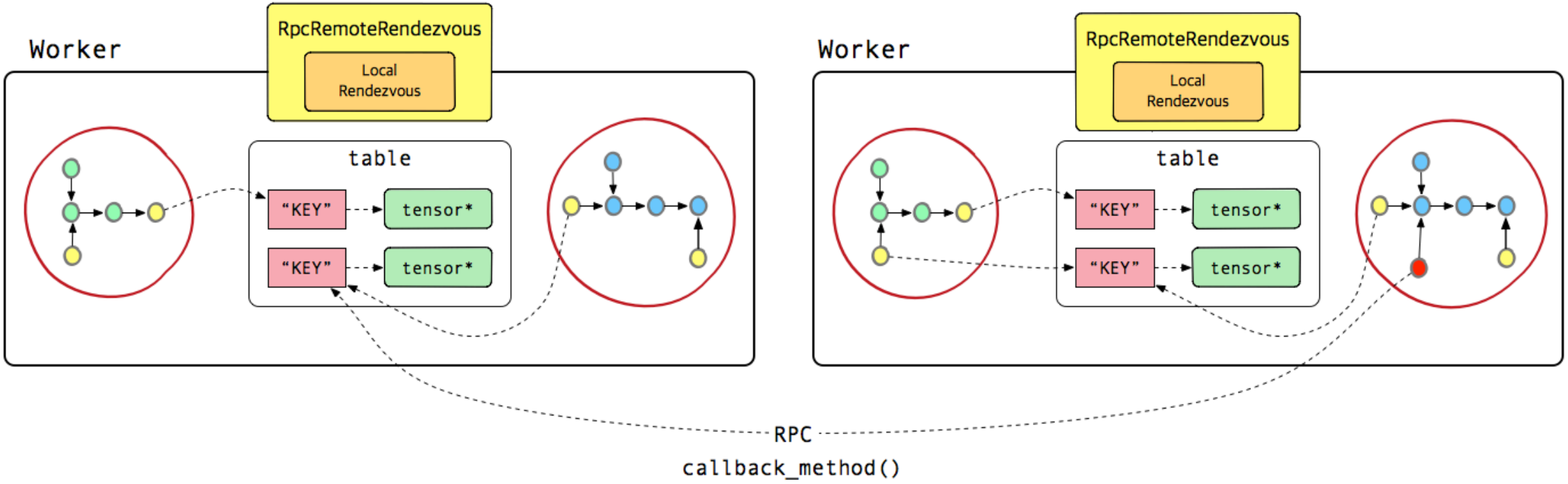
1. SimplePlacer로 각 노드에 디바이스 할당
2. Task 서버 단위 파티셔닝 (recv, send 없음)
3. 생성된 GraphDef 를 각 task 서버에 전달

[워커 서버]

