Lecture 3: Deep Learning Frameworks

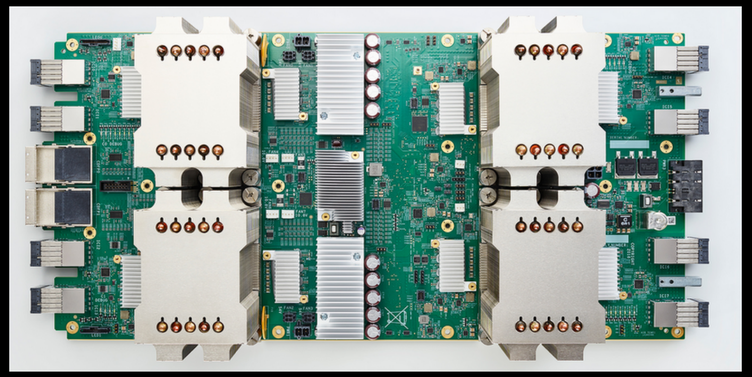
CS 256: Systems and Machine Learning

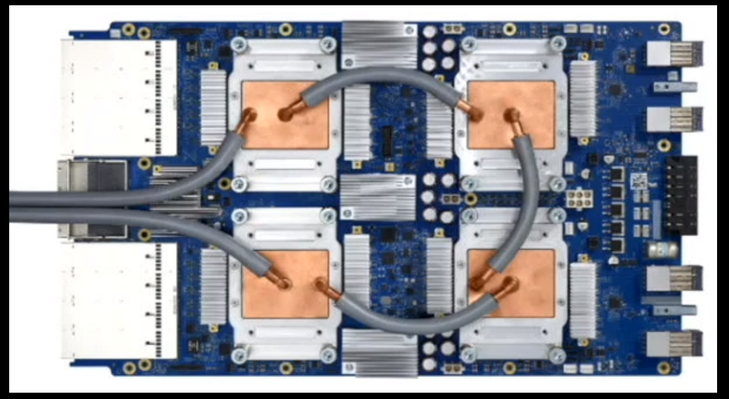
Sangeetha Abdu Jyothi

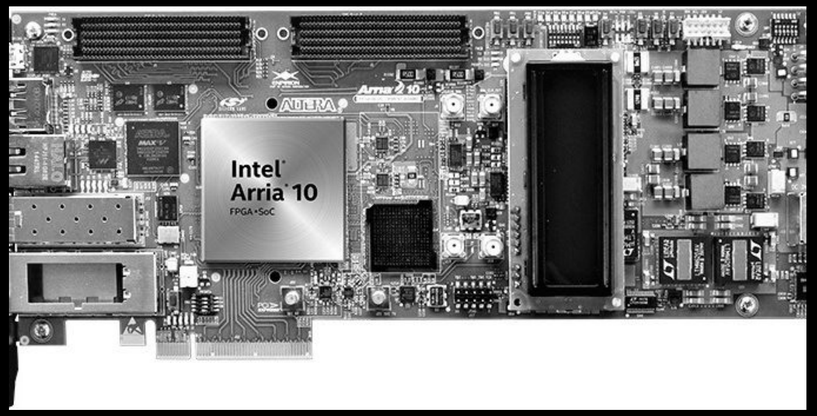
UCIrvine

Parts of this lecture were adapted from CS 231 at Stanford, CS 6787 at Cornell and CS 15-849 at CMU

