15-884: Machine Learning Systems

ML Frameworks and Abstractions

Instructor: Tianqi Chen

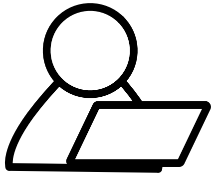
Carnegie Mellon University
School of Computer Science

catalyst

Class Information

- Website: https://catalyst.cs.cmu.edu/15-884-mlsys-sp21
  - Bookmark this, contains links all resources(including ones below)
- Piazza: discussions and announcements
- Use Zoom for lectures, recordings are available via Canvas
- Gradscope: used for all assignments

Machine Learning Systems



Researcher

ResNet ....

Transformer

ML Research

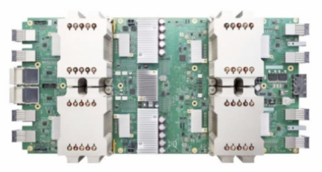
100 lines of python A few hours

System Abstractions

Systems (ML Frameworks) TensorFlow mxnet

IMAGENET **Data**

NVIDIA CUDA



Compute

Machine Learning Systems

We won't focus on a specific one, but will discuss the common and useful elements of these systems

A Typical Deep Learning System Stack

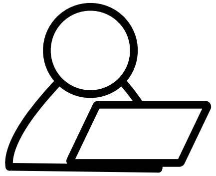
User API
- Programming Abstraction
- Automatic Differentiation

System Components
- Graph IR Optimizations and Transformations
- Runtime and Parallel Scheduling

Architecture
- Optimized Device Code, Libraries
- Accelerators and Hardware Backends

We will dive into each part in follow up lectures

A Typical Deep Learning System Stack



ResNet ....
Transformer

ML Research

Programming Abstraction

Automatic Differentiation

Graph IR Optimizations and Transformations

Runtime and Parallel Scheduling

Optimized Device Code, Libraries

Accelerators and Hardware Backends

Quick Recap: Elements of Machine Learning

Model

$$x_i = \begin{bmatrix} \text{feature}_0 \\ \text{feature}_1 \\ \cdots \\ \text{feaure}_m \end{bmatrix}$$

$$\hat{y}_i = \frac{1}{1 + \exp\left(-w^T x_i\right)}$$

Objective

$$L(w) = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \lambda \|w\|^2$$

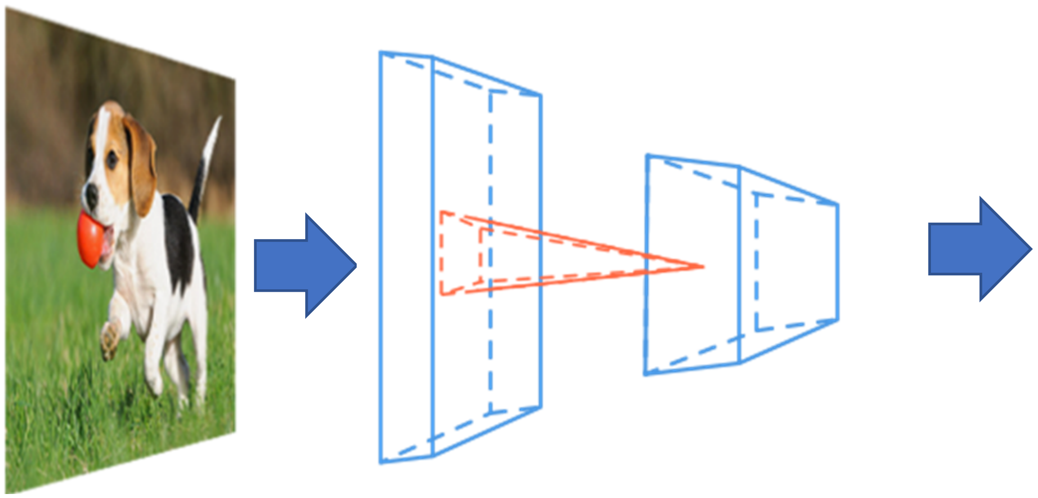
Training
(Optimization)

$$w \leftarrow w - \eta \nabla_w L(w)$$

Quick Recap: Deep Learning

data layer1 layer2 predictor

Compositional Model



$$\hat{y}_i = \frac{1}{1 + \exp\left(-w^T x_i\right)}$$

End to end training

Ingredients of a Deep Learning

- Model and architecture

- Objective function and training techniques
  - Which feedback should be used to guide the learning?
  - Supervised, self-supervised, RL, adversarial learning

- Regularization, normalization and initialization (coupled with modeling)
  - Batch norm, dropout, Xavier

- Get good amount of data

Application affects System Design

	Data Management	Data Processing
Application	Data Management	Data Processing
System Design	Declarative language(SQL) Execution planner Storage engine	Distributed Primitive(MapReduce) Fault tolerance layer Workload migration

Ingredients of a Deep Learning

- Model and architecture

- Objective function and training techniques
  - Which feedback should be used to guide the learning?
  - Supervised, self-supervised, RL, adversarial learning

- Regularization, normalization and initialization (coupled with modeling)
  - Batch norm, dropout, Xavier

