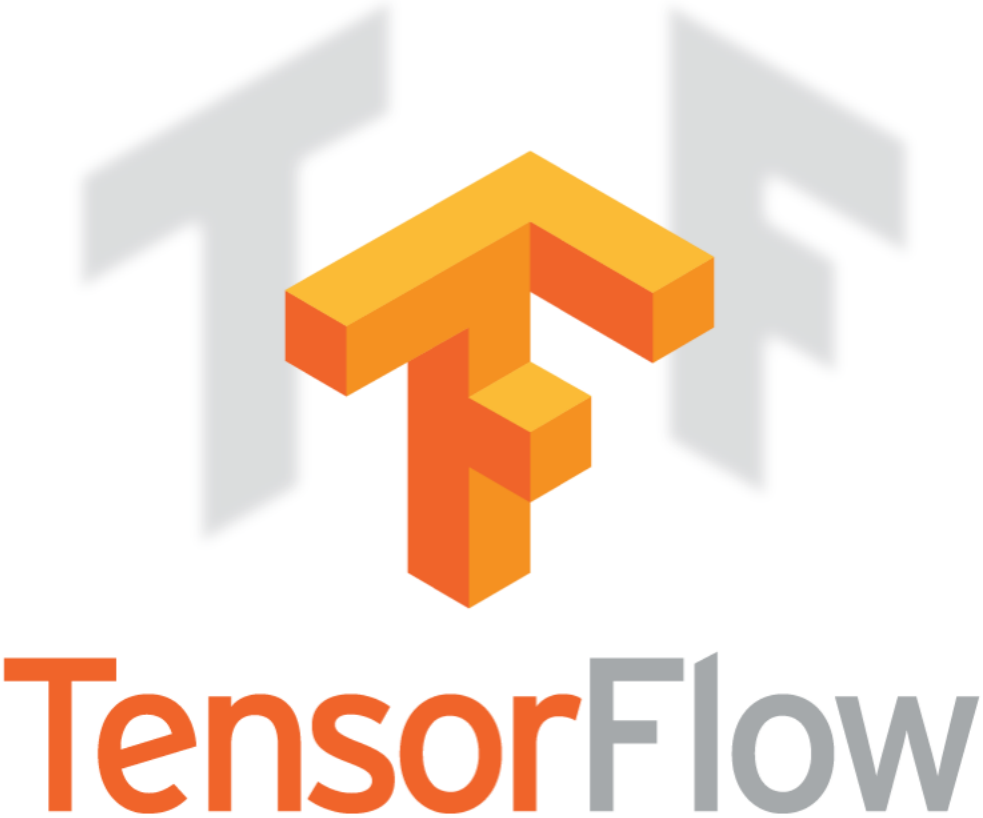


TensorFlow

Goals for this talk

Intro to main TensorFlow concepts

- Useful to have a mental model for how the system behaves
- …but mostly you can forget everything and use the high-level wrappers

Questions!

Background

Google Brain project started in 2011, with a focus on pushing state-of-the-art in neural networks. Initial emphasis:

use large datasets, and

large amounts of computation

to push boundaries of what is possible in perception and language understanding

Two Generations of Distributed ML Systems

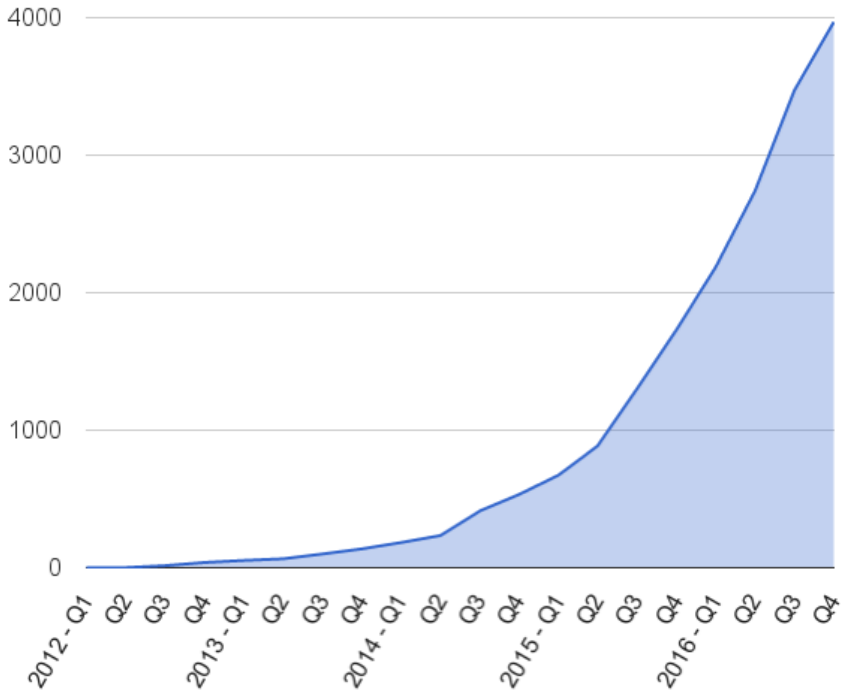
1st generation - DistBelief (Dean *et al.*, NIPS 2012)

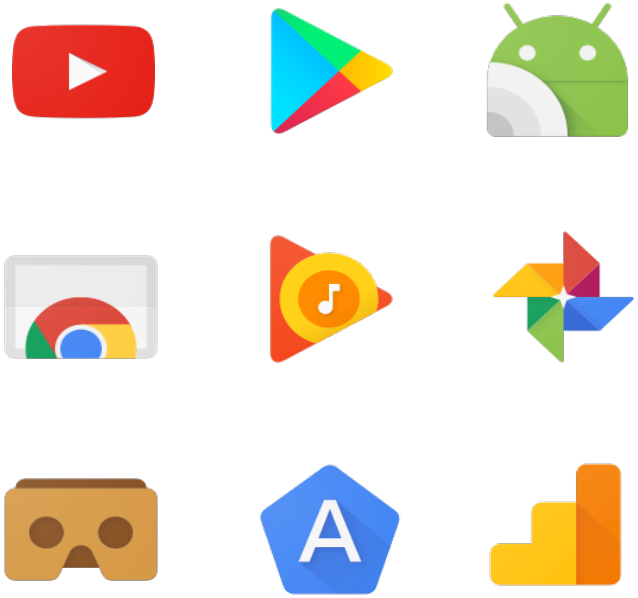
- Scalable, good for production, but not very flexible for research

2nd generation - TensorFlow (see tenorflow.org)

- Ease of expression: for lots of crazy ML ideas/algorithms
- Scalability: can run experiments quickly
- Portability: can run on wide variety of platforms
- Reproducibility: easy to share and reproduce research

Google directories containing model description files





and many more . . . .

TensorFlow Design Overview

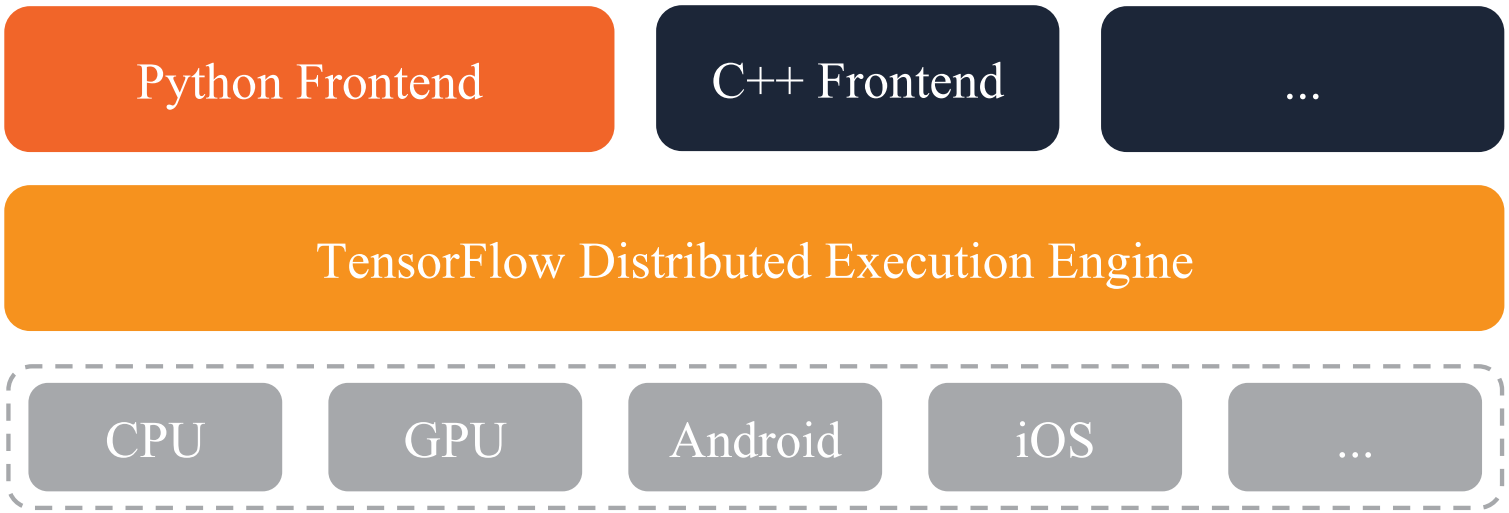
General Approach

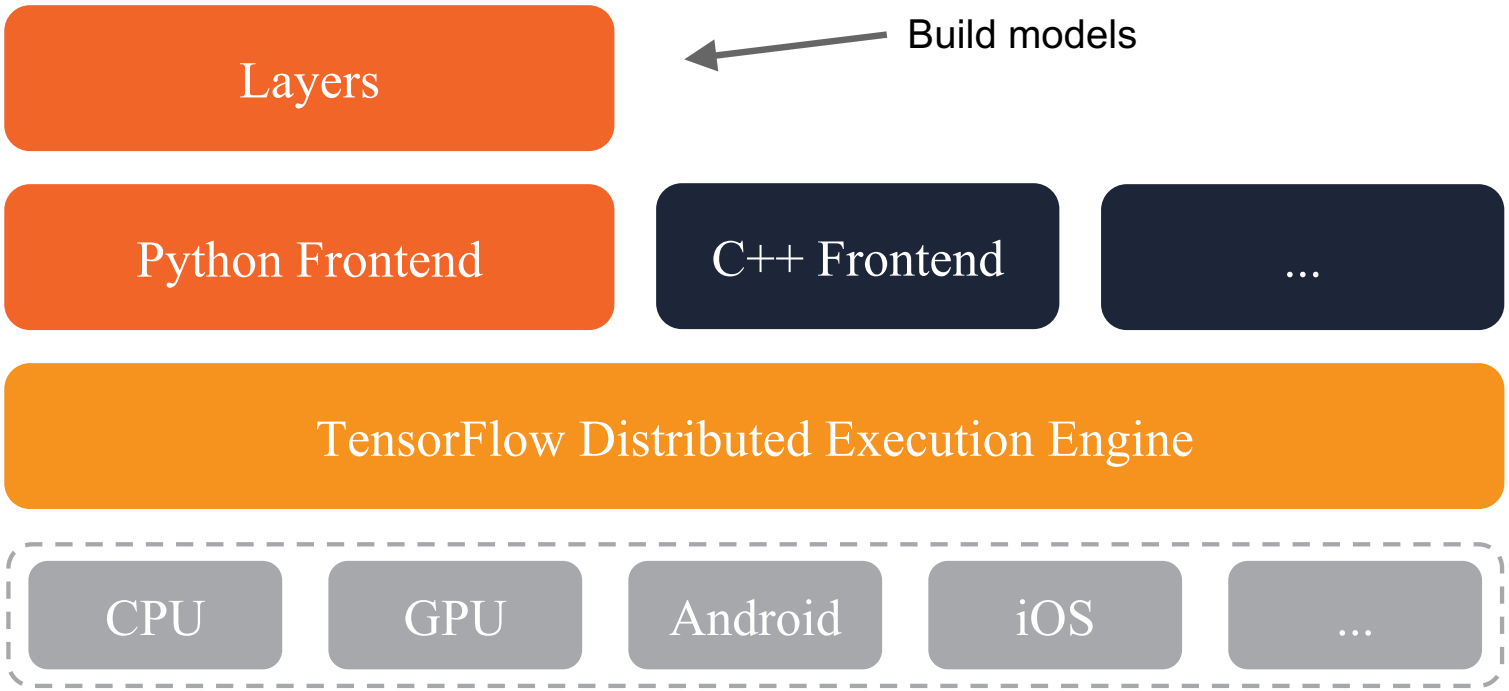
Define a general-purpose computation graph

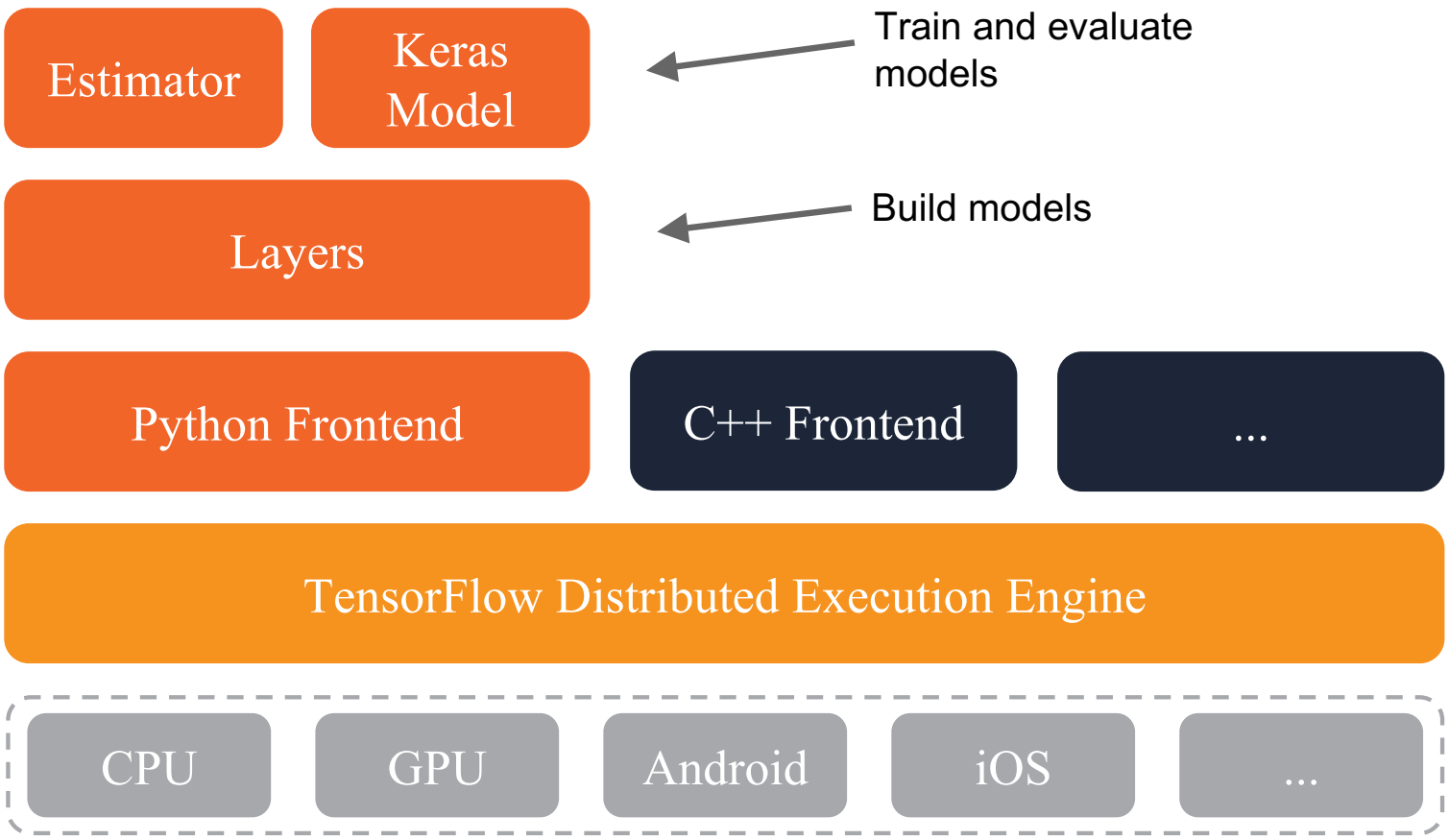
- Achieves ease of expression

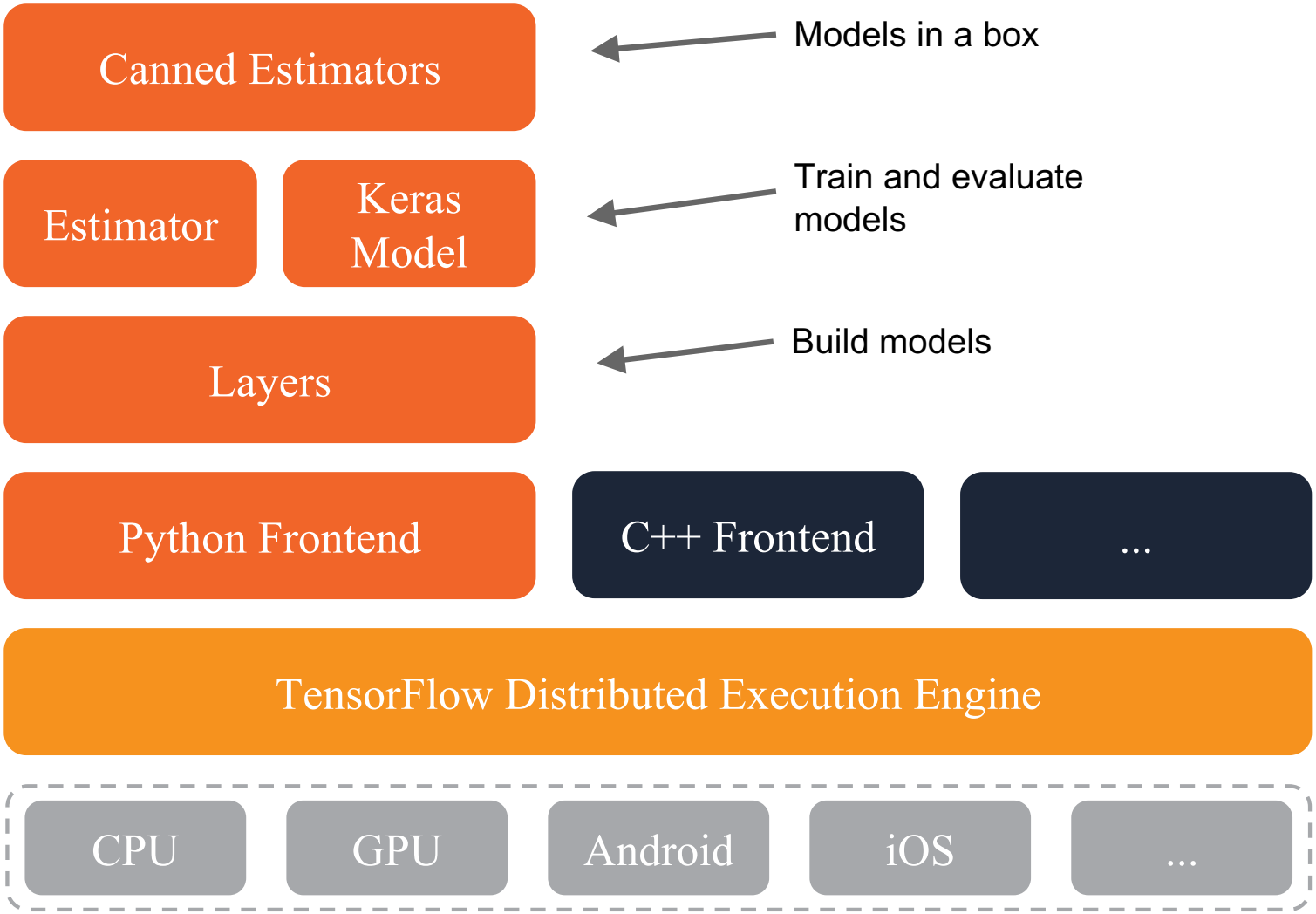
Create tools for running in different environments

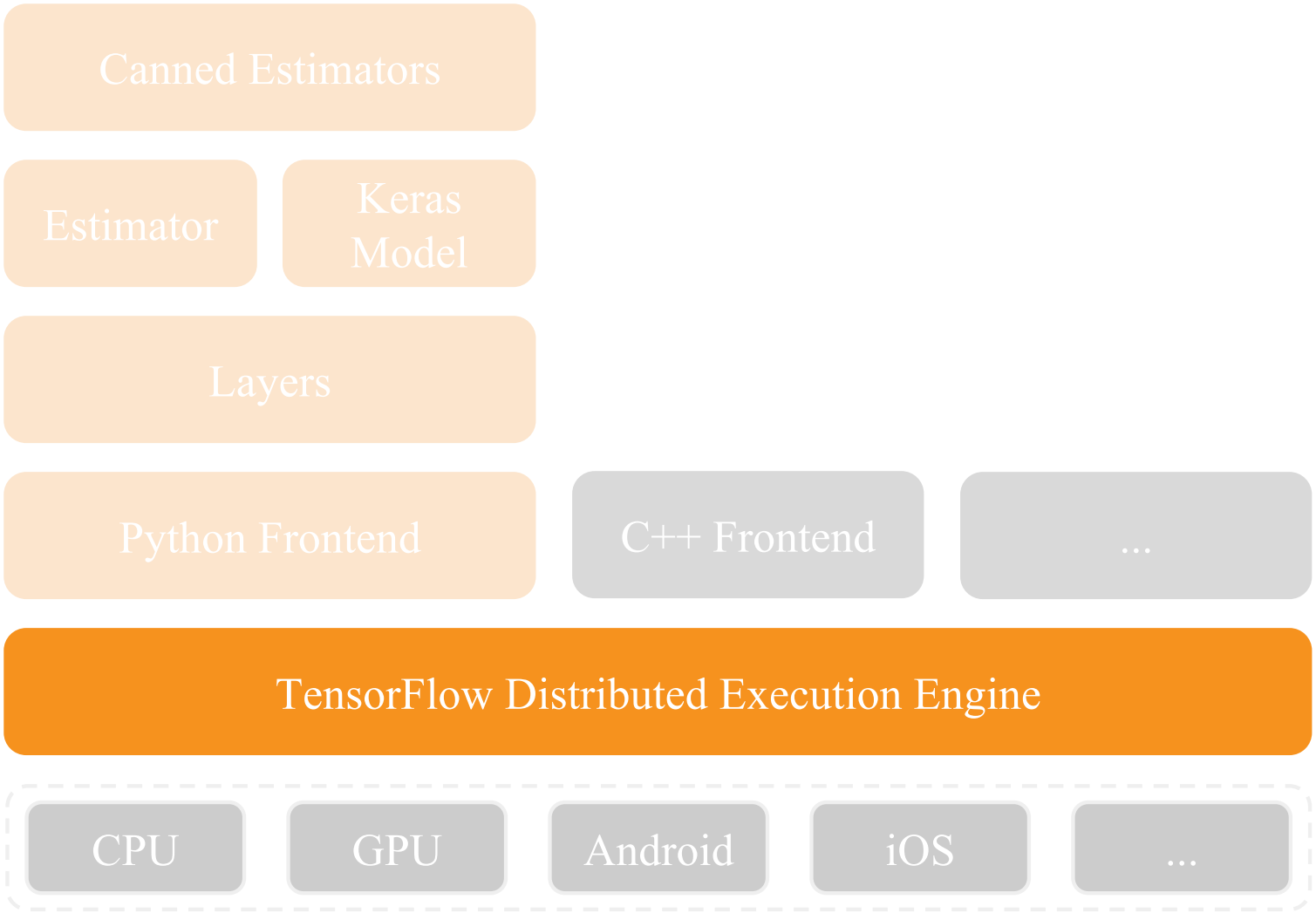
- Achieves portability, scalability, reproducibility and production-readiness


Python Frontend
C++ Frontend
...
TensorFlow Distributed Execution Engine
CPU
GPU
Android
iOS
...

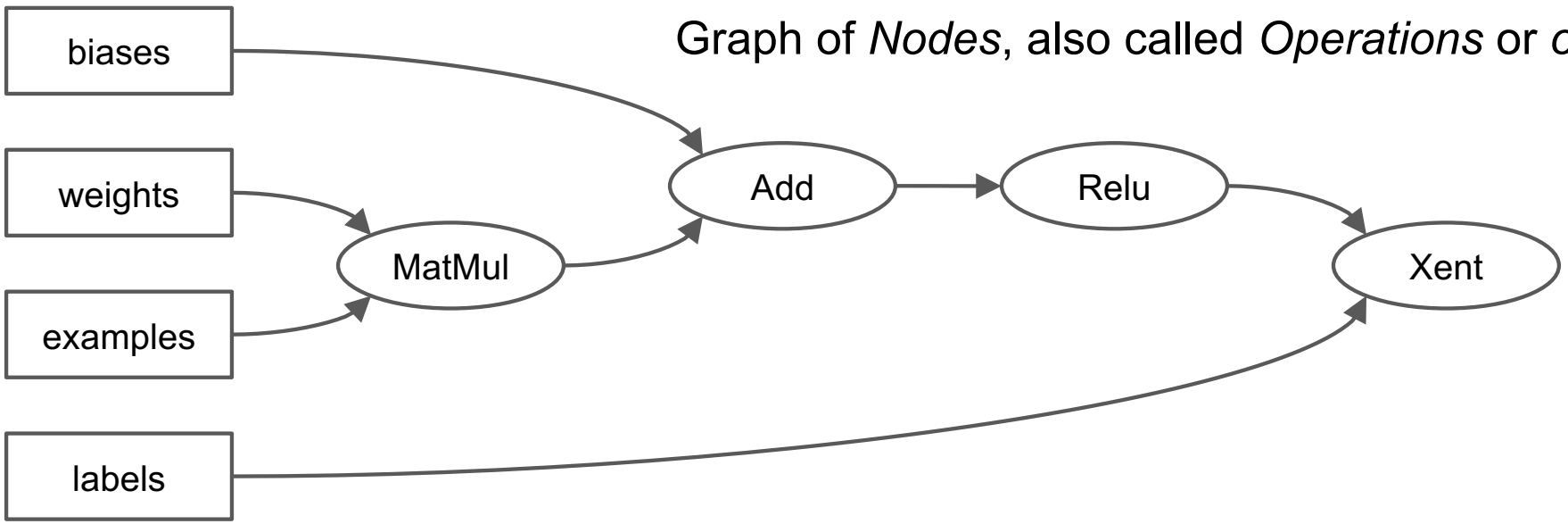

Layers
Build models
Python Frontend
C++ Frontend
...
TensorFlow Distributed Execution Engine
CPU
GPU
Android
iOS
...


Estimator
Keras Model
Train and evaluate models
Layers
Build models
Python Frontend
C++ Frontend
...
TensorFlow Distributed Execution Engine
CPU
GPU
Android
iOS
...


Canned Estimators
Models in a box
Estimator
Keras Model
Train and evaluate models
Layers
Build models
Python Frontend
C++ Frontend
...
TensorFlow Distributed Execution Engine
CPU
GPU
Android
iOS
...

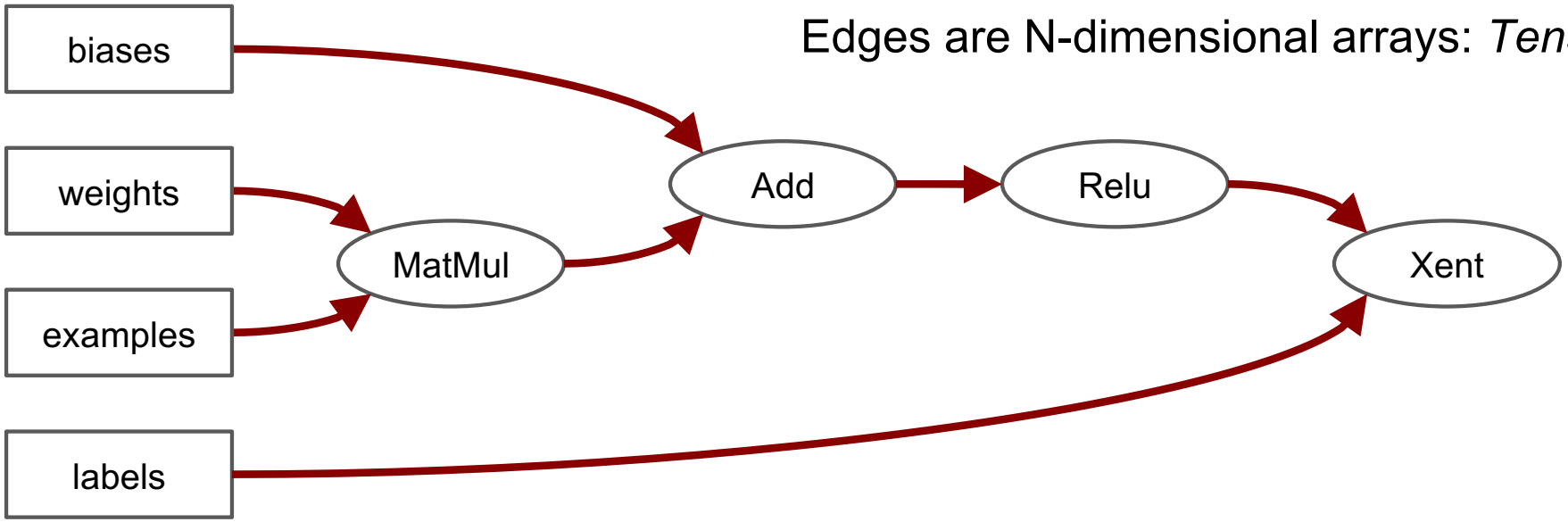

Canned Estimators
Estimator
Keras Model
Layers
Python Frontend
C++ Frontend
...
TensorFlow Distributed Execution Engine
CPU
GPU
Android
iOS
...