Last Class: Hardware for Deep Learning



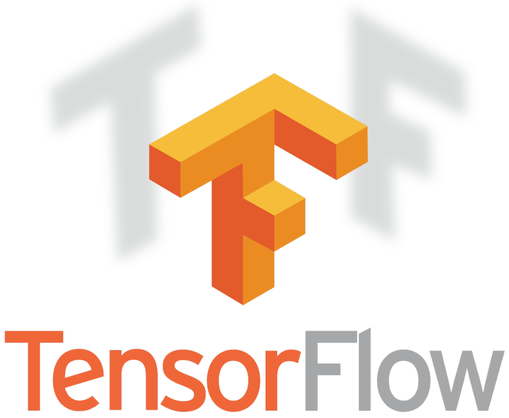


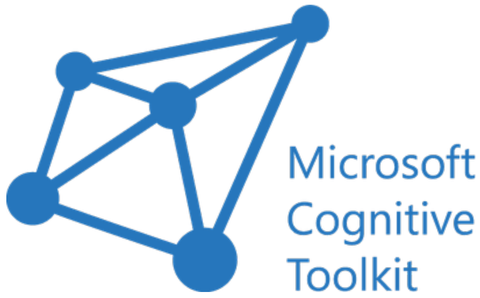


Deep Learning Frameworks

- Can be viewed as the scaffolding of the Machine Learning Revolution
- Enable easy design, training, and validation of deep neural networks
- High level programming interface makes ML accessible to non-experts

Deep Learning Frameworks





Deep Learning Software Evolution - Early 2000s

- Early tools to describe and develop neural networks
- Frameworks: MATLAB, Torch, OpenNN
- Limited functionality
- Complex APIs
- Some didn’t support GPUs

Deep Learning Software Evolution - Early 2010s

- Early deep learning frameworks - Caffe, Chainer, Theano, etc.
- Easy to build complex neural networks
- Multi-GPU training support

Deep Learning Software Evolution - Late 2010s

- More deep learning frameworks - TensorFlow, Caffe2, PyTorch, CNTK, MXNet, Keras, etc.
- Well-defined user APIs
- Optimizations for multi-GPU training and distributed training
- Several model zoos and toolkits

Deeper dive into the internals

- Common Features of ML frameworks

- Overview of framework design (TensorFlow and PyTorch)

Common Features of ML Frameworks

Tensors

- Multi-dimensional array
- E.g., CIFAR10 dataset consists of 60000 32x32 color images

$$T_{CIFAR-10} \in \mathbb{R}^{32*32*3*60000}$$

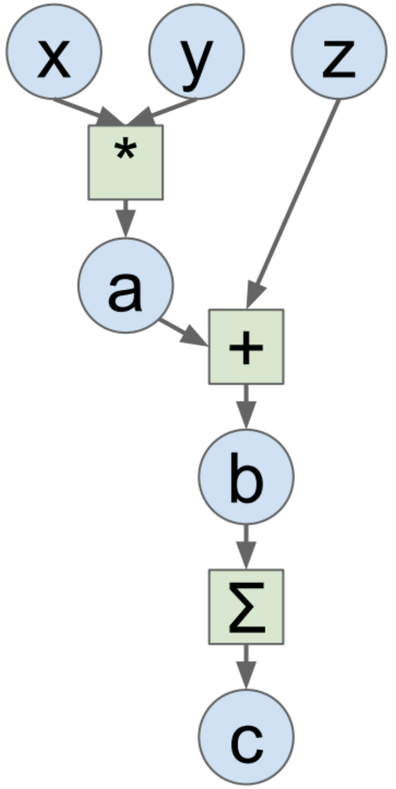
- Gradients for deep learning can also be tensors

$$G \in \mathbb{R}^{100*100*32}$$

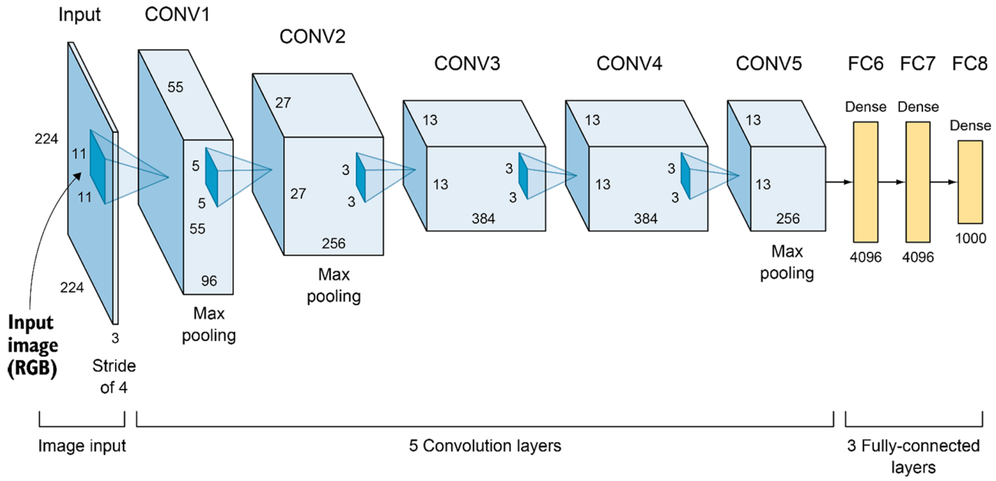
- During deep learning, we are performing various mathematical operations on these tensors: Elementwise operations, matrix multiply-like operations, etc.