1. Task 서버에서는 각 그래프 개별 실행
2. 결과 반환



RpcRemoteRendezvous



Handling very large models

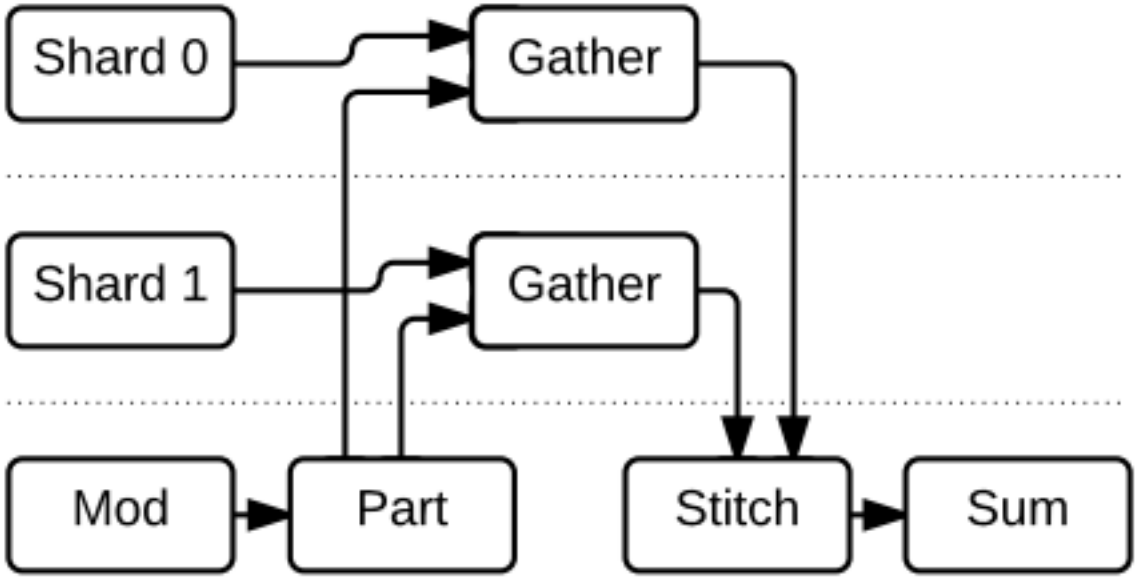


Figure 3: Schematic dataflow graph for a sparse embedding layer containing a two-way sharded embedding matrix.

