

Lecture 7:

Machine Learning Compilers

Sitao Huang

sitaoh@uci.edu

February 22, 2022

(Ref: UIUC ECE 498 ICC: IoT and Cognitive Computing)



Tentative Schedule

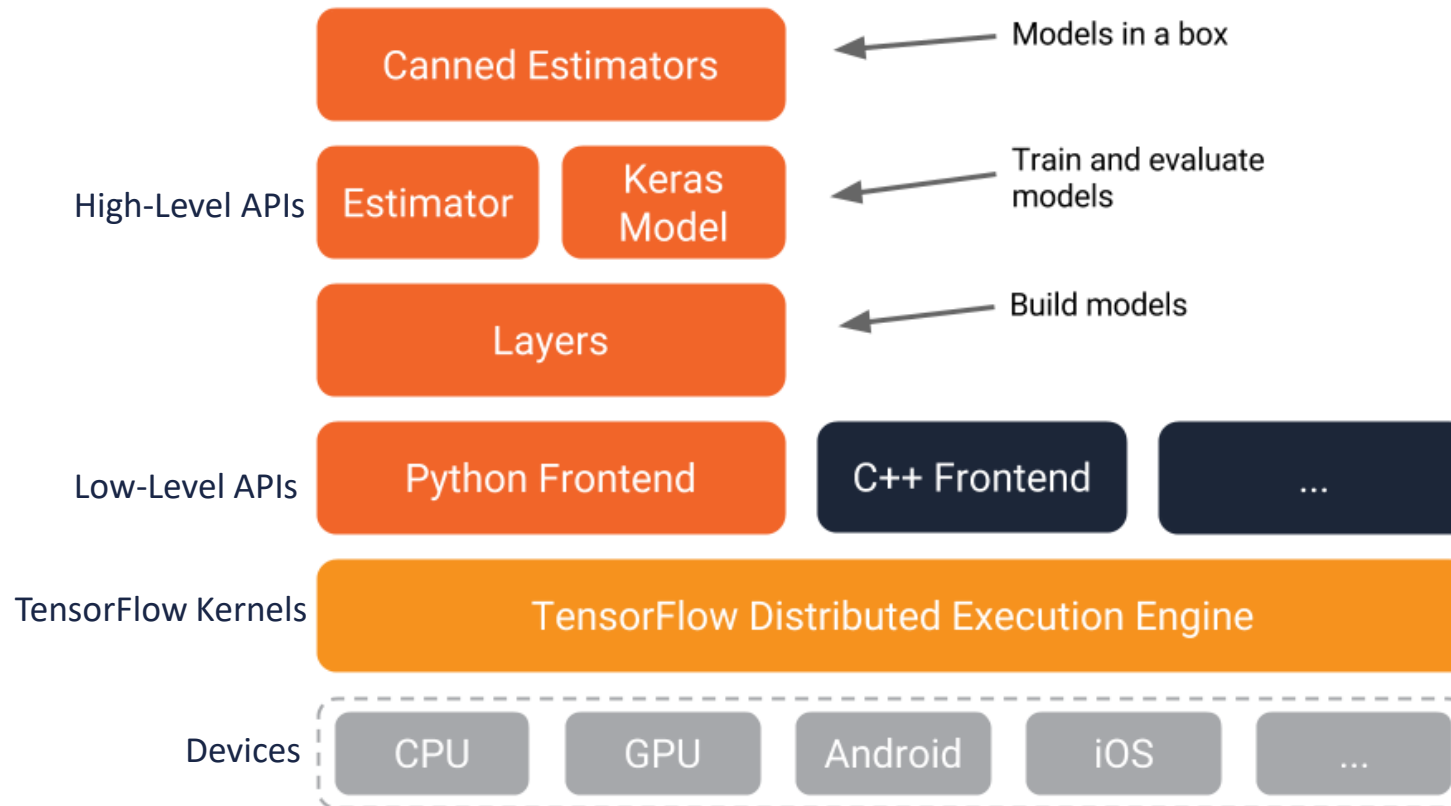
- **Week 1** (1/4, 1/6): Course Introduction
- **Week 2** (1/11, 1/13): Hardware Accelerators
- **Week 3** (1/18, 1/20): Language and Compiler Basics
- **Week 4** (1/25, 1/27): Reconfigurable Accelerators
- **Week 5** (2/1, 1/3): High-Level Synthesis
- **Week 6** (2/8, 2/10): *Midterm*
- **Week 7** (2/15, 2/17): Compilers for Accelerators
- **Week 8 (2/22, 2/24): Machine Learning Compilers**
- **Week 9** (3/1, 3/3): Emerging Architectures and Compilers
- **Week 10** (3/8, 3/10): *Project Presentations*



TensorFlow

TensorFlow

- TensorFlow is an open-source machine learning library for research and production.

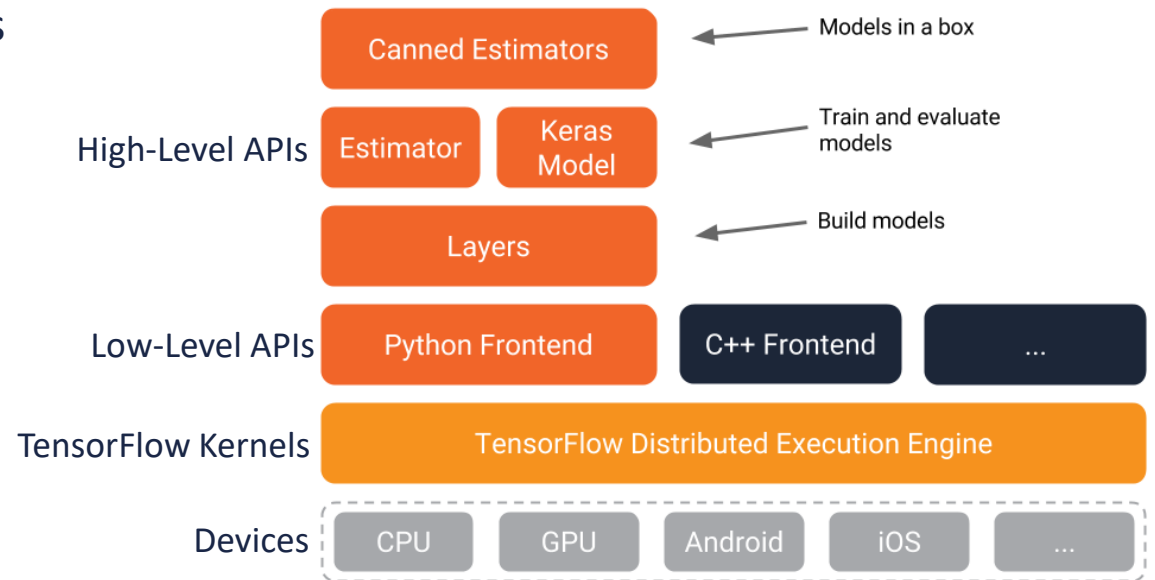


TensorFlow

- High-level APIs
 - [Keras](#), TensorFlow's high-level API for building and training deep learning models.
 - [Eager Execution](#), an API for writing TensorFlow code imperatively, without building graph.
 - [Importing Data](#), easy input pipelines to bring your data into your TensorFlow program.
 - [Estimators](#), a high-level API that provides fully-packaged models ready for large-scale training and production.
- Low-level APIs
 - [Tensors](#), the fundamental object in TensorFlow.
 - [Variables](#), represent shared, persistent state in program.
 - [Graphs](#), TensorFlow's representation of computations as dependencies between operations
 - [Sessions](#), TensorFlow's mechanism for running dataflow graphs across one or more local or remote devices.

If you are programming with the **low-level** TensorFlow API, graphs and sessions unit is essential.

If you are programming with a **high-level** TensorFlow API such as Estimators or Keras, the high-level API creates and manages graphs and sessions for you, but understanding graphs and sessions can still be helpful



Tensors

- Tensor
 - Generalization of vectors and matrices
 - Represented as n-dimensional arrays of base datatypes, `tf.Tensor`
- TensorFlow: a framework to define and run computations involving tensors
 - Build a graph of `tf.Tensor` objects
 - Detail how each tensor is computed based on the other available tensors
 - Run parts of this graph to achieve the desired results

Some types of Tensors:

- `tf.Variable`
- `tf.constant`
- `tf.placeholder`
- `tf.SparseTensor`

• tf.Variable

- A tf.Variable represents a tensor whose value can be changed by running ops on it.
- You add a variable to the graph by constructing an instance of the class Variable.

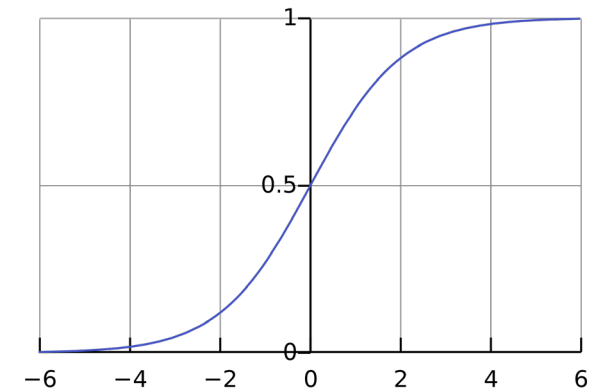
```
import tensorflow as tf

# Create a variable.
w = tf.Variable(<initial-value>, name=<optional-name>)

# Use the variable in the graph like any Tensor.
y = tf.matmul(w, ...another variable or tensor...)

# The overloaded operators are available too.
z = tf.sigmoid(w + y)

# Assign a new value to the variable with `assign()` or a related method.
w.assign(w + 1.0)
w.assign_add(1.0)
```



Sigmoid function

• tf.placeholder

```
tf.placeholder(  
    dtype,  
    shape=None,  
    name=None  
)
```

- Inserts a placeholder for a tensor that will be always fed.
- This tensor will produce an error if evaluated. Its value must be fed using the `feed_dict` optional argument to `Session.run()`, `Tensor.eval()`, or `Operation.run()`.

```
x = tf.placeholder(tf.float32, shape=(1024, 1024))  
y = tf.matmul(x, x)  
  
with tf.Session() as sess:  
    print(sess.run(y)) # ERROR: will fail because x was not fed.  
  
    rand_array = np.random.rand(1024, 1024)  
    print(sess.run(y, feed_dict={x: rand_array})) # Will succeed.
```

- **tf.SparseTensor**

- Represents a sparse tensor.
- TensorFlow represents a sparse tensor as three separate dense tensors: indices, values, and dense_shape.
 - indices: A 2-D int64 tensor of shape [N, ndims], which specifies the indices of the elements in the sparse tensor that contain N nonzero values (elements are zero-indexed), each is ndims. For example, indices=[[1,3], [2,4]] specifies that the elements with indexes of [1,3] and [2,4] have nonzero values.
 - values: A 1-D tensor of any type and shape [N], which supplies the values for each element in indices. For example, given indices=[[1,3], [2,4]], the parameter values=[18, 3.6] specifies that element [1,3] of the sparse tensor has a value of 18, and element [2,4] of the tensor has a value of 3.6.
 - dense_shape: A 1-D int64 tensor of shape [ndims], which specifies the dense_shape of the sparse tensor. Takes a list indicating the number of elements in each dimension. For example, dense_shape=[3,6] specifies a two-dimensional 3x6 tensor, dense_shape=[2,3,4] specifies a three-dimensional 2x3x4 tensor, and dense_shape=[9] specifies a one-dimensional tensor with 9 elements.

```
SparseTensor(indices=[[0, 0], [1, 2]], values=[1, 2], dense_shape=[3, 4])
```