Computational Graphs

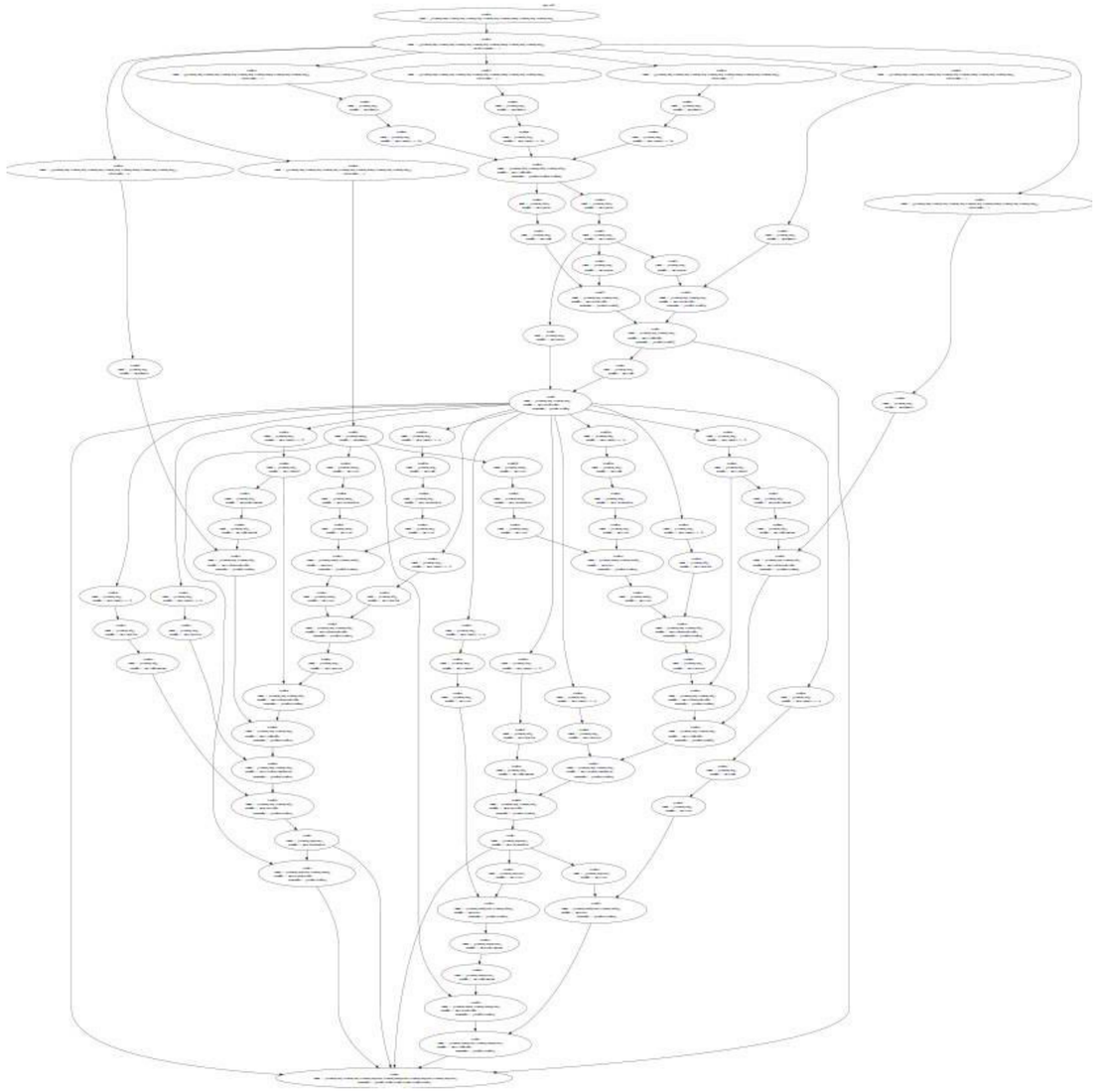
- Deep learning process represented using computational graphs
- Computational graphs are DAGs (Directed Acyclic Graphs)
- Nodes are operations and variables
- Edges are dependencies (direction of data flow)