- Get good amount of data

Discussion how can these ingredients affect the system design of ML frameworks

A Typical Deep Learning System Stack

User API

Programming Abstraction

Automatic Differentiation

Graph IR Optimizations and Transformations

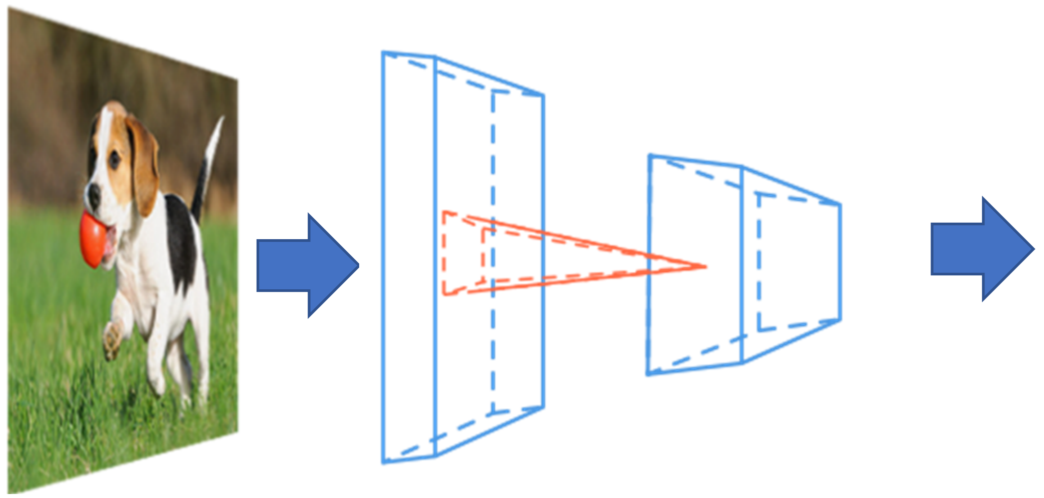
Runtime and Parallel Scheduling

Optimized Device Code, Libraries

Accelerators and Hardware Backends

Quick Recap: Deep Learning

Compositional Model



$$\hat{y}_i = \frac{1}{1 + \exp\left(-w^T x_i\right)}$$

End to end training

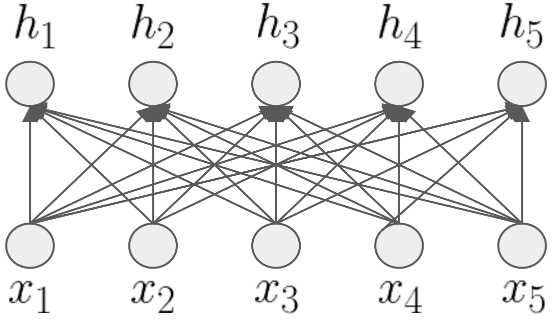
Example: Logistic Regression

Input | **Linear Layer** | **Softmax**

$$x_i = \begin{bmatrix} \text{pixel}_1 \\ \text{pixel}_2 \\ \cdots \\ \text{pixel}_m \end{bmatrix}$$

$$h_k = w_k^T x_i$$

$$P(y_i = k | x_i) = \frac{\exp(h_k)}{\sum_{j=1}^{10} \exp(h_i)}$$



Logistic Regression in Numpy

```
import numpy as np
from tinyflow.datasets import get_mnist
def softmax(x):
    x = x - np.max(x, axis=1, keepdims=True)
    x = np.exp(x)
    x = x / np.sum(x, axis=1, keepdims=True)
    return x
# get the mnist dataset
mnist = get_mnist(flatten=True, onehot=True)
learning_rate = 0.5 / 100
W = np.zeros((784, 10))
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    # forward
    y = softmax(np.dot(batch_xs, W))
    # backward
    y_grad = y - batch_ys
    W_grad = np.dot(batch_xs.T, y_grad)
    # update
    W = W - learning_rate * W_grad
```

Forward computation:
Compute probability of each class y given input

- Matrix multiplication
  - `np.dot(batch_xs, W)`
- Softmax transform the result
  - `softmax(np.dot(batch_xs, W))`

Logistic Regression in Numpy

```
import numpy as np
from tinyflow.datasets import get_mnist
def softmax(x):
   x = x - np.max(x, axis=1, keepdims=True)
   x = np.exp(x)
   x = x / np.sum(x, axis=1, keepdims=True)
   return x
# get the mnist dataset
mnist = get_mnist(flatten=True, onehot=True)
learning_rate = 0.5 / 100
W = np.zeros((784, 10))
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
   # forward
   y = softmax(np.dot(batch_xs, W))
   # backward
   y_grad = y - batch_ys
   W_grad = np.dot(batch_xs.T, y_grad)
   # update
   W = W - learning_rate * W_grad
```

Manually calculate the gradient of weight with respect to the log-likelihood loss.

Exercise: Try to derive the gradient rule by yourself.