Computation is a dataflow graph



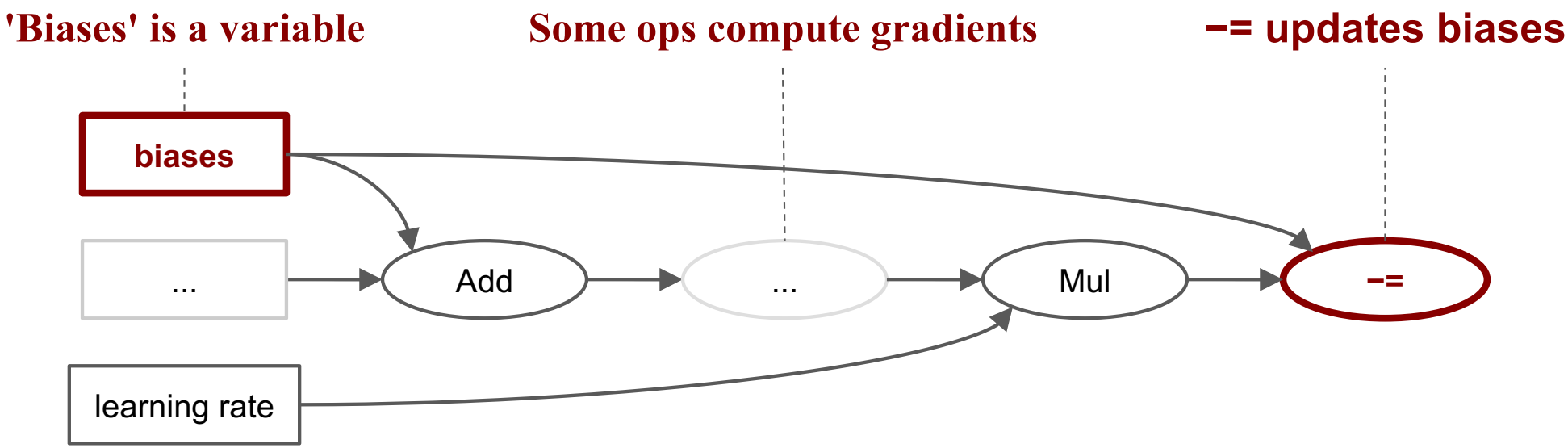
Computation is a dataflow graph

with tensors



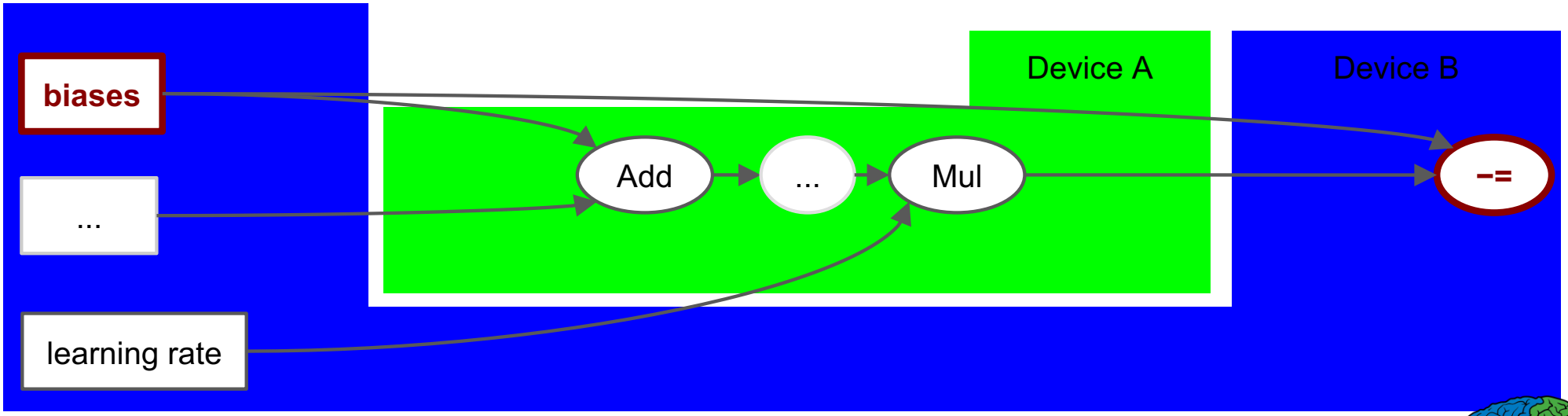
Computation is a dataflow graph

with state

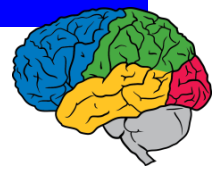


Computation is a dataflow graph

distributed

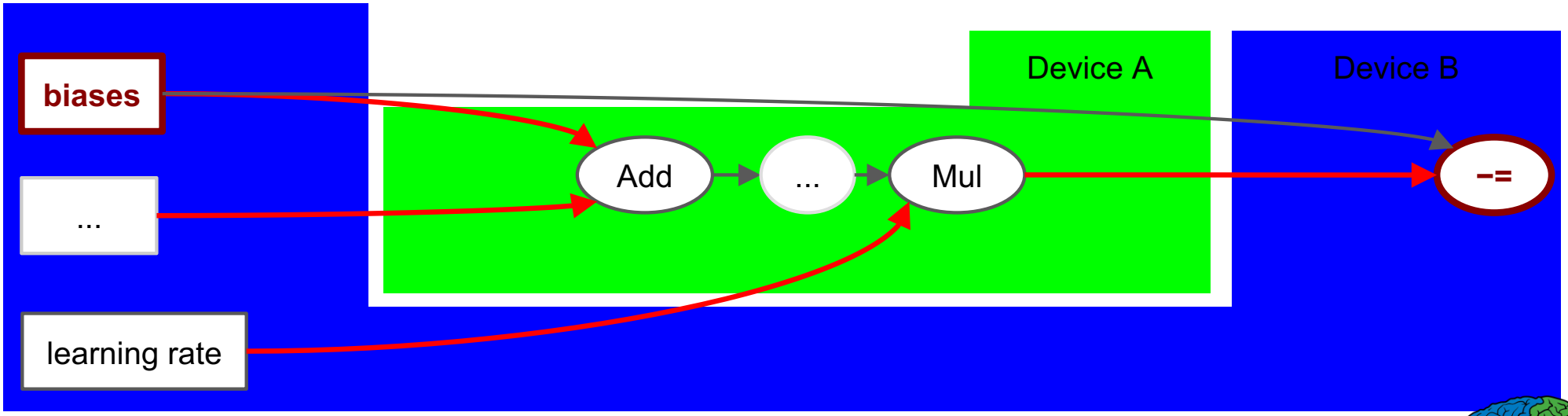


Devices: Processes, Machines, GPUs, etc

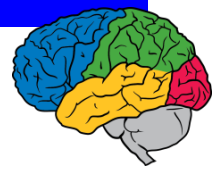


Send and Receive Nodes

distributed

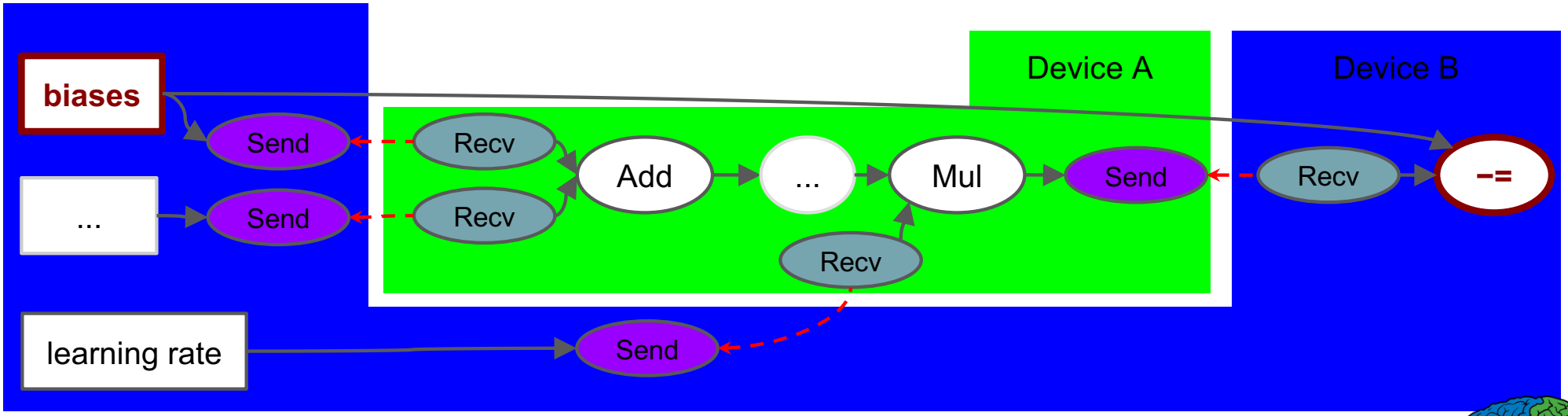


Devices: Processes, Machines, GPUs, etc

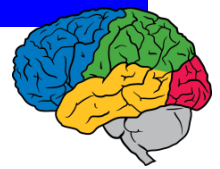


Send and Receive Nodes

distributed



Devices: Processes, Machines, GPUs, etc



Send and Receive Implementations

Implementing send/recv in the runtime system ⇒ optimization

Different implementations depending on source/dest devices

e.g. GPUs on same machine: **local GPU → GPU copy**

e.g. CPUs on different machines: **cross-machine RPC**

e.g. GPUs on different machines: **RDMA**

Extensibility

Core system defines a number of standard ***operations*** and ***kernels*** (device-specific implementations of operations)

Easy to define new operators and/or kernels

Sessions

Primary interface to the runtime

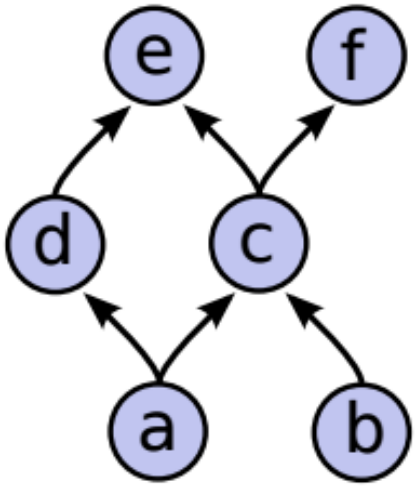
`Extend`: add nodes to computation graph

`Run`: execute an arbitrary subgraph

optionally feeding in Tensor inputs and retrieving Tensor output

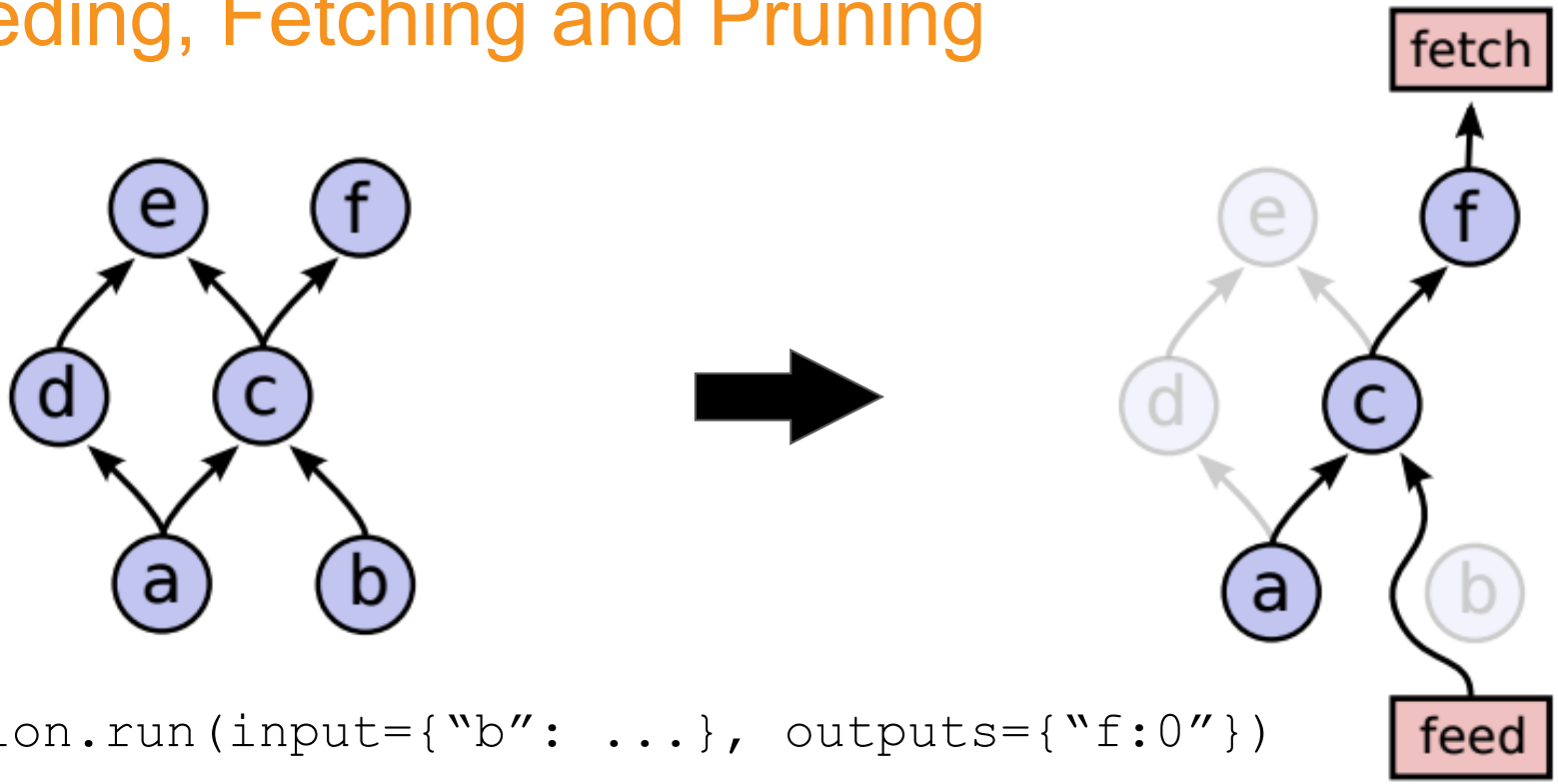
Typically, setup a graph with one or a few `Extend` calls and then `Run` it thousands or millions or times

Feeding, Fetching and Pruning



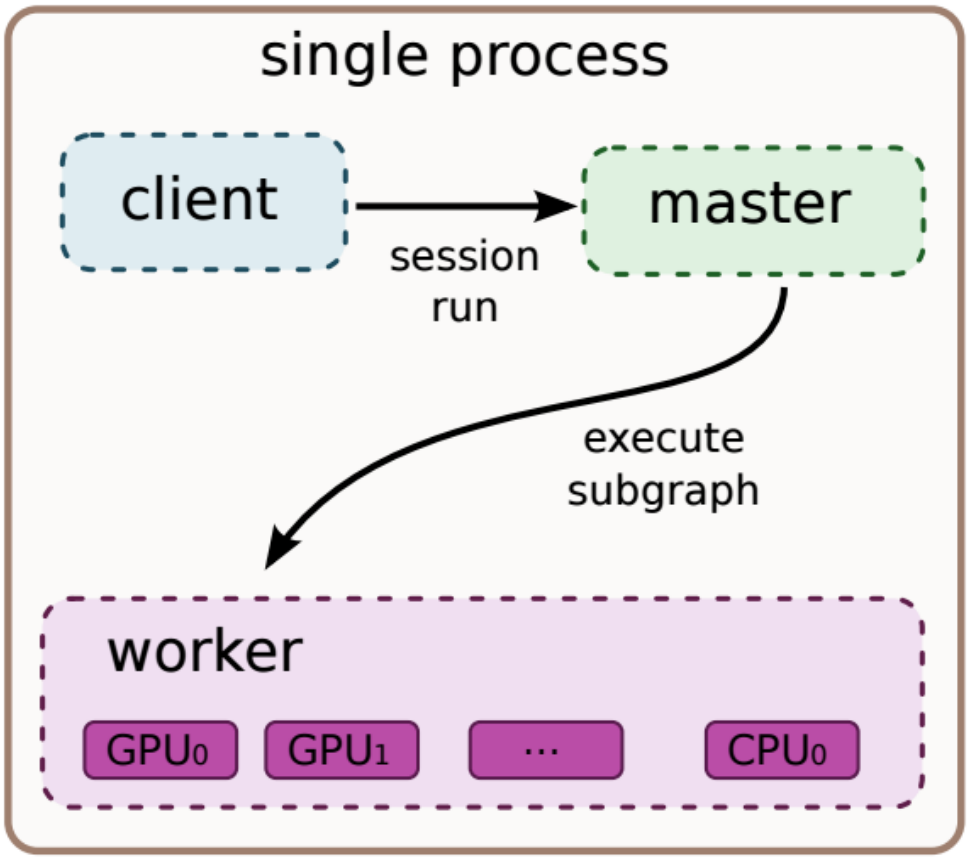

```
session.run(input={"b": ...}, outputs={"f:0"})
```

Feeding, Fetching and Pruning

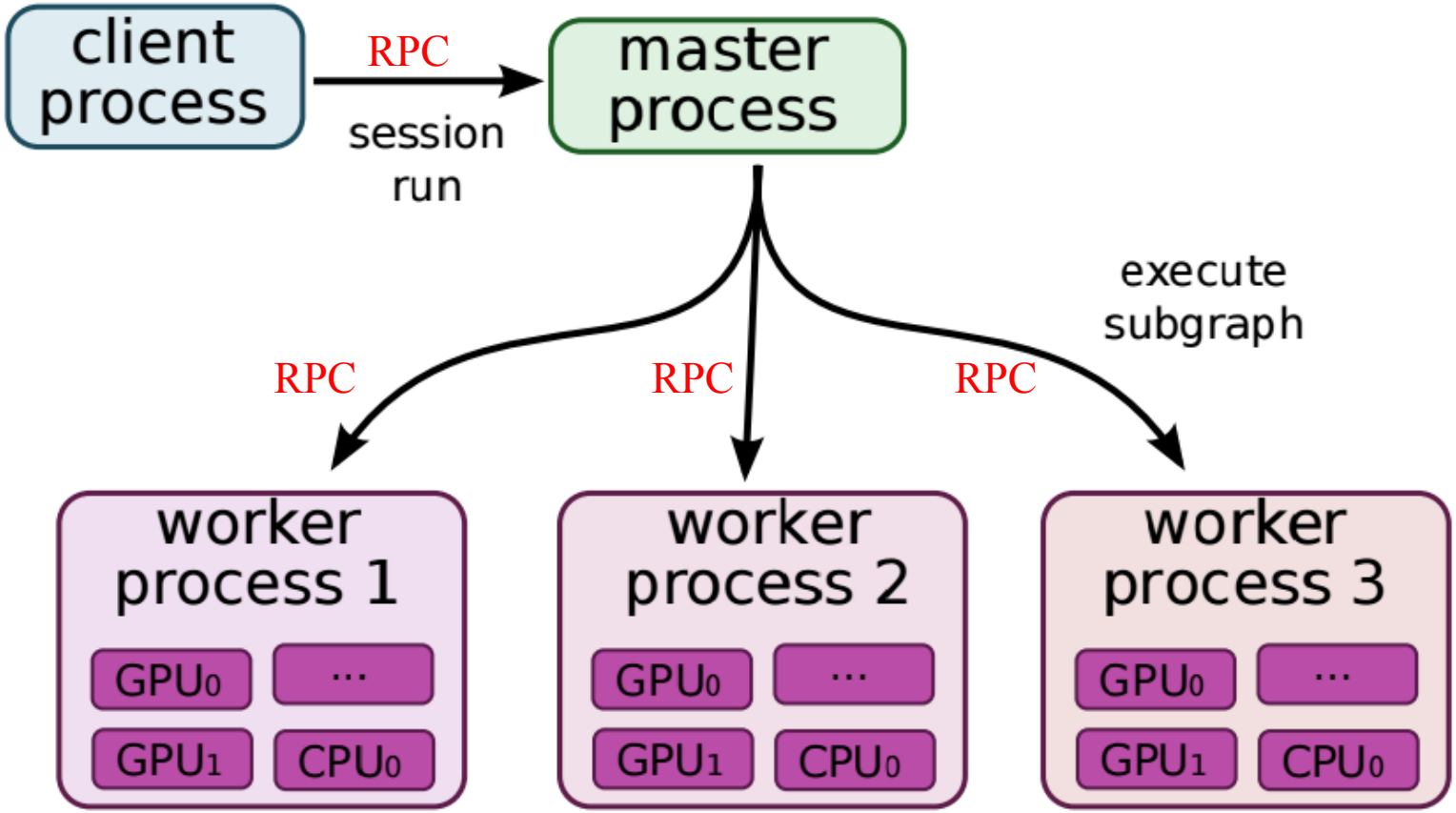


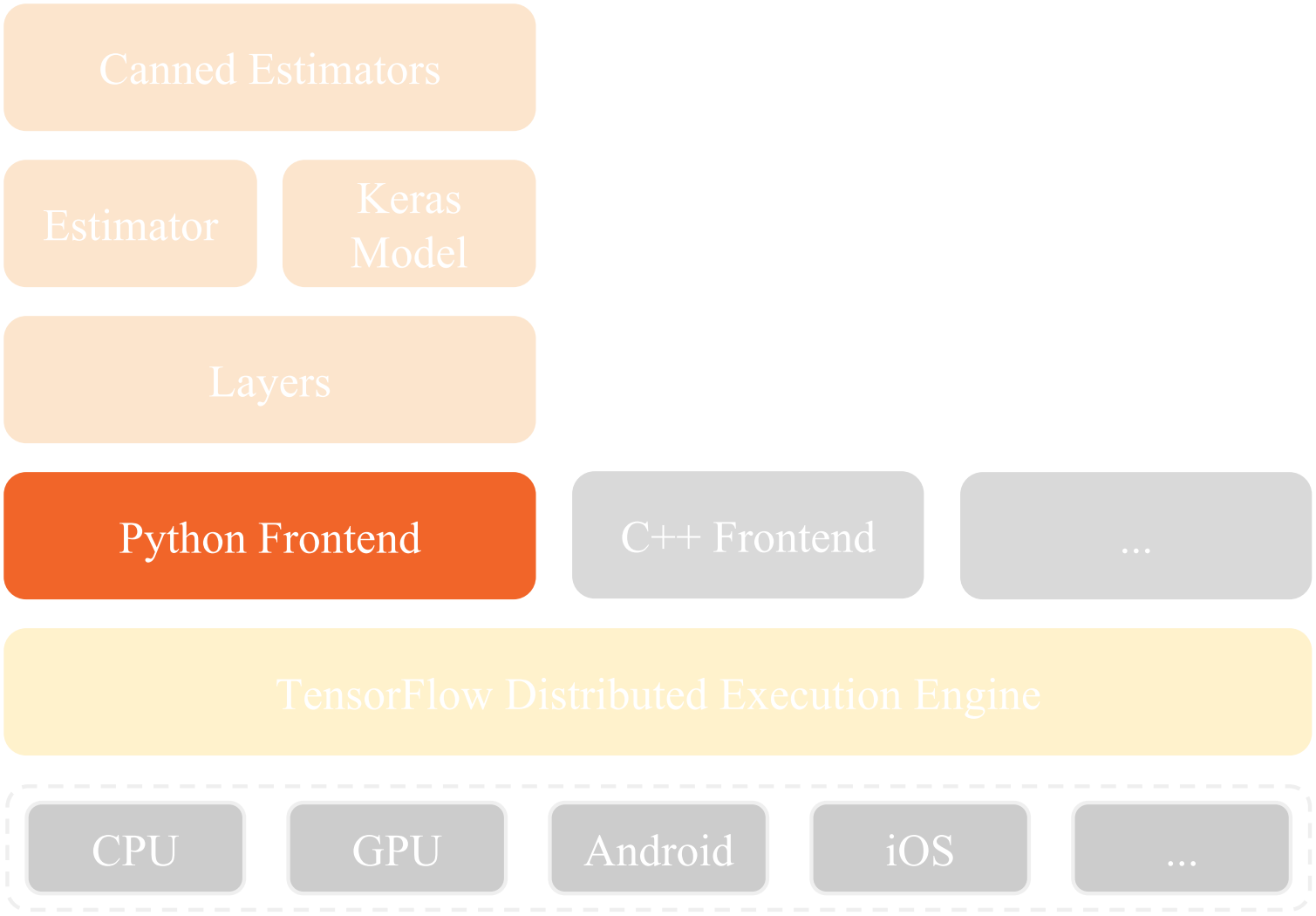

```
session.run(input={"b": ...}, outputs={"f:0"})
```

Single Process Configuration

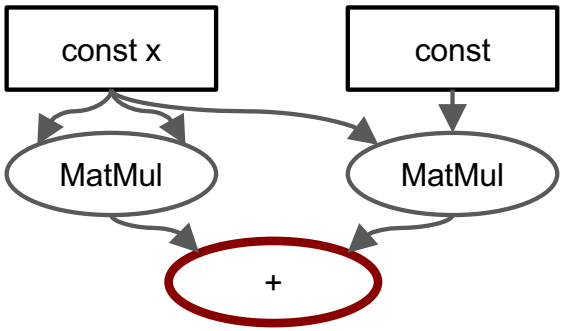


Distributed Configuration



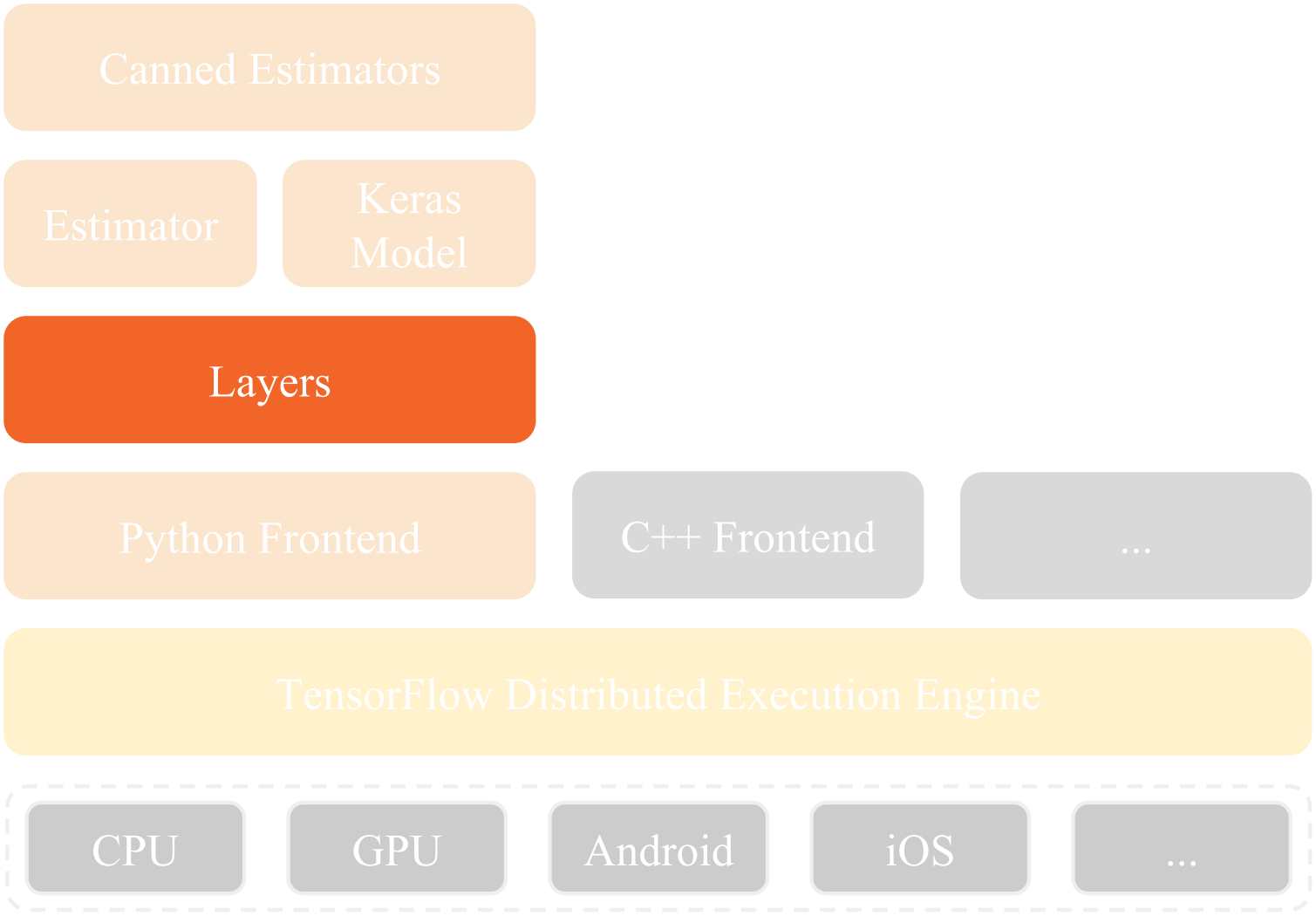

Canned Estimators
Estimator
Keras Model
Layers
Python Frontend
C++ Frontend
...
TensorFlow Distributed Execution Engine
CPU
GPU
Android
iOS
...