Real-world Computational Graphs are Complex



AlexNet Model



Need for Automatic Differentiation

- Gradient computation in back propagation is a tedious process
- We need a mechanism to automate this process
    - Write the forward pass and have the backward pass implementation automated
- Solution: Automatic Differentiation [Automatic Differentiation in PyTorch, NeurIPS 2017]
    - Apply chain rule directly to operations in a program
    - Allows computing exact solutions
    - Cheaper to compute than symbolic differentiation
    - No round-off errors like numerical differentiation

Need for high-level API

- Several low-level libraries: CUDA, cuDNN, cuBLAS, etc.
- Programming in low-level libraries is cumbersome
- We want to run efficiently on heterogenous hardware, but write program using high-level APIs

Goal of Deep Learning Frameworks

- Run and build complex computational graphs easily
- Compute gradients in computational graphs easily
- Run efficiently on heterogenous hardware

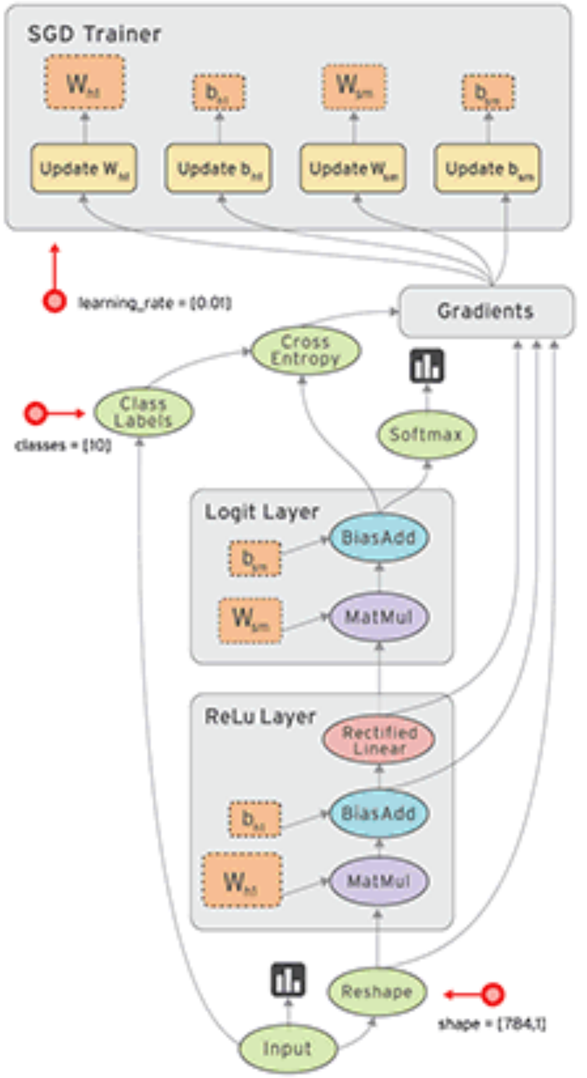
TensorFlow

TensorFlow

- Originated from DistBelief
  - PS architecture
  - Stateless worker processes for computation
  - Stateful Parameter Server processes for maintaining latest version of parameters
  - Layers are C++ classes
  - Experimentation (e.g., adding new optimization methods) was difficult
- TensorFlow used static dataflow graphs to represent computations and shared states (static in TF 1.x only)
- Allows applications to be easily deployed in any settings: distributed clusters, local workstations, mobile devices, etc.