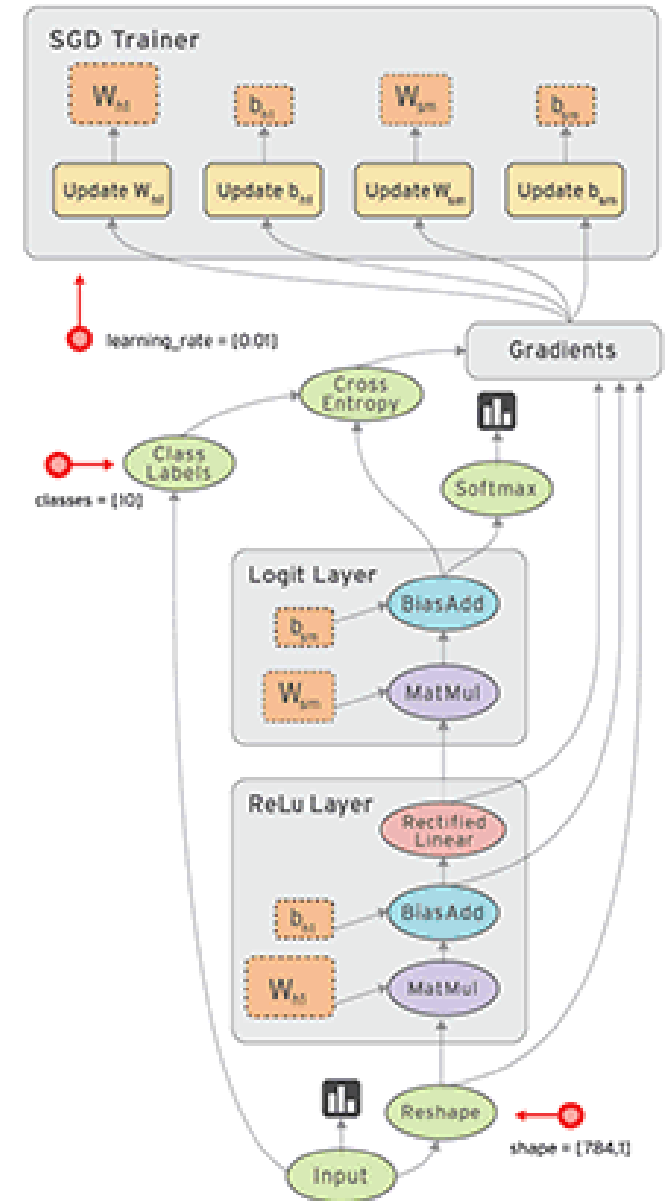
represents

```
[[1, 0, 0, 0]  
 [0, 0, 2, 0]  
 [0, 0, 0, 0]]
```

Dataflow Graphs

TensorFlow uses a **dataflow graph** (`tf.Graph`) to represent your computation in terms of the dependencies between individual operations.

- Low-level programming in TensorFlow:
 - first define the dataflow graph
 - then create a TensorFlow **session** to run parts of the graph across a set of local and remote devices
- Why Dataflow Graphs?
 - **Parallelism**: explicit data dependencies
 - **Distributed execution**: partition to multiple devices
 - **Compilation**: enable compiler optimization e.g. fusion
 - **Portability**: language-independent



Sessions

- TensorFlow uses the `tf.Session` class to represent a connection between the client program and the runtime.
 - provide access to devices in the local machine
 - provide access to remote devices using the distributed TensorFlow runtime
 - caches `tf.Graph` information
- Creating a `tf.Session`

```
# Create a default in-process session.  
with tf.Session() as sess:  
    # ...  
  
# Create a remote session.  
with tf.Session("grpc://example.org:2222"):  
    # ...
```

- `tf.Session` owns physical resources (such as GPUs and network connections)
- used as a context manager (in a `with` block) that automatically closes the session when you exit the block.

Executing a graph in a tf.Session

The `tf.Session.run` method is the main mechanism for running a `tf.Operation` or evaluating a `tf.Tensor`. You can pass one or more `tf.Operation` or `tf.Tensor` objects to `tf.Session.run`, and TensorFlow will execute the operations that are needed to compute the result.

- Example:

```
x = tf.constant([[37.0, -23.0], [1.0, 4.0]])
w = tf.Variable(tf.random_uniform([2, 2]))
y = tf.matmul(x, w)
output = tf.nn.softmax(y)
init_op = w.initializer
```

```
with tf.Session() as sess:
    # Run the initializer on `w`.
    sess.run(init_op)
```

```
# Evaluate `output`. `sess.run(output)` will return a NumPy array containing
# the result of the computation.
print(sess.run(output))
```

```
# Evaluate `y` and `output`. Note that `y` will only be computed once, and its
# result used both to return `y_val` and as an input to the `tf.nn.softmax()`
# op. Both `y_val` and `output_val` will be NumPy arrays.
y_val, output_val = sess.run([y, output])
```

Initializer

When you launch the graph, variables have to be explicitly initialized before you can run Ops that use their value. You can initialize a variable by running its initializer op, restoring the variable from a save file, or simply running an assign Op that assigns a value to the variable. In fact, the variable initializer op is just an assign Op that assigns the variable's initial value to the variable itself.

Nearest Neighbor Example

Load libraries, import data:

```
import numpy as np
import tensorflow as tf

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

Construct dataflow graph:

```
Training images (Xtr) and labels (Ytr) # In this example, we limit mnist data
Xtr, Ytr = mnist.train.next_batch(5000) #5000 for training (nn candidates)
Xte, Yte = mnist.test.next_batch(200) #200 for testing
```

Testing images (Xte) and labels (Yte)

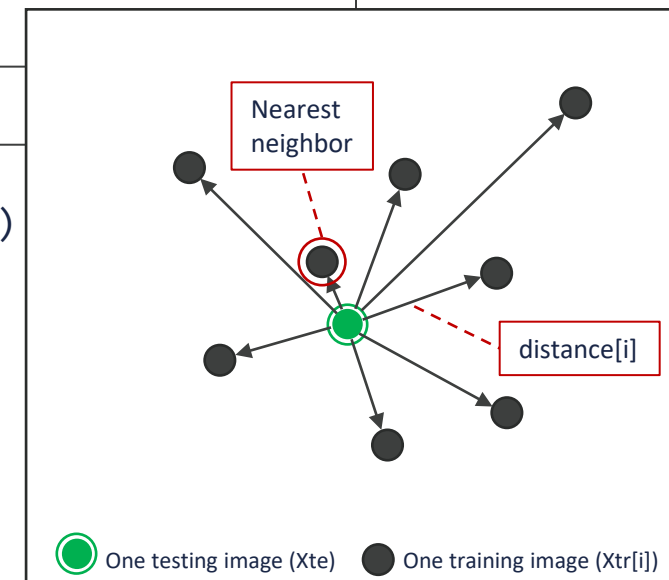
```
# tf Graph Input
xtr = tf.placeholder("float", [None, 784])
xte = tf.placeholder("float", [784])
```

Each image has $28 \times 28 = 784$ pixels, used as feature vector

```
# Nearest Neighbor calculation using L1 Distance
# Calculate L1 Distance
distance = tf.reduce_sum(tf.abs(tf.add(xtr, tf.negative(xte))), reduction_indices=1)
# Prediction: Get min distance index (Nearest neighbor)
pred = tf.argmin(distance, 0)
```

```
accuracy = 0.
```

```
# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```



L1 distance:

$$\|x^{\text{tr}} - x^{\text{te}}\|_1 = \sum_i |x_i^{\text{tr}} - x_i^{\text{te}}|$$

Nearest Neighbor Example

Training:

```
with tf.Session() as sess:
    sess.run(init)

    # loop over test data
    for i in range(len(Xte)):
        # Get nearest neighbor
        nn_index = sess.run(pred, feed_dict={xtr: Xtr, xte: Xte[i, :]})
        # Get nearest neighbor class label and compare it to its true label
        print "Test", i, "Prediction:", np.argmax(Ytr[nn_index]), \
              "True Class:", np.argmax(Yte[i])
        # Calculate accuracy
        if np.argmax(Ytr[nn_index]) == np.argmax(Yte[i]):
            accuracy += 1./len(Xte)
    print "Done!"
    print "Accuracy:", accuracy
```

Using GPUs

If a TensorFlow operation has both CPU and GPU implementations, the GPU devices will be given priority when the operation is assigned to a device*. For example, matmul has both CPU and GPU kernels. On a system with devices cpu:0 and gpu:0, gpu:0 will be selected to run matmul.

(*need to use GPU-enabled TensorFlow)

Using a single GPU in multi-GPU system

```
# Creates a graph.
with tf.device('/device:GPU:2'):
    a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3], name='a')
    b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2], name='b')
    c = tf.matmul(a, b)
# Creates a session with log_device_placement set to True.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
# Runs the op.
print(sess.run(c))
```

Using multiple GPUs

```
# Creates a graph.
c = []
for d in ['/device:GPU:2', '/device:GPU:3']:
    with tf.device(d):
        a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[2, 3])
        b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], shape=[3, 2])
        c.append(tf.matmul(a, b))
with tf.device('/cpu:0'):
    sum = tf.add_n(c)
# Creates a session with log_device_placement set to True.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
# Runs the op.
print(sess.run(sum))
```




```
Device mapping:
/job:localhost/replica:0/task:0/device:GPU:0 -> device: 0, name: Tesla K20m,
pci bus
id: 0000:02:00.0
/job:localhost/replica:0/task:0/device:GPU:1 -> device: 1, name: Tesla K20m,
pci bus
id: 0000:03:00.0
/job:localhost/replica:0/task:0/device:GPU:2 -> device: 2, name: Tesla K20m,
pci bus
id: 0000:83:00.0
/job:localhost/replica:0/task:0/device:GPU:3 -> device: 3, name: Tesla K20m,
pci bus
id: 0000:84:00.0
Const_3: /job:localhost/replica:0/task:0/device:GPU:3
Const_2: /job:localhost/replica:0/task:0/device:GPU:3
MatMul_1: /job:localhost/replica:0/task:0/device:GPU:3
Const_1: /job:localhost/replica:0/task:0/device:GPU:2
Const: /job:localhost/replica:0/task:0/device:GPU:2
MatMul: /job:localhost/replica:0/task:0/device:GPU:2
AddN: /job:localhost/replica:0/task:0/cpu:0
[[ 44.  56.]
 [ 98. 128.]]
```