Synchronous replica coordination

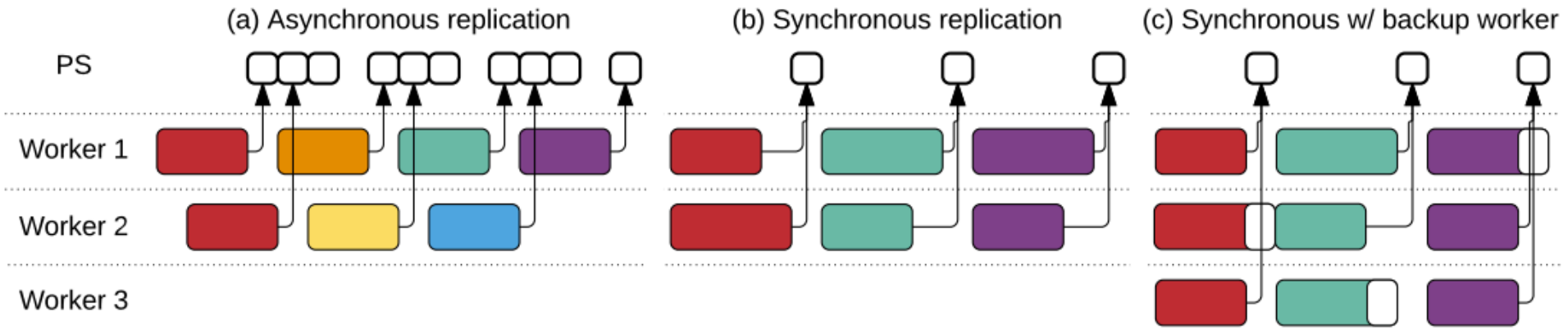
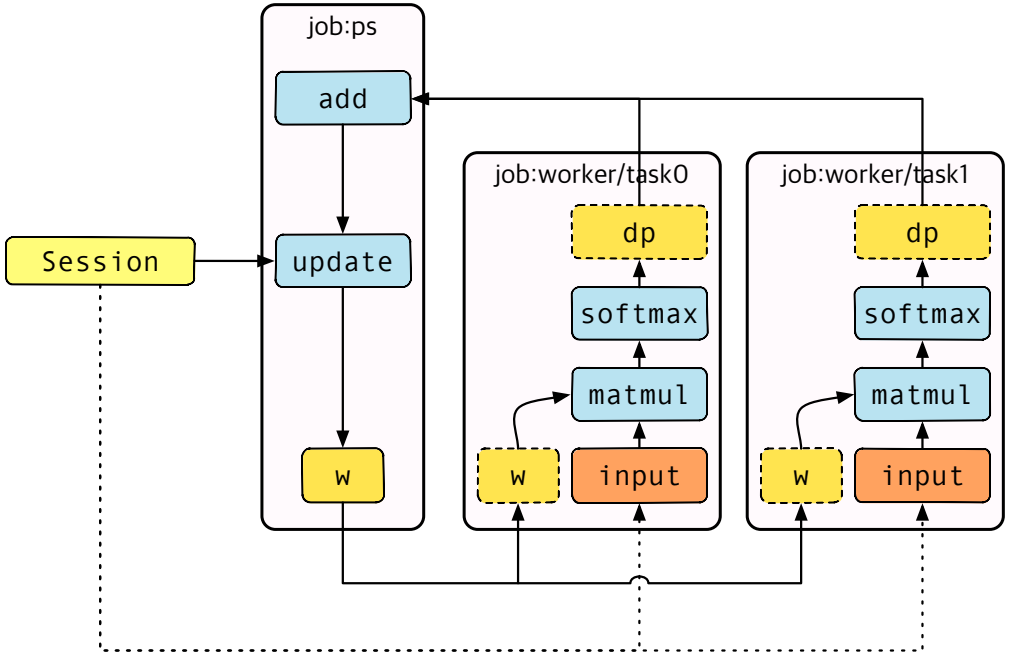
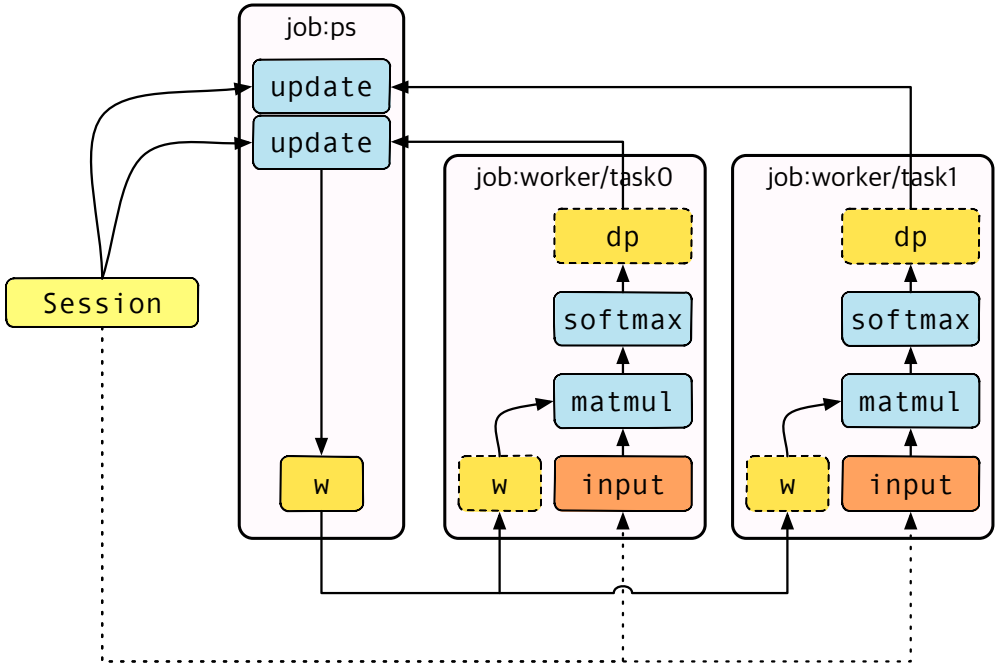


Figure 4: Three parameter synchronization schemes for a single parameter in data-parallel training (§4.4): (a) asynchronous, (b) synchronous without backup workers, and (c) synchronous with backup workers.