Design Principles

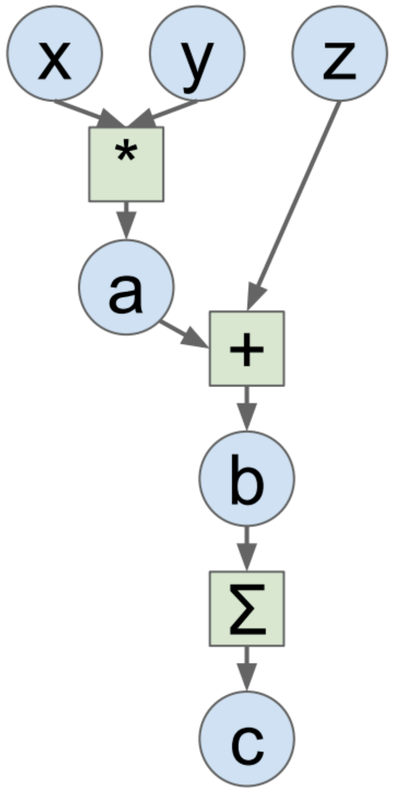
- TensorFlow differs from batch dataflow systems
  - Multiple concurrent executions on overlapping subgraphs of the overall graph
- Deferred execution
  - First define symbolic dataflow graph
  - Then execute optimized version of the program
- Common abstraction for heterogenous accelerators
  - Tensors are dense at the lowest level for efficient memory allocation