Cluster Specification

The cluster specification dictionary `tf.train.ClusterSpec` maps job names to lists of network addresses. For example,

```
tf.train.ClusterSpec({"local": ["localhost:2222",  
                                "localhost:2223"]})
```



```
/job:local/task:0  
/job:local/task:1
```

```
tf.train.ClusterSpec(  
    "worker": [  
        "worker0.example.com:2222",  
        "worker1.example.com:2222",  
        "worker2.example.com:2222"  
    ],  
    "ps": [  
        "ps0.example.com:2222",  
        "ps1.example.com:2222"  
    ]  
)
```



```
/job:worker/task:0  
/job:worker/task:1  
/job:worker/task:2  
/job:ps/task:0  
/job:ps/task:1
```

Placing operations on different devices

- A **device specification** has the following form:

```
/job:<JOB_NAME>/task:<TASK_INDEX>/device:<DEVICE_TYPE>:<DEVICE_INDEX>
```

```
# Operations created outside either context will run on the "best possible"
# device. For example, if you have a GPU and a CPU available, and the operation
# has a GPU implementation, TensorFlow will choose the GPU.
weights = tf.random_normal(...)

with tf.device("/device:CPU:0"):
    # Operations created in this context will be pinned to the CPU.
    img = tf.decode_jpeg(tf.read_file("img.jpg"))

with tf.device("/device:GPU:0"):
    # Operations created in this context will be pinned to the GPU.
    result = tf.matmul(weights, img)
```

If you are deploying TensorFlow in a typical distributed configuration, you might specify the job name and task ID to place variables on a task in the parameter server job ("/job:ps"), and the other operations on task in the worker job ("/job:worker"):

```
with tf.device("/job:ps/task:0"):
    weights_1 = tf.Variable(tf.truncated_normal([784, 100]))
    biases_1 = tf.Variable(tf.zeros([100]))

with tf.device("/job:ps/task:1"):
    weights_2 = tf.Variable(tf.truncated_normal([100, 10]))
    biases_2 = tf.Variable(tf.zeros([10]))

with tf.device("/job:worker"):
    layer_1 = tf.matmul(train_batch, weights_1) + biases_1
    layer_2 = tf.matmul(train_batch, weights_2) + biases_2
```

TensorFlow Computation Graphs (Visualized in TensorBoard)

TensorBoard

EVENTS IMAGES GRAPH HISTOGRAMS



Fit to screen

Run

cifar-train

Upload

Choose File

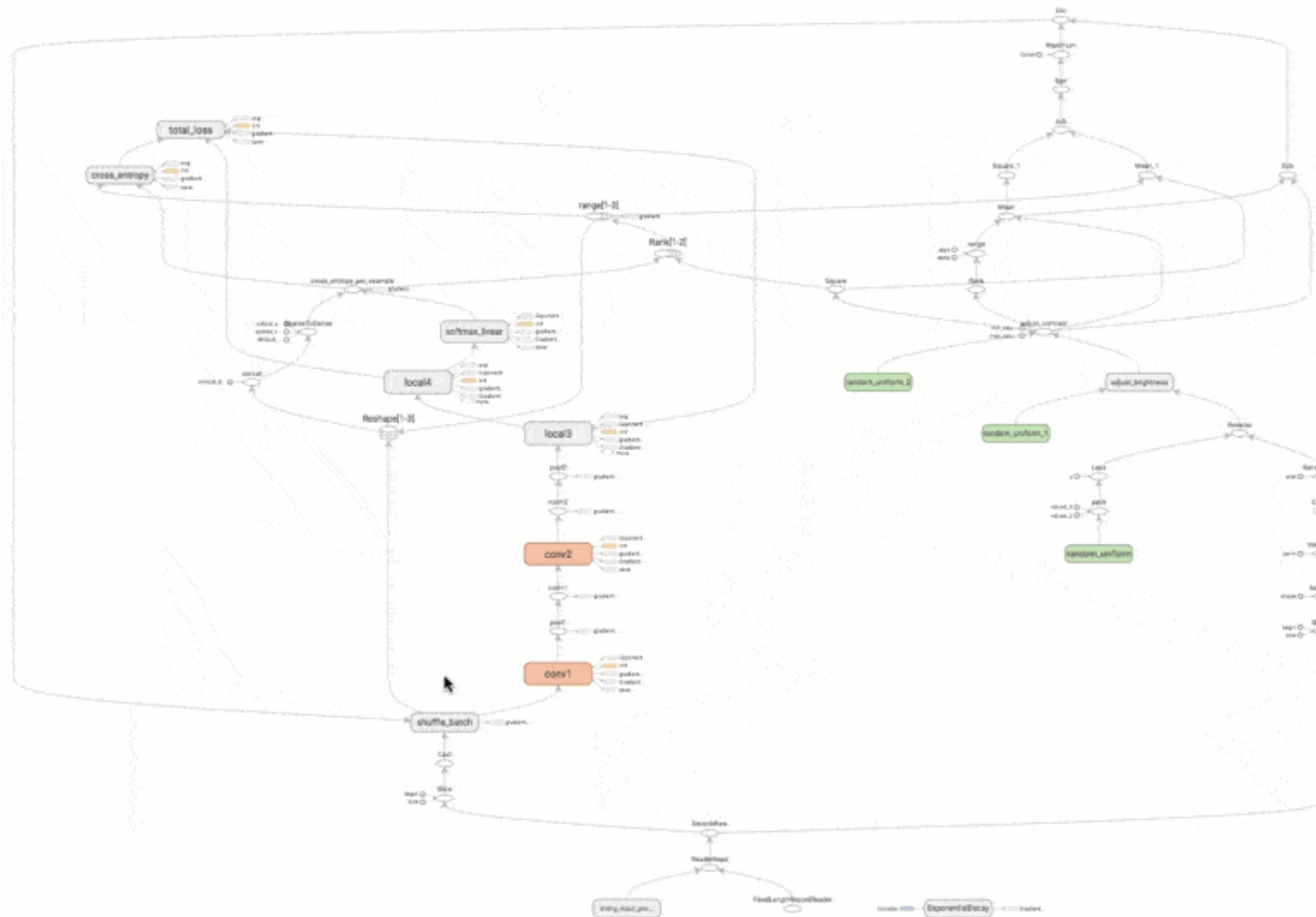
Color

Structure

color: same substructure
gray: unique substructure

Main Graph

Auxiliary nodes



Variable

GlobalStep

LearningRate

WeightDecay

DropoutRate

BatchNormOffset

BatchNormScale

BatchNormOffset

BatchNormScale

BatchNormOffset

BatchNormScale

BatchNormOffset

BatchNormScale

BatchNormOffset

BatchNormScale

BatchNormOffset

BatchNormScale

BatchNormOffset

BatchNormScale

BatchNormOffset

BatchNormScale

BatchNormOffset

BatchNormScale

BatchNormOffset

BatchNormScale

BatchNormOffset

BatchNormScale

BatchNormOffset

BatchNormScale

BatchNormOffset

BatchNormScale

BatchNormOffset

BatchNormScale

BatchNormOffset

BatchNormScale

BatchNormOffset

BatchNormScale

BatchNormOffset

BatchNormScale

BatchNormOffset

BatchNormScale

BatchNormOffset

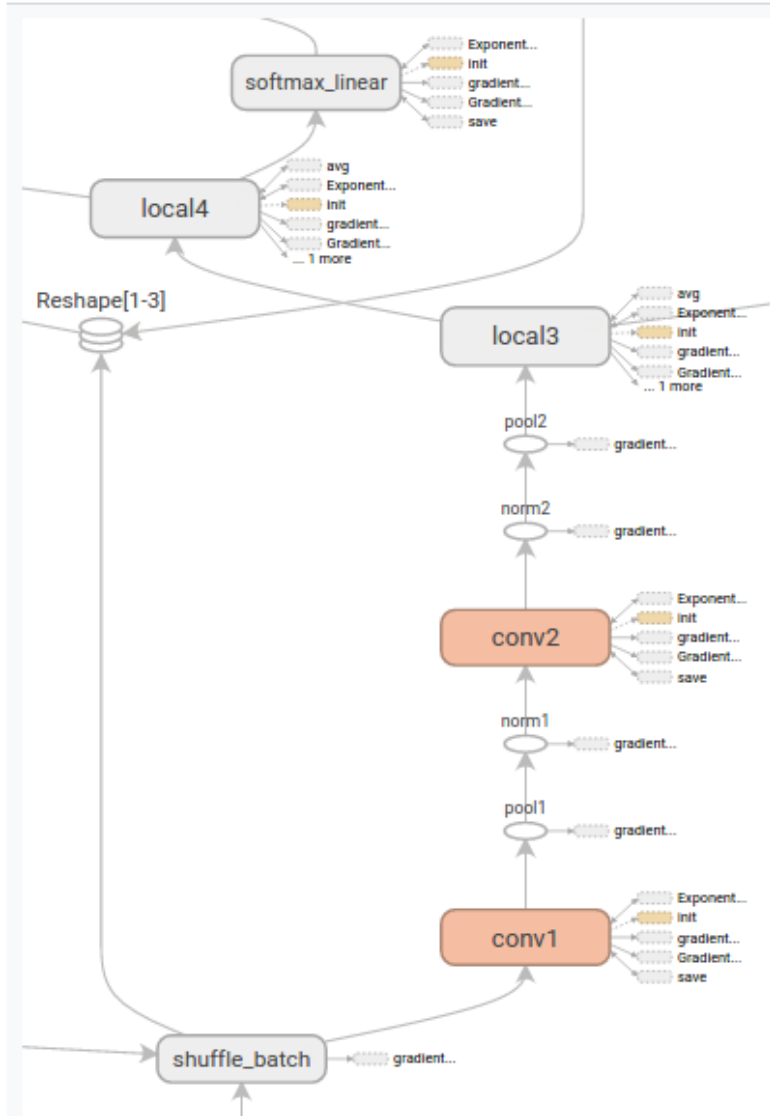
BatchNormScale

Graph

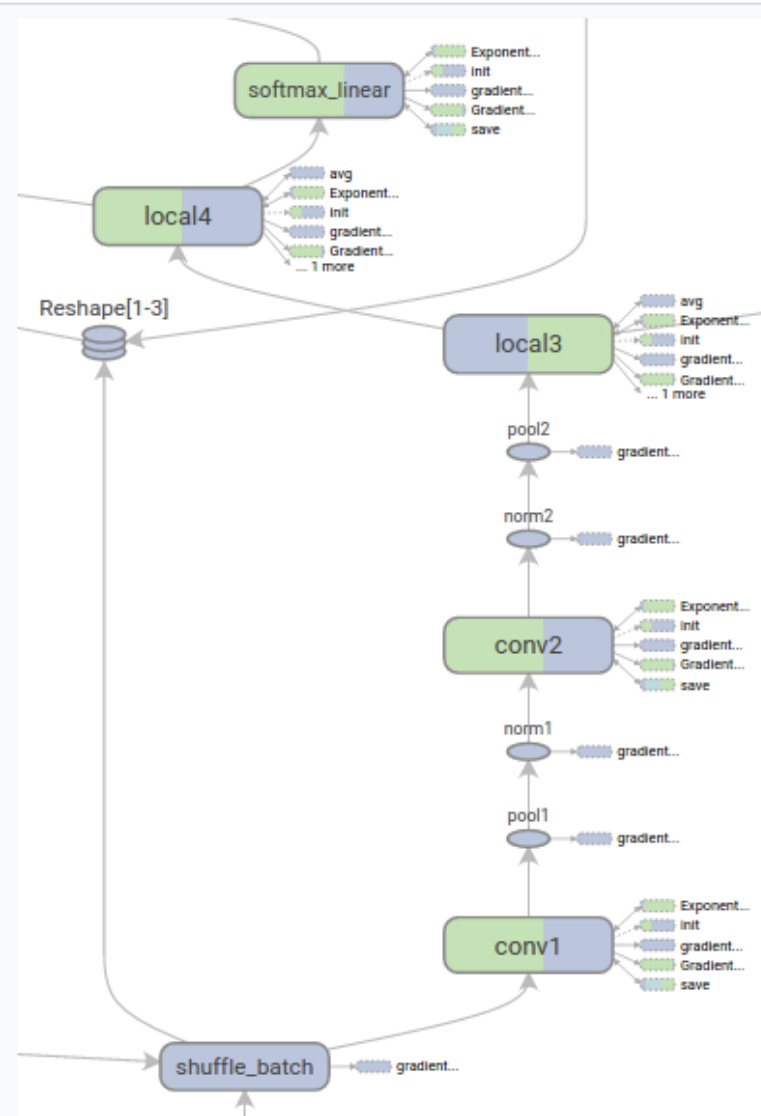
(* = expandable)