Details

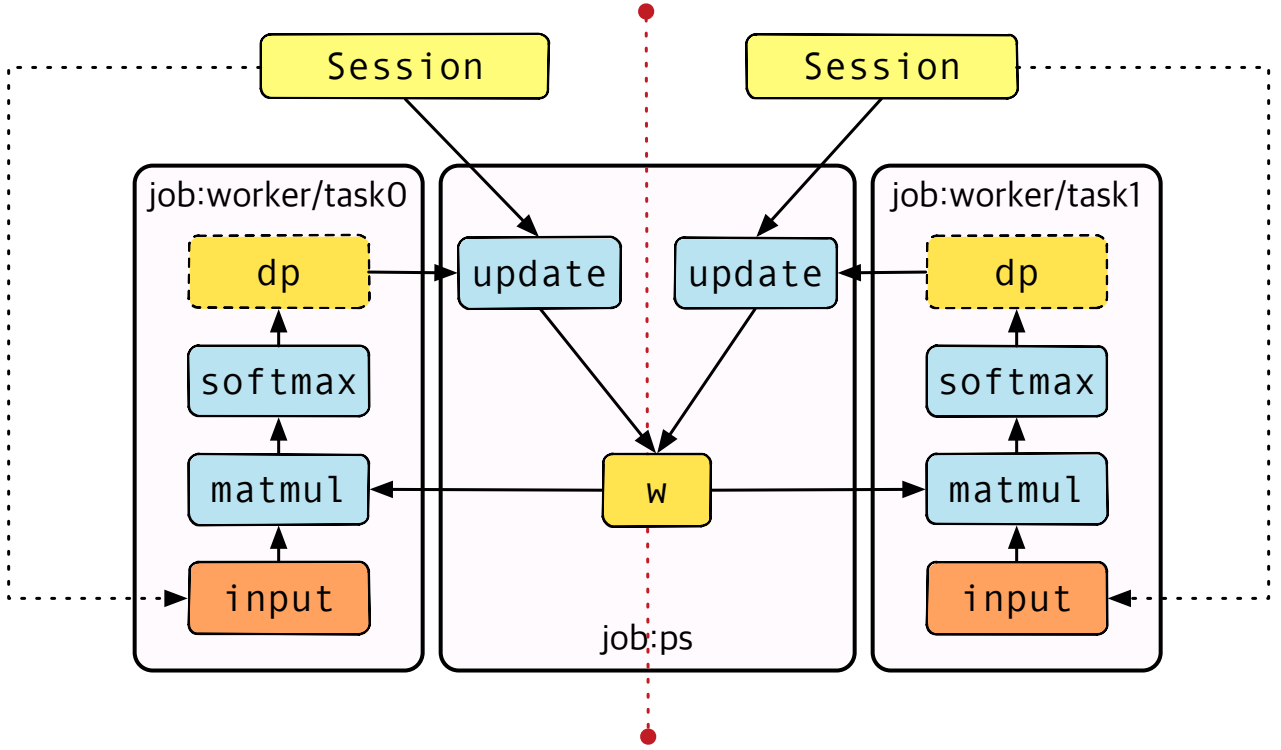


Synchronous data parallelism



Asynchronous data parallelism

Replicated Training (Cont'd)



Shared variables between different graphs

Evaluation

- Focus on system performance metric rather than accuracy.

Library	Training step time (ms)			
	AlexNet	Overfeat	OxfordNet	GoogleNet
Caffe [36]	324	823	1068	1935
Neon [56]	87	**211**	**320**	**270**
Torch [17]	**81**	268	529	470
TensorFlow	**81**	279	540	445

Table 1: Step times for training four convolutional models with different libraries, using one GPU. All results are for training with 32-bit floats. The fastest library for each model is shown in bold.