Logistic Regression in Numpy

```
import numpy as np
from tinyflow.datasets import get_mnist
def softmax(x):
   x = x - np.max(x, axis=1, keepdims=True)
   x = np.exp(x)
   x = x / np.sum(x, axis=1, keepdims=True)
   return x
# get the mnist dataset
mnist = get_mnist(flatten=True, onehot=True)
learning_rate = 0.5 / 100
W = np.zeros((784, 10))
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
   # forward
   y = softmax(np.dot(batch_xs, W))
   # backward
   y_grad = y - batch_ys
   W_grad = np.dot(batch_xs.T, y_grad)
   # update
   W = W - learning_rate * W_grad
```

Weight Update via SGD

$$w \leftarrow w - \eta \nabla_w L(w)$$

Discussion: Numpy based Program

```
import numpy as np
from tinyflow.datasets import get_mnist
def softmax(x):
   x = x - np.max(x, axis=1, keepdims=True)
   x = np.exp(x)
   x = x / np.sum(x, axis=1, keepdims=True)
   return x
# get the mnist dataset
mnist = get_mnist(flatten=True, onehot=True)
learning_rate = 0.5 / 100
W = np.zeros((784, 10))
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
   # forward
   y = softmax(np.dot(batch_xs, W))
   # backward
   y_grad = y - batch_ys
   W_grad = np.dot(batch_xs.T, y_grad)
   # update
   W = W - learning_rate * W_grad
```

- What do we need to do to support deeper neural networks
- What are the complications

Logistic Regression in Numpy

- Computation in Tensor Algebra
  - `softmax(np.dot(batch_xs, W))`
- Manually calculate the gradient
  - `y_grad = y - batch_ys`
  - `W_grad = np.dot(batch_xs.T, y_grad)`
- SGD Update Rule
  - `W = W - learning_rate * W_grad`

Logistic Regression in TinyFlow (TF-1.x like API)

```
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
   sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

Forward Computation
Declaration

Logistic Regression in TinyFlow

```
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

Loss function **Declaration**

$$P(\text{label} = k) = y_k$$

$$L(y) = \sum_{k=1}^{10} I(\text{label} = k) \log(y_i)$$

Logistic Regression in TinyFlow

```
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
   sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

Automatic Differentiation:
Next incoming topic

Logistic Regression in TinyFlow

```
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
   sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

SGD update rule

Logistic Regression in TinyFlow

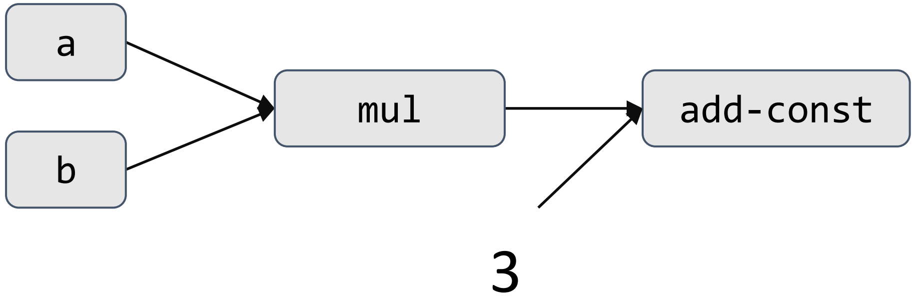
```
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
   sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

Real execution happens here!

The Declarative Language: Computation Graph

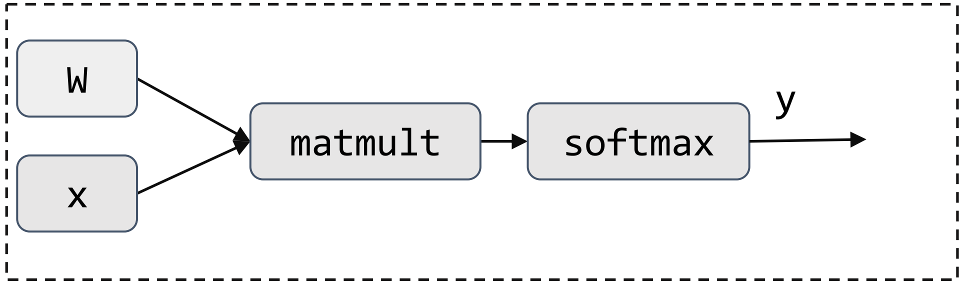
- Nodes represents the computation (operation)
- Edge represents the data dependency between operations

Computational Graph for `a * b +3`



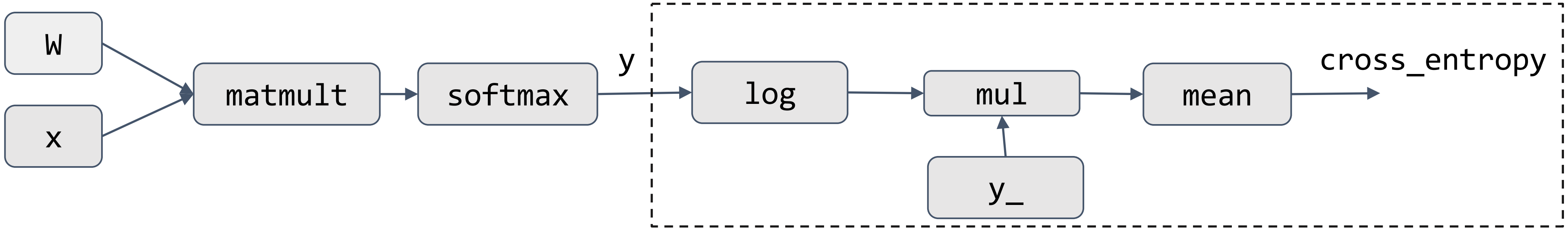
Computational Graph Construction by Step