- Namespace*
- OpNode
- Unconnected series*
- Connected series*
- Constant
- Summary
- Dataflow edge
- Control dependency edge
- Reference edge

TensorFlow Computation Graphs (Visualized in TensorBoard)



Structure view: The gray nodes have unique structure. The orange **conv1** and **conv2** nodes have the same structure, and analogously for nodes with other colors.



Device view: Name scopes are colored proportionally to the fraction of devices of the operation nodes inside them. Here, purple means GPU and the green is CPU.



TensorFlow Lite

TensorFlow Lite

A set of tools to help developers run TensorFlow models on mobile, embedded, and IoT devices.

Main Components

- [TensorFlow Lite interpreter](#): runs specially optimized models on many different hardware types, including mobile phones, embedded Linux devices, and microcontrollers.
- [TensorFlow Lite converter](#): *converts* TensorFlow models into an efficient form for use by the interpreter, and it can introduce *optimizations* to improve binary size and performance.

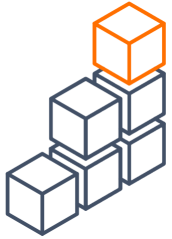
Key Features

- [Interpreter](#) tuned for on-device ML.
- Diverse platform support, covering [Android](#) and [iOS](#) devices, embedded Linux, and microcontrollers.
- APIs for multiple languages including Java, Swift, Objective-C, C++, and Python.
- High performance, with [hardware acceleration](#) on supported devices, device-optimized kernels, and [pre-fused activations and biases](#).
- Model optimization tools, including [quantization](#), that can reduce size and increase performance of models without sacrificing accuracy.
- Efficient model format, using a [FlatBuffer](#) that is optimized for small size and portability.
- [Pre-trained models](#) for common machine learning tasks that can be customized to your application.
- [Samples and tutorials](#) that show you how to deploy machine learning models on supported platforms.

Limitations

- Supports a limited subset of TensorFlow operators. You can use [TensorFlow Select](#) to include TensorFlow operations not yet supported.
- Currently no support for on-device training

TensorFlow Lite Development Flow



Pick a model

Bring your own TensorFlow model, find a model online, or pick a model from [Pre-trained models](#) to drop in or retrain.



Convert the model

If you're using a custom model, use the [TensorFlow Lite converter](#) and a few lines of Python to convert it to the TensorFlow Lite format.



Deploy to your device

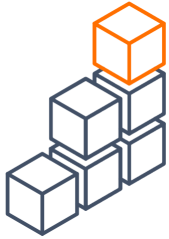
Run your model on-device with the [TensorFlow Lite interpreter](#), with APIs in many languages.



Optimize your model

Use [Model Optimization Toolkit](#) to reduce your model's size and increase its efficiency with minimal impact on accuracy.

TensorFlow Lite Development Flow



Pick a model

Bring your own TensorFlow model, find a model online, or pick a model from [pre-trained models](#) to drop in or retrain.

- A TensorFlow model is a data structure that contains the logic and knowledge of a machine learning network trained to solve a particular problem.
- To use a model with TensorFlow Lite, you must start with a regular TensorFlow model, and then [convert the model](#). You cannot create or train a model using TensorFlow Lite.

Options

- Use a pre-trained TensorFlow / TensorFlow Lite model (list of pre-trained [models](#))
- Re-train a model (transfer learning) to your applications
- Train a custom model (covered in last lecture)

TensorFlow Lite Development Flow



Convert the model

If you're using a custom model, use the [TensorFlow Lite converter](#) and a few lines of Python to convert it to the TensorFlow Lite format.

Convert a TensorFlow `SavedModel` into the TensorFlow Lite format:

```
import tensorflow as tf

converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
tflite_model = converter.convert()
open("converted_model.tflite", "wb").write(tflite_model)
```

It can also introduce optimizations, which are covered in later slides.

The converter can also be used from the [command line](#), but the Python API is recommended.

Input types for the converter:

- For TensorFlow 1.x models: [SavedModel directories](#), Frozen GraphDef (models generated by [freeze_graph.py](#)), [Keras](#) HDF5 models, Models taken from a `tf.Session`.
- For TensorFlow 2.x models: [SavedModel directories](#), [tf.keras models](#), [Concrete functions](#)

TensorFlow Lite Development Flow



Deploy to your device

Run your model on-device with the [TensorFlow Lite interpreter](#), with APIs in many languages.

TensorFlow Lite Interpreter

The [TensorFlow Lite interpreter](#) is a library that takes a model file, executes the operations it defines on input data, and provides access to the output. It provides a simple API for running TensorFlow Lite models from Java, Swift, Objective-C, C++, and Python.

GPU Acceleration and Delegates

The [GPU Delegate](#) allows the interpreter to run appropriate operations on the device's GPU. An example of GPU Delegate being used from Java:

```
GpuDelegate delegate = new GpuDelegate();
Interpreter.Options options = (new Interpreter.Options()).addDelegate(delegate);
Interpreter interpreter = new Interpreter(tensorflow_lite_model_file, options);
```

To add support for new hardware accelerators you can [define your own delegate](#).

Platforms

- [Android](#) and [iOS](#)
- [Linux](#)
- Microcontrollers: Use [TensorFlow Lite for Microcontrollers](#)

TensorFlow Lite Development Flow



Optimize your model

Use [Model Optimization Toolkit](#) to reduce your model's size and increase its efficiency with minimal impact on accuracy.

Performance

The goal of model optimization is to reach the ideal balance of performance, model size, and accuracy on a given device. [Performance best practices](#) can help guide you through this process.

Quantization

TensorFlow Lite supports reducing precision of values from full floating point to half-precision floats (float16) or 8-bit integers: [post-training quantization](#). An example of quantizing: a `SavedModel`:

```
import tensorflow as tf
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_dir)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_quantized_model = converter.convert()
open("converted_model.tflite", "wb").write(tflite_quantized_model)
```

Converts only the weights
from floating point to integer

Can also force all inputs and outputs to be quantized as well:

```
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8 # or tf.uint8
converter.inference_output_type = tf.int8 # or tf.uint8
```

Other Optimization

Check out [Model Optimization](#) for more information.

TensorFlow Lite for *Microcontrollers*

Microcontrollers

- A small computer on a single integrated circuit (IC) chip, contains CPU(s), memory and I/O peripherals
- Small memory (a few kilobytes), low power (a few milliwatts or even less than a milliwatt)
- Executes instructions directly (bare metal, no OS) or only a tailored minimum OS (no complete Linux)

Ref: Arm Cortex-A9 CPU:
500mW

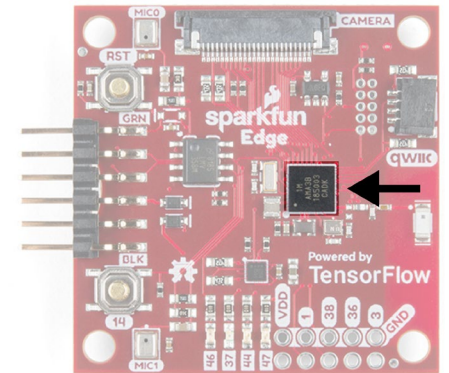
(Note: Raspberry Pi has more powerful hardware and embedded Linux, so we don't consider it as microcontroller here. Raspberry Pi can support standard TensorFlow Lite)

Why ML on Microcontrollers is Important?

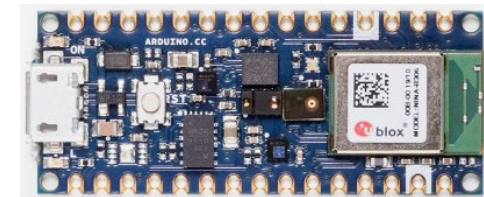
- Microcontrollers are embedded in many devices to provide basic computation
- ML on microcontroller can boost the intelligence of billions of devices used in our lives
- Preserve privacy, no data leaves the device

TensorFlow Lite for *Microcontrollers*

- A special version of TensorFlow Lite designed to run ML models on *microcontrollers* and other devices with *only few kilobytes of memory*
- Doesn't require operating system support, any standard C or C++ libraries, or dynamic memory allocation
- List of [supported microcontrollers](#)



The SparkFun Edge board
(source: sparkfun.com)



Arduino Nano 33 BLE Sense board
(source: arduino.cc)

Run Inference on *Microcontrollers*

Let's look at a very basic "Hello World" example that loads model, prepares input, checks input shape, runs inference, and finally checks output.