Example on TensorFlow

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

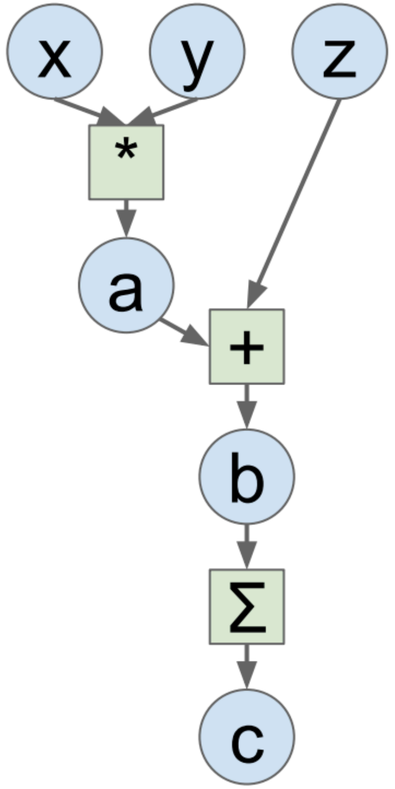
Create forward pass



Example on TensorFlow

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

Ask TF to create gradients



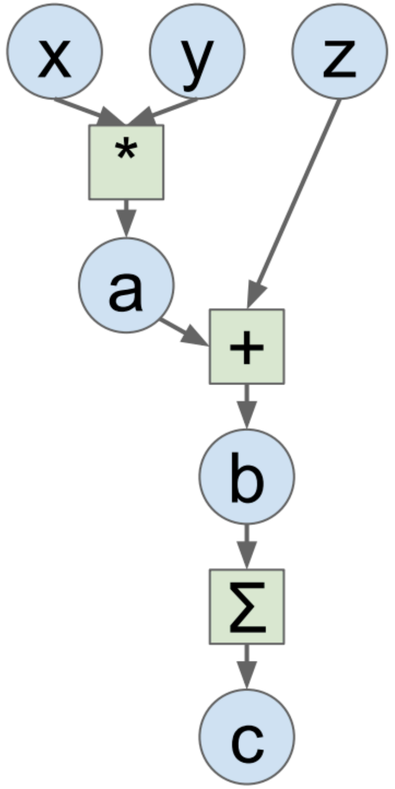
Example on TensorFlow

```
# Basic computational graph
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = tf.placeholder(tf.float32)

a = x * y
b = a + z
c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])
```

```
with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

Define a computational graph

Run the graph



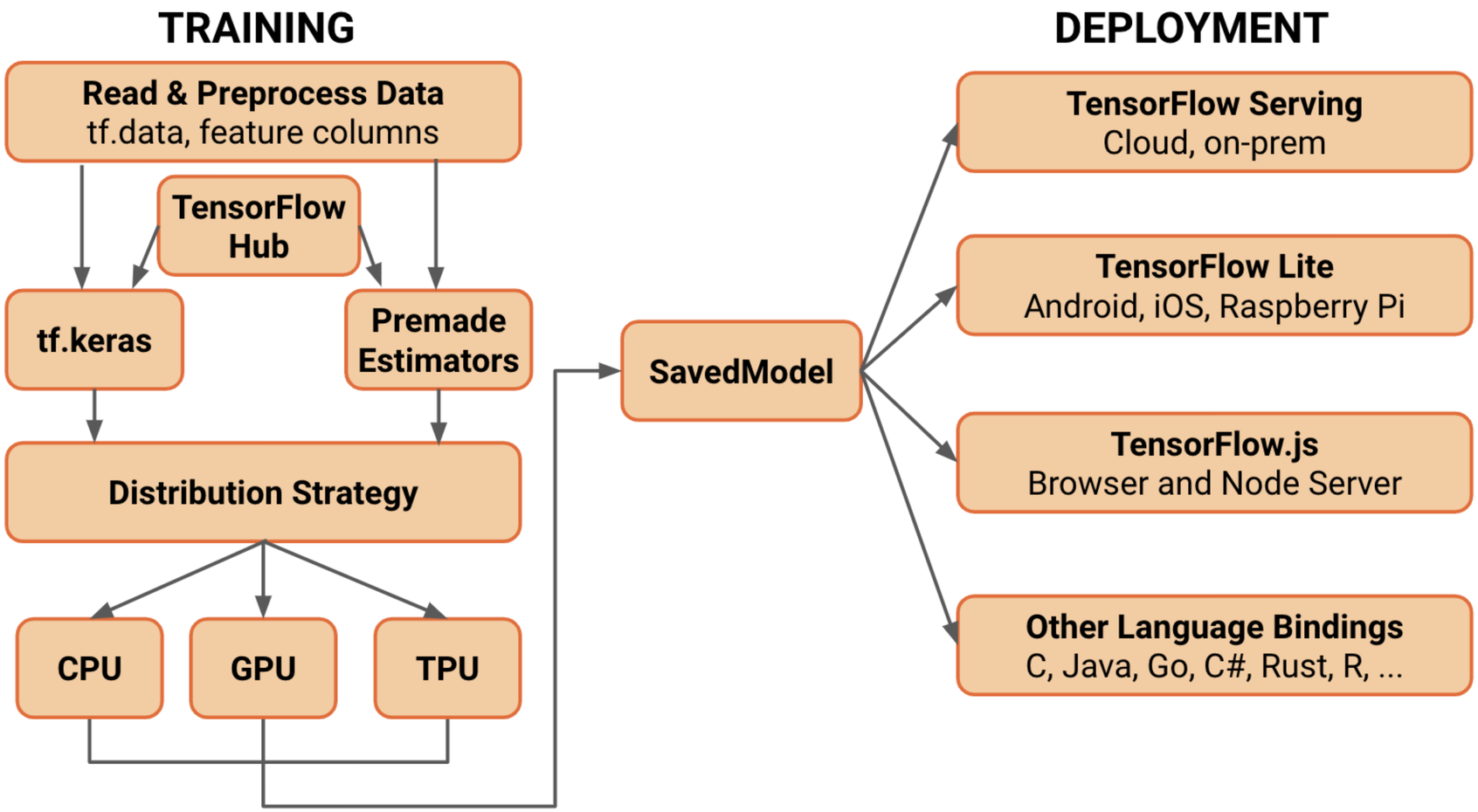
Distributed Execution

- Each operation resides on a particular device (such as GPU or CPU)
- A per-device subgraph contains all operations assigned to that device
- Send and Recv operations replace edges across device boundaries

TensorFlow 2.0

- Dynamic graphs as default, static graphs are optional
- Standardized on Keras
    - High-level object-oriented API
    - Build the model as a stack of layers
- Distribution Strategy API to distribute training across multiple GPUs, multiple machines or TPUs
- A direct path to production for models: TensorFlow Serving, TensorFlow Lite, TensorFlow.js

TensorFlow 2.0



PyTorch

PyTorch

- Imperative Style
- High-Performance

Pythonic

- Interfaces are simple and consistent
- Integrates naturally with standard plotting, debugging, and data processing tools
- Layers are Python classes