Compile and Run

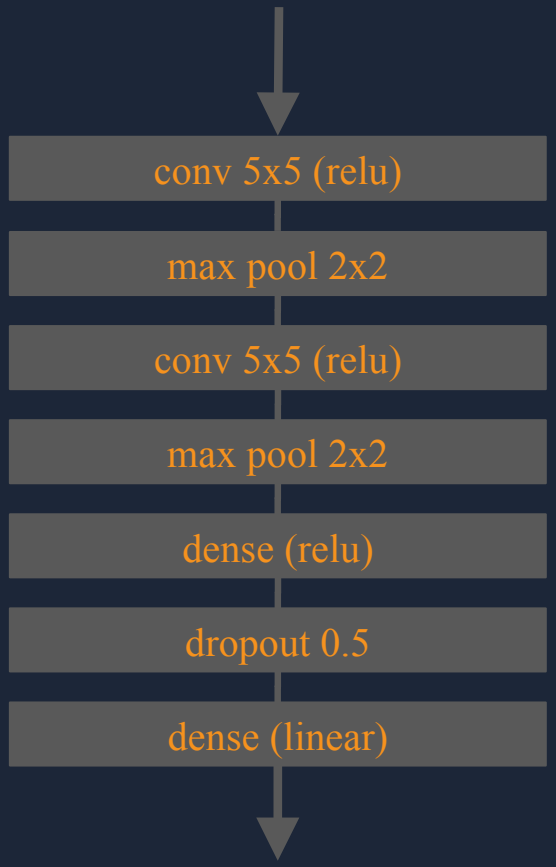


Operations build the dataflow graph; `eval()` fetches the result.

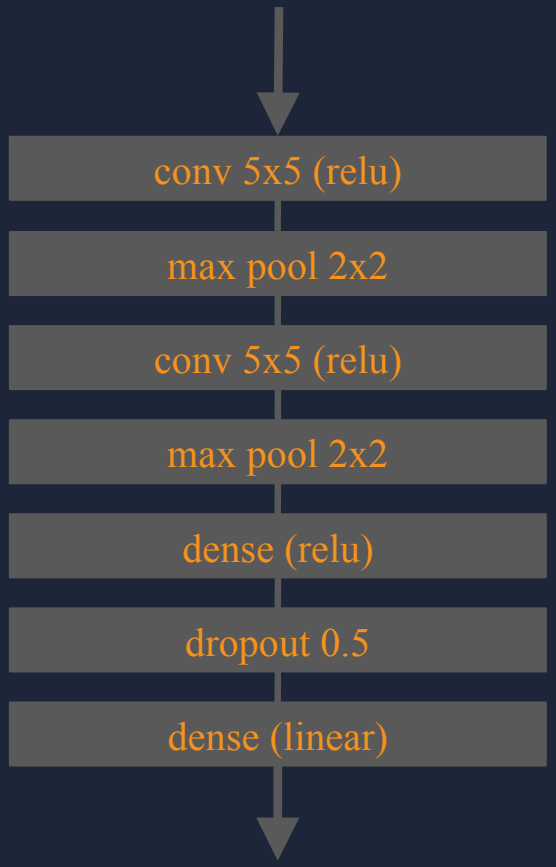
```
import tensorflow as tf

with tf.Session():

  x = tf.constant([[5, 6], [7, 8]])

  z = tf.matmul(x, x) + tf.matmul(x, [[1, 0], [0, 1]])

  # Run graph to fetch z.

  result = z.eval()  # Shorthand for sess.run(z)
```


Canned Estimators
Estimator
Keras Model
Layers
Python Frontend
C++ Frontend
...
TensorFlow Distributed Execution Engine
CPU
GPU
Android
iOS
...

Layers

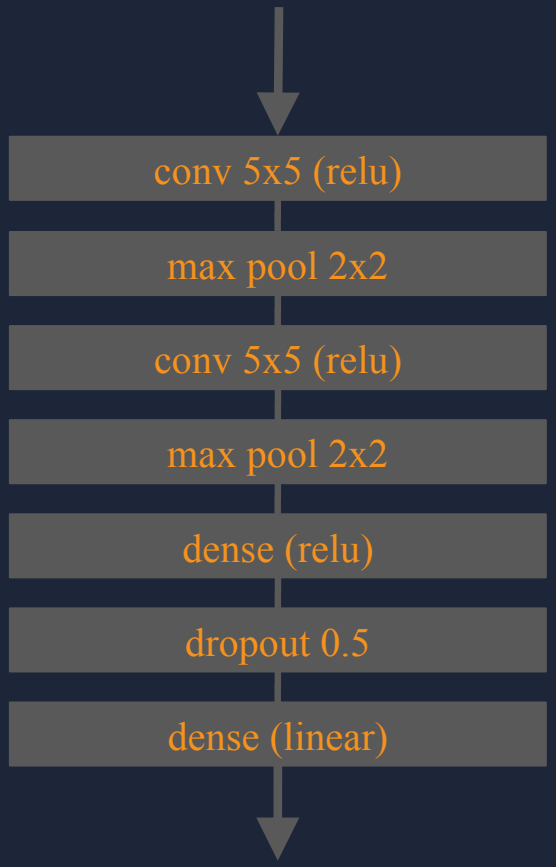

conv 5x5 (relu)
max pool 2x2
conv 5x5 (relu)
max pool 2x2
dense (relu)
dropout 0.5
dense (linear)

Layers




```
x = tf.layers.conv2d(x, kernel_size=[5,5], ...)
```

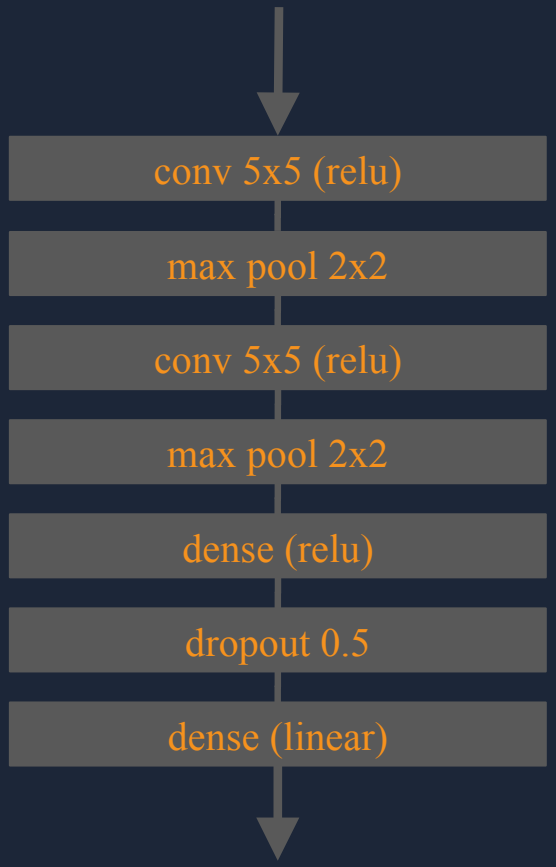
Layers



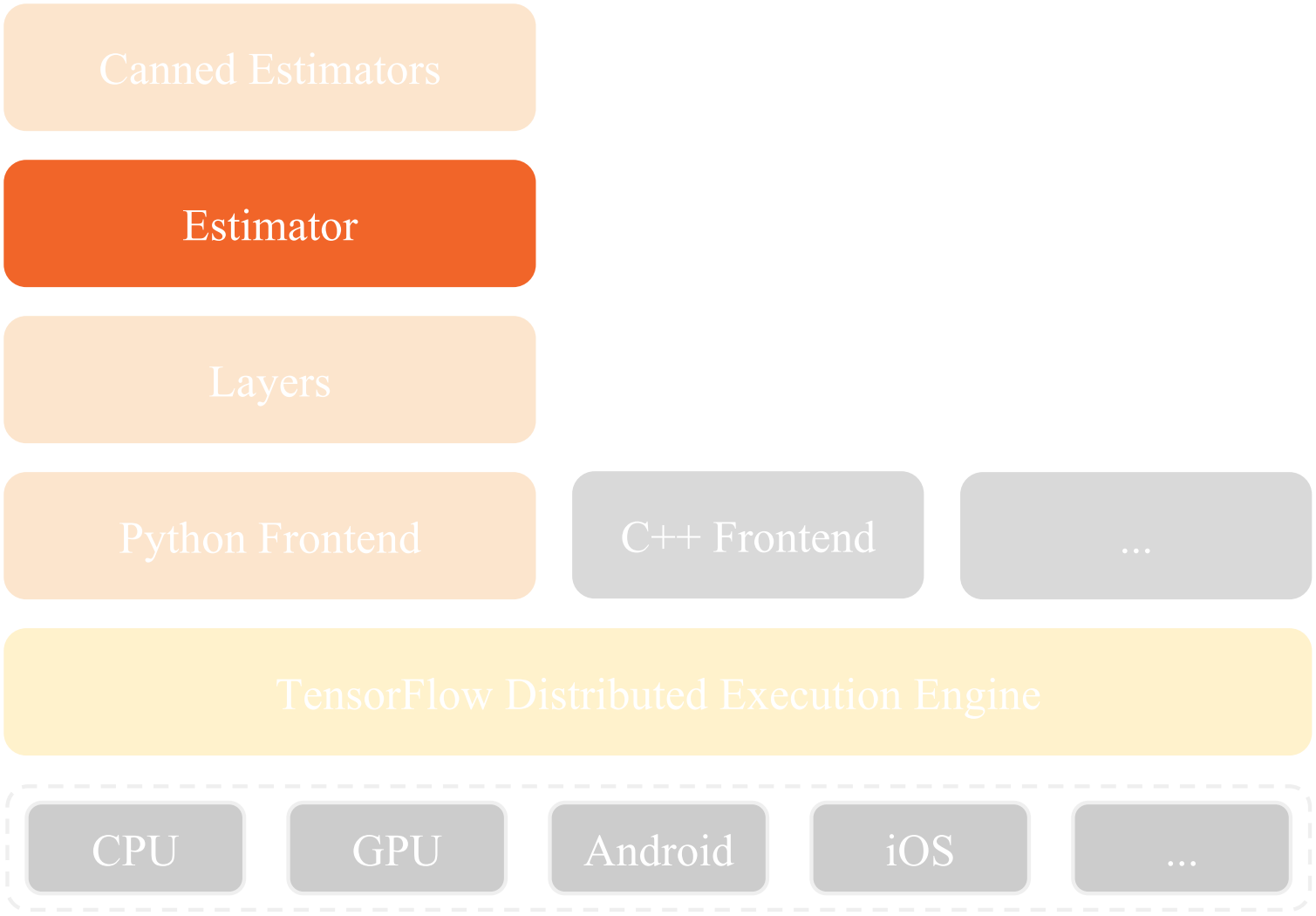

```
x = tf.layers.conv2d(x, kernel_size=[5,5], ...)
```

```
x = tf.layers.max_pooling2d(x, kernel_size=[2,2], ...)
```

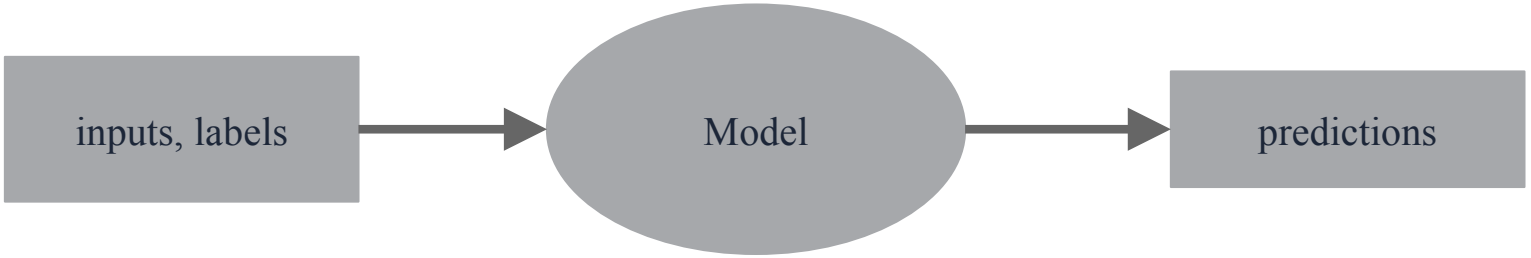
Layers



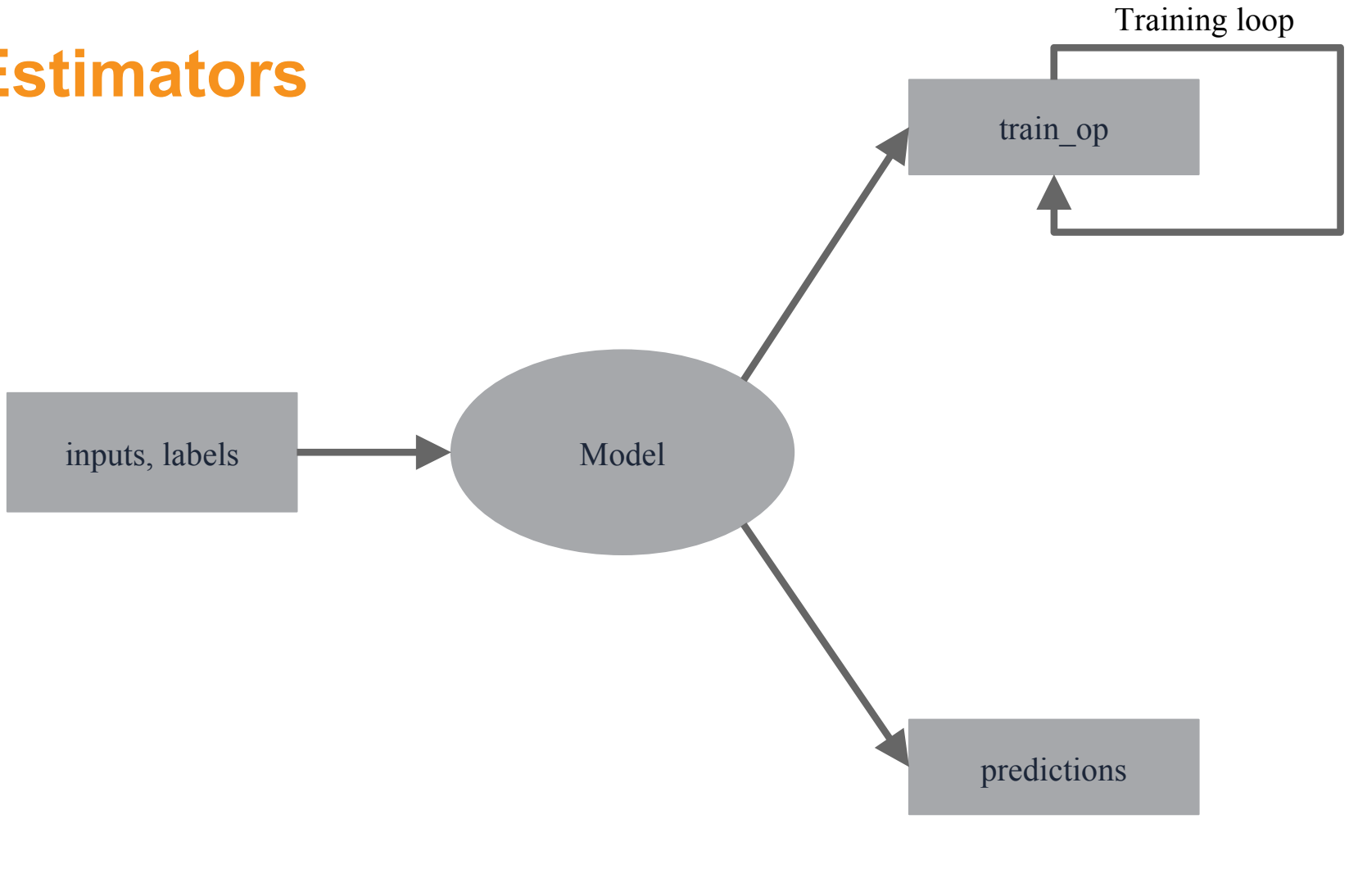

```
x = tf.layers.conv2d(x, kernel_size=[5,5], ...)
x = tf.layers.max_pooling2d(x, kernel_size=[2,2], ...)
x = tf.layers.conv2d(x, kernel_size=[5,5], ...)
x = tf.layers.max_pooling2d(x, kernel_size=[2,2], ...)
x = tf.layers.dense(x, activation_fn=tf.nn.relu)
x = tf.layers.dropout(x, 0.5)
x = tf.layers.dense(x)
```


Canned Estimators
Estimator
Layers
Python Frontend
C++ Frontend
...
TensorFlow Distributed Execution Engine
CPU
GPU
Android
iOS
...