1. Include the headers

```
// library headers
#include "tensorflow/lite/micro/all_ops_resolver.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"
// model headers (contains g_model, a C array of weights)
#include "tensorflow/lite/micro/examples/hello_world/model.h"
// unit test framework headers
#include "tensorflow/lite/micro/testing/micro_test.h"
```

2. Set up logging

```
tflite::MicroErrorReporter micro_error_reporter;
tflite::ErrorReporter* error_reporter = &micro_error_reporter;
```

3. Load a model

```
const tflite::Model* model = ::tflite::GetModel(g_model);
if (model->version() != TFLITE_SCHEMA_VERSION) {
    TF_LITE_REPORT_ERROR(error_reporter,
        "Model provided is schema version %d not equal "
        "to supported version %d.\n",
        model->version(), TFLITE_SCHEMA_VERSION);
}
```

Run Inference on *Microcontrollers*

4. Instantiate operations resolver (used by interpreter to access the operations used by the model)

```
tflite::AllOpsResolver resolver;
```

5. Allocate memory (for input, output and intermediate arrays)

```
const int tensor_arena_size = 2 * 1024;  
uint8_t tensor_arena[tensor_arena_size];
```

6. Instantiate interpreter

```
tflite::MicroInterpreter interpreter(model, resolver, tensor_arena,  
                                     tensor_arena_size, error_reporter);
```

7. Allocate tensors

```
interpreter.AllocateTensors();
```

8. Validate input shape

```
// Obtain a pointer to the model's input tensor  
TfLiteTensor* input = interpreter.input(0);  
  
// Make sure the input has the properties we expect  
TF_LITE_MICRO_EXPECT_NE(nullptr, input);  
TF_LITE_MICRO_EXPECT_EQ(2, input->dims->size);  
// The value of each element gives the length of the corresponding tensor.  
// We should expect two single element tensors (one is contained within the  
// other).  
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[0]);  
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[1]);  
// The input is a 32 bit floating point value  
TF_LITE_MICRO_EXPECT_EQ(kTfLiteFloat32, input->type);
```

Run Inference on *Microcontrollers*

9. Provide an input value

```
input->data.f[0] = 0.;
```

10. Run the model

```
TfLiteStatus invoke_status = interpreter.Invoke();  
if (invoke_status != kTfLiteOk) {  
    TF_LITE_REPORT_ERROR(error_reporter, "Invoke failed\n");  
}
```

11. Obtain the output

```
TfLiteTensor* output = interpreter.output(0);  
TF_LITE_MICRO_EXPECT_EQ(2, output->dims->size);  
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[0]);  
TF_LITE_MICRO_EXPECT_EQ(1, input->dims->data[1]);  
TF_LITE_MICRO_EXPECT_EQ(kTfLiteFloat32, output->type);  
  
// Obtain the output value from the tensor  
float value = output->data.f[0];  
// Check that the output value is within 0.05 of the expected value  
TF_LITE_MICRO_EXPECT_NEAR(0., value, 0.05);
```

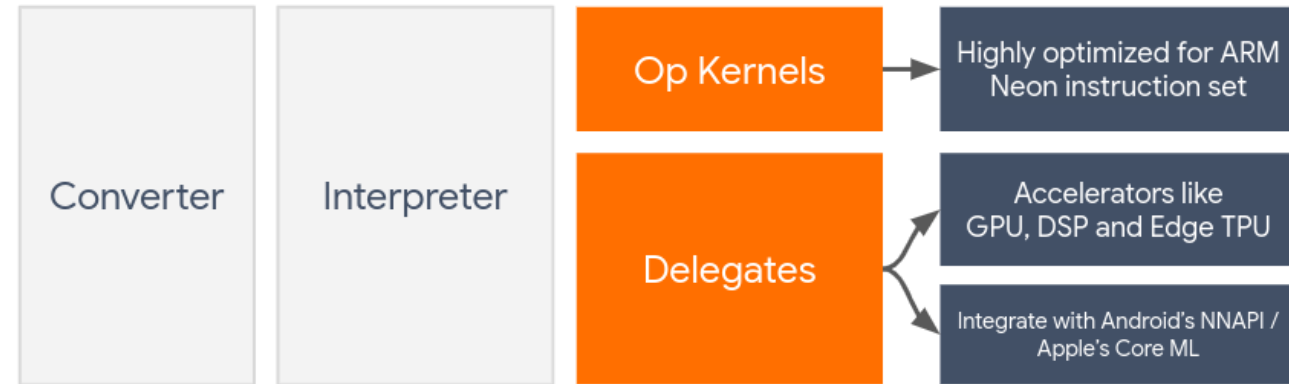
Complete code is here: [hello world test.cc](https://github.com/tensorflow/tf-lite/blob/master/examples/hello_world_test.cc).

TensorFlow Lite Delegates

- **Delegates** enable hardware acceleration of TensorFlow Lite models (part or whole) by leveraging on-device accelerators such as GPU, DSP, and Edge TPU.
- TensorFlow Lite default backend: CPU kernels optimized for [ARM Neon](#) instruction set.
- Most modern cell phones or embedded platforms contain specialized processors, each programmed with special APIs.
- TensorFlow Lite's Delegate API acts as a bridge between the TFLite runtime and these special lower-level APIs.

Delegate Examples

- **GPU delegate**: used on both Android and iOS, optimized to run 32-bit and 16-bit floating-point based models on GPU
- **NNAPI delegate** (Android devices with GPU, DSP, and NPU) and Hexagon delegate (Android devices with Hexagon DSP)
- **Core ML delegate** for Neural Engines in newer iPhones and iPads (A12 SoC or higher), accelerates inference for 32-bit or 16-bit floating-point models



Model Type	GPU	NNAPI	Hexagon	Core ML
Floating-point (32 bit)	Yes	Yes	No	Yes
Post-training float16 quantization	Yes	No	No	Yes
Post-training dynamic range quantization	Yes	Yes	No	No
Post-training integer quantization	Yes	Yes	Yes	No
Quantization-aware training	Yes	Yes	Yes	No

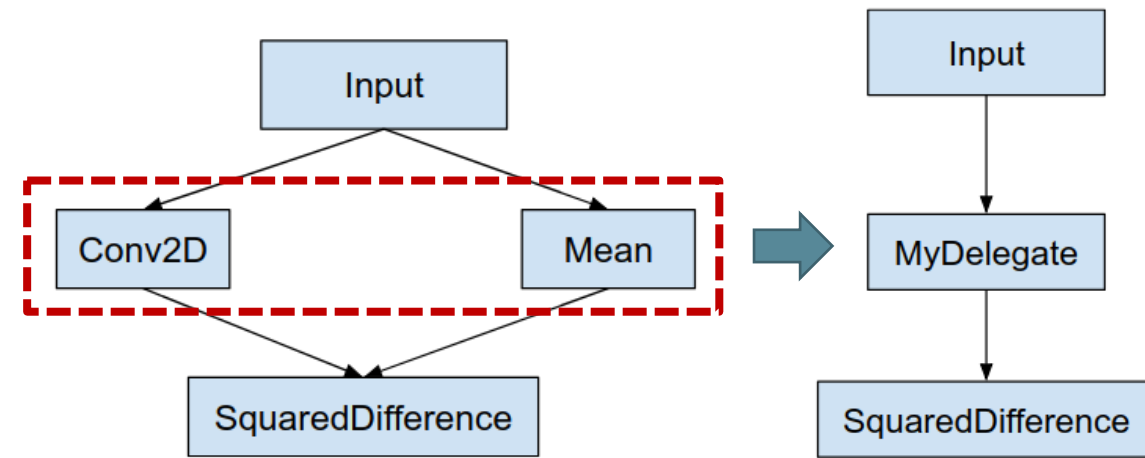
TensorFlow Lite Delegates – Custom Delegates

In some cases, you may need to create custom delegate:

- Integrate a **new ML engine** not supported by any existing delegate
- Have a **custom hardware accelerator** that improves runtime for known scenarios
- Developing CPU **optimizations** (such as operator fusing) that can speed up certain models

TensorFlow Lite Graph Transformation Rules (when applying custom delegates):

- Specific operations that could be handled by the delegate are put into a partition while still satisfying the original computing workflow dependencies among operations.
(supported operations are put into a partition)
- Each to-be-delegated partition only has input and output nodes that are not handled by the delegate.
(the sizes of these partitions are maximized/aggregated whenever possible)



Example: MyDelegate accelerates Conv2D and Mean operations

Performance Considerations

- Depending on the model and the operations supported by custom delegate, the final graph may have **one or more nodes**
- If some ops are not supported by the delegate, there will be **more than one node** in the final graph
- It's better to avoid multiple partitions handled by the delegate, because **data movements** happen at the boundaries between delegate and main graph, which results in performance overhead

TensorFlow Lite Delegates – Custom Delegates

Code Example

```
// MyDelegate implements the interface of SimpleDelegateInterface.
// This holds the Delegate capabilities.
class MyDelegate : public SimpleDelegateInterface {
public:
    bool IsNodeSupportedByDelegate(const TfLiteRegistration* registration,
                                   const TfLiteNode* node,
                                   TfLiteContext* context) const override {
        // Only supports Add and Sub ops.
        if (kTfLiteBuiltinAdd != registration->builtin_code &&
            kTfLiteBuiltinSub != registration->builtin_code)
            return false;
        // This delegate only supports float32 types.
        for (int i = 0; i < node->inputs->size; ++i) {
            auto& tensor = context->tensors[node->inputs->data[i]];
            if (tensor.type != kTfLiteFloat32) return false;
        }
        return true;
    }
    ...

    std::unique_ptr<SimpleDelegateKernelInterface> CreateDelegateKernelInterface()
        override {
            return std::make_unique<MyDelegateKernel>();
        }
};
```

```
// My delegate kernel.
class MyDelegateKernel : public SimpleDelegateKernelInterface {
public:
    ...
}
```



Resources – TensorFlow Lite

- [TensorFlow Lite Guide](#)
- [TensorFlow Lite Examples and Pre-Trained Models](#)
- [TensorFlow Lite Tutorials](#)
- [TensorFlow Lite Inference](#)
- [TensorFlow Lite Optimization](#)

Summary – TensorFlow Lite

TensorFlow Lite is a set of tools to help developers run TensorFlow models on mobile, embedded, and IoT devices.

TensorFlow Lite Key Features

- [*Interpreter*](#) tuned for on-device ML.
- Diverse platform support, covering [Android](#) and [iOS](#) devices, embedded Linux, and microcontrollers.
- APIs for multiple languages including Java, Swift, Objective-C, C++, and Python.
- High performance, with [hardware acceleration](#) on supported devices, device-optimized kernels, and [pre-fused activations and biases](#).
- Model optimization tools, including [quantization](#), that can reduce size and increase performance of models without sacrificing accuracy.
- Efficient model format, using a [FlatBuffer](#) that is optimized for small size and portability.
- [Pre-trained models](#) for common machine learning tasks that can be customized to your application.
- [Samples and tutorials](#) that show you how to deploy machine learning models on supported platforms.