```
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
```



Computational Graph Construction by Step

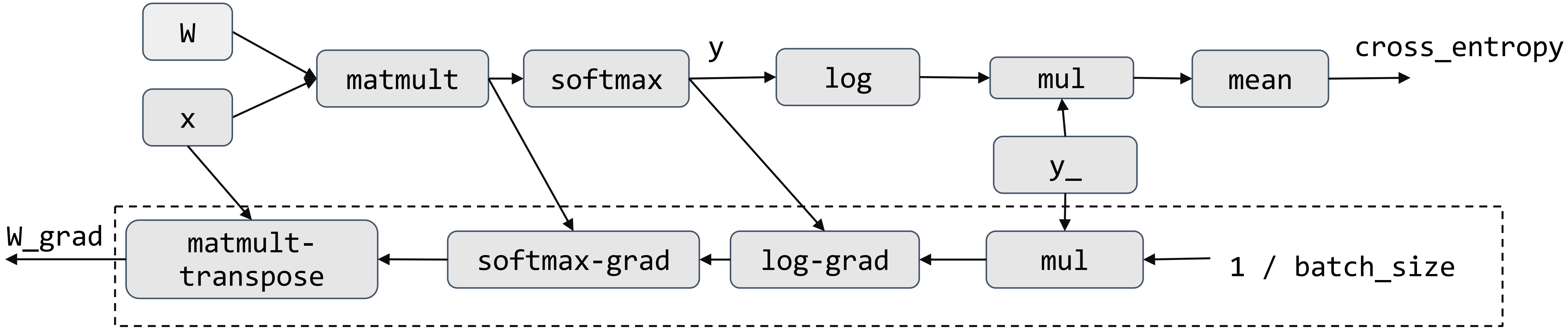
```
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
```



Computational Graph Construction by Step

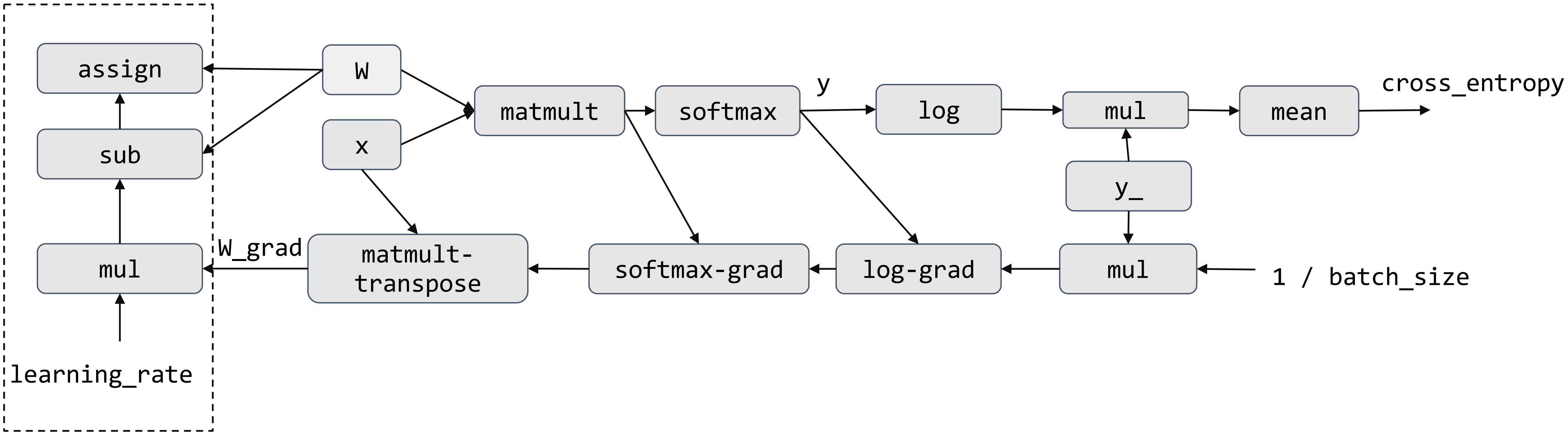
```
W_grad = tf.gradients(cross_entropy, [W])[0]
```

Automatic Differentiation, more details in follow up lectures



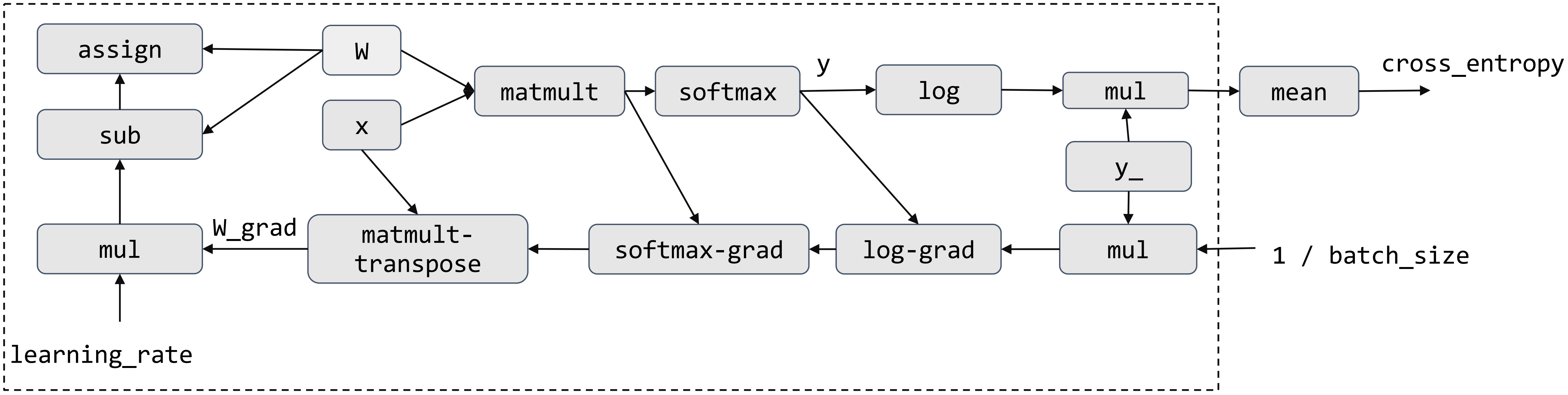
Computational Graph Construction by Step