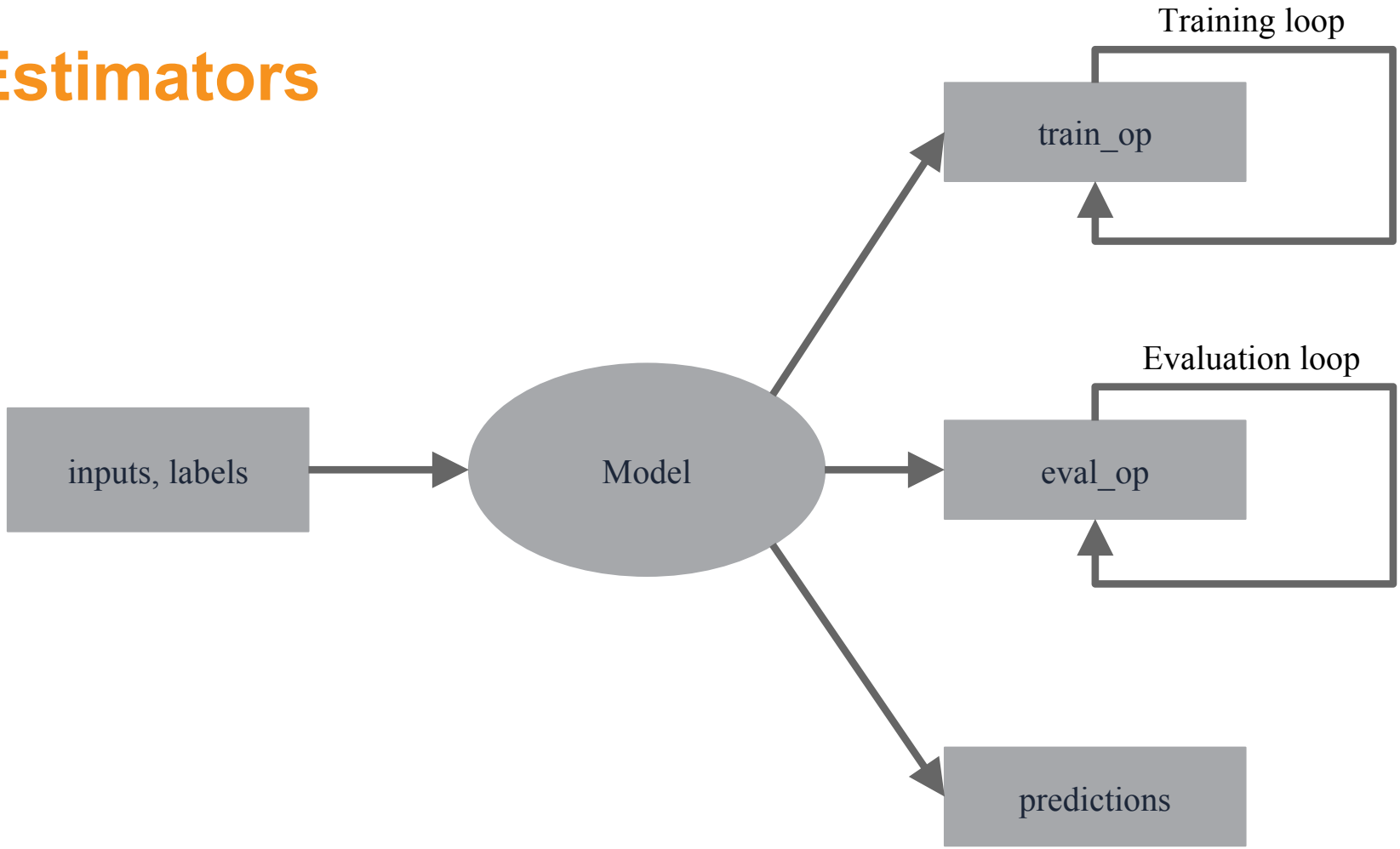
Estimators

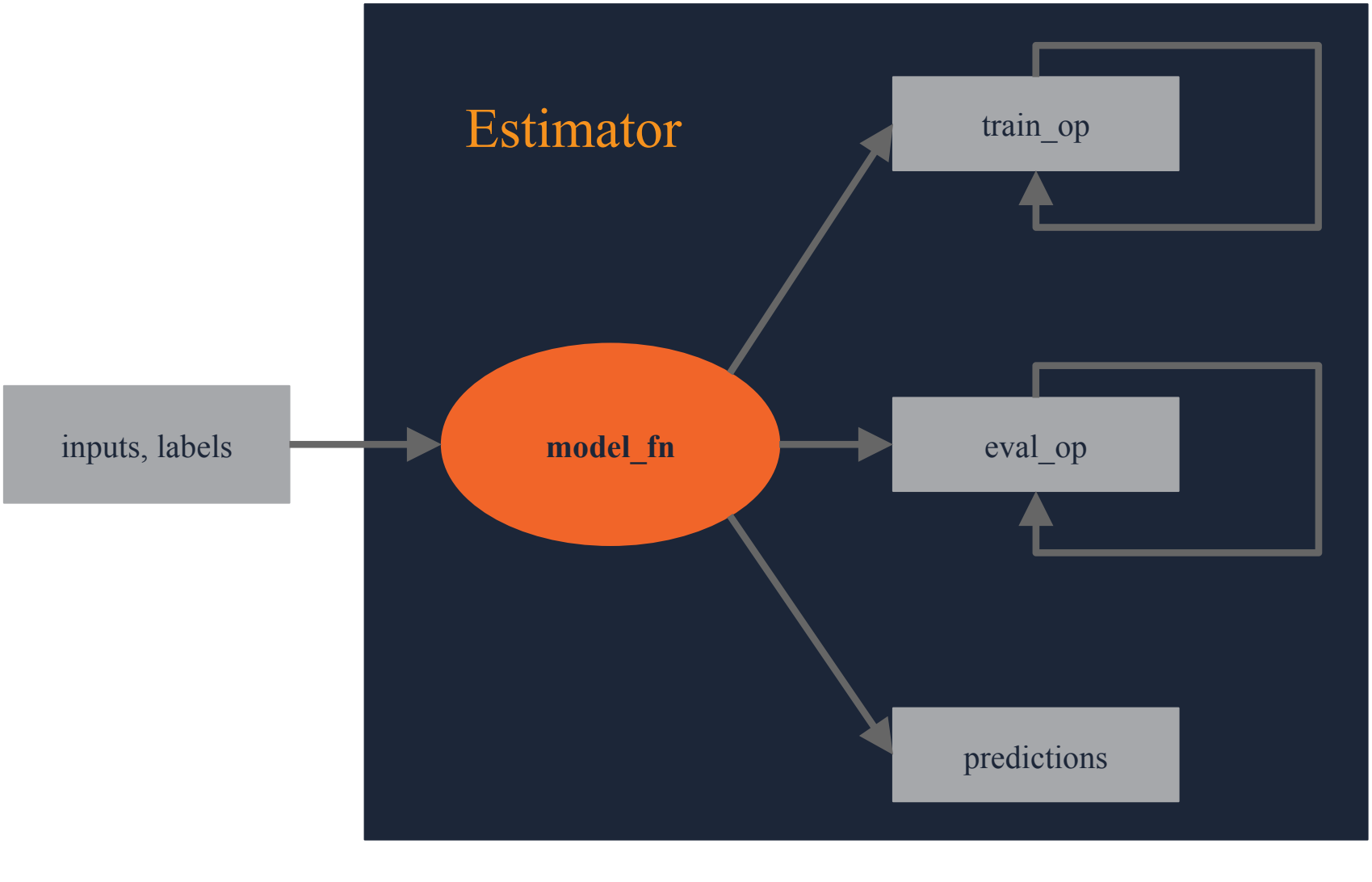


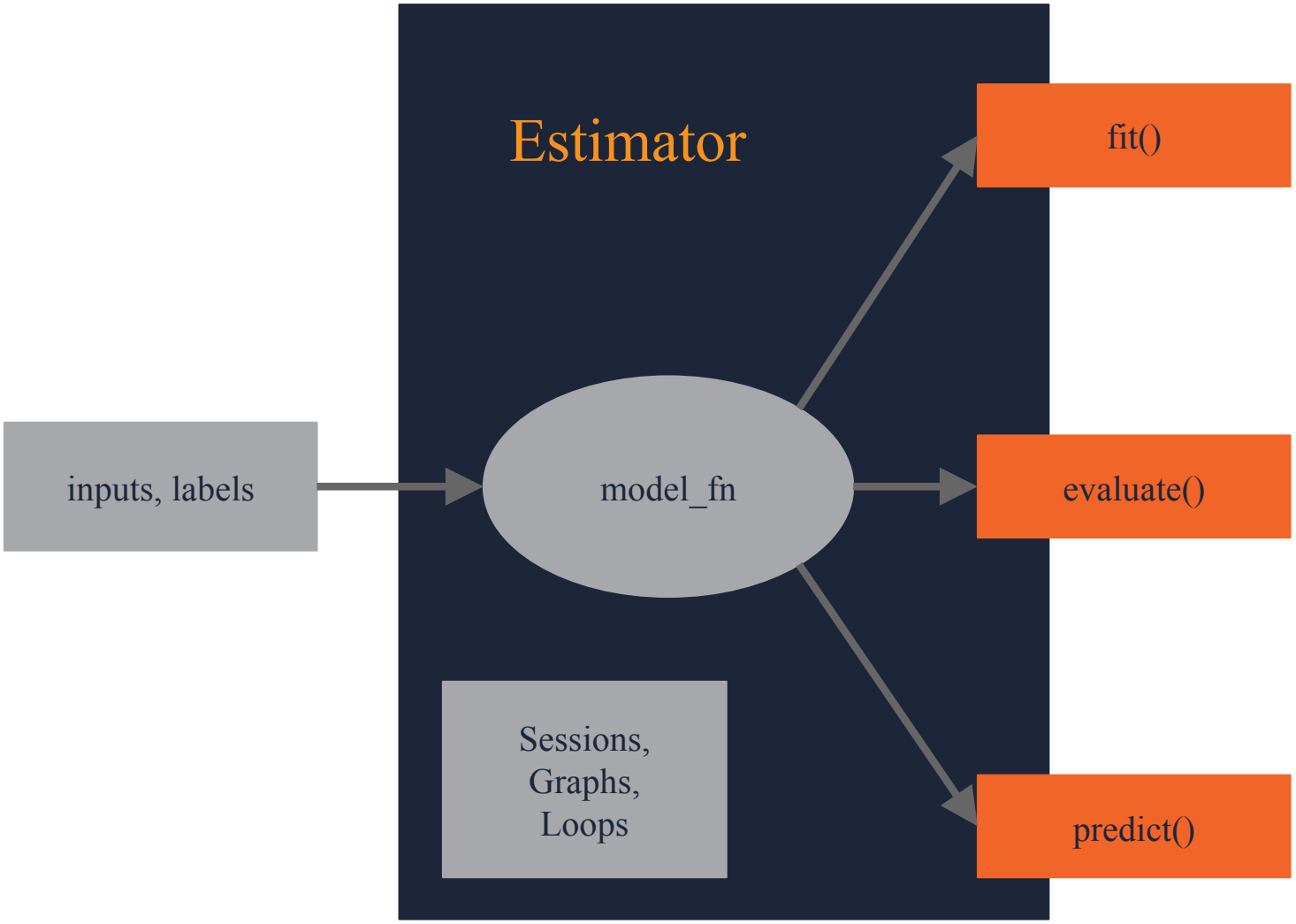
Estimators

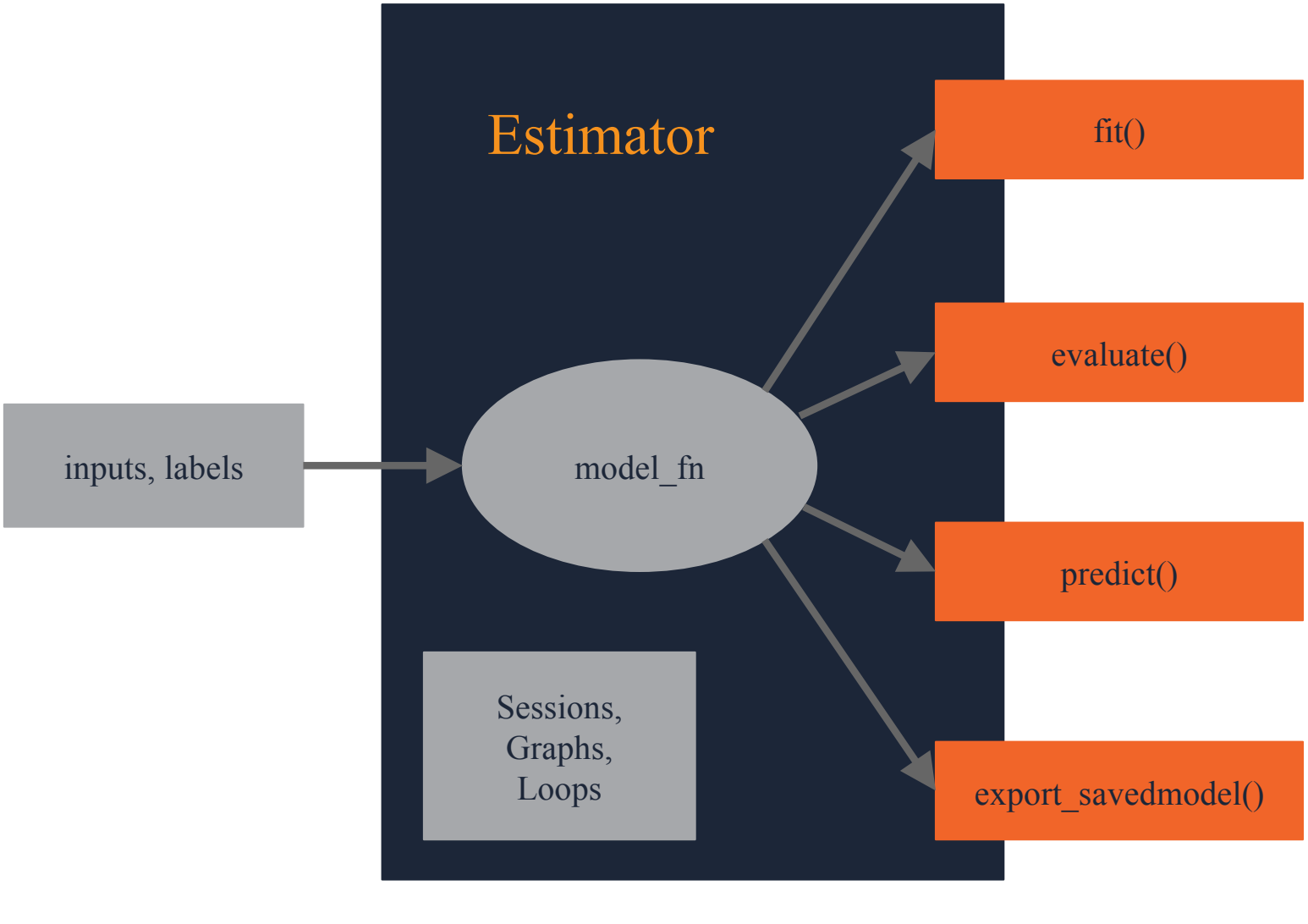


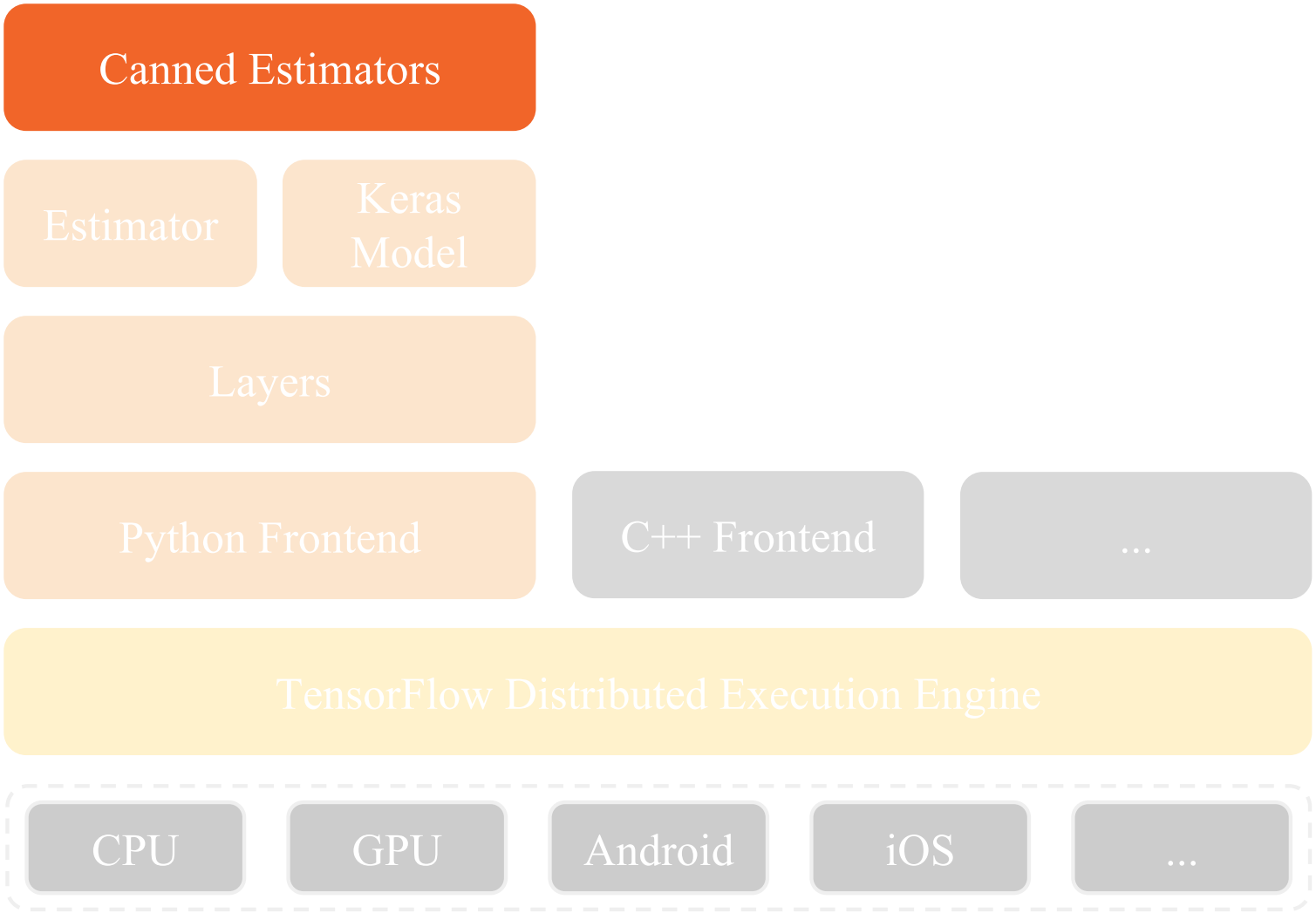
Estimators


Training loop
train_op
Evaluation loop
inputs, labels
Model
eval_op
predictions


Estimator
train_op
inputs, labels
model_fn
eval_op
predictions


Estimator
fit()
inputs, labels
model_fn
evaluate()
Sessions,
Graphs,
Loops
predict()


Estimator
fit()
evaluate()
predict()
export_savedmodel()
inputs, labels
model_fn
Sessions,
Graphs,
Loops


Canned Estimators
Estimator
Keras Model
Layers
Python Frontend
C++ Frontend
...
TensorFlow Distributed Execution Engine
CPU
GPU
Android
iOS
...

Canned Estimators

```
area = real_valued_column(“square_foot”),
rooms = real_valued_column(“num_rooms”),
zip_code = sparse_column_with_integerized_feature(“zip_code”, 100000)


regressor = LinearRegressor(feature_columns=[area, rooms, zip_code])


regressor.fit(train_input_fn)

regressor.evaluate(eval_input_fn)
```

Canned Estimators

```
area = real_valued_column("square_foot"),
rooms = real_valued_column("num_rooms"),
zip_code = sparse_column_with_integerized_feature("zip_code", 100000)


regressor = LinearRegressor(feature_columns=[area, rooms, zip_code])


regressor.fit(train_input_fn)

regressor.evaluate(eval_input_fn)
```

Canned Estimators

```
area = real_valued_column(“square_foot”),
rooms = real_valued_column(“num_rooms”),
zip_code = sparse_column_with_integerized_feature(“zip_code”, 100000)


regressor = DNNRegressor(
    feature_columns=[area, rooms, embedding_column(zip_code, 8)],
    hidden_units=[1024, 512, 256])

regressor.fit(train_input_fn)

regressor.evaluate(eval_input_fn)
```

Canned Estimators

```
area = real_valued_column(“square_foot”),
rooms = real_valued_column(“num_rooms”),
zip_code = sparse_column_with_integerized_feature(“zip_code”, 100000)


regressor = DNNRegressor(
    feature_columns=[area, rooms, embedding_column(zip_code, 8)],
    hidden_units=[1024, 512, 256])

regressor.fit(train_input_fn)

regressor.evaluate(eval_input_fn)
```

Distributed Execution

Experiment Turnaround Time and Research Productivity

Minutes, Hours:

Interactive research! Instant gratification!

1-4 days

Tolerable

Interactivity replaced by running many experiments in parallel

1-4 weeks

High value experiments only

Transition

How do you do this at scale?

How does TensorFlow make distributed training easy?

Model Parallelism

Best way to decrease training time:

decrease the step time

Many models have lots of inherent parallelism

Problem is distributing work so communication doesn't kill you

- local connectivity (as found in CNNs)
- towers with little or no connectivity between towers (e.g. AlexNet)
- specialized parts of model active only for some examples

Exploiting Model Parallelism

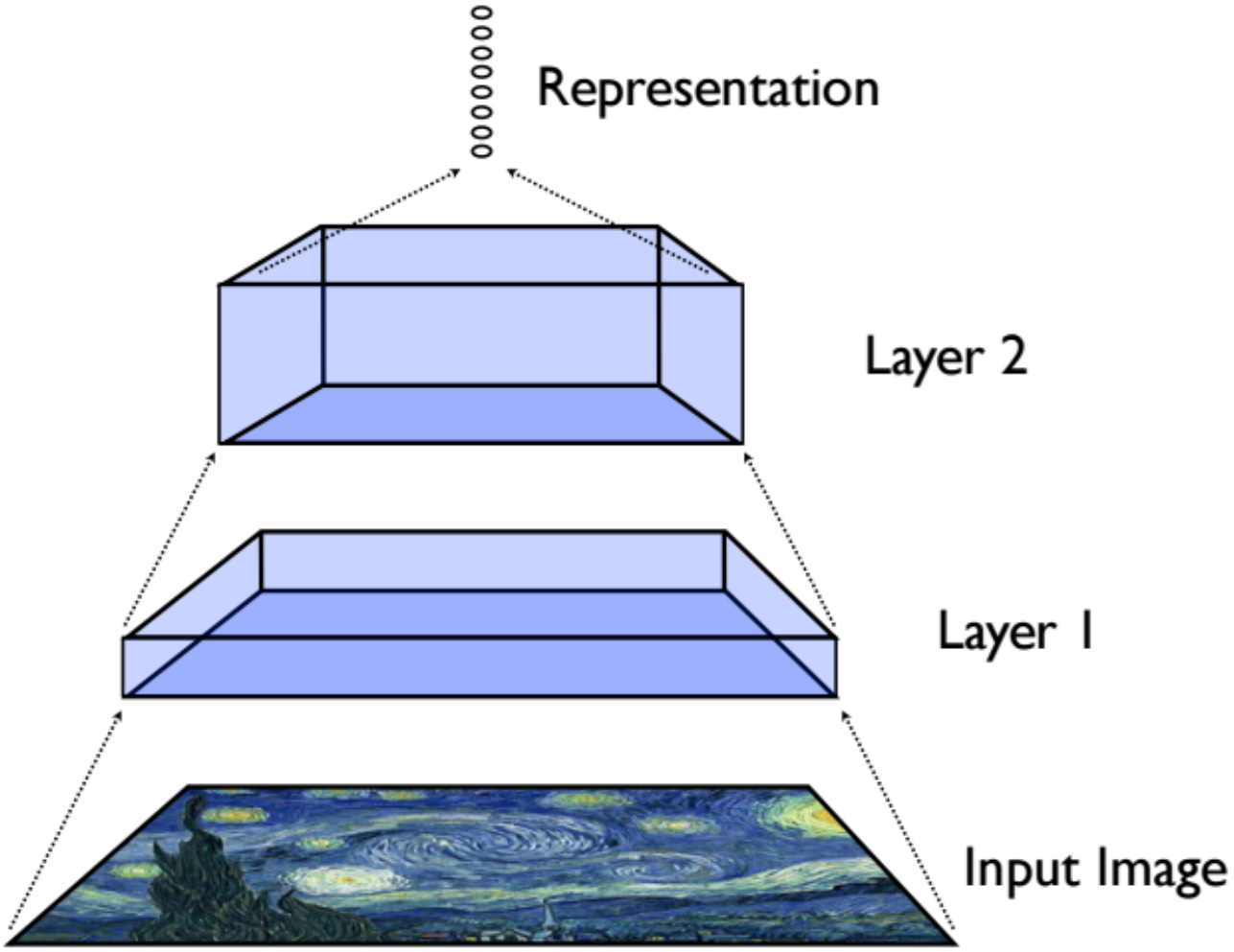
On a single core: Instruction parallelism (SIMD). Pretty much free.

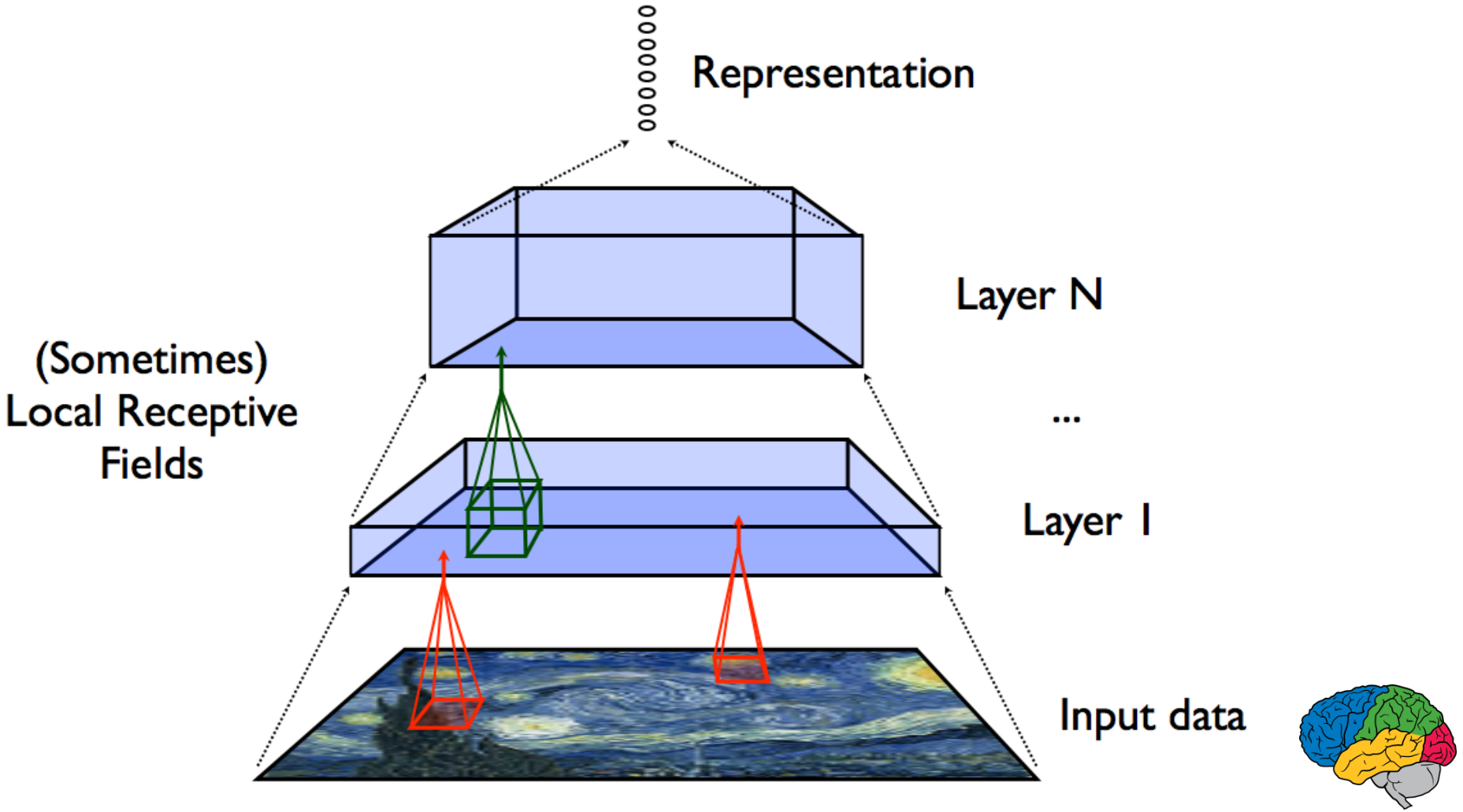
Across cores: thread parallelism. Almost free, unless across sockets, in which case inter-socket bandwidth matters.

Across devices: for GPUs, often limited by PCIe bandwidth.

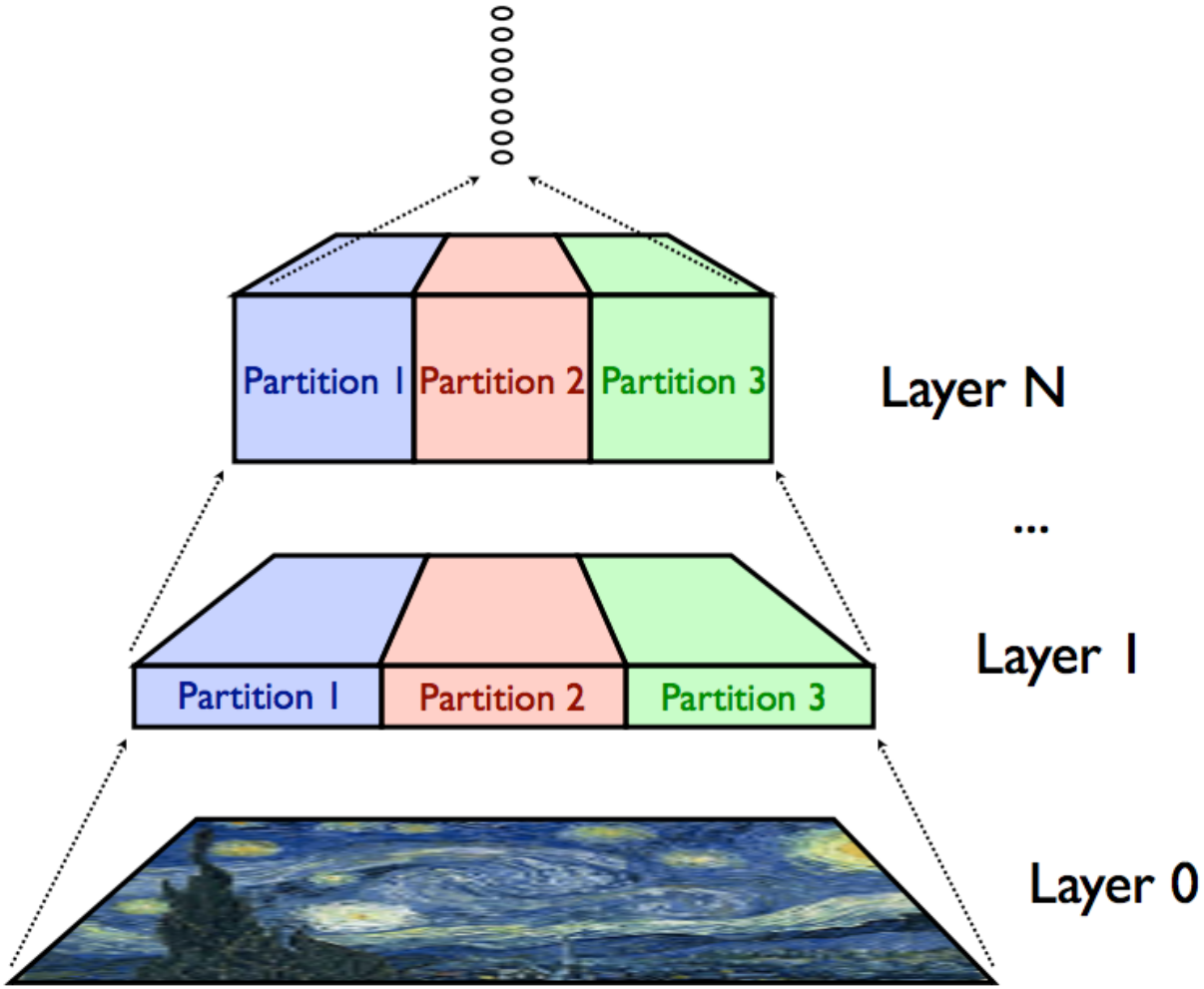
Across machines: limited by network bandwidth / latency.

Model Parallelism

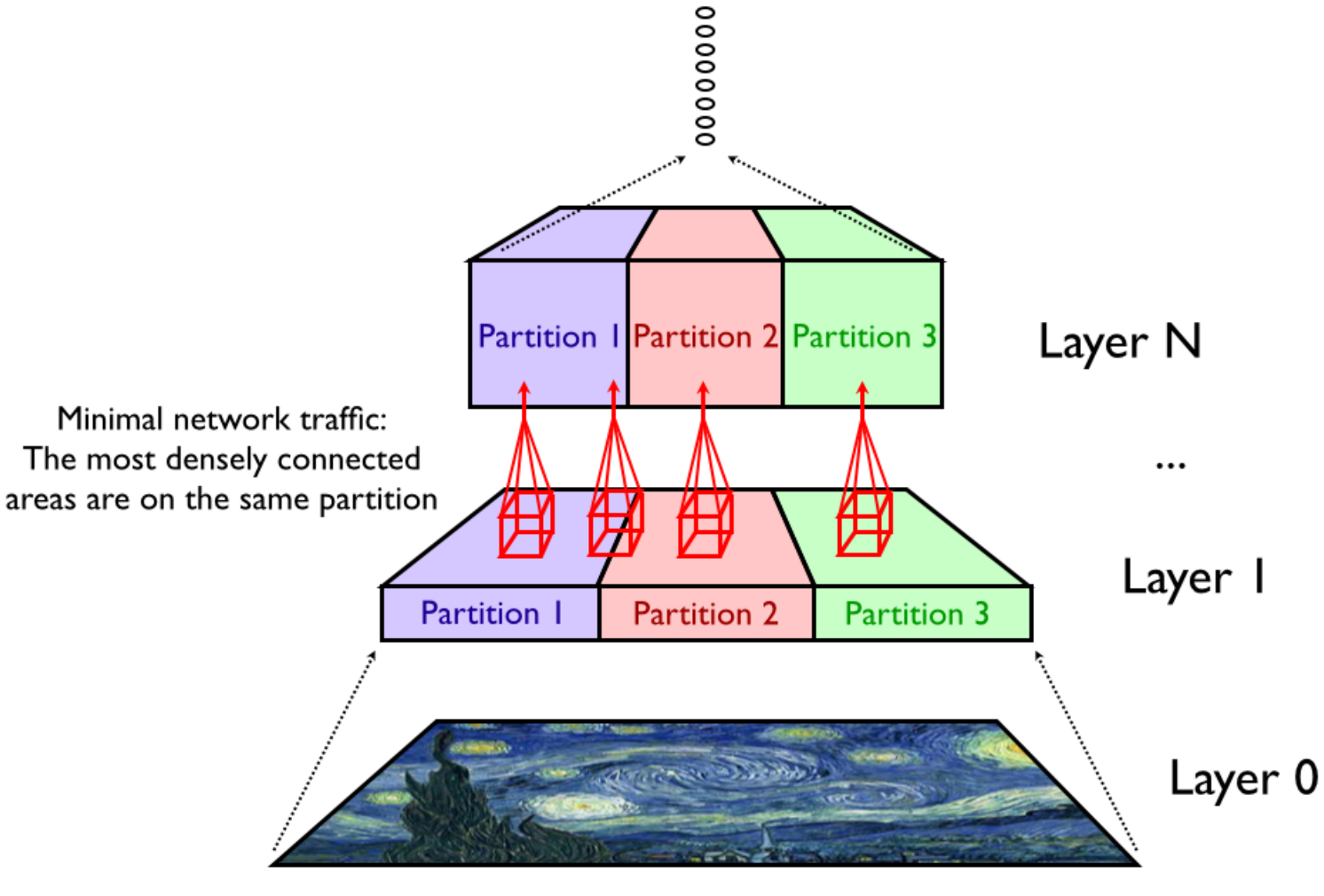



Representation
Layer N
...
Layer 1
Input data
(Sometimes)
Local Receptive
Fields

Model Parallelism: Partition model across machines



Model Parallelism: Partition model across machines



Data Parallelism

Use multiple model replicas to process different examples at the same time

All collaborate to update model state (parameters) in shared parameter server(s)

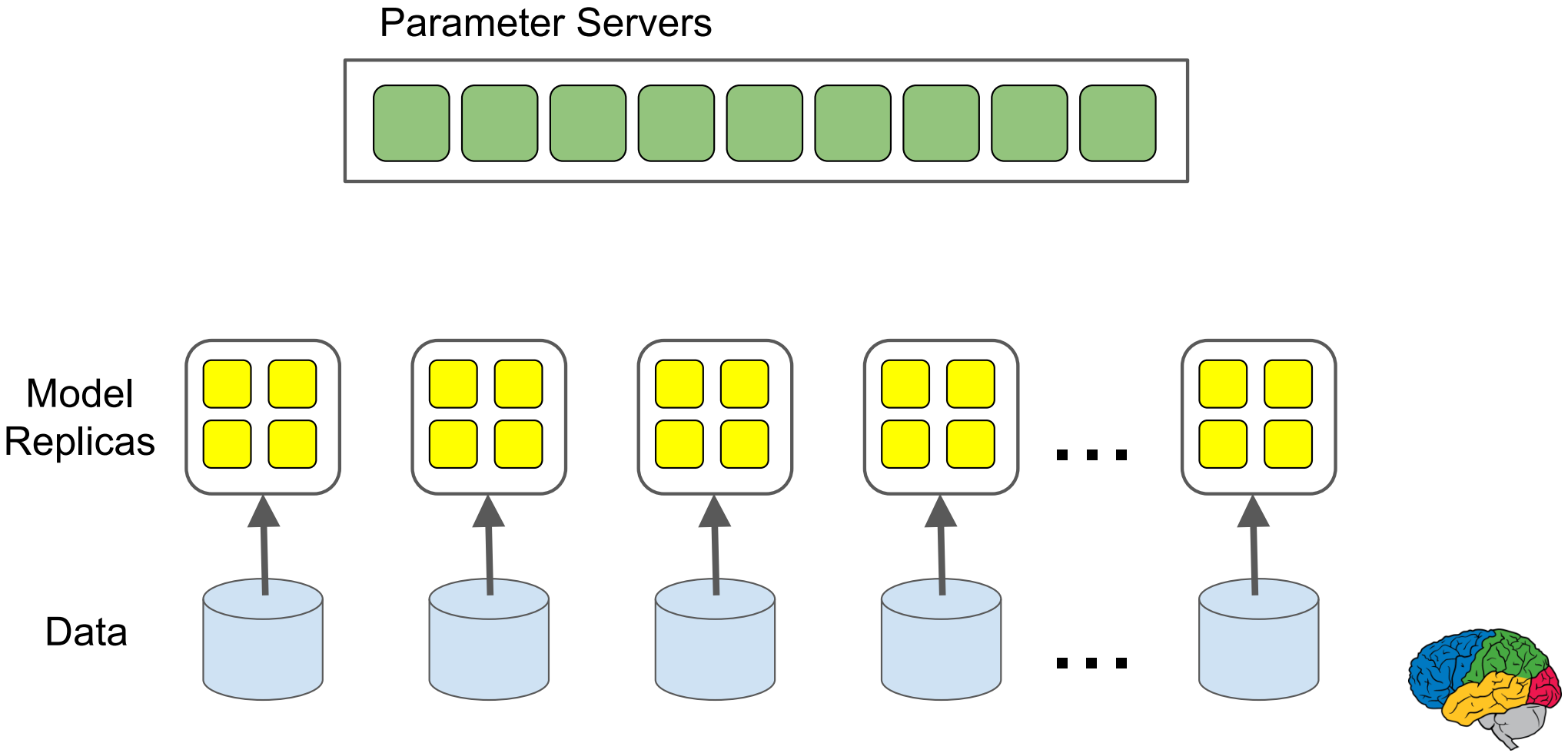
Speedups depend highly on kind of model

Dense models: 10-40X speedup from 50 replicas

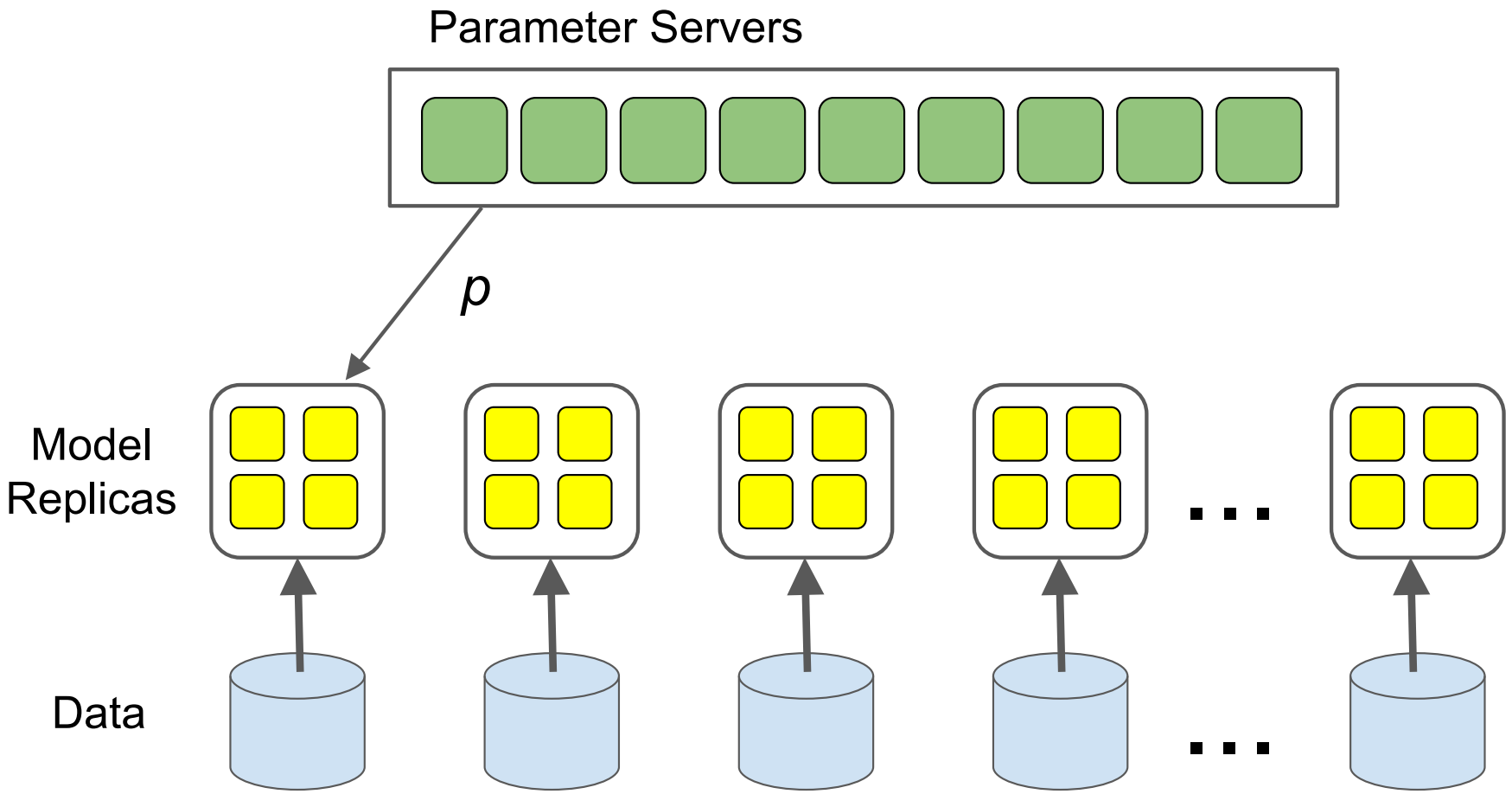
Sparse models:

support many more replicas

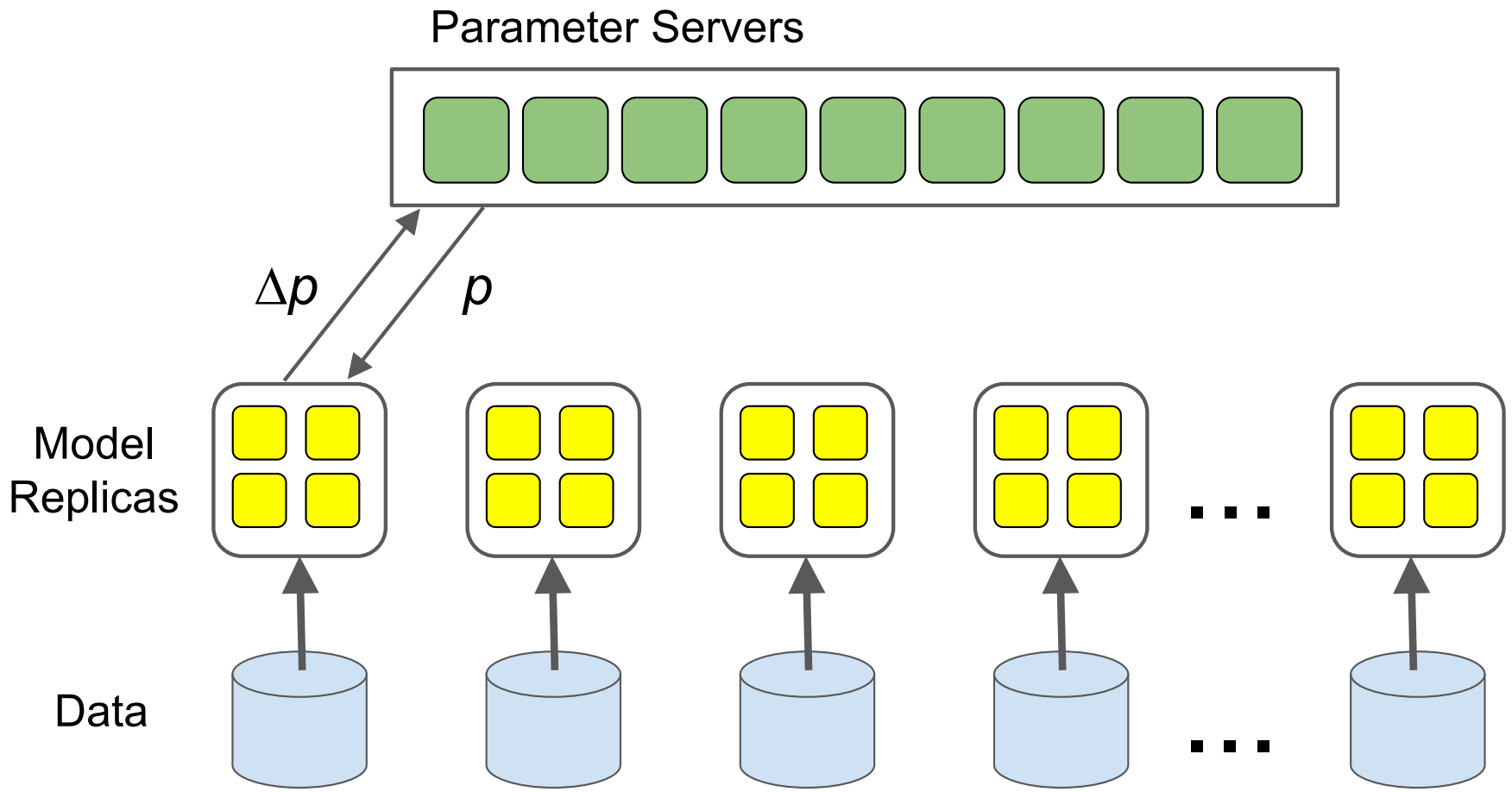
Data Parallelism



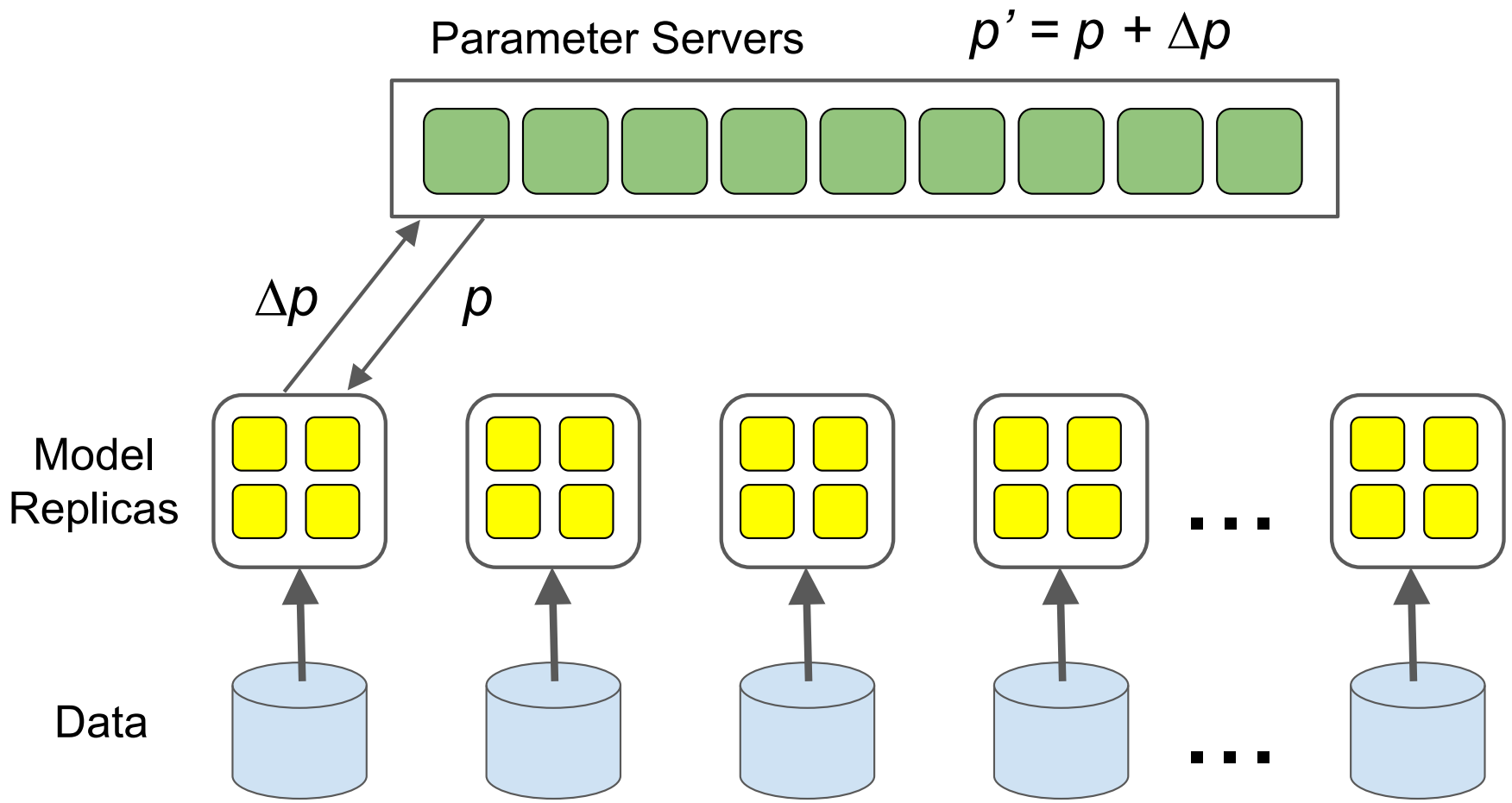
Data Parallelism



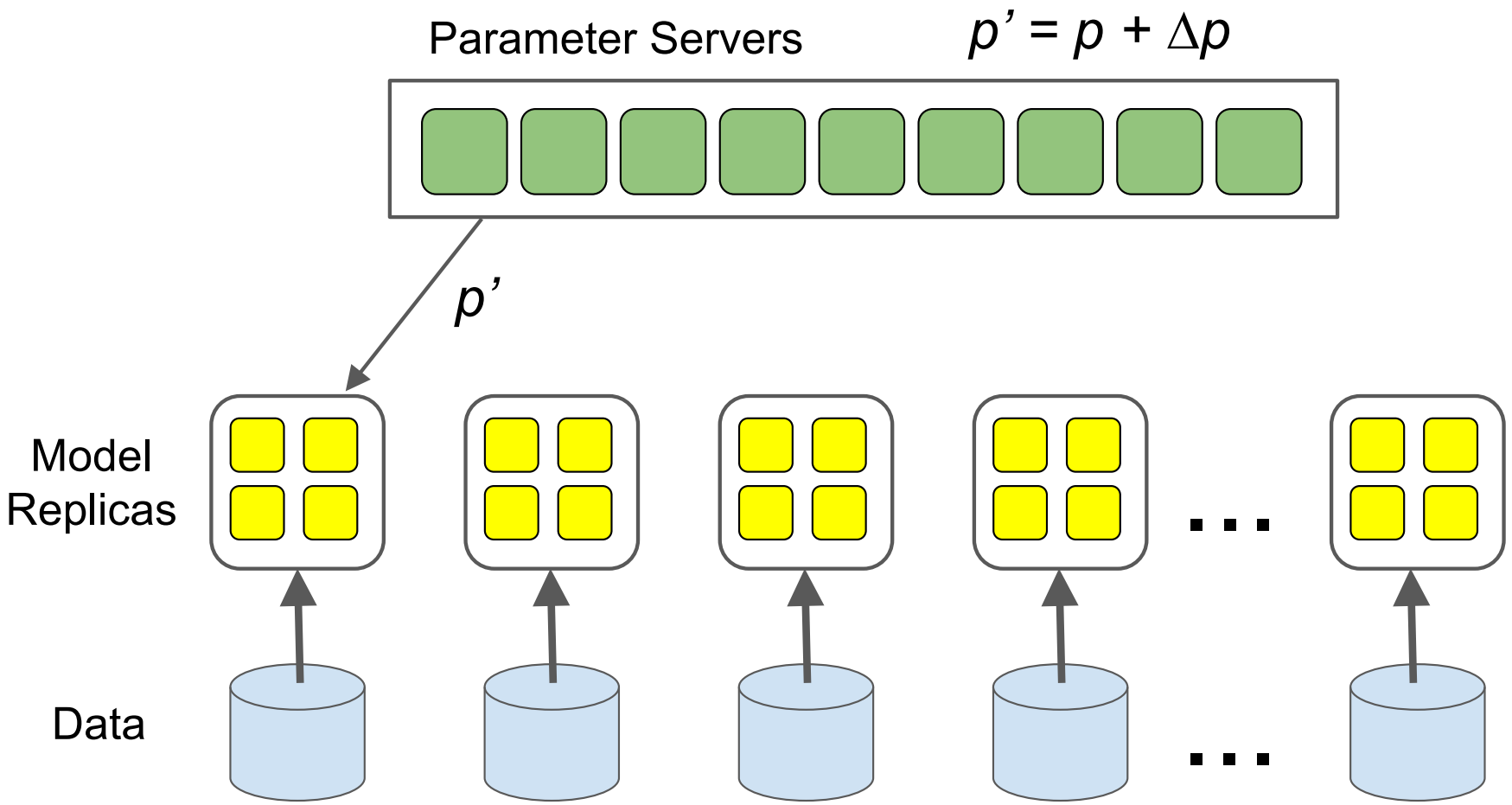
Data Parallelism



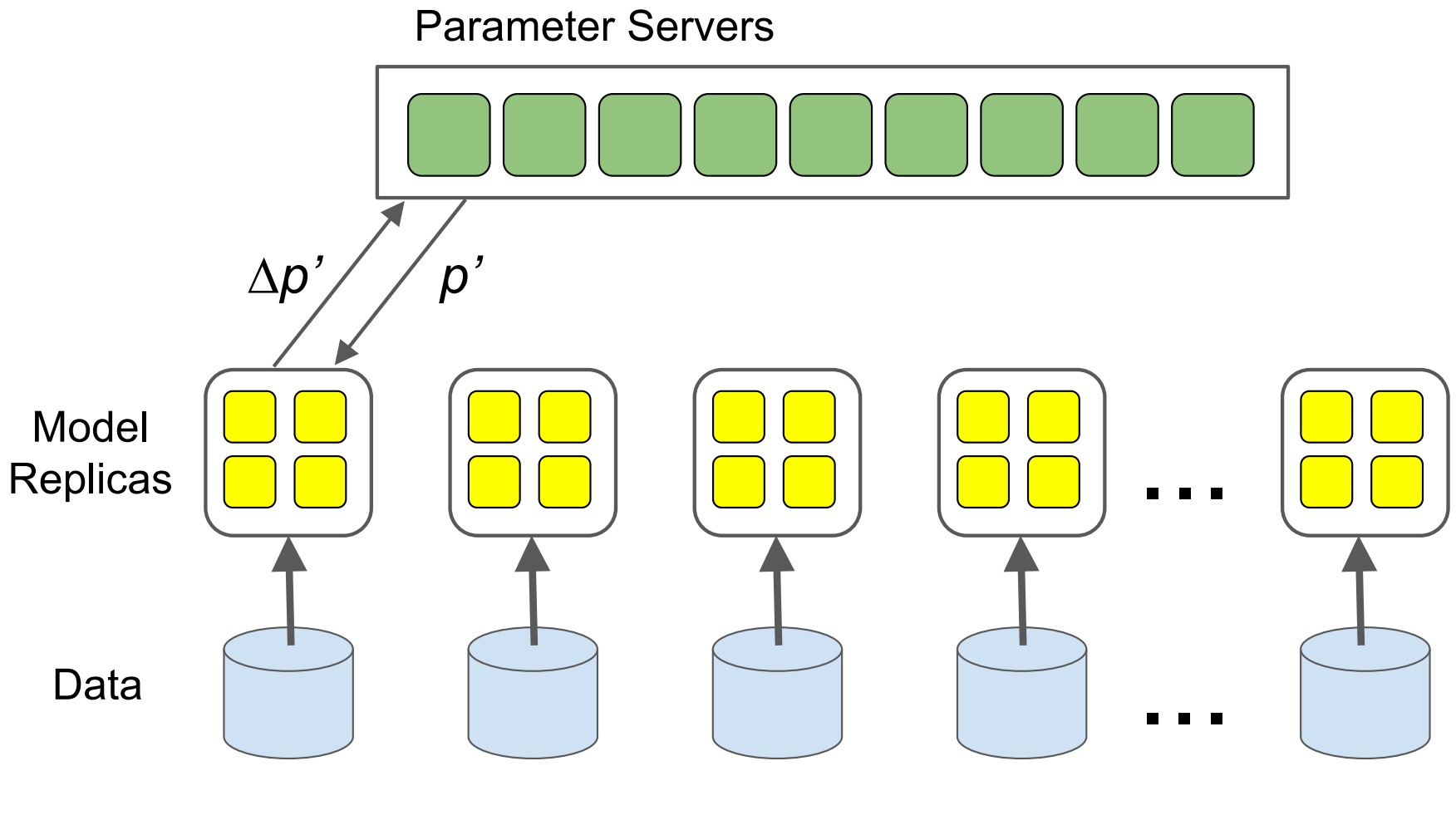
Data Parallelism



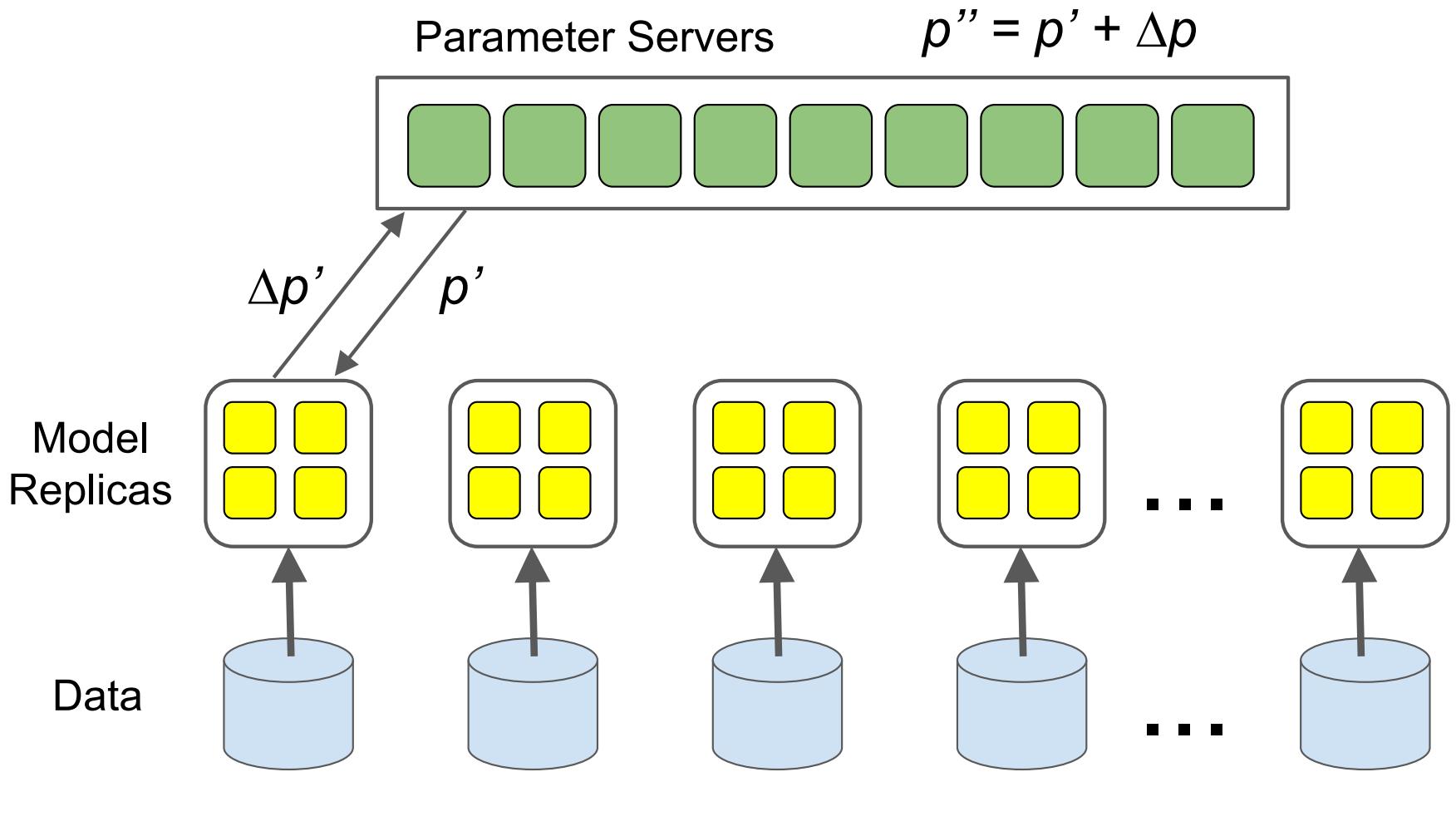
Data Parallelism



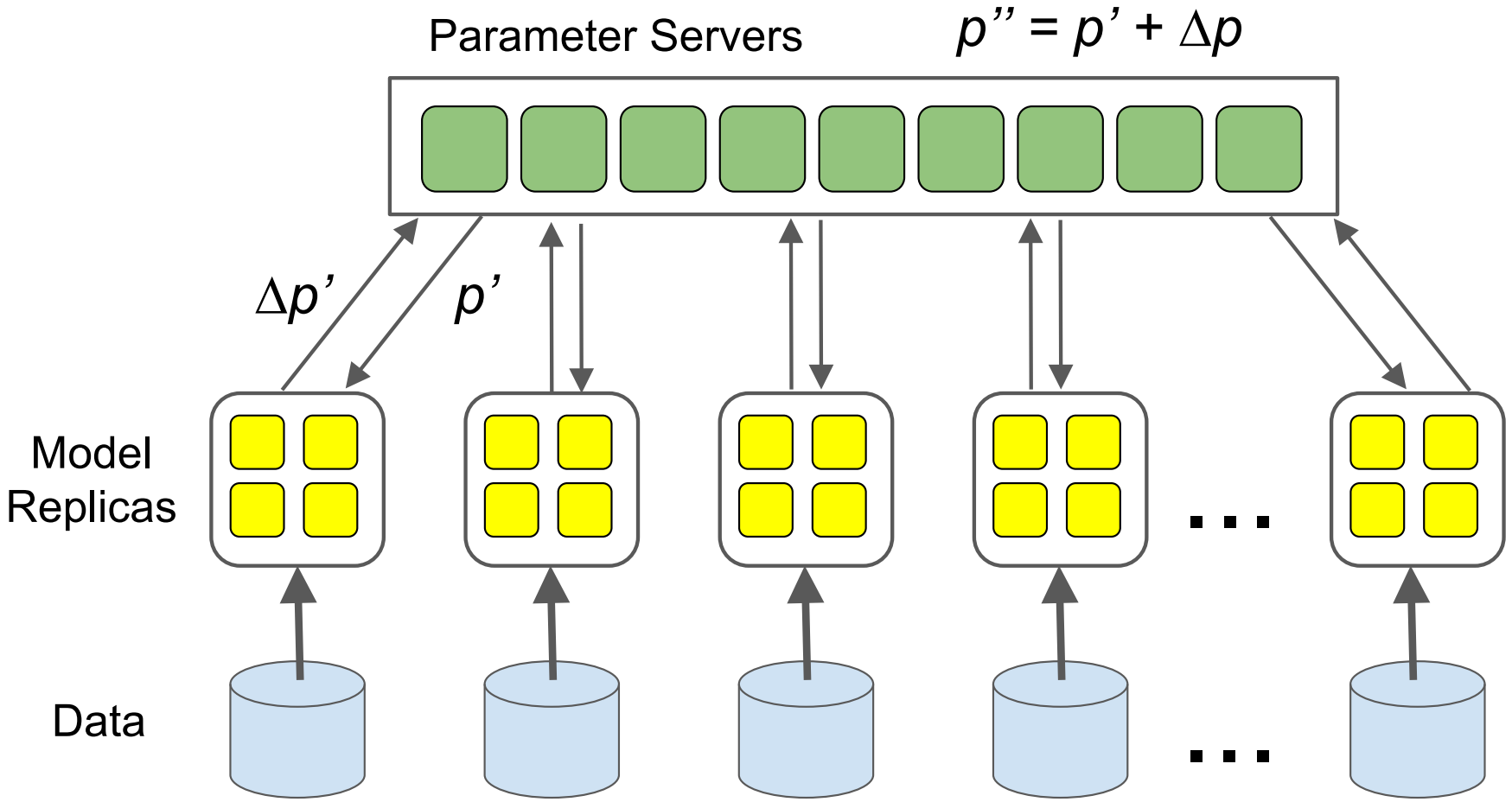
Data Parallelism



Data Parallelism



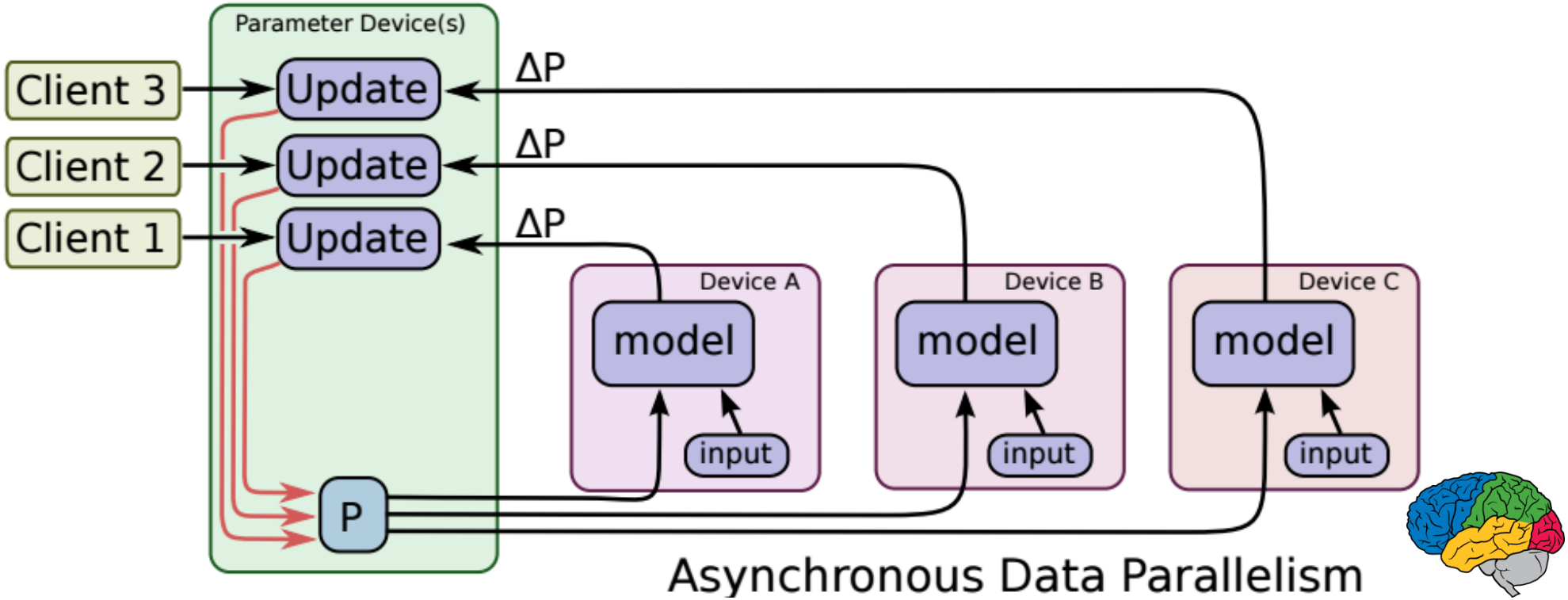
Data Parallelism



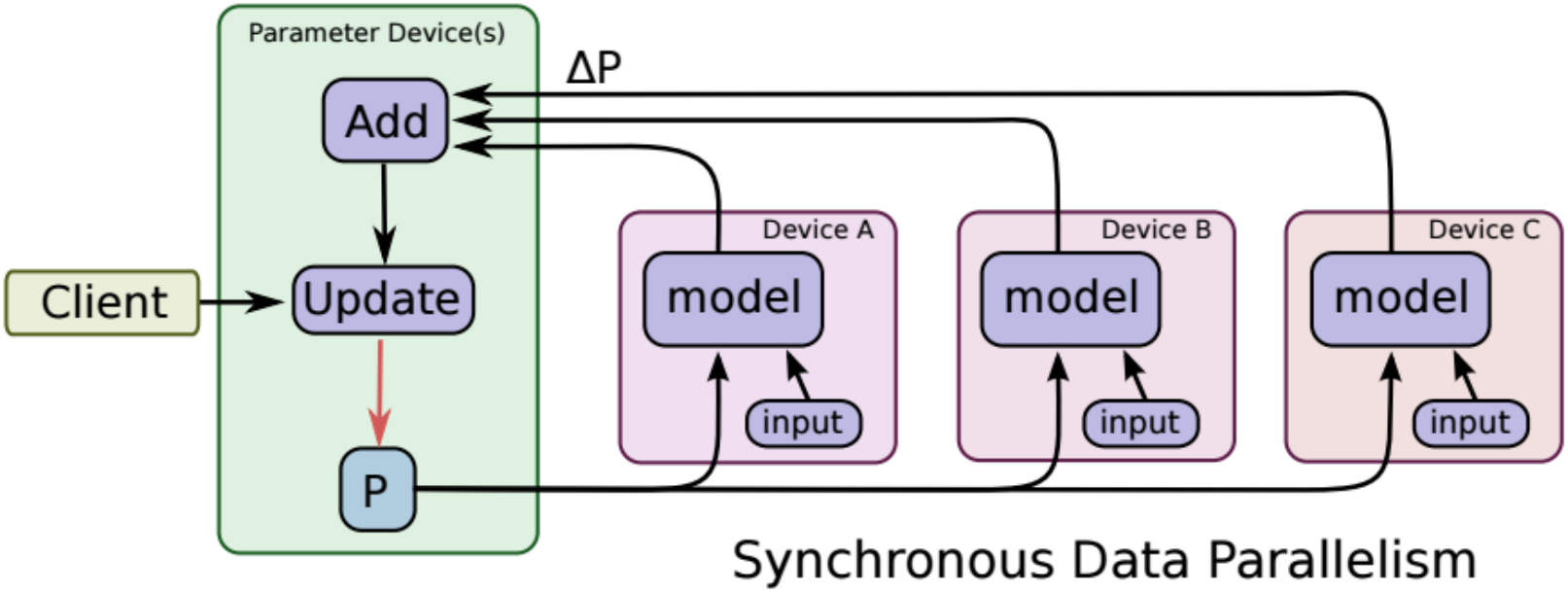
Asynchronous Training

No separate parameter server system:

Parameters are just stateful nodes in the graph



Synchronous Variant



Data Parallelism Choices

Can do this **synchronously**:

- **N replicas** equivalent to an **N times larger batch size**
- Pro: No gradient staleness
- Con: Less fault tolerant (requires some recovery if any single machine fails)

Can do this **asynchronously**:

- Pro: Relatively fault tolerant (failure in model replica doesn't block other replicas)
- Con: Gradient staleness means each gradient less effective

Data Parallelism Considerations

Want model computation time to be large relative to time to send/receive parameters over network

Models with fewer parameters, that reuse each parameter multiple times in the computation

- Mini-batches of size *B* reuse parameters *B* times

Certain model structures **reuse each parameter** many times within each example:

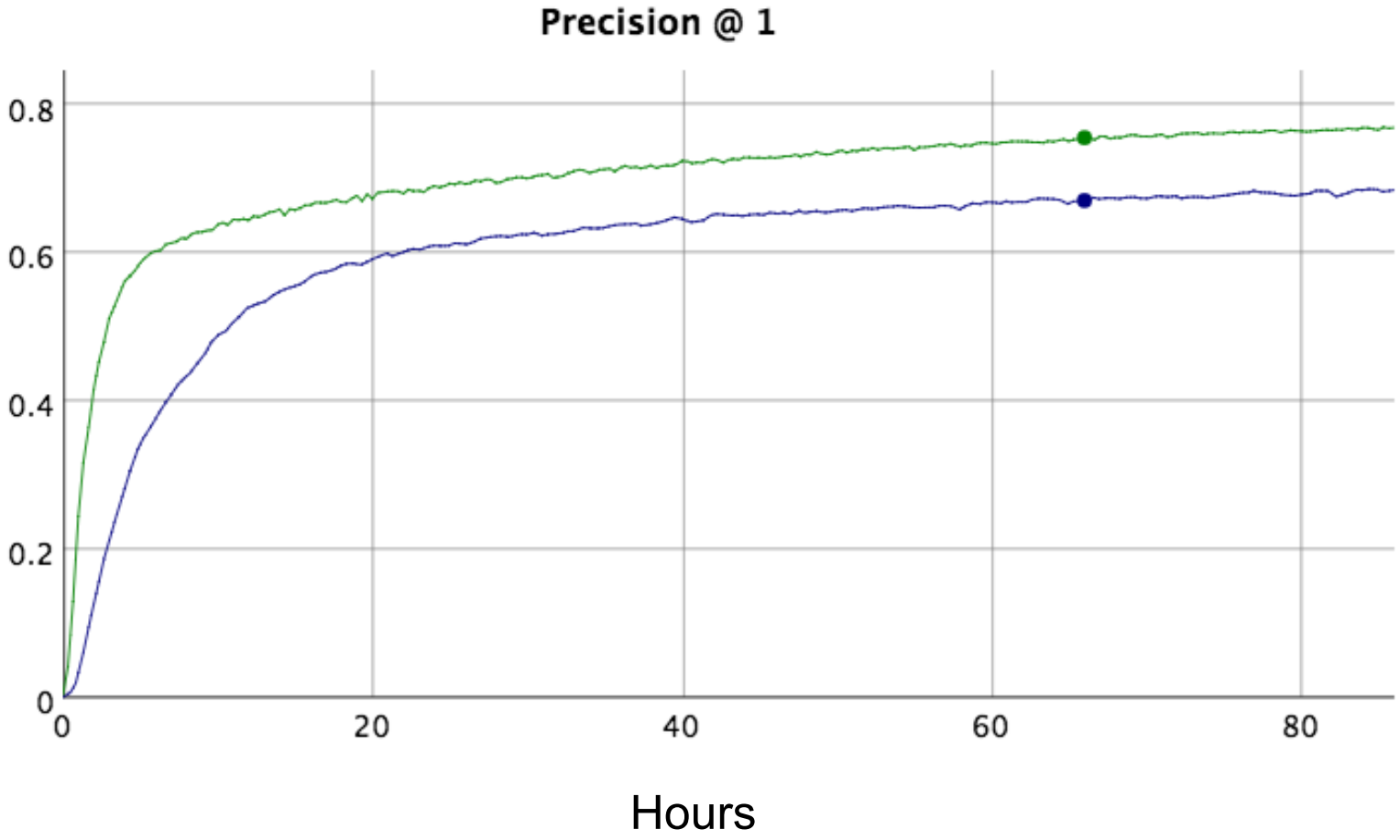
- **Convolutional models** tend to reuse hundreds or thousands of times per example (for different spatial positions)
- **Recurrent models** (LSTMs, RNNs) tend to reuse tens to hundreds of times (for unrolling through *T* time steps during training)

Success of Data Parallelism

Data parallelism is **really important** for many of Google's problems (very large datasets, large models):

- RankBrain uses 500 replicas
- ImageNet Inception training uses 50 GPUs, ~40X speedup
- SmartReply uses 16 replicas, each with multiple GPUs
- State-of-the-art on LM "One Billion Word" Benchmark model uses both data and model parallelism on 32 GPUs

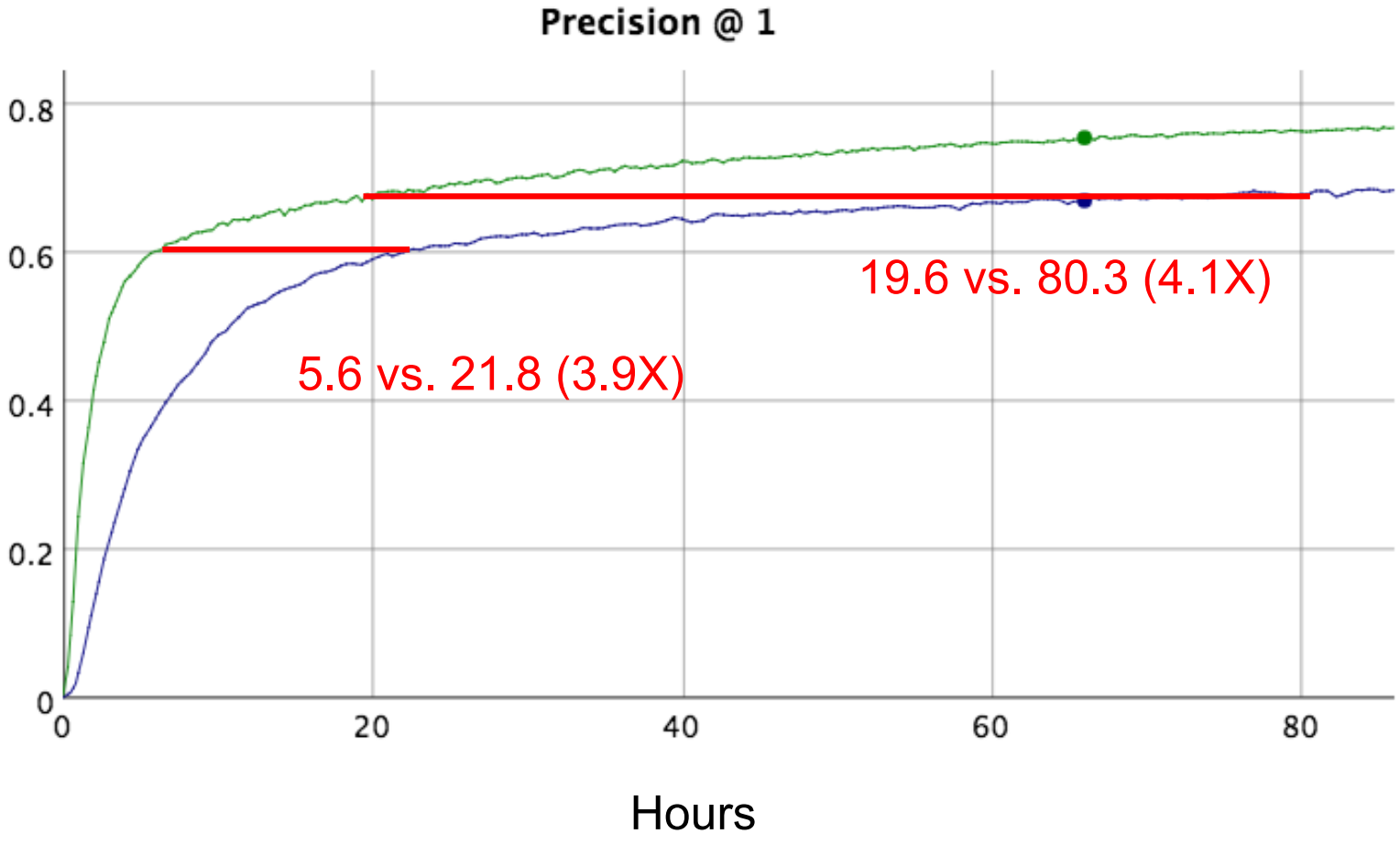
10 vs 50 Replica Inception Synchronous Training



50 replicas

10 replicas

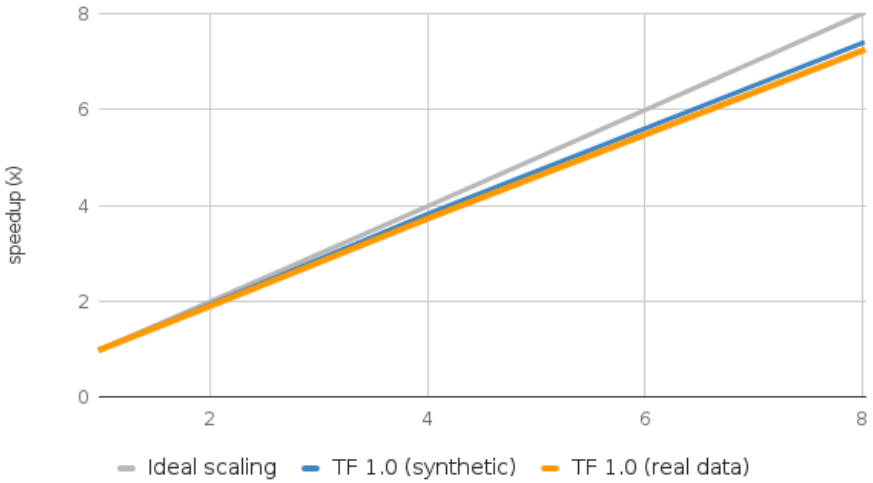
10 vs 50 Replica Inception Synchronous Training



Inception-v3 Training

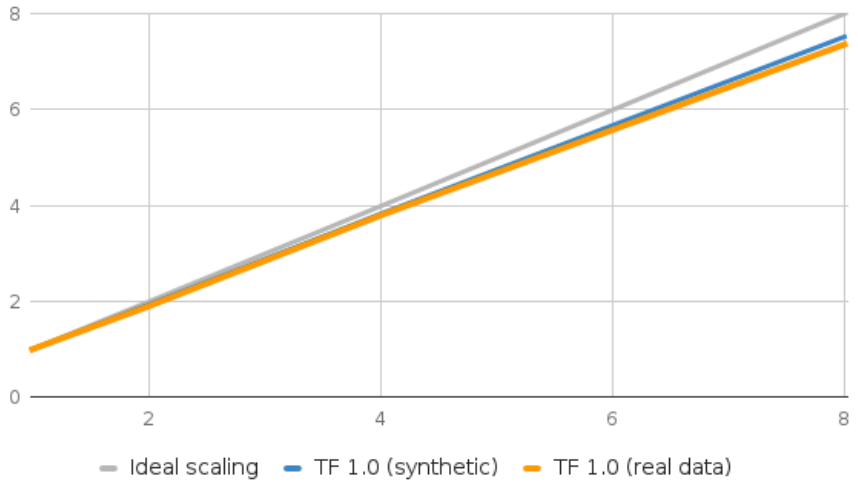
NVIDIA® DGX-1™

7.2x speedup at 8 GPUs



NVIDIA® Telsa® K80

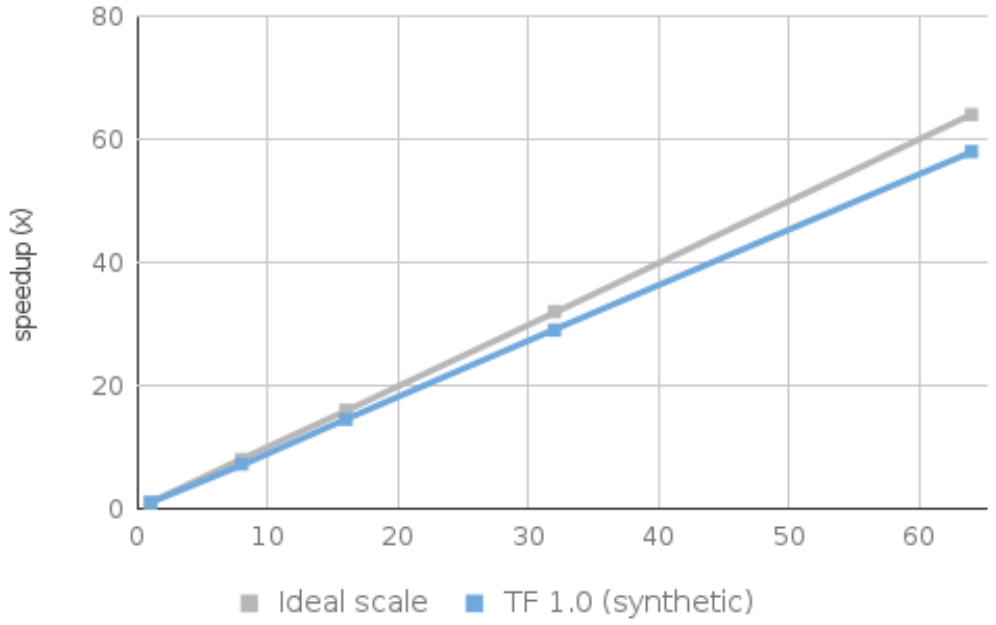
7.3x speedup at 8 GPUs



Inception-v3 Distributed Training

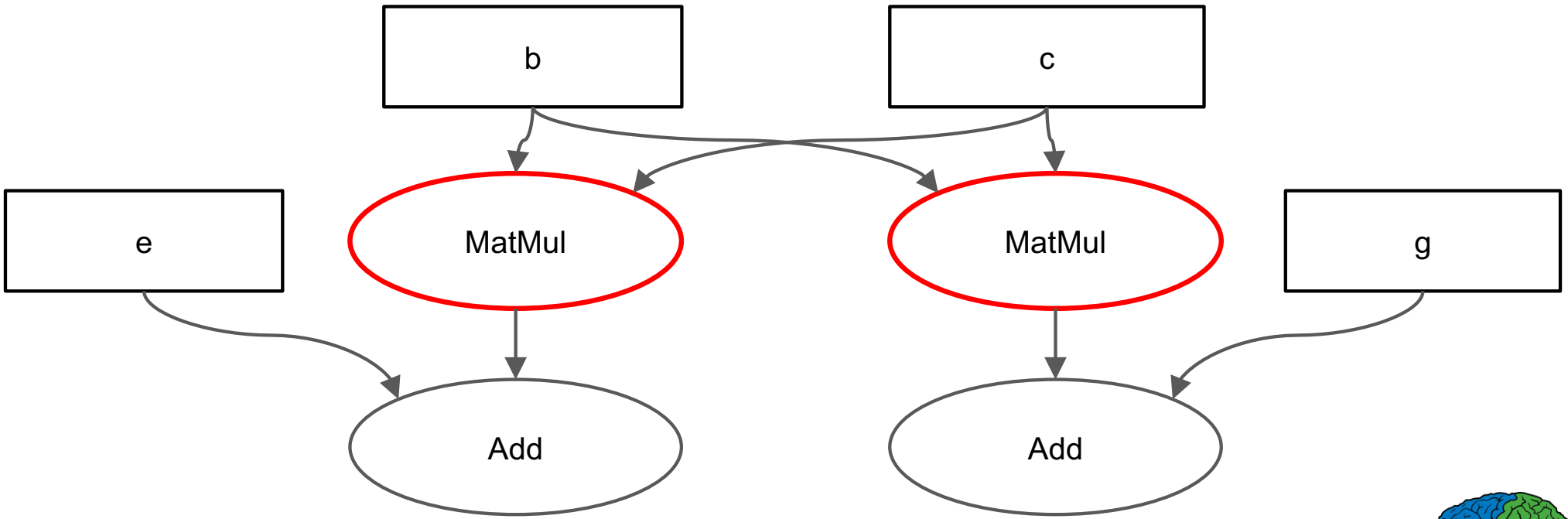
58x speedup at 64 GPUs (8 Servers / 8 GPUs each)