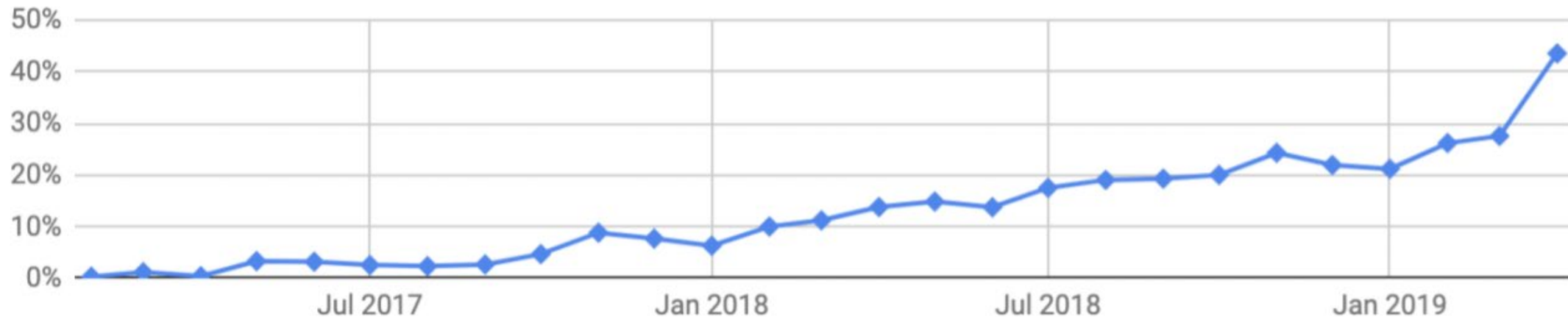
PyTorch

- Introduction
- Quick overview of building DNNs in PyTorch
- Comparison against TensorFlow



PyTorch

- PyTorch is a Python-based scientific computing package targeted at two sets of audiences:
 - A replacement for NumPy to use the power of GPUs
 - A deep learning research platform that provides maximum flexibility and speed

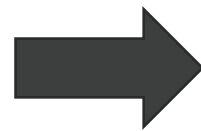
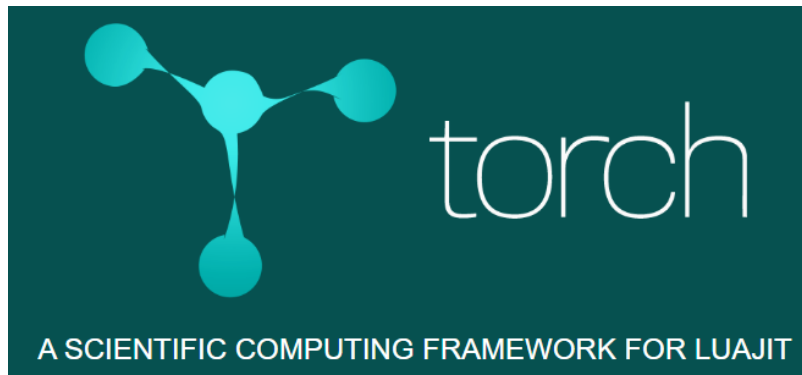


Among arXiv papers each month that mention common deep learning frameworks, percentage of them that mention PyTorch. (From the figure 3 in “PyTorch: An Imperative Style, High-Performance Deep Learning Library”)

PyTorch

- Brief history

- Before PyTorch was created, there was and still is another framework called Torch
- Torch is an open source library for machine learning and scientific computing based on the [Lua programming language](#)
 - Lua is a lightweight scripting language first introduced in 1993. It supports multiple programming models; it has its origins in application extensibility. Lua is compact and written in C, which makes it able to run on constrained embedded platforms.
- As the Lua version of Torch was aging, Torch developers started a newer version written in Python. As a result, PyTorch came to be.



PyTorch Components (1/4)

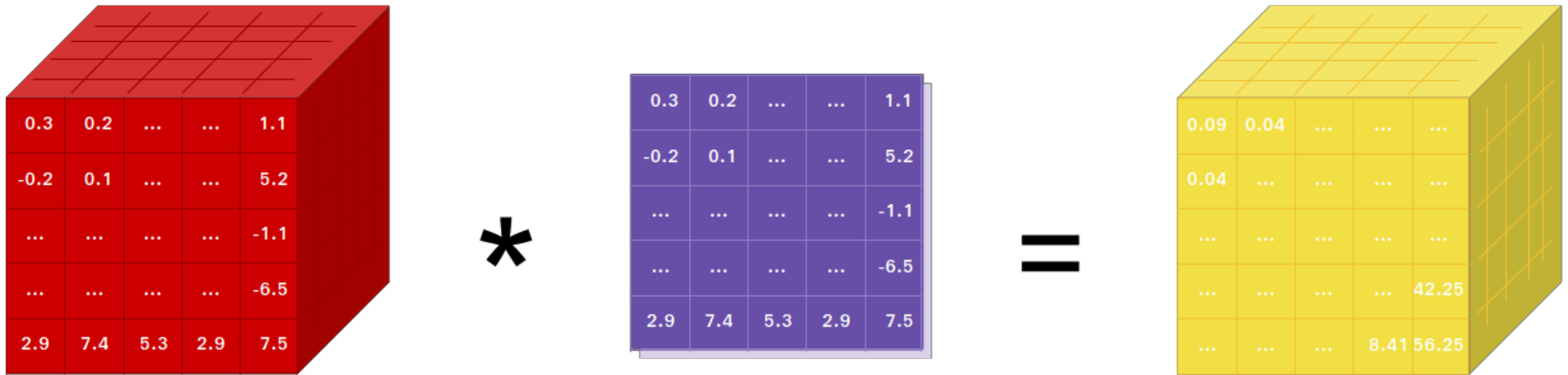
- PyTorch consists the following components:

Component	Description
<code>torch</code>	a Tensor library like NumPy, with strong GPU support
<code>torch.autograd</code>	a tape-based automatic differentiation library that supports all differentiable Tensor operations in torch
<code>torch.jit</code>	a compilation stack (TorchScript) to create serializable and optimizable models from PyTorch code
<code>torch.nn</code>	a neural networks library deeply integrated with autograd designed for maximum flexibility
<code>torch multiprocessing</code>	Python multiprocessing, but with magical memory sharing of torch Tensors across processes. Useful for data loading and Hogwild training
<code>torch.utils</code>	DataLoader and other utility functions for convenience

<https://github.com/pytorch/pytorch>

PyTorch Components (2/4)

- A GPU-Ready Tensor Library
 - PyTorch provides Tensors that can live either on the CPU or the GPU, and accelerates the computation by a huge amount.



<https://github.com/pytorch/pytorch>

PyTorch Components (3/4)

- Dynamic Neural Network construction
 - Most frameworks such as TensorFlow / Caffe have a static view of the world. One has to build a neural network, and reuse the same structure again and again. Changing the way the network behaves means that one has to start from scratch.

In PyTorch, a technique called **reverse-mode auto-differentiation** is applied, which allows you to change neural network behaviors arbitrarily with zero lag or overhead.

A graph is created on the fly



```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```

- DNN graph is created on the fly
- Backpropagation uses the dynamically-created graph



PyTorch Components (4/4)

- Acceleration libraries:
 - PyTorch integrates acceleration libraries such as Intel MKL and NVIDIA cuDNN / NCCL to maximize speed. At the core, its CPU and GPU Tensor and neural network backends (TH, THC, THNN, THCUNN) are mature and have been tested for years.



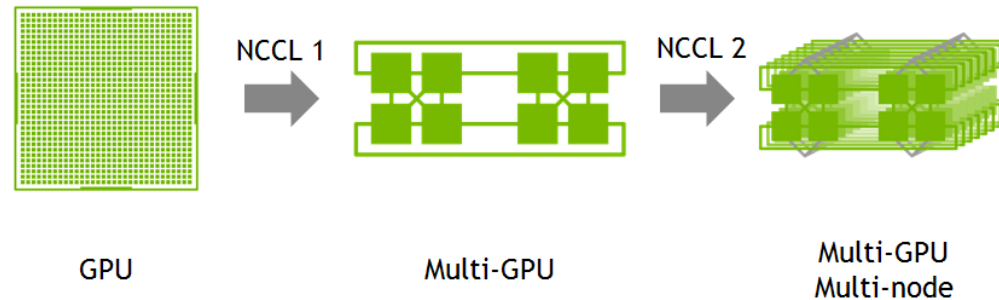
Accelerate math processing routines, increase application performance, and reduce development time. This ready-to-use math library includes:

Linear Algebra | Fast Fourier Transforms (FFT) | Vector Statistics & Data Fitting | Vector Math & Miscellaneous Solvers

<https://software.intel.com/en-us/mkl>



The NVIDIA CUDA® Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks.



The NVIDIA Collective Communications Library (NCCL) implements multi-GPU and multi-node collective communication primitives that are performance optimized for NVIDIA GPUs.

<https://developer.nvidia.com>

Tensors

- Tensors are similar to NumPy's ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing

Construct a randomly initialized matrix:

```
x = torch.rand(5, 3)
print(x)
```



```
tensor([[0.7799, 0.1989, 0.5753],
        [0.4624, 0.5366, 0.7436],
        [0.8719, 0.9631, 0.3204],
        [0.8461, 0.0422, 0.3457],
        [0.9910, 0.8967, 0.5962]])
```

Addition operation:

```
y = torch.rand(5, 3)
```

Syntax 1 `torch.add(x, y)`

Syntax 2 `result = torch.empty(5, 3)`
`torch.add(x, y, out=result)`

Syntax 3 `y.add_(x)`

NumPy Array to Torch Tensor:

```
import numpy as np
a = np.ones(5)
b = torch.from_numpy(a)
```

Torch Tensor to NumPy Array:

```
a = torch.ones(5)
b = a.numpy()
```

Autograd: Automatic Differentiation

- The autograd package provides automatic differentiation for all operations on Tensors. It is a define-by-run framework, which means that your backprop is defined by how your code is run, and that every single iteration can be different.

```
x = torch.ones(2, 2, requires_grad=True)
y = x + 2
z = y * y * 3
out = z.mean()
```

$$\text{out} = 3 / 4 * (x + 2)^2$$

Let's backprop now. Because out contains a single scalar, out.backward() is equivalent to out.backward(torch.tensor(1.)).