```
train_step = tf.assign(W, W - learning_rate * W_grad)
```



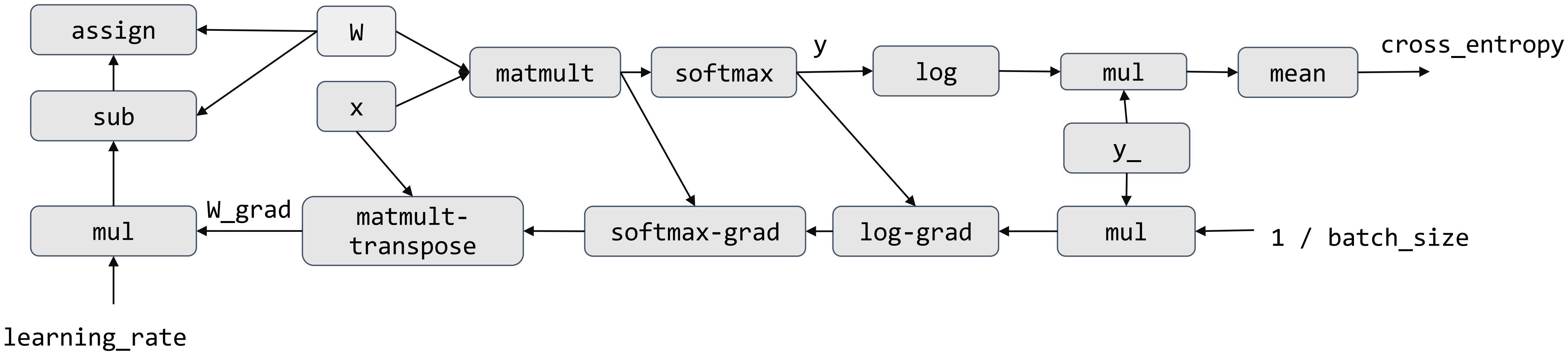
Execution only Touches the Needed Subgraph

```
sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```



Discussion: Computational Graph

- What is the benefit of computational graph?
- How can we deploy the model to mobile devices?



Imperative AutoGrad

```
import autograd.numpy as np
from autograd import grad

def softmax(x):
    x = x - np.max(x, axis=1, keepdims=True)
    x = np.exp(x)
    x = x / np.sum(x, axis=1, keepdims=True)
    return x

def loss(W, batch_xs, batch_ys):
    y = softmax(np.dot(batch_xs, W))
    return cross_entropy_loss(y, batch_ys)

# get the mnist dataset
mnist = get_mnist(flatten=True, onehot=True)
learning_rate = 0.5 / 100
W = np.zeros((784, 10))
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    W_grad = grad(loss, argnum=0)(W, batch_xs, batch_ys)
    # update
    W = W - learning_rate * W_grad
```