- GPU: Tesla K80
- Network: 20 Gb/sec

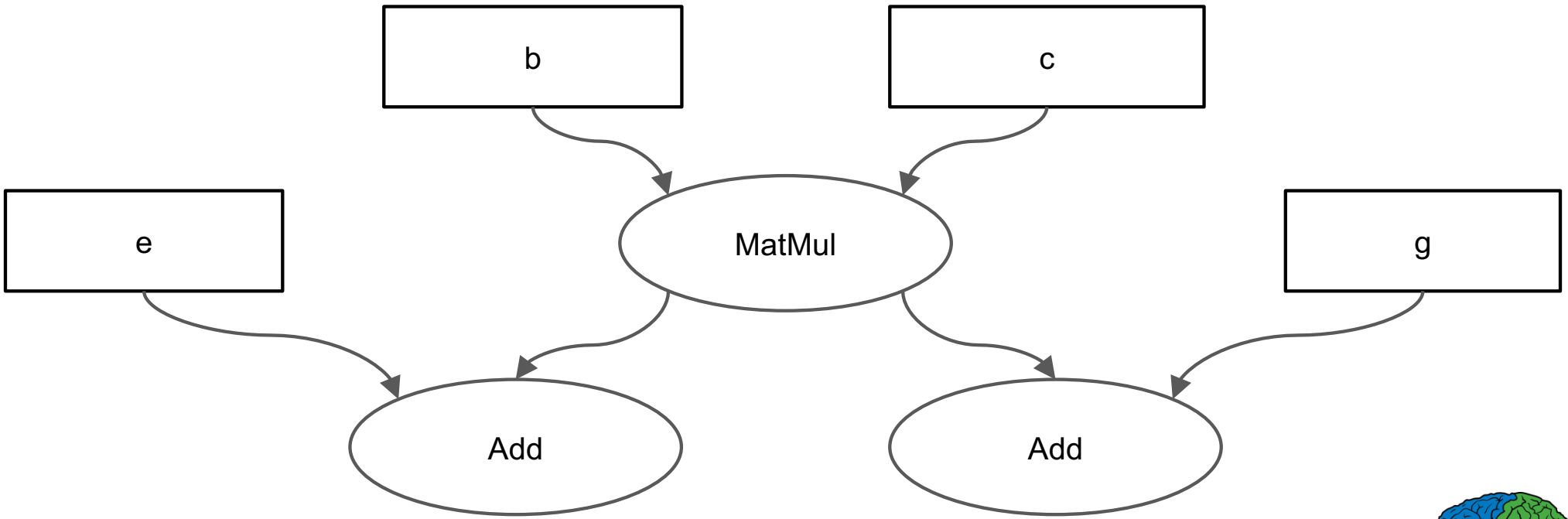


Runtime Optimizations

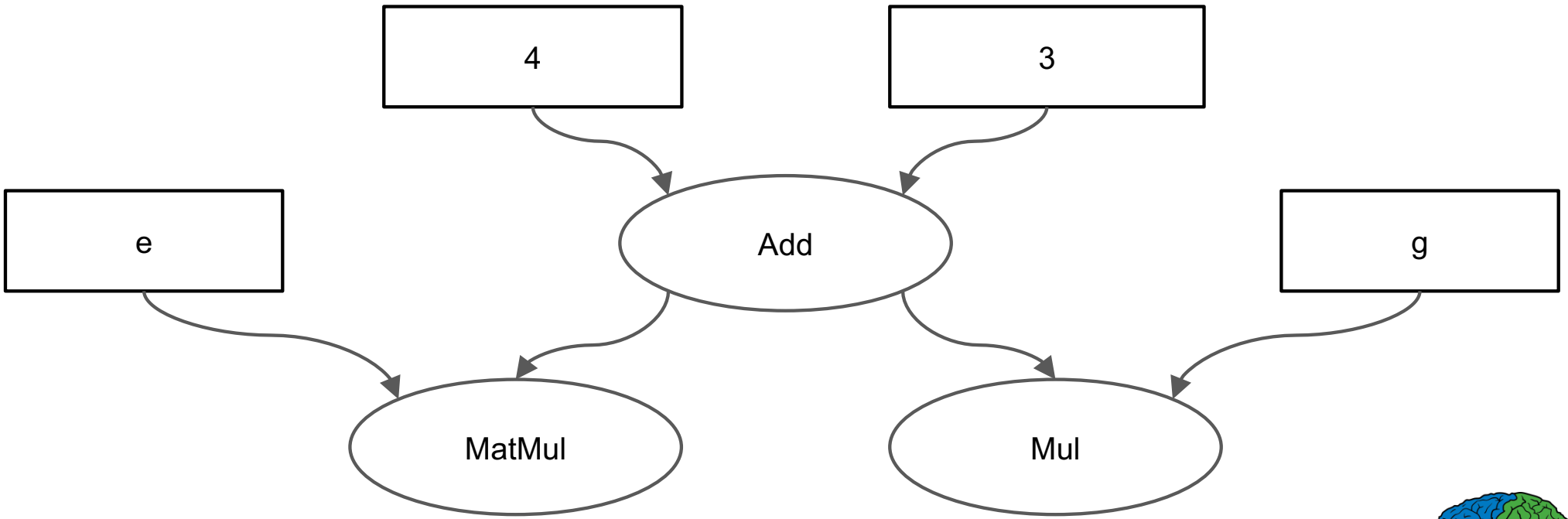
Common Subexpression Elimination



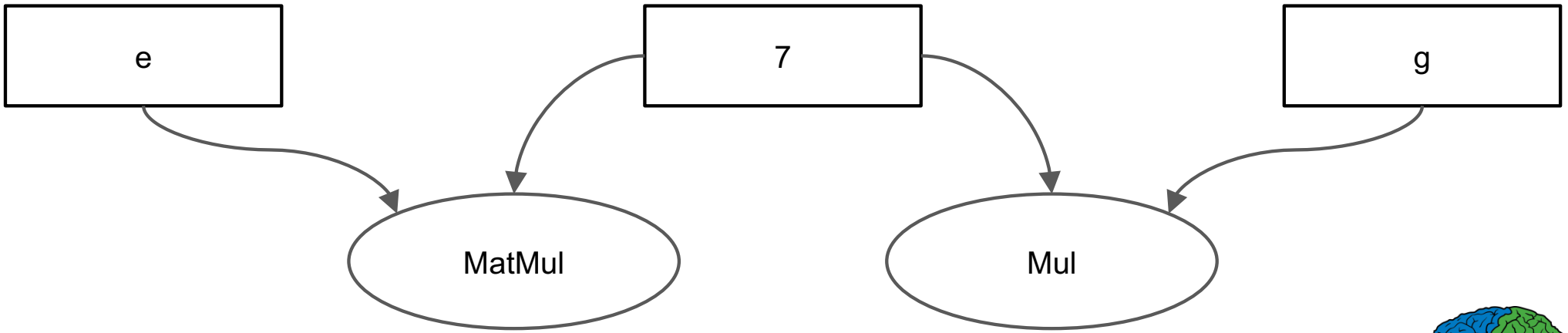
Common Subexpression Elimination



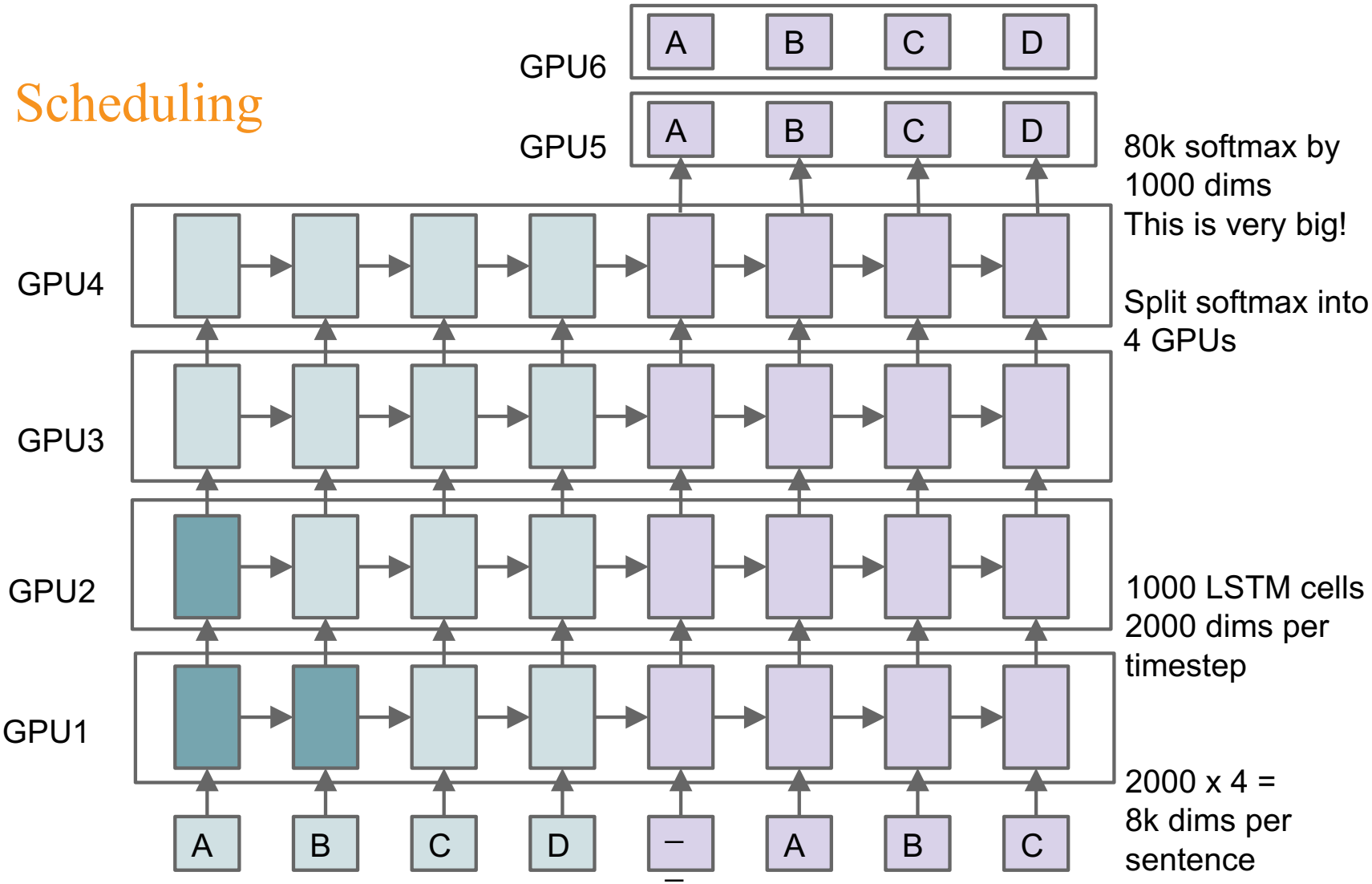
Constant Folding



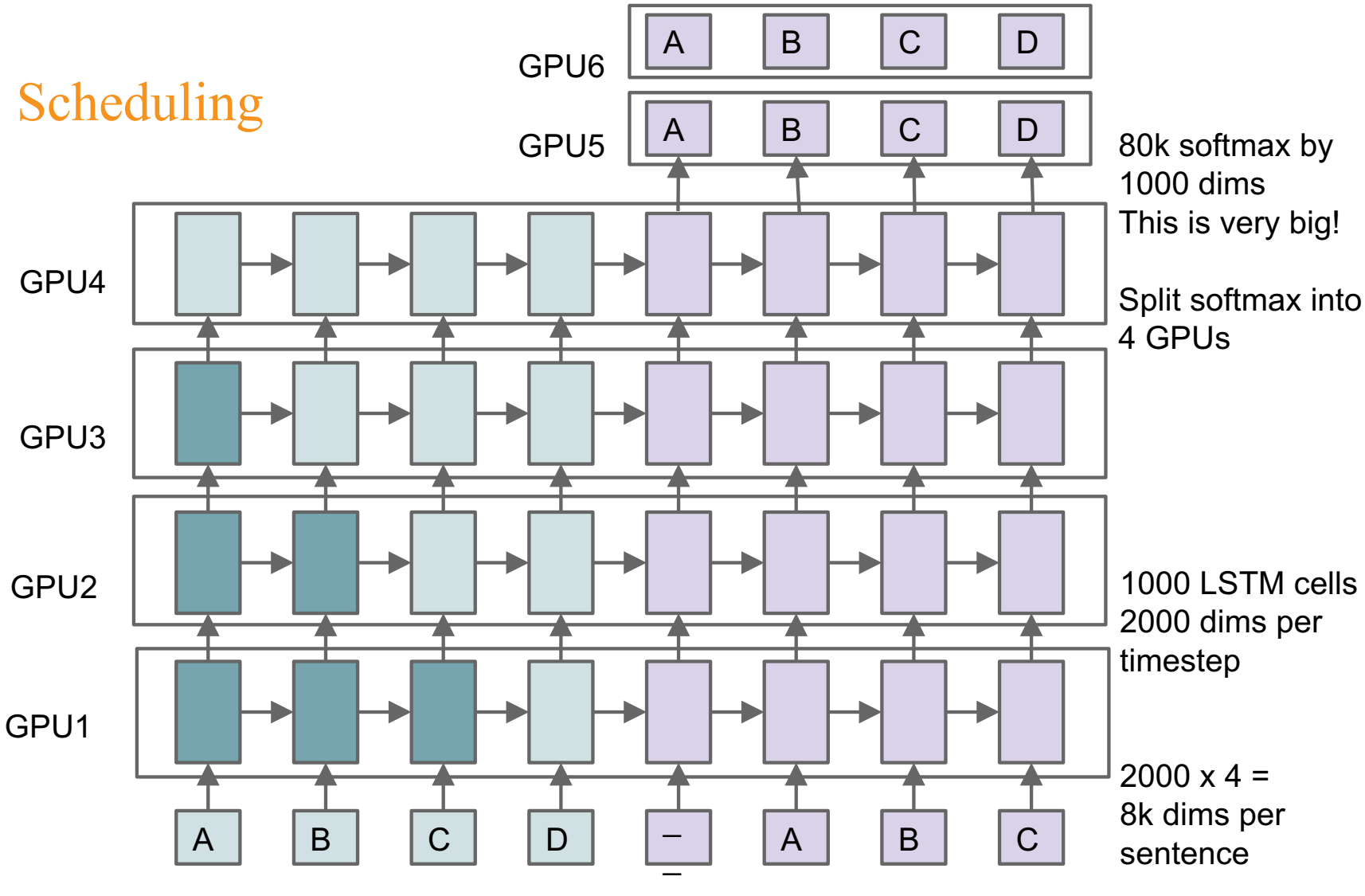
Constant Folding



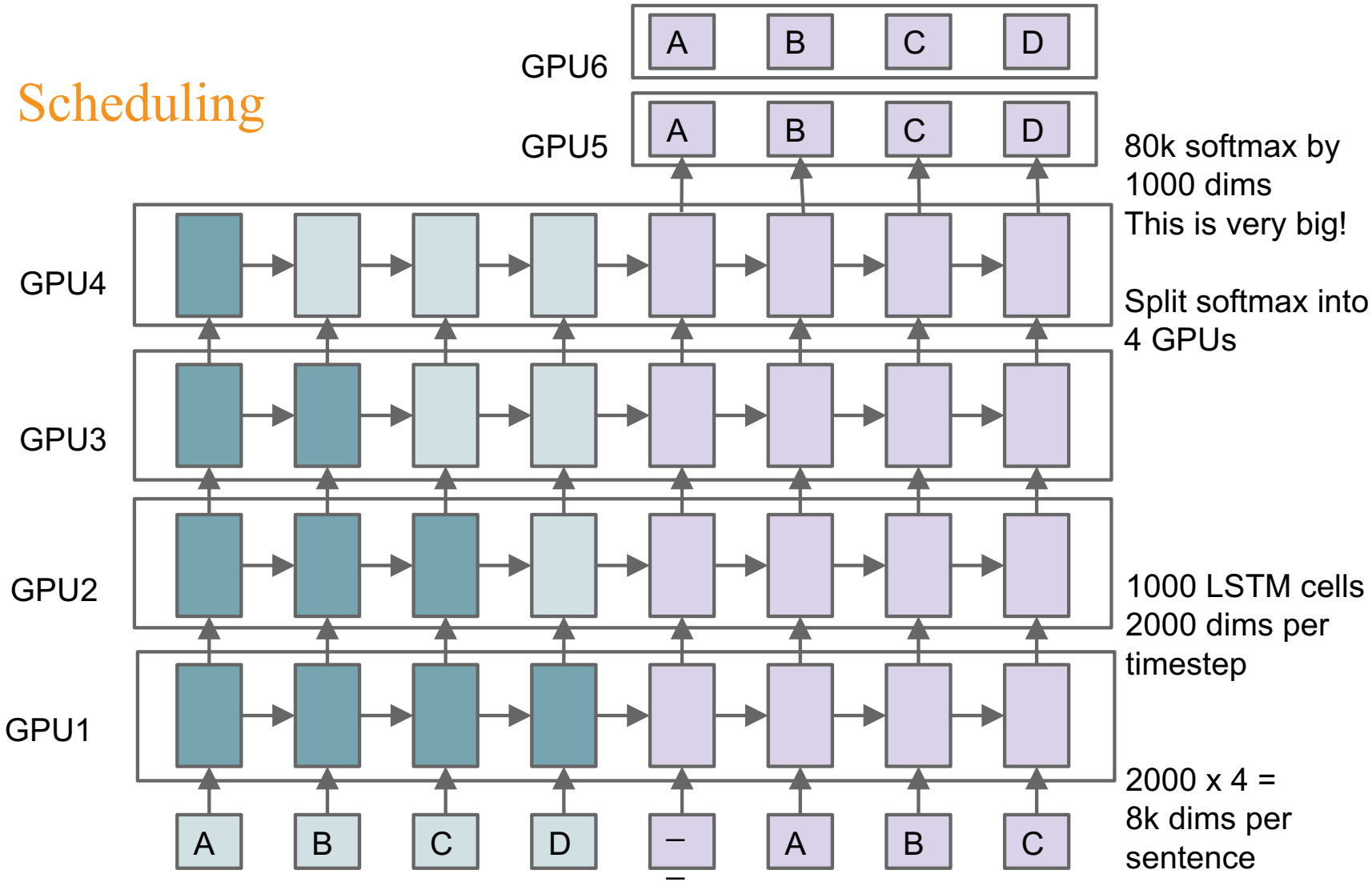
Scheduling


GPU6
A
B
C
D
GPU5
A
B
C
D
80k softmax by
1000 dims
This is very big!
GPU4
Split softmax into
4 GPUs
GPU3
GPU2
1000 LSTM cells
2000 dims per
timestep
GPU1
2000 x 4 =
8k dims per
sentence
A
B
C
D
–
A
B
C

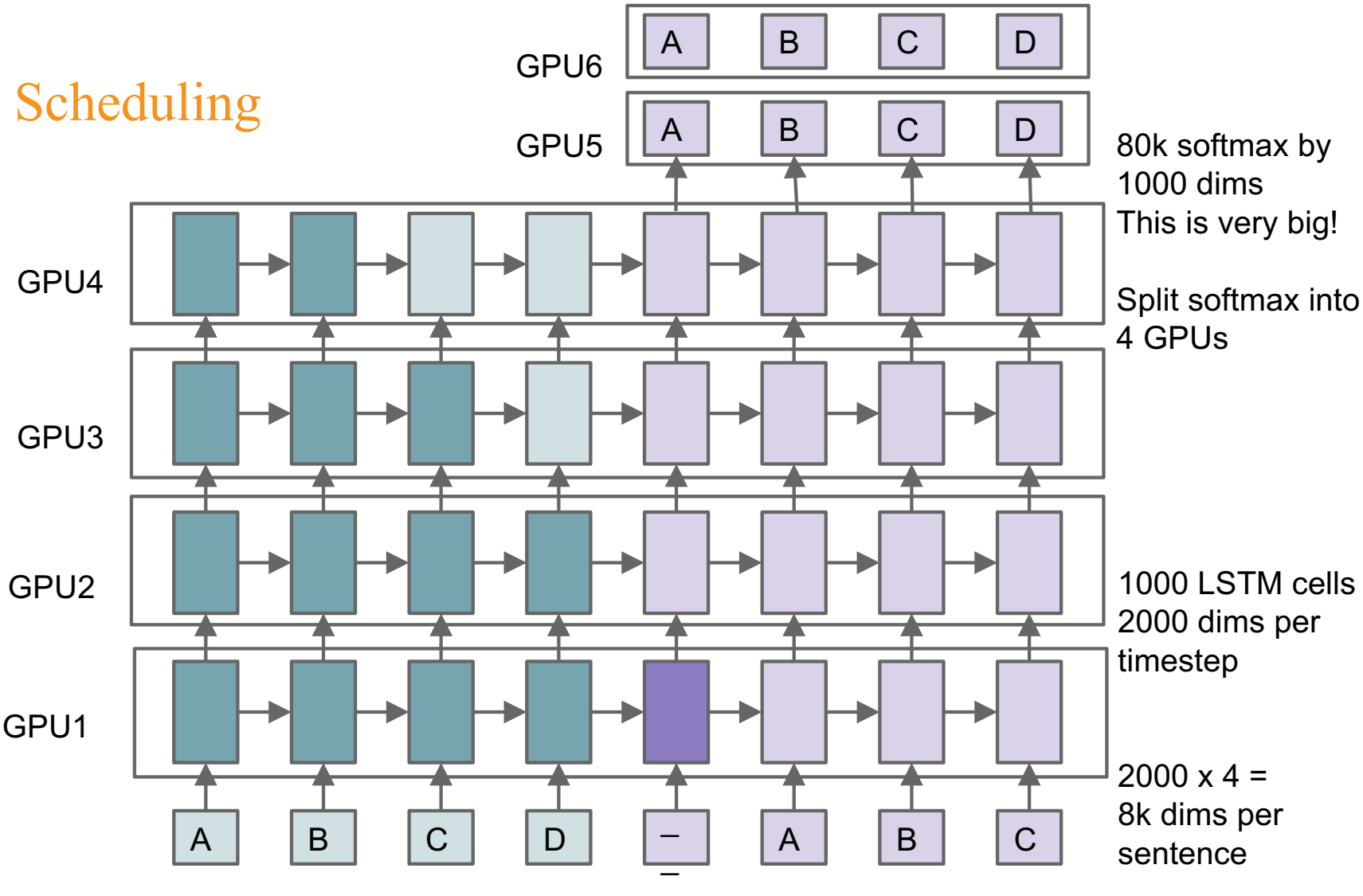
Scheduling


GPU6
GPU5
GPU4
GPU3
GPU2
GPU1
A
B
C
D
–
80k softmax by 1000 dims
This is very big!
Split softmax into 4 GPUs
1000 LSTM cells 2000 dims per timestep
2000 x 4 = 8k dims per sentence

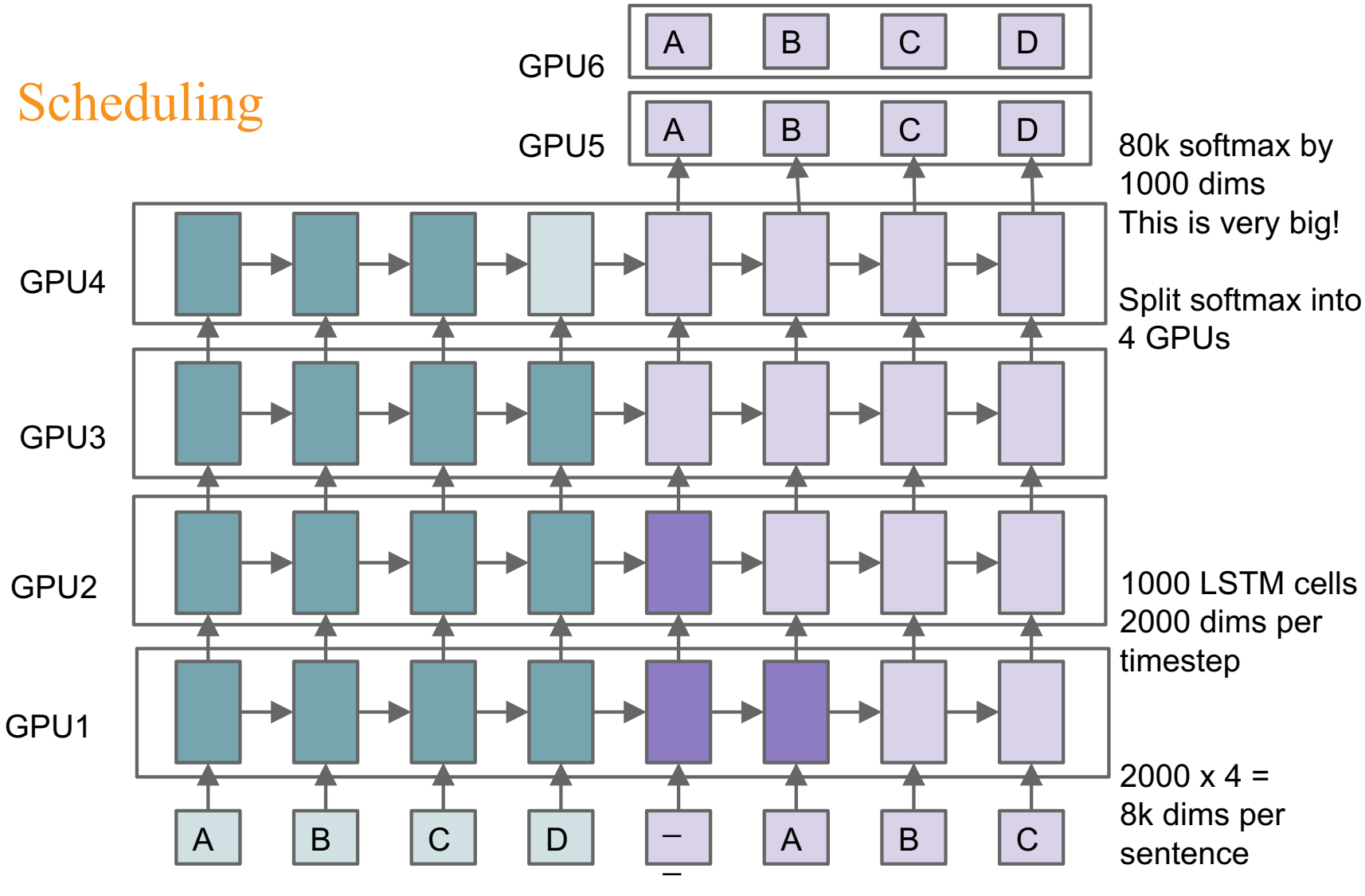
Scheduling


GPU6
A
B
C
D
GPU5
A
B
C
D
80k softmax by
1000 dims
This is very big!
GPU4
Split softmax into
4 GPUs
GPU3
GPU2
1000 LSTM cells
2000 dims per
timestep
GPU1
2000 x 4 =
8k dims per
sentence
A
B
C
D
–
A
B
C

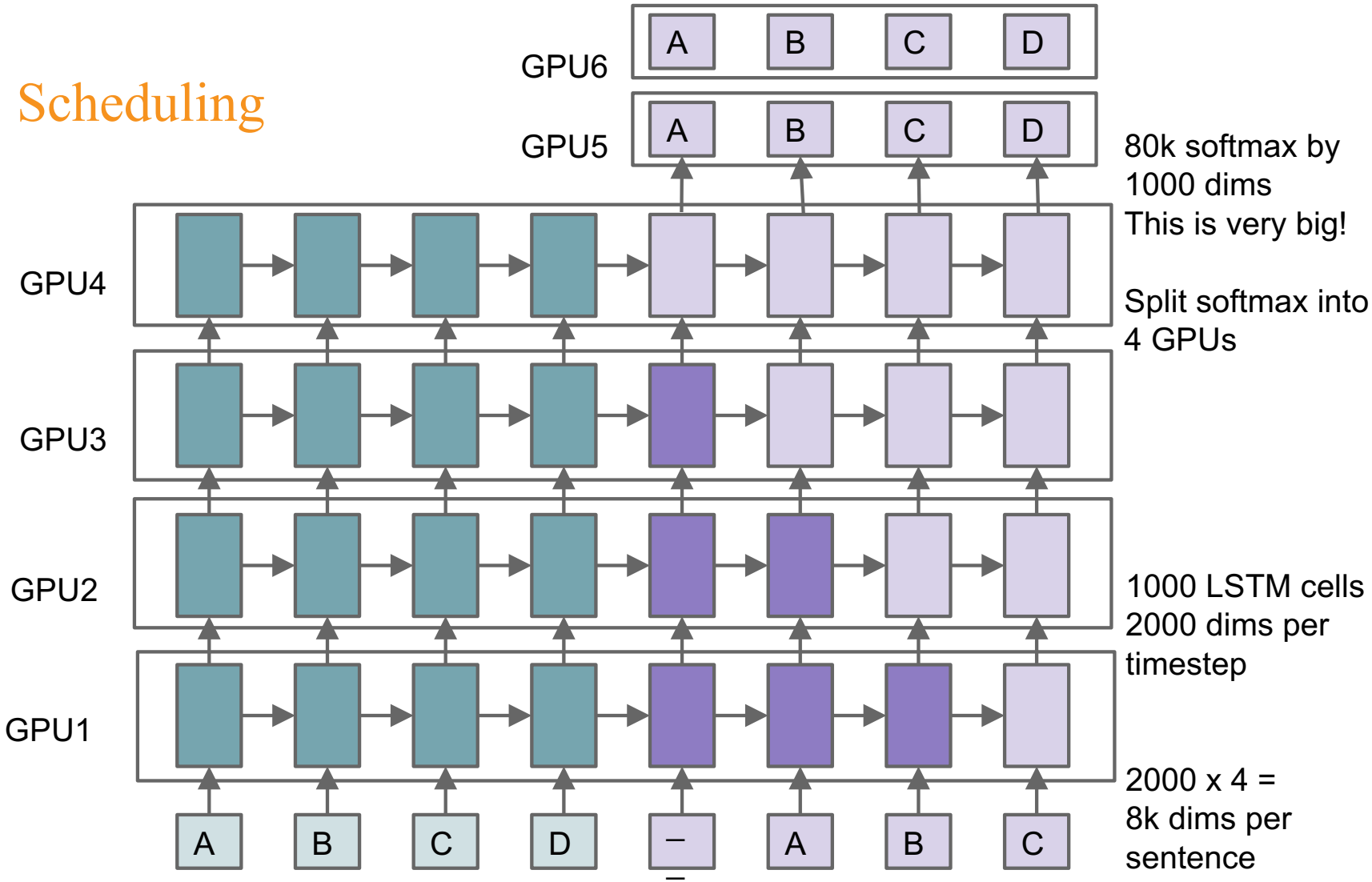
Scheduling


GPU6
A
B
C
D
GPU5
A
B
C
D
80k softmax by
1000 dims
This is very big!
GPU4
Split softmax into
4 GPUs
GPU3
GPU2
1000 LSTM cells
2000 dims per
timestep
GPU1
2000 x 4 =
8k dims per
sentence
A
B
C
D
–
A
B
C

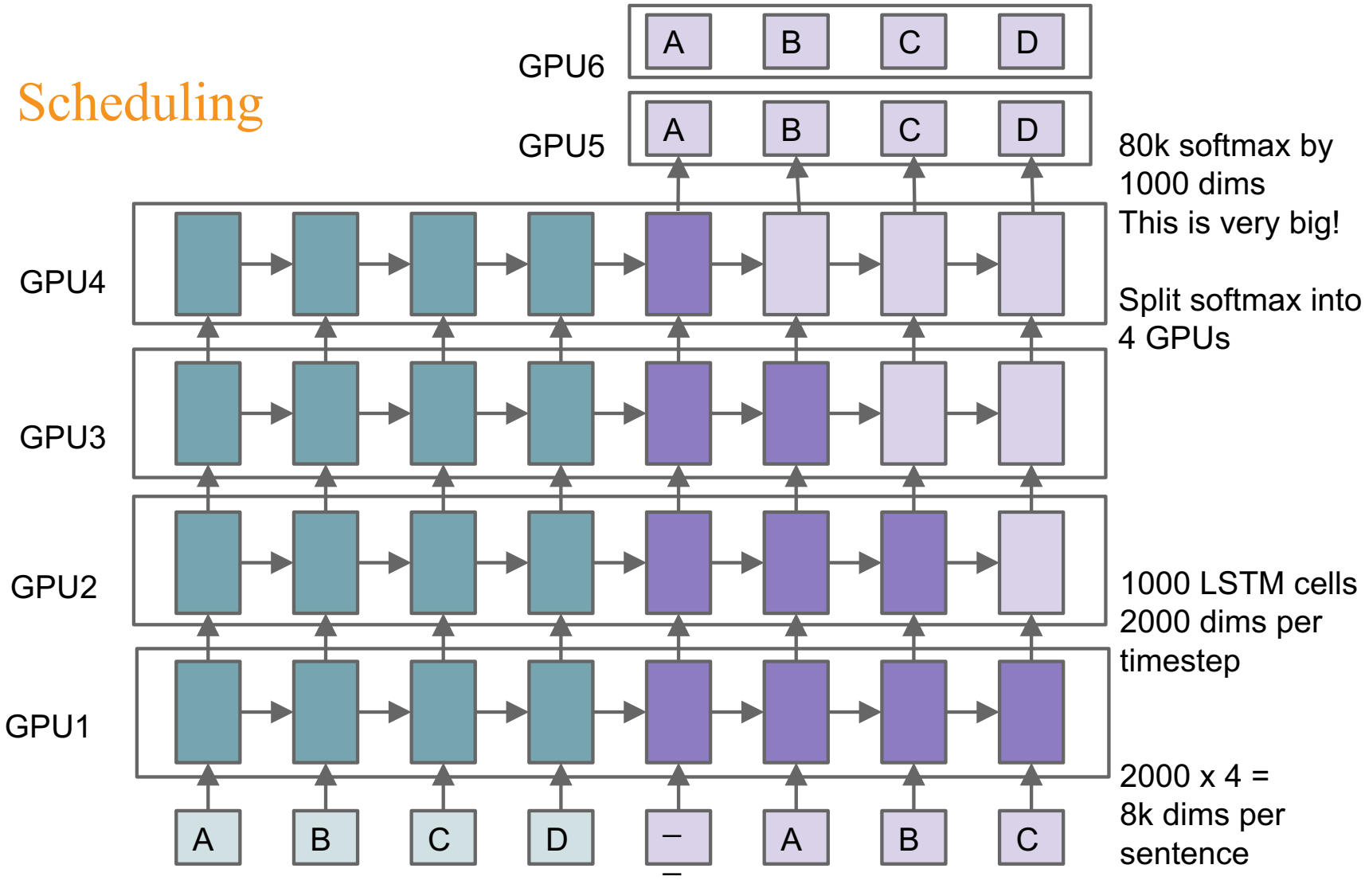
Scheduling


GPU6
A
B
C
D
GPU5
A
B
C
D
80k softmax by
1000 dims
This is very big!
GPU4
Split softmax into
4 GPUs
GPU3
GPU2
1000 LSTM cells
2000 dims per
timestep
GPU1
2000 x 4 =
8k dims per
sentence
A
B
C
D
–
A
B
C

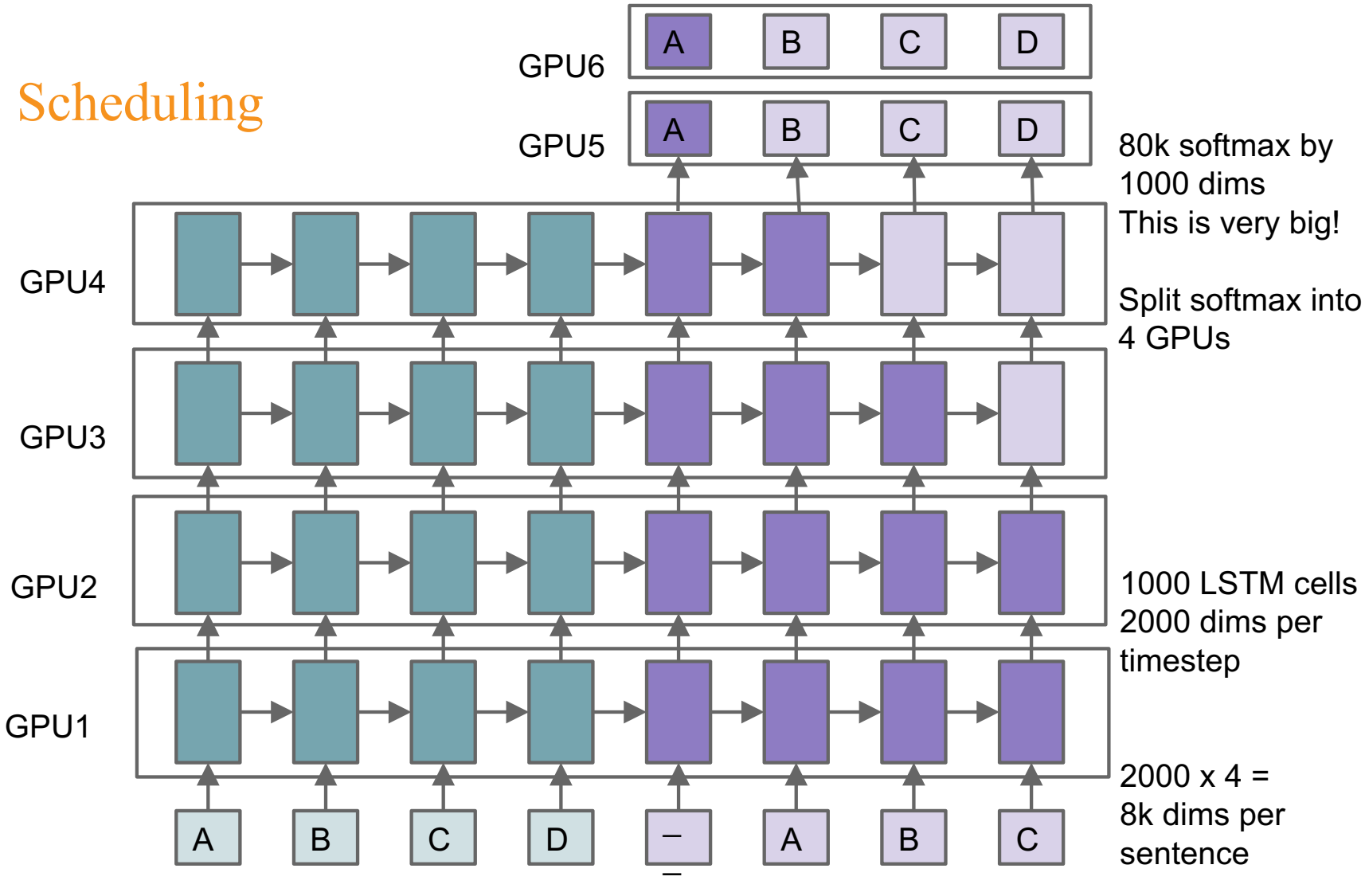
Scheduling


GPU6
A
B
C
D
GPU5
A
B
C
D
80k softmax by
1000 dims
This is very big!
GPU4
Split softmax into
4 GPUs
GPU3
GPU2
1000 LSTM cells
2000 dims per
timestep
GPU1
2000 x 4 =
8k dims per
sentence
A
B
C
D
–
A
B
C