Single Machine

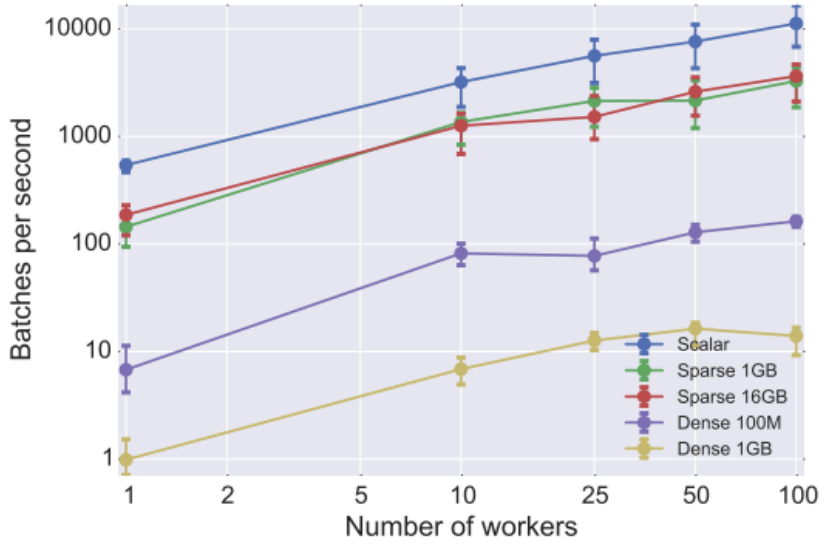


Figure 6: Baseline throughput for synchronous replication with a null model. Sparse accesses enable TensorFlow to handle larger models, such as embedding matrices (§4.2).

Evaluation (Cont'd)

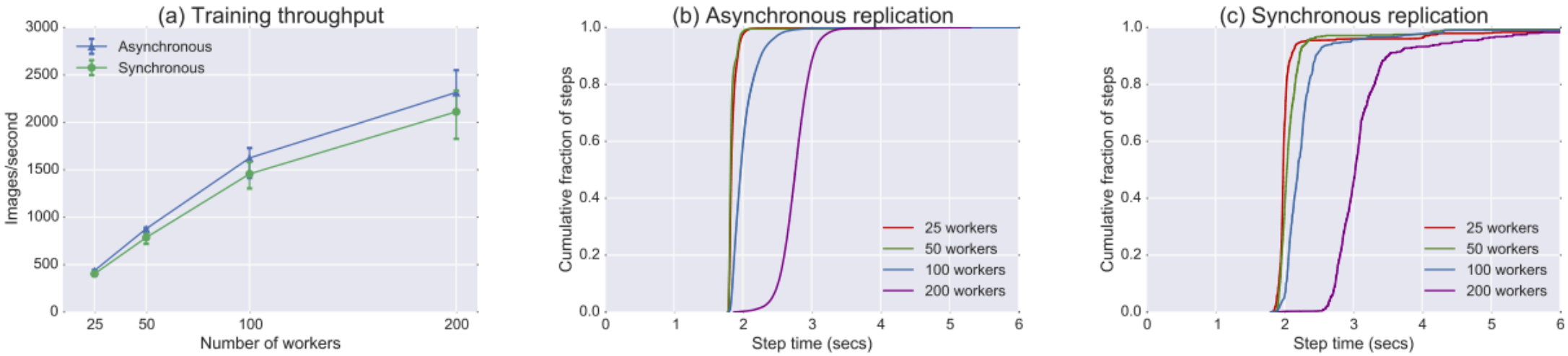


Figure 7: (a) Inception-v3 training throughput increases with up to 200 workers. However, adding more workers gets diminishing returns because the step time increases for both (b) asynchronous and (c) synchronous replication.