Compute gradient via tracing through python executions

Discussion: Imperative vs Declarative Program

Benefit/drawback of the TF v1 model(declarative) vs Numpy(imperative) Model

```python
import autograd.numpy as np
from autograd import grad

def softmax(x):
    x = x - np.max(x, axis=1, keepdims=True)
    x = np.exp(x)
    x = x / np.sum(x, axis=1, keepdims=True)
    return x
def loss(W, batch_xs, batch_ys):
    y = softmax(np.dot(batch_xs, W))
    return cross_entropy_loss(y, batch_ys)

mnist = get_mnist(flatten=True, onehot=True)
learning_rate = 0.5 / 100
W = np.zeros((784, 10))
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    W_grad = grad(loss, argnum=0)(W, batch_xs, batch_ys)
    # update
    W = W - learning_rate * W_grad
```

```python
import tinyflow as tf
from tinyflow.datasets import get_mnist
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)
# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
   batch_xs, batch_ys = mnist.train.next_batch(100)
   sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```

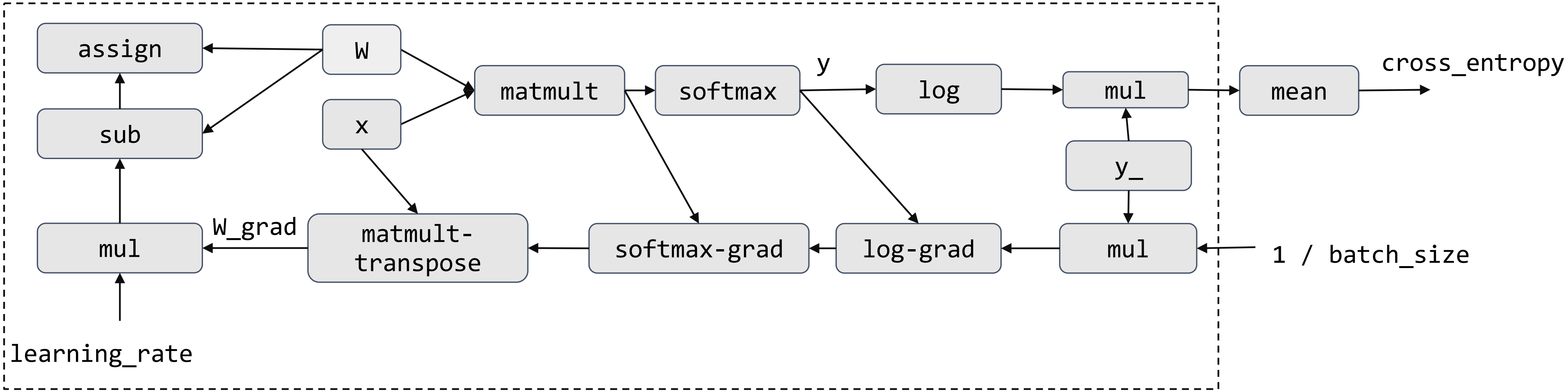
A Typical Deep Learning System Stack

System Components

- Programming Abstraction
- Automatic Differentiation
- Graph IR Optimizations and Transformations
- Runtime and Parallel Scheduling
- Optimized Device Code, Libraries
- Accelerators and Hardware Backends

Computation Graph Optimization

- E.g. Deadcode elimination
- Memory planning and optimization
- What other possible optimization can we do given a computational graph?

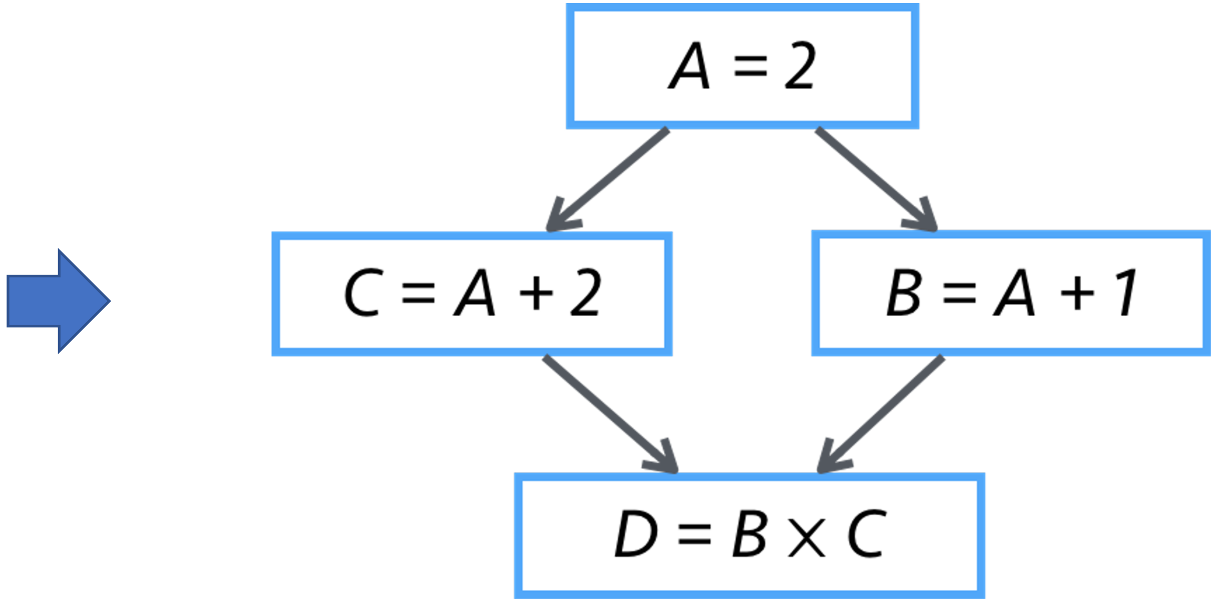


Parallel Scheduling

- Code need to run parallel on multiple devices and worker threads
- Detect and schedule parallelizable patterns
- Detail lecture on later

MXNet Example