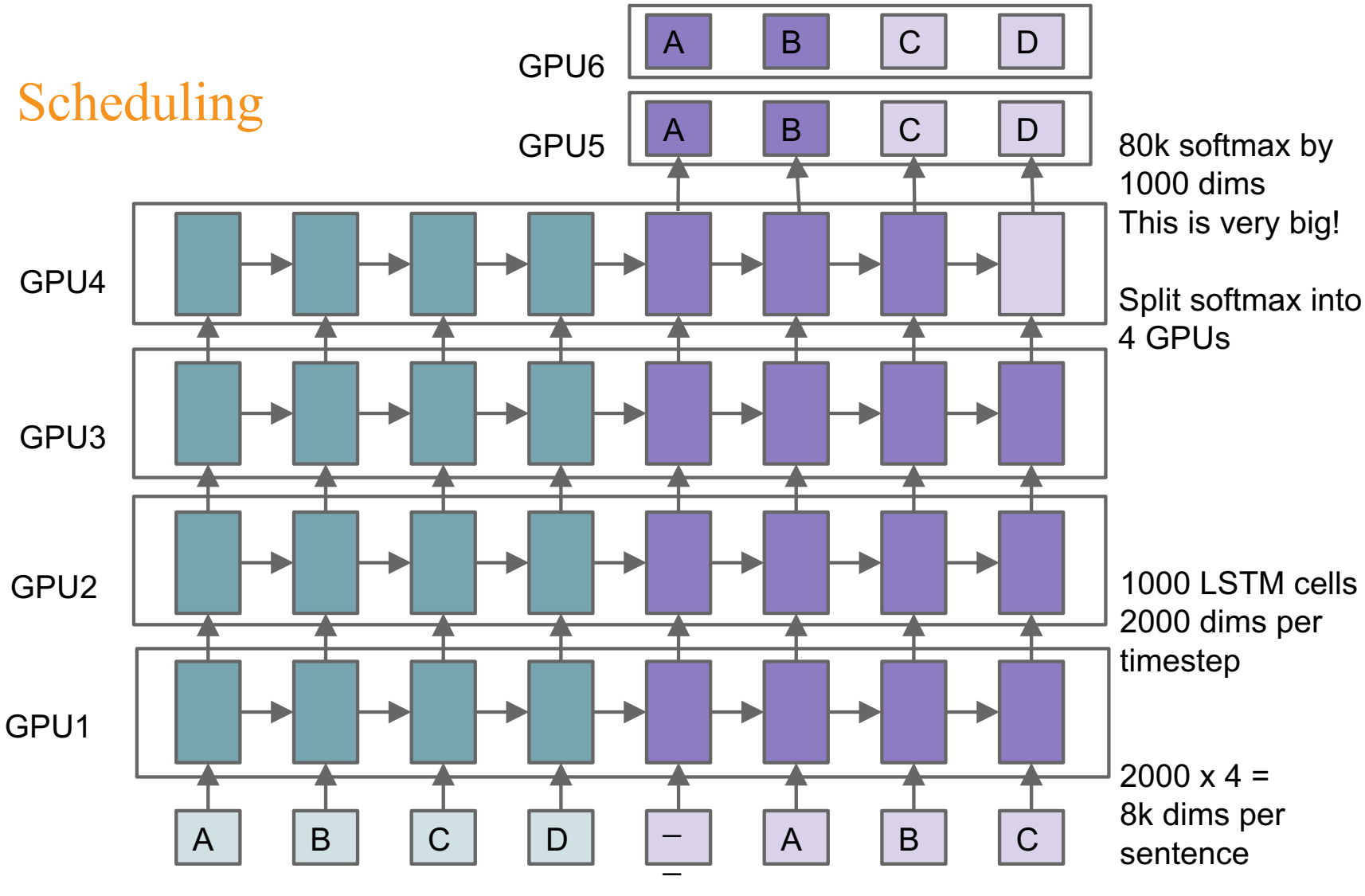
Scheduling


GPU6
A
B
C
D
GPU5
A
B
C
D
80k softmax by
1000 dims
This is very big!
GPU4
Split softmax into
4 GPUs
GPU3
GPU2
1000 LSTM cells
2000 dims per
timestep
GPU1
2000 x 4 =
8k dims per
sentence
A
B
C
D
–
A
B
C

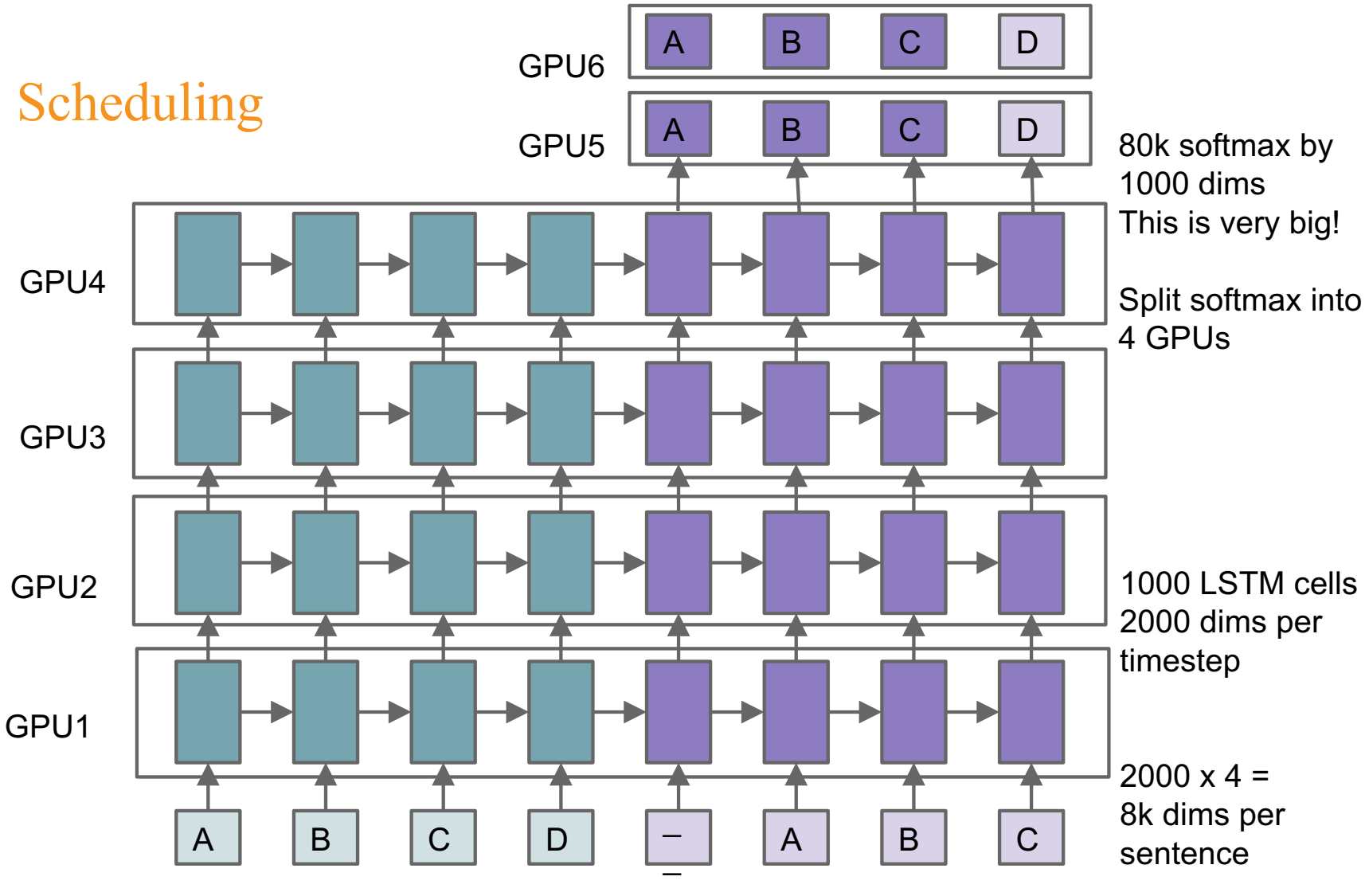
Scheduling


GPU6
A
B
C
D
GPU5
A
B
C
D
80k softmax by
1000 dims
This is very big!
GPU4
Split softmax into
4 GPUs
GPU3
GPU2
1000 LSTM cells
2000 dims per
timestep
GPU1
2000 x 4 =
8k dims per
sentence
A
B
C
D
–
A
B
C

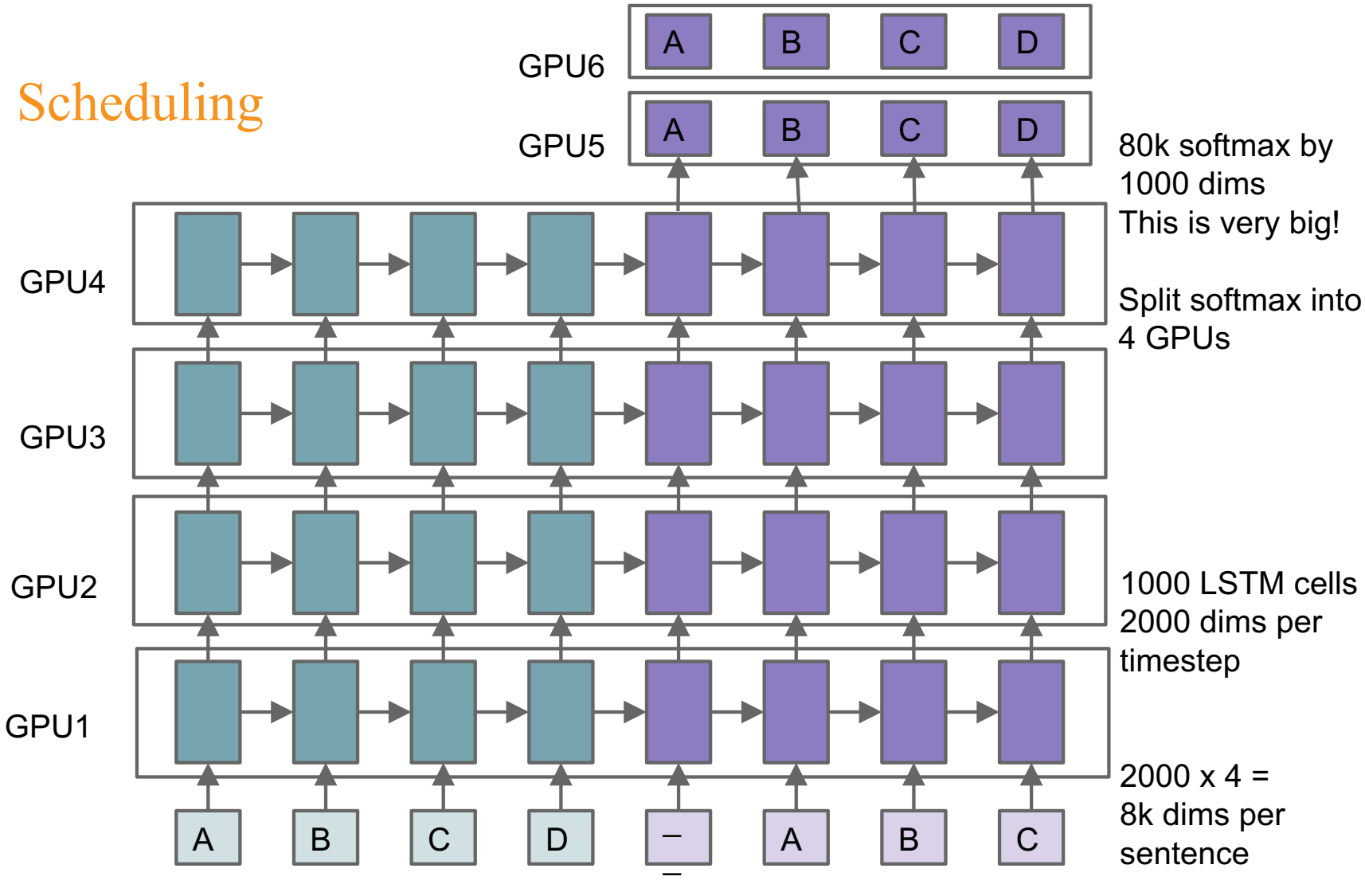
Scheduling


GPU6
A
B
C
D
GPU5
A
B
C
D
80k softmax by
1000 dims
This is very big!
GPU4
Split softmax into
4 GPUs
GPU3
GPU2
1000 LSTM cells
2000 dims per
timestep
GPU1
2000 x 4 =
8k dims per
sentence
A
B
C
D
–
A
B
C

Scheduling


GPU6
GPU5
GPU4
GPU3
GPU2
GPU1
A
B
C
D
–
80k softmax by 1000 dims
This is very big!
Split softmax into 4 GPUs
1000 LSTM cells 2000 dims per timestep
2000 x 4 = 8k dims per sentence

Scheduling


GPU6
A
B
C
D
GPU5
A
B
C
D
80k softmax by
1000 dims
This is very big!
GPU4
Split softmax into
4 GPUs
GPU3
GPU2
1000 LSTM cells
2000 dims per
timestep
GPU1
2000 x 4 =
8k dims per
sentence
A
B
C
D
–
A
B
C

TensorFlow Tools

TensorFlow Tools

- Pre-existing models: ML Toolkit
- Tensorboard
- TensorFlow Serving
- Embedding Visualizer
- Hyperparameter Tuning
  - https://cloud.google.com/ml-engine/docs/how-tos/using-hyperparameter-tuning
- ….

ML Toolkit

- Linear / Logistic regression
- KMeans Clustering
- Gaussian Mixture Model
- WALS Matrix Factorization
- Support Vector Machine
- Stochastic Dual Coordinate Ascent
- Random Forest
- DNN, RNN, LSTM, Wide & Deep, ...

What is Serving?

Serving is how you *apply* a ML model, *after* you've trained it

TensorFlow Serving

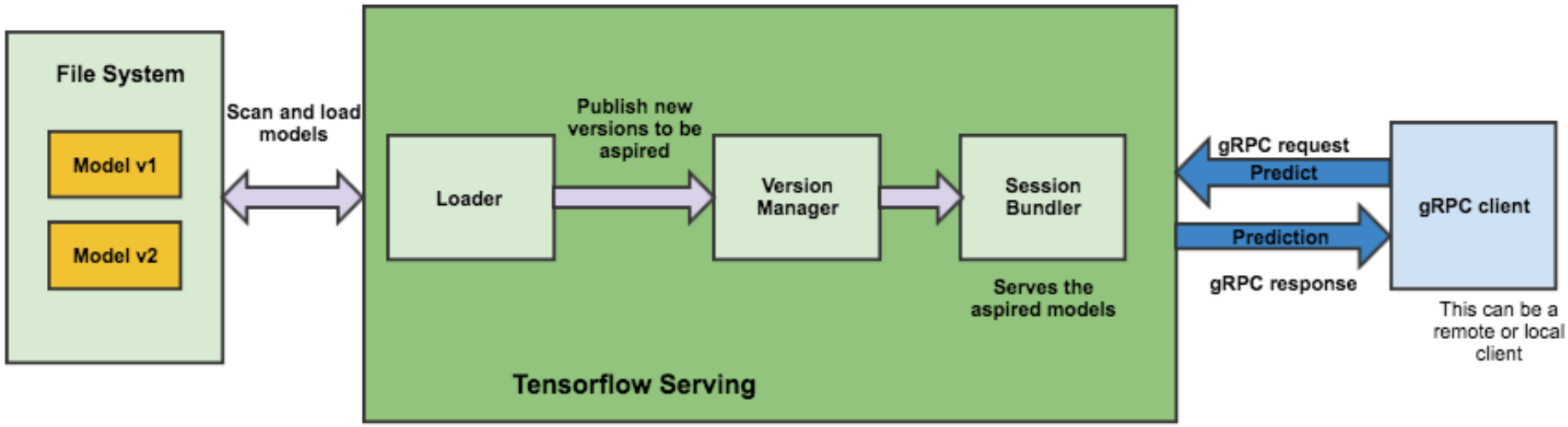
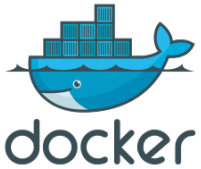


Image: Zendesk Engineering medium.com/zendesk-engineering/how-zendesk-serves-tensorflow-models-in-production-751ee22f0f4b

TensorFlow Serving

- C++ Libraries
  - TensorFlow model save / export formats
  - Generic core platform
- Binaries
  - Best practices out of the box
  - Docker containers, K8s tutorial
- Hosted Service across
  - Google Cloud ML



TensorBoard

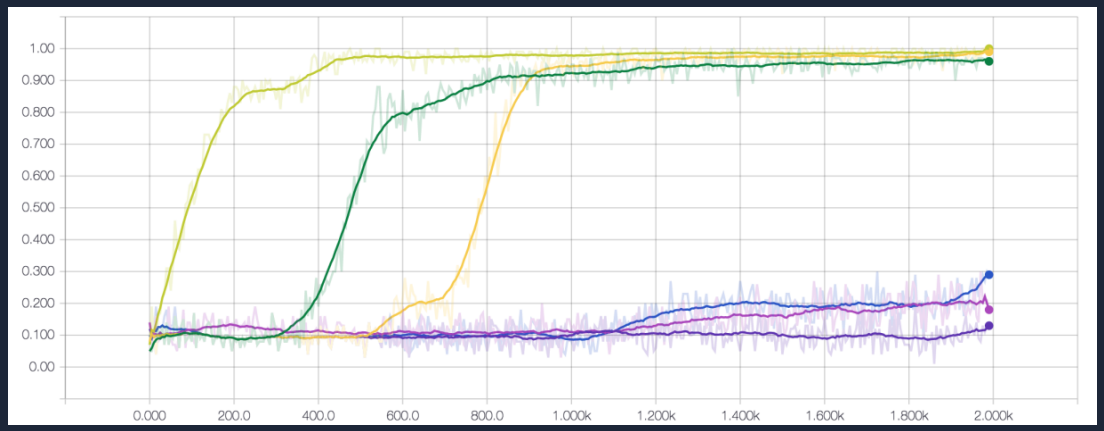
Collect some summaries

summary (n):

a TensorFlow op that output protocol buffers containing "summarized" data

Examples:

- **tf.summary.scalar**



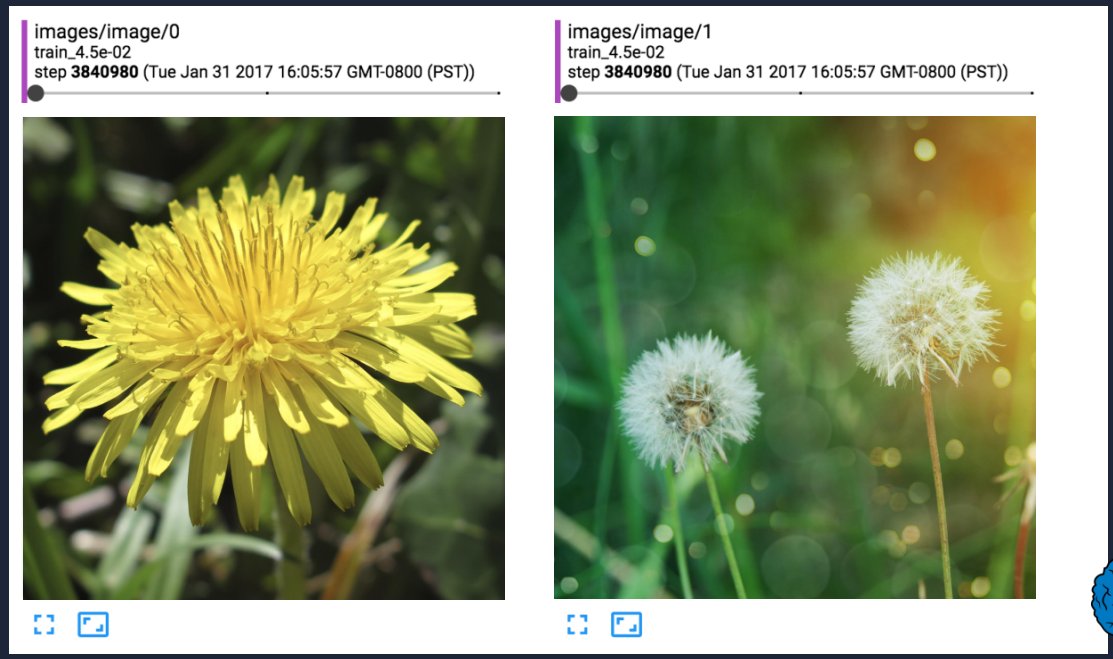
Collect some summaries

summary (n):

a TensorFlow op that output protocol buffers containing "summarized" data

Examples:

- tf.summary.scalar
- **tf.summary.image**



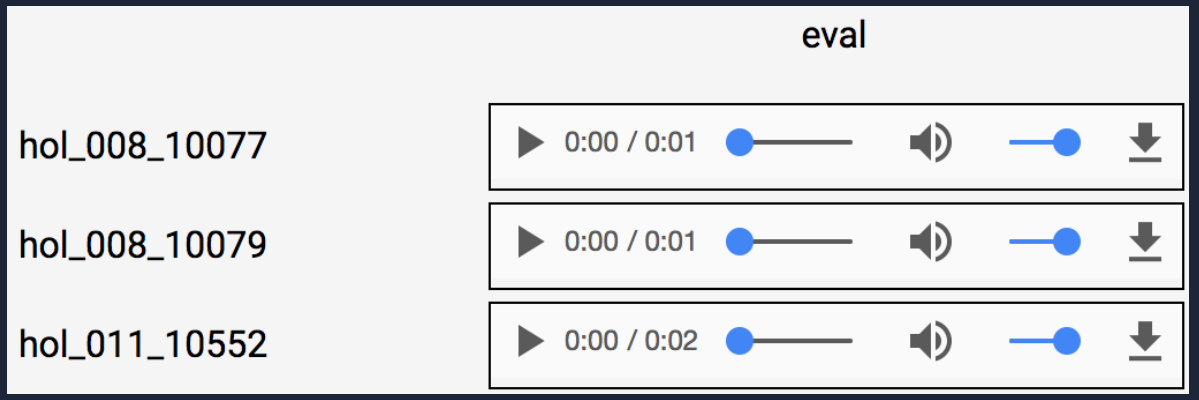
Collect some summaries

summary (n):

a TensorFlow op that output protocol buffers containing "summarized" data

Examples:

- tf.summary.scalar
- tf.summary.image
- **tf.summary.audio**



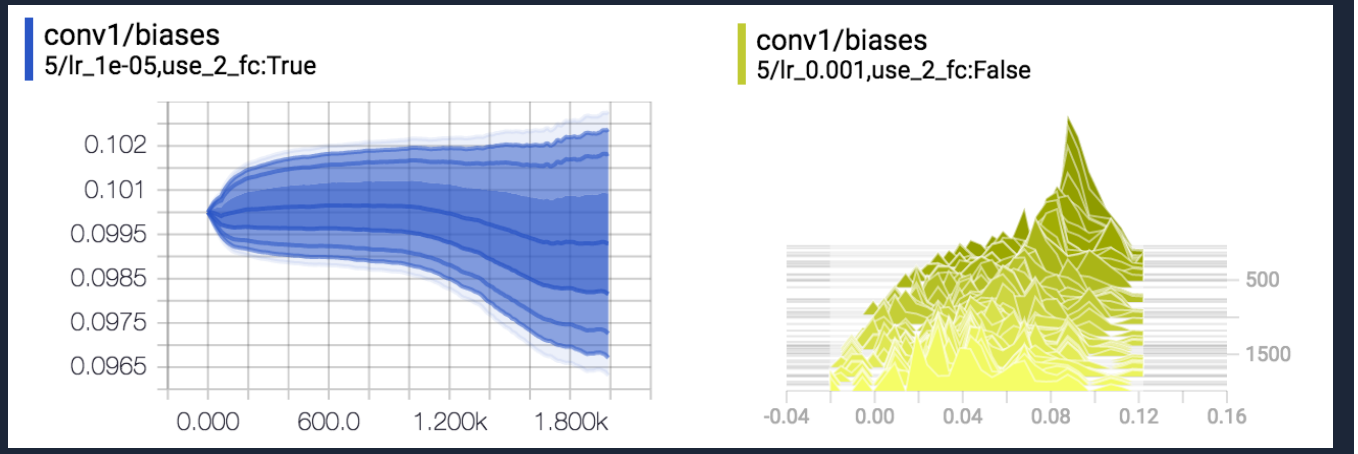
Collect some summaries

summary (n):

a TensorFlow op that output protocol buffers containing "summarized" data

Examples:

- tf.summary.scalar
- tf.summary.image
- tf.summary.audio
- **tf.summary.histogram**



TensorBoard SCALARS IMAGES AUDIO GRAPHS DISTRIBUTIONS HISTOGRAMS EMBEDDINGS

Write a regex to create a tag group

Split on underscores

Data download links

Tooltip sorting method: default

Smoothing

0.6

Horizontal Axis

STEP RELATIVE WALL

Runs

Write a regex to filter runs

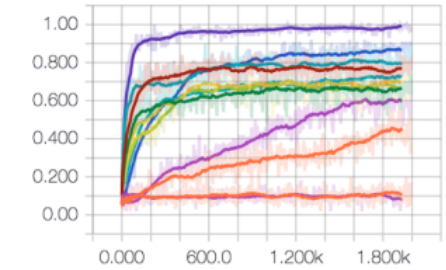
lr_1E-05,conv=1,fc=1
lr_1E-05,conv=1,fc=2
lr_1E-05,conv=2,fc=1
lr_1E-05,conv=2,fc=2
lr_1E-04,conv=1,fc=1

TOGGLE ALL RUNS

/usr/local/google/home/dandelion/mnist_tutorial/5

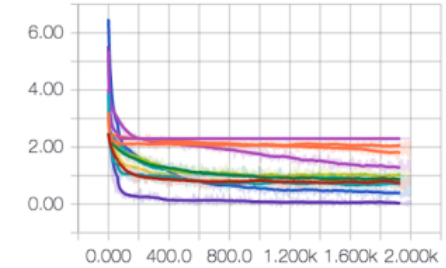
accuracy

accuracy/accuracy

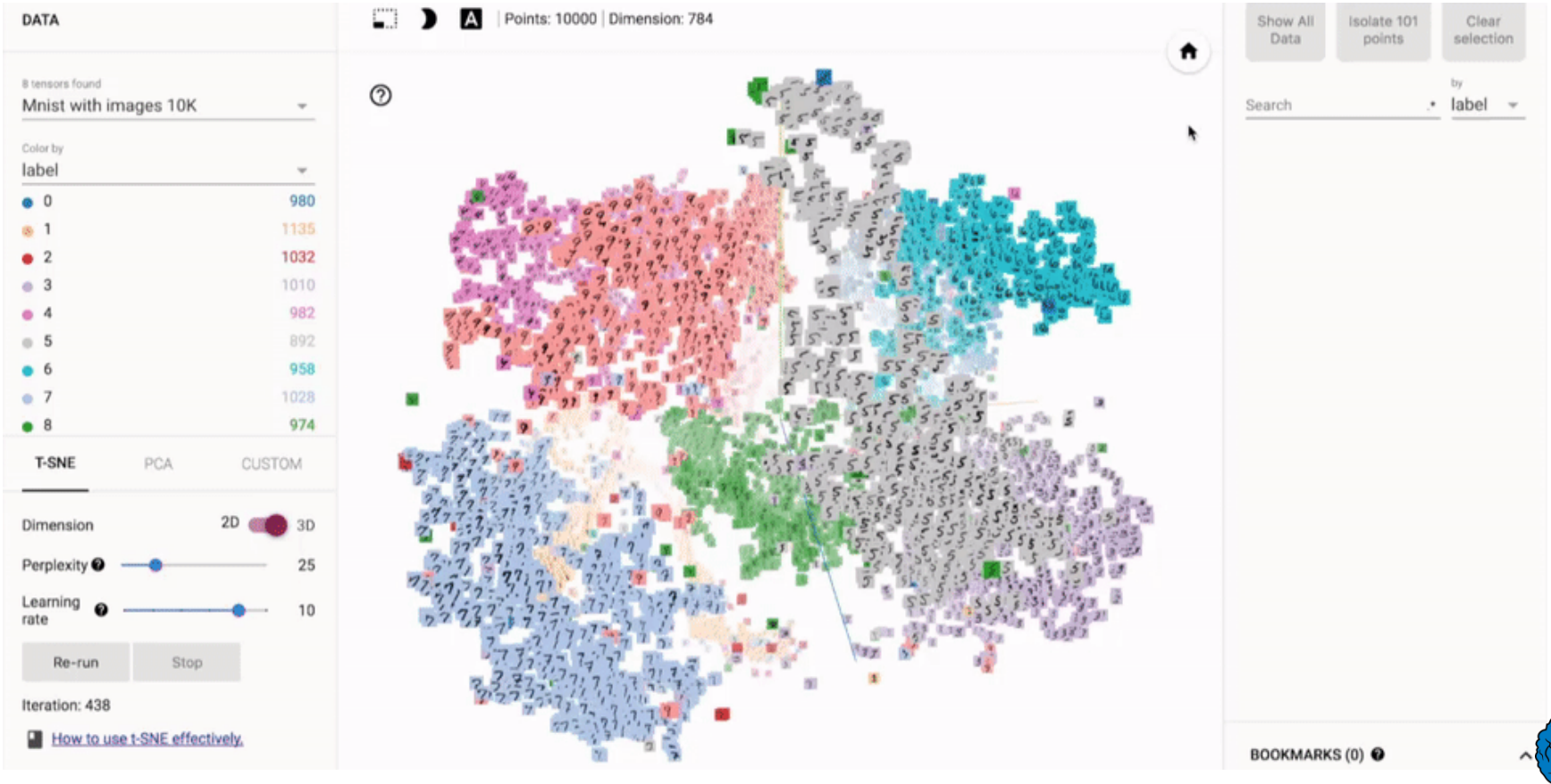


xent

xent/xent_1



Embedding Visualizer




158 M 742
github.com/jnaulty/metermaid-monitor