Evaluation (Cont'd)

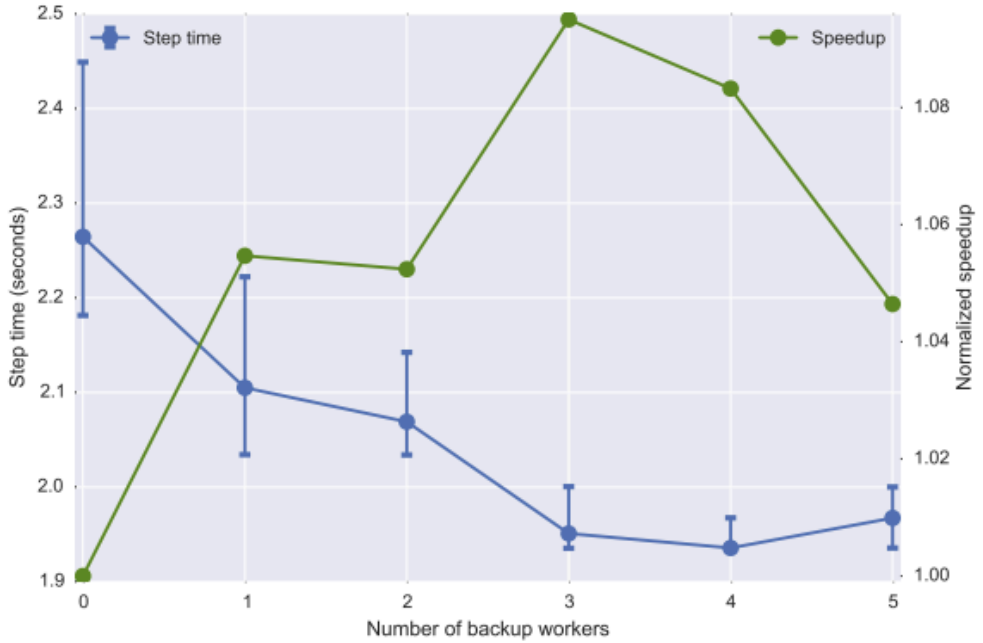


Figure 8: Backup workers reduce the step time for 50-worker Inception-v3 training. 4 backup workers give the shortest overall step time, but 3 backup workers are most efficient when we normalize for the total resources used.

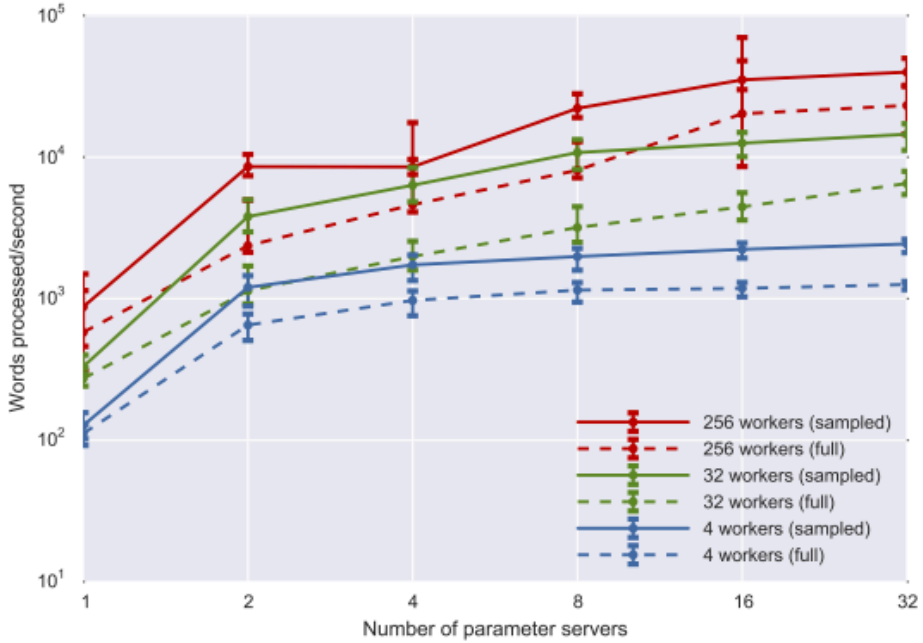


Figure 9: Increasing the number of PS tasks leads to increased throughput for language model training, by parallelizing the softmax computation. Sampled softmax increases throughput by performing less computation.