```
>>> import mxnet as mx
>>> A = mx.nd.ones((2,2)) *2
>>> C = A + 2
>>> B = A + 1
>>> D = B * C
```



A Typical Deep Learning System Stack

Programming Abstraction

Automatic Differentiation

Graph IR Optimizations and Transformations

Runtime and Parallel Scheduling

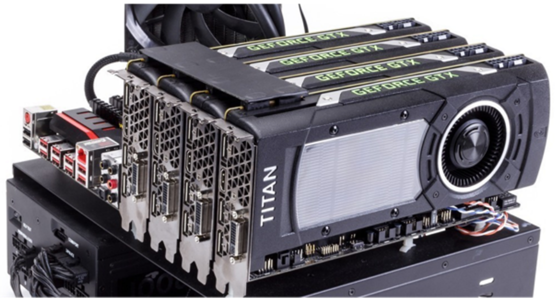
Architecture

Optimized Device Code, Libraries

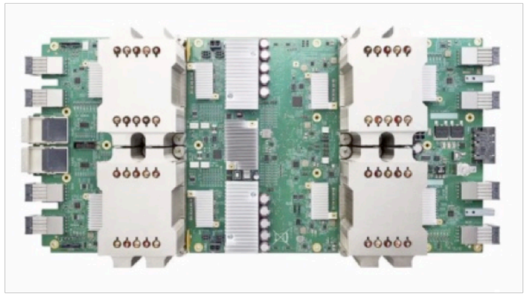
Accelerators and Hardware Backends

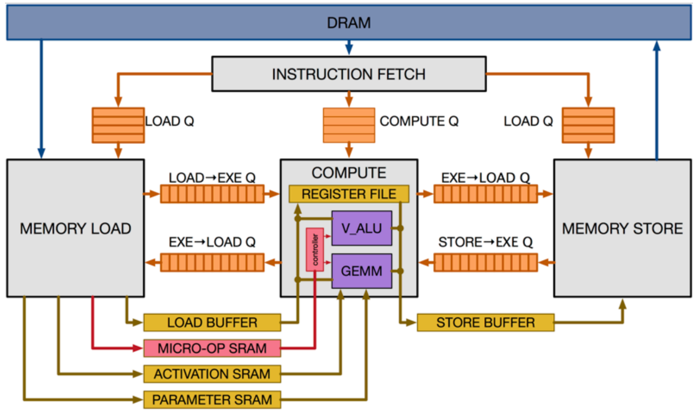
GPU Acceleration

- Most existing deep learning programs runs on GPUs
- Modern GPU have Teraflops of computing power

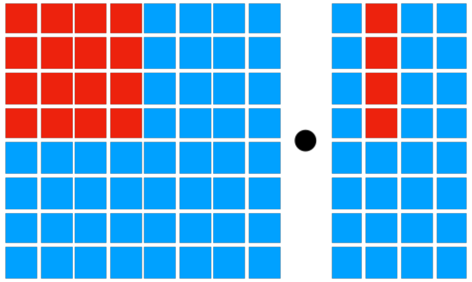


Specialized Accelerators



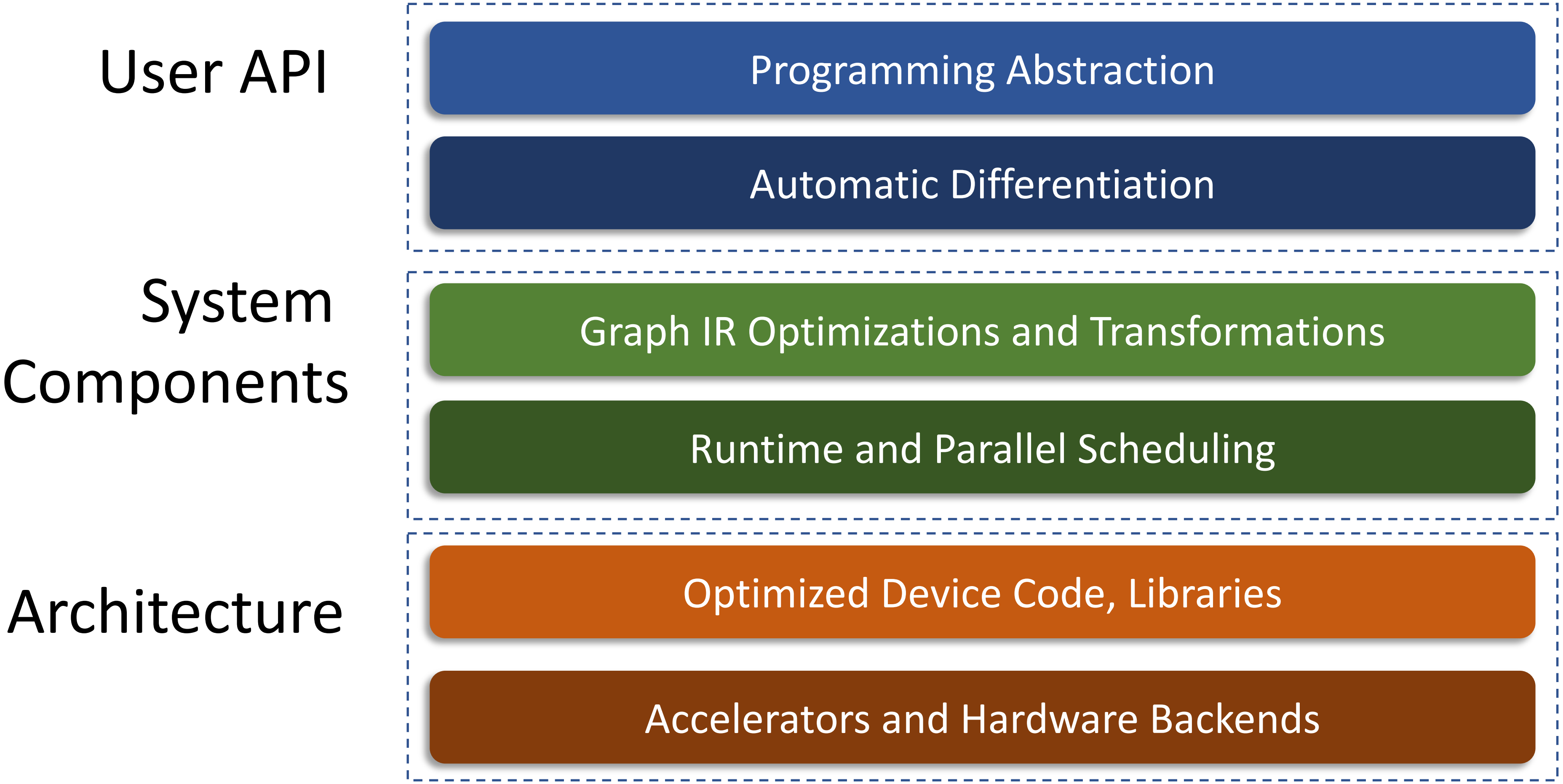


Tensor
Compute Primitives



Explicitly Managed
Memory Subsystem

A Typical Deep Learning System Stack



Not a comprehensive list of elements,
the systems are still rapidly evolving :)

Differentiable Programming

Differentiable Programming language

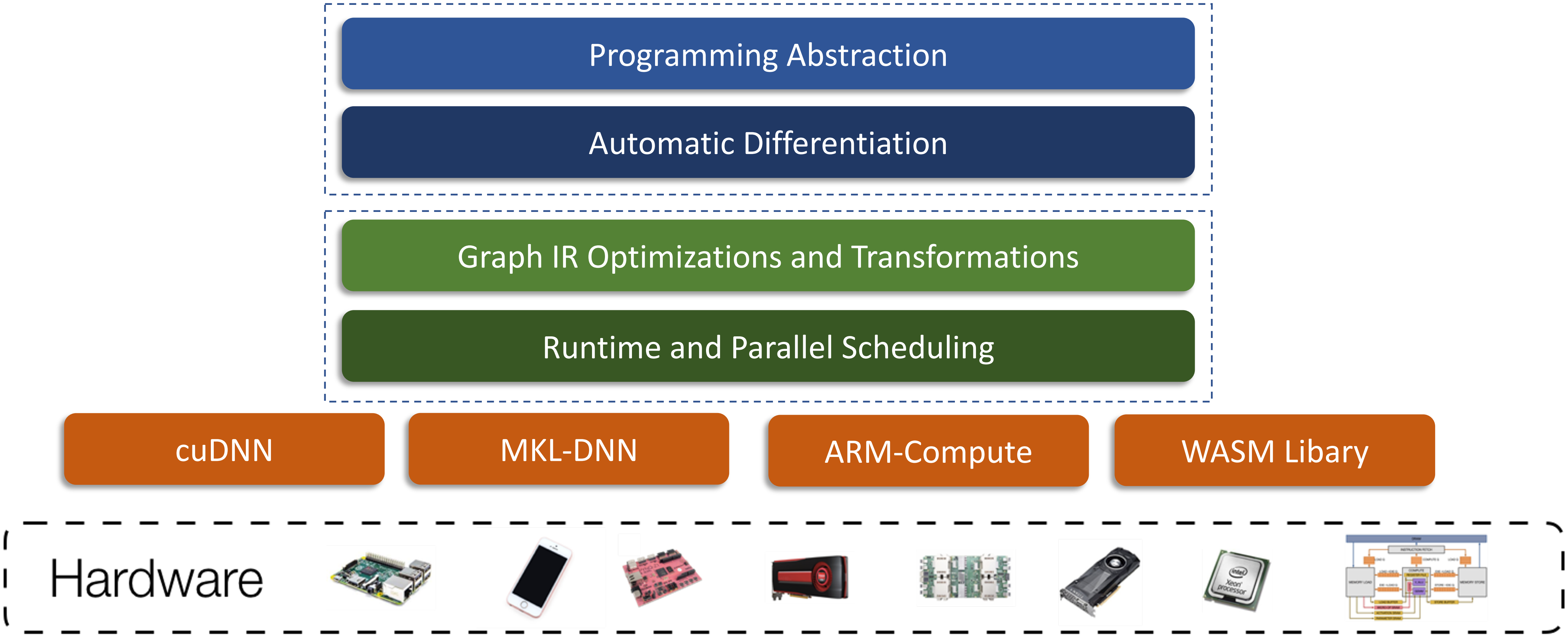
Compiler IR Optimizations and Transformations

Runtime and Parallel Scheduling

Optimized Device Code, Libraries

Accelerators and Hardware Backends

Each Hardware backend requires a software stack



Compiler Based Approach

Programming Abstraction

Automatic Differentiation

High-level IR Optimizations and Transformations

Tensor Operator level Optimization

Direct code generation

Hardware

Other ML Frameworks

This lecture focused on deep learning frameworks

- Common components
  - Distributed learning primitives (allreduce, parameter server)
  - Data loading and processing
  - Hyper parameter tuning
- Model specific optimizations
  - Approximate summary (for trees)

Logistics

- First discussion session next Tuesday about ML Frameworks!
- Submit paper reviews before Tuesday’s lecture
- Presentation assignment will be out today.
- Start to think about project ideas and find teammates.

Questions

Programming Abstraction

Automatic Differentiation

Graph IR Optimizations and Transformations

Runtime and Parallel Scheduling

Optimized Device Code, Libraries

Accelerators and Hardware Backends