```
out.backward()
print(x.grad)
```

$d(\text{out}) / dx$



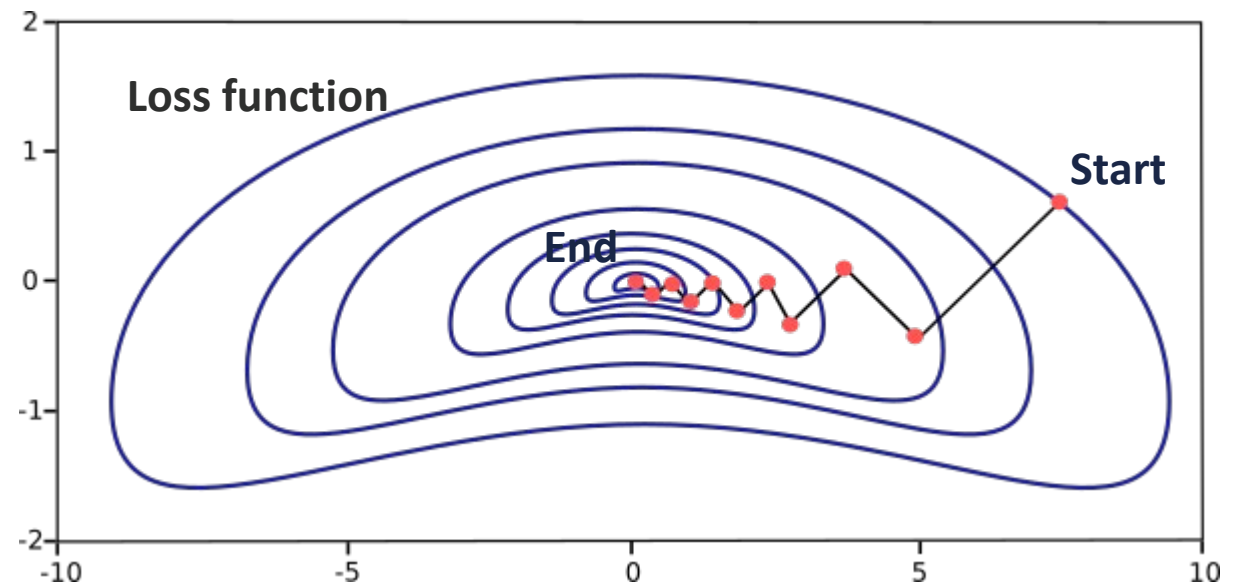
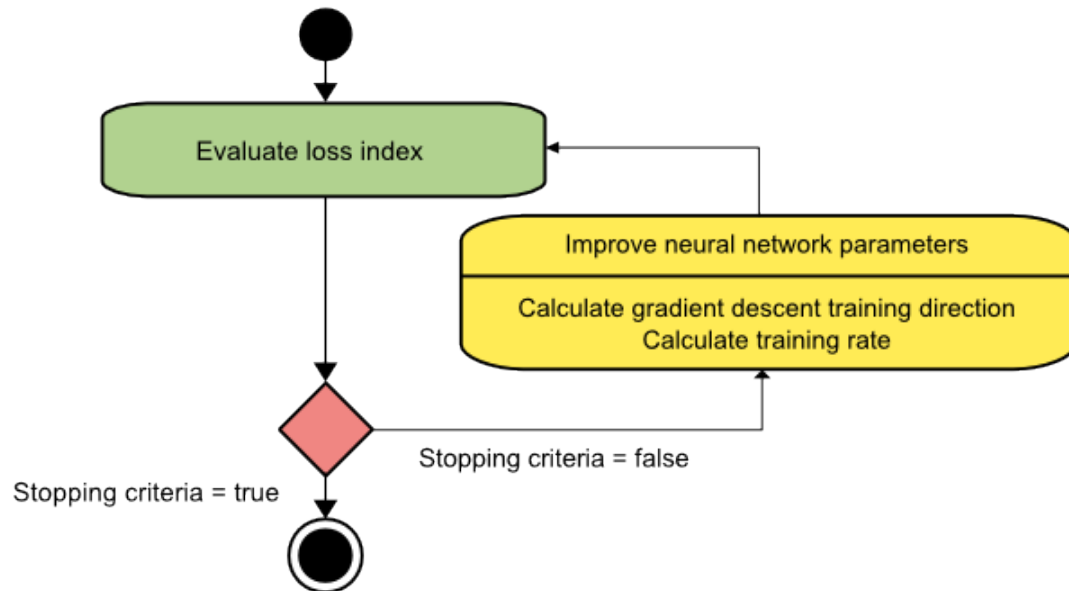
```
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

Gradient & Gradient Descent for DNN training

- The gradient is a vector, and each of its components is a partial derivative with respect to one specific variable.

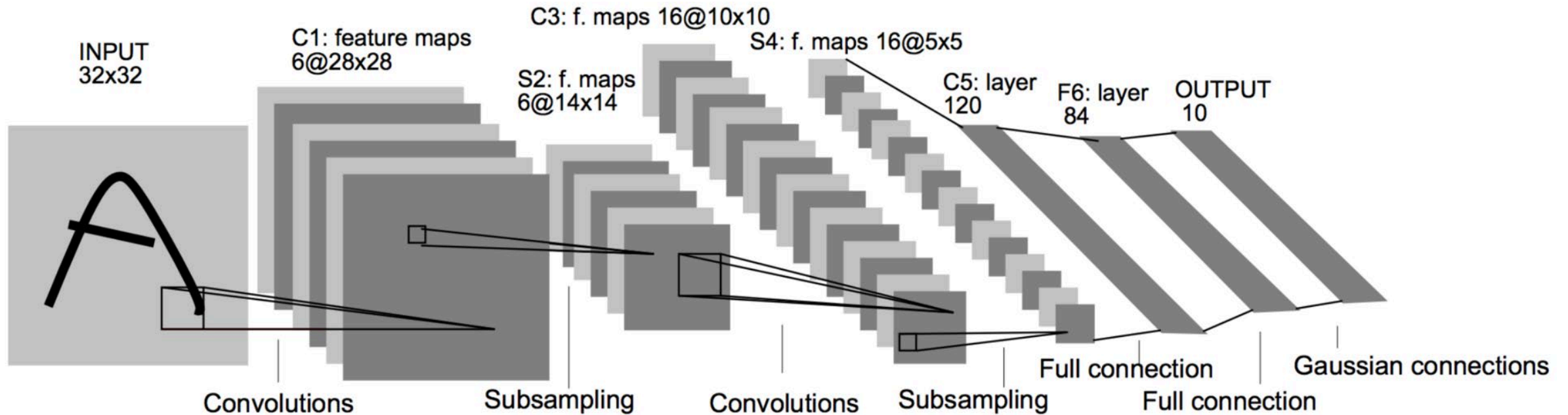
$$\text{grad}f(x_0, x_1, \dots, x_n) = \left(\frac{\partial f}{\partial x_0}, \dots, \frac{\partial f}{\partial x_j}, \dots, \frac{\partial f}{\partial x_n} \right)$$

- The gradient descent is an optimization algorithm used to minimize the loss function (evaluating the distance between the ground truth and the prediction) by iteratively updating DNN parameters in the direction of steepest descent as defined by the negative of the gradient. Less loss means better DNN performance.



Creating a neural network (1/5)

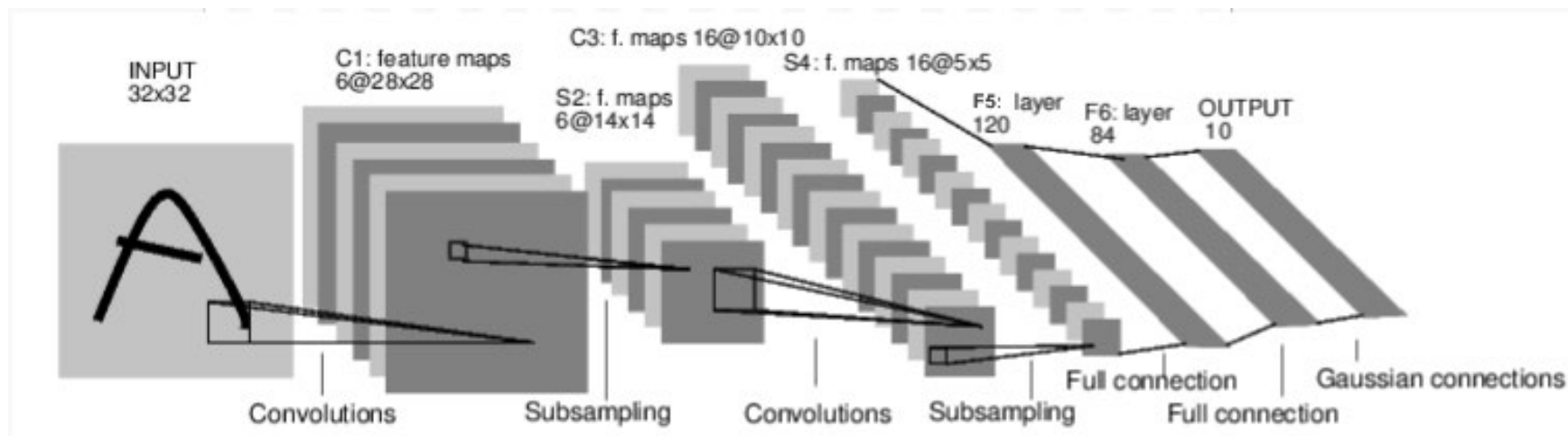
- Neural networks can be constructed using the torch.nn package.
 - Example: ConvNet, a network that classifies digit images



Two convolutional (CONV) layers and three fully-connected (FC) layers

Creating a neural network (2/5)

- It is a simple feed-forward network. It takes the input, feeds it through several layers one after the other, and then finally gives the output.
- A typical training procedure for a neural network is as follows:
 - Define the neural network that has some learnable parameters (or weights)
 - Iterate over a dataset of inputs
 - Process input through the network
 - Compute the loss (how far is the output from being correct)
 - Propagate gradients back into the network's parameters
 - Update the weights of the network, typically using a simple update rule: $\text{weight} = \text{weight} - \text{learning_rate} * \text{gradient}$




```

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 3x3 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 6 * 6, 120) #
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), 2)
        # If the size is a square you can only specify one dimension
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:] # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features

```

- Define the network

```

net = Net()
print(net)

```



```

Net(
  (conv1): Conv2d(1, 6, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=576, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)

```

- Prepare the input (random input as example) and Process input through the network

```

input = torch.randn(1, 1, 32, 32)
out = net(input)

```


Creating a neural network (4/5)

- Compute the loss

A loss function takes the (output, target) pair of inputs, and computes a value that estimates how far away the output is from the target.

```
output = net(input)
target = torch.randn(10)  # a dummy target, for example
target = target.view(1, -1)  # make it the same shape as output
criterion = nn.MSELoss()

loss = criterion(output, target)
```



```
input -> conv2d -> relu -> maxpool2d -> conv2d -> relu -> maxpool2d
      -> view -> linear -> relu -> linear -> relu -> linear
      -> MSELoss
      -> loss
```

Forward

- Backprop

To backpropagate the error all we have to do is to `loss.backward()`

```
net.zero_grad()  # zeroes the gradient buffers of all parameters

print('conv1.bias.grad before backward')
print(net.conv1.bias.grad)

loss.backward()

print('conv1.bias.grad after backward')
print(net.conv1.bias.grad)
```

```
conv1.bias.grad before backward
tensor([0., 0., 0., 0., 0., 0.])
conv1.bias.grad after backward
tensor([ 0.0187, -0.0201,  0.0286, -0.0095,  0.0078, -0.0092])
```

Creating a neural network (5/5)

- Update the weights

The simplest update rule used in practice is the Stochastic Gradient Descent (SGD):

```
weight = weight - learning_rate * gradient
```

PyTorch provides various different update rules such as SGD, Nesterov-SGD, Adam, RMSProp, etc. To enable this, use a small package: `torch.optim` that implements all these methods.

```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()    # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()         # Does the update
```

Visualizing Models/Data/Training with TensorBoard

PyTorch integrates with TensorBoard, a tool designed for visualizing the results of neural network training runs. We can intuitively inspect dataset, model architecture, training process, etc.