```
class LinearLayer(Module):
   def __init__(self, in_sz, out_sz):
      super().__init__()
      t1 = torch.randn(in_sz, out_sz)
      self.w = nn.Parameter(t1)
      t2 = torch.randn(out_sz)
      self.b = nn.Parameter(t2)

   def forward(self, activations):
      t = torch.mm(activations, self.w)
      return t + self.b
```

```
class FullBasicModel(nn.Module):
   def __init__(self):
      super().__init__()
      self.conv = nn.Conv2d(1, 128, 3)
      self.fc = LinearLayer(128, 10)

   def forward(self, x):
      t1 = self.conv(x)
      t2 = nn.functional.relu(t1)
      t3 = self.fc(t1)
      return nn.functional.softmax(t3)
```

Listing 1: A custom layer used as a building block for a simple but complete neural network.

Imperative Programming Model

- Execute the computational graph while constructing it
- Easy to debug
  - e.g., printing intermediate tensors
- Mutable tensors
  - Update the value of the tensor in place (e.g. update the weights during backpropagation)
- Flexible Control Flow
  - Can use a mix of Python control flow and Pytorch Ops to construct the graph

```
import torch

v1 = torch.Tensor([1,2,3])
v2 = torch.exp(v1)
v3 = v2 + 1
print(v3)
V4 = v2 * v3
```

```
if  v2.numpy() > 1:
      v3 = v2 * 2
else:
      v3 = v2 * 0.5
```

High Performance

- Simplicity and ease of use are primary goals, but also provide compelling performance
- Efficient C++ core
    - Python as a host language
    - Data structures and ops are implemented in C++ and CUDA
- Separate control and data flow
- Multiprocessing
    - Extend Python multiprocessing to torch.multiprocessing
- Custom caching tensor allocator

PyTorch Features

- Imperative programming and Dynamic Eager Execution
- Dynamic computational graph generation
- Easy debugging
- Interoperability with Python libraries
- Reverse-mode automatic differentiation
- Script mode for production use

Performance Comparison

Framework	*Throughput (higher is better)*					
	AlexNet	VGG-19	ResNet-50	MobileNet	GNMTv2	NCF
Chainer	778 ± 15	N/A	**219** ± 1	N/A	N/A	N/A
CNTK	845 ± 8	84 ± 3	210 ± 1	N/A	N/A	N/A
MXNet	**1554** ± 22	113 ± 1	218 ± 2	444 ± 2	N/A	N/A
PaddlePaddle	933 ± 123	112 ± 2	192 ± 4	**557** ± 24	N/A	N/A
TensorFlow	1422 ± 27	66 ± 2	200 ± 1	216 ± 15	$9631 \pm 1.3\%$	$4.8e6 \pm 2.9\%$
PyTorch	1547 ± 316	**119** ± 1	212 ± 2	463 ± 17	**15512** $\pm 4.8\%$	**5.4e6** $\pm 3.4\%$

Table 1: Training speed for 6 models using 32bit floats. Throughput is measured in images per second for the AlexNet, VGG-19, ResNet-50, and MobileNet models, in tokens per second for the GNMTv2 model, and in samples per second for the NCF model. The fastest speed for each model is shown in bold.

PyTorch moving to Linux Foundation

3 MIN READ

Welcoming PyTorch to the Linux Foundation

JIM ZEMLIN | 12 SEPTEMBER 2022

Today we are more than thrilled to welcome PyTorch to the Linux Foundation. Honestly, it's hard to capture how big a deal this is for us in a single post but I'll try.

What's next?

- PyTorch and TensorFlow have become the two dominant players
- Future
  - Compiler-based operator optimization
  - Unified API standards
    - e.g., JAX with numpy compatible API
  - Data movement as a first-class citizen

Thanks!