TensorBoard IMAGES

☐ Show actual image size

Filter tags (regular expressions supported)

Brightness adjustment

Contrast adjustment

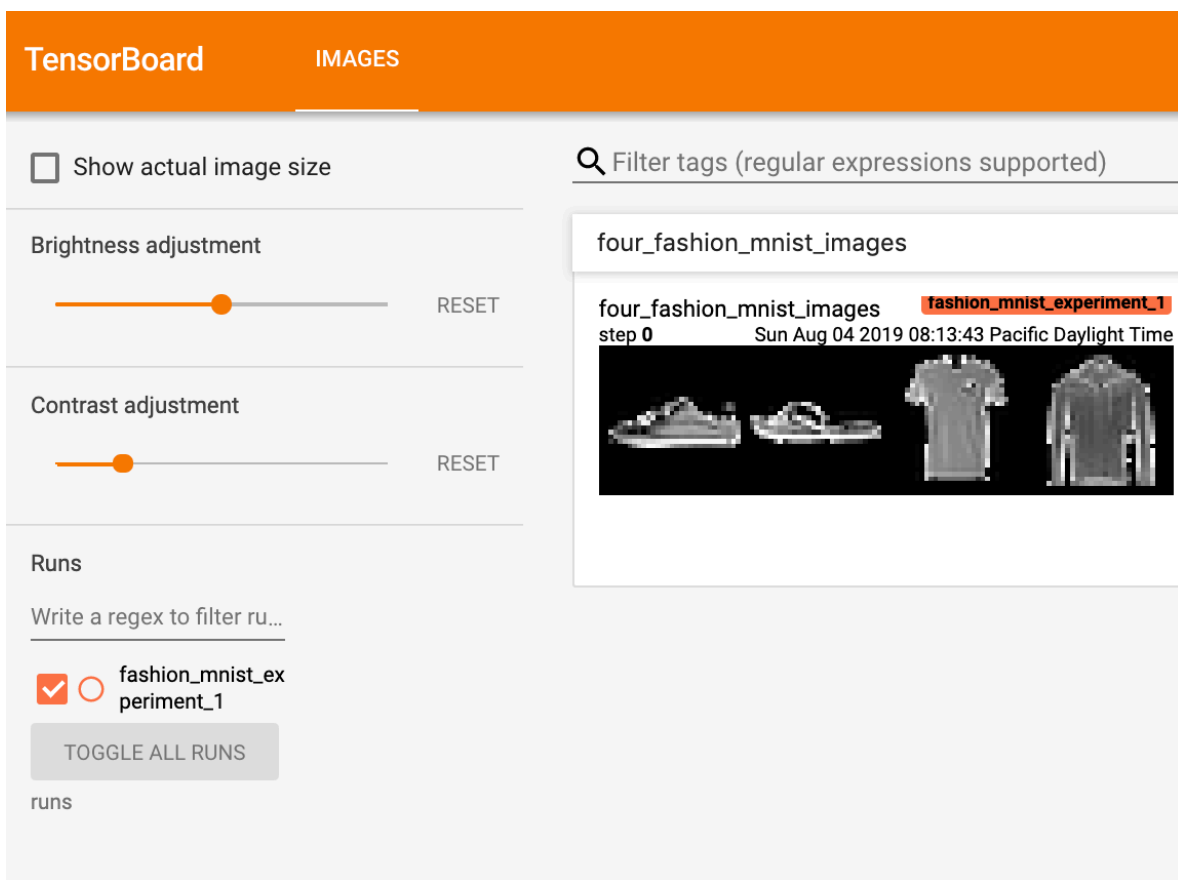
four_fashion_mnist_images

four_fashion_mnist_images **fashion_mnist_experiment_1**
step 0 Sun Aug 04 2019 08:13:43 Pacific Daylight Time

☒ ☐ fashion_mnist_experiment_1

TOGGLE ALL RUNS

runs



TensorBoard SCALARS IMAGES GRAPHS PROJECTOR

☐ Show data download links
☒ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.6

Horizontal Axis
STEP RELATIVE WALL

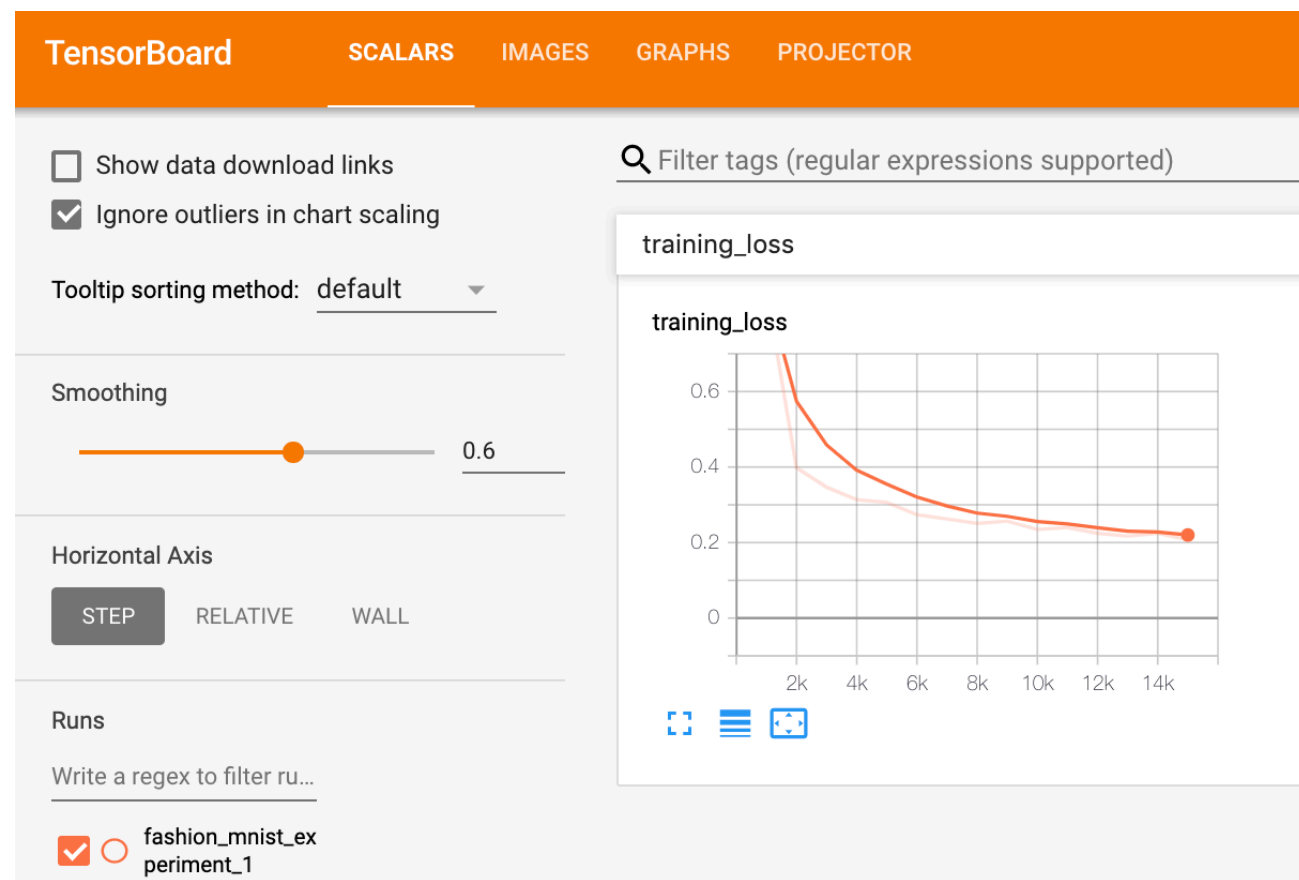
Runs
Write a regex to filter ru...

☒ ☐ fashion_mnist_experiment_1

Filter tags (regular expressions supported)

training_loss

training_loss



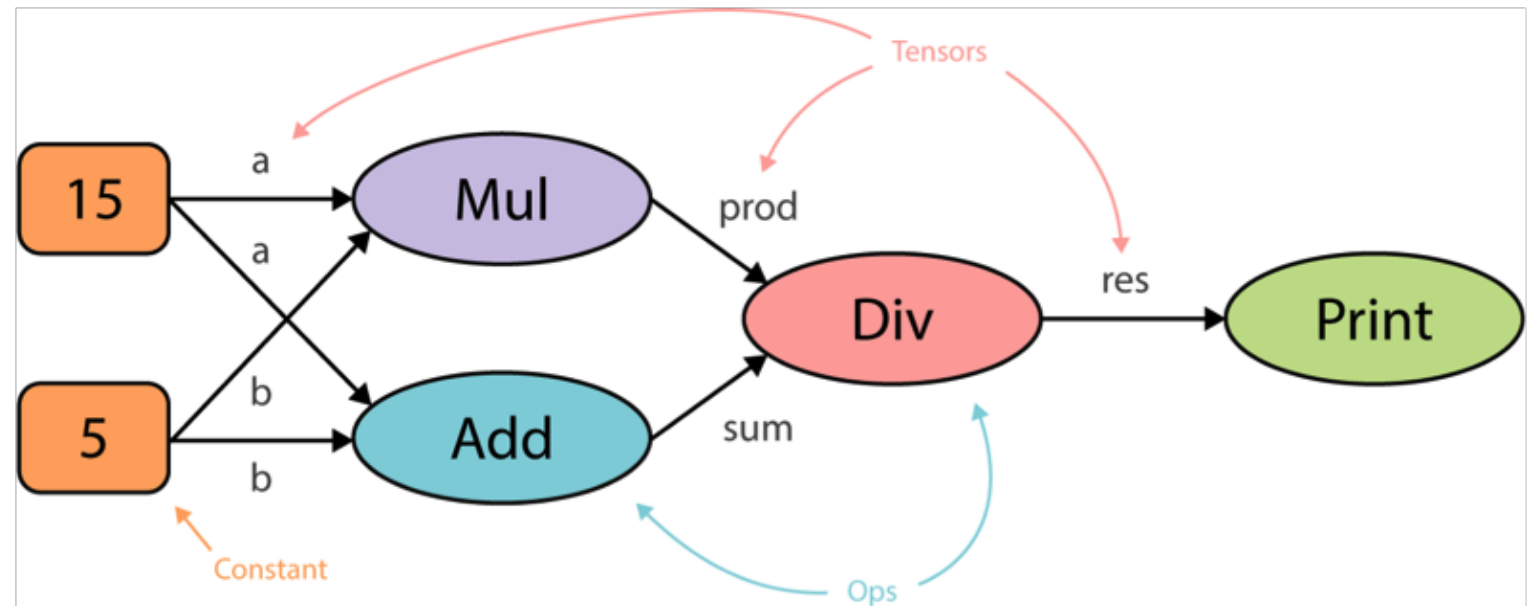
Step	Loss
2k	0.65
4k	0.40
6k	0.30
8k	0.25
10k	0.22
12k	0.20
14k	0.18

Differences: PyTorch vs TensorFlow (1/2)

1. Dynamic VS Static graph definition

- TensorFlow is a framework composed of two core building blocks:
 - A library for defining computational graphs and runtime for executing such graphs on a variety of different hardware.
 - A computational graph

```
a = 15
b = 15
prod = a * b
sum = a + b
result = prod / sum
print(result)
```



A computational graph is generated in a static way before the code is run in TensorFlow.

Differences: PyTorch vs TensorFlow (2/2)

Dynamic VS Static graph definition

- PyTorch also has two core building blocks:
 - Imperative and dynamic building of computational graphs.
 - Autograd: Performs automatic differentiation of the dynamic graphs.
- PyTorch is more tightly integrated with the Python language (more pythonic)

A graph is created on the fly

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```



The graphs change and execute nodes as you go with no special session interfaces or placeholders.



EECS 221:

Languages and Compilers for Hardware Accelerators

(Winter 2022)

Sitao Huang

sitaoh@uci.edu

