10-414/714 – Deep Learning Systems: Algorithms and Implementation

Model Deployment

Fall 2022

J. Zico Kolter and Tianqi Chen (this time)
Carnegie Mellon University

Outline

Model deployment overview

Machine learning compilation

Outline

Model deployment overview

Machine learning compilation

What we have learned so far in this class

How to build a deep learning system that trains deep learning models efficiently on a standard computing environment (with GPUs).

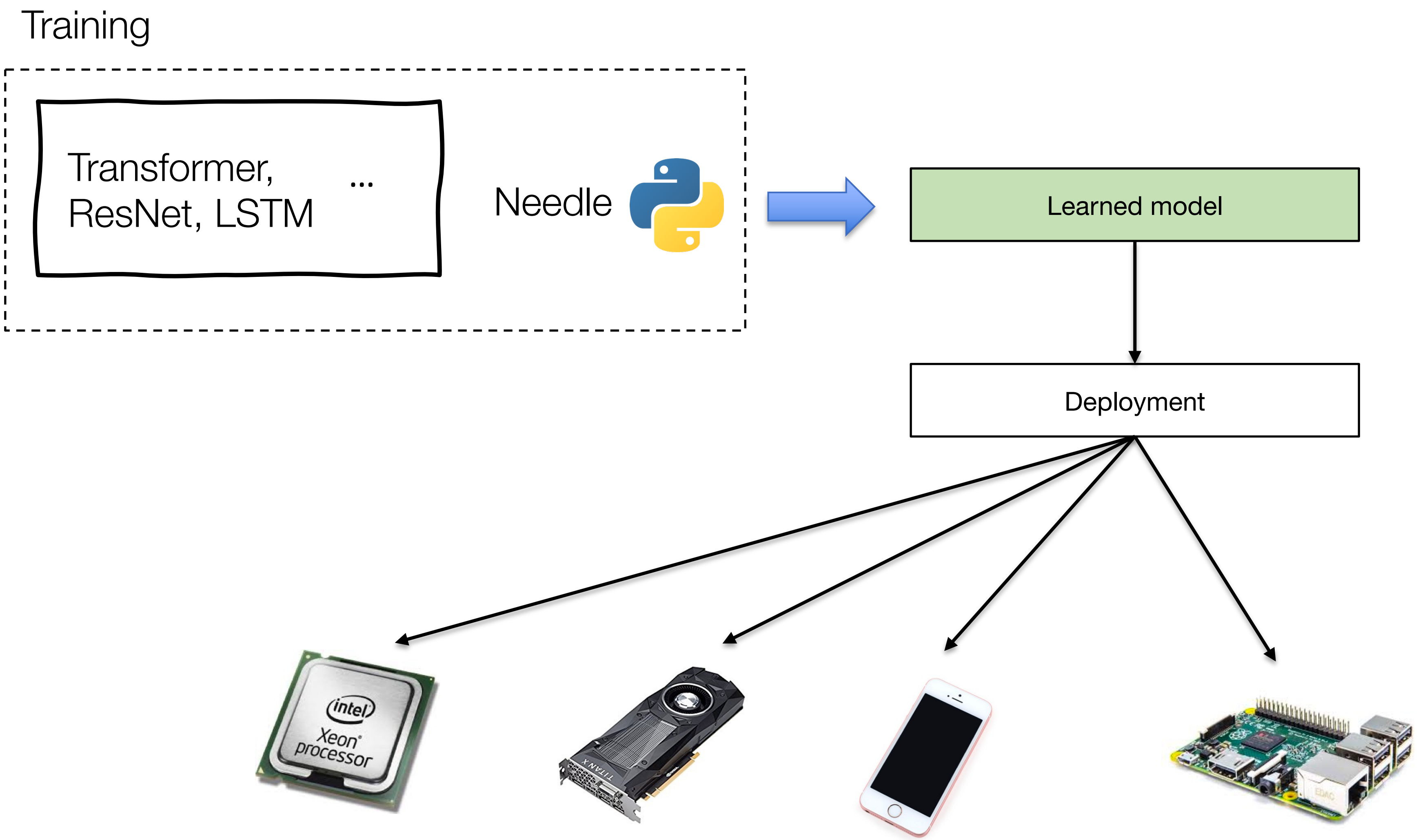
Automatic differentiation

Deep learning modeling techniques

Hardware accelerations and scale up

Normalization, initialization, optimization

Model deployment



Bring learned models to different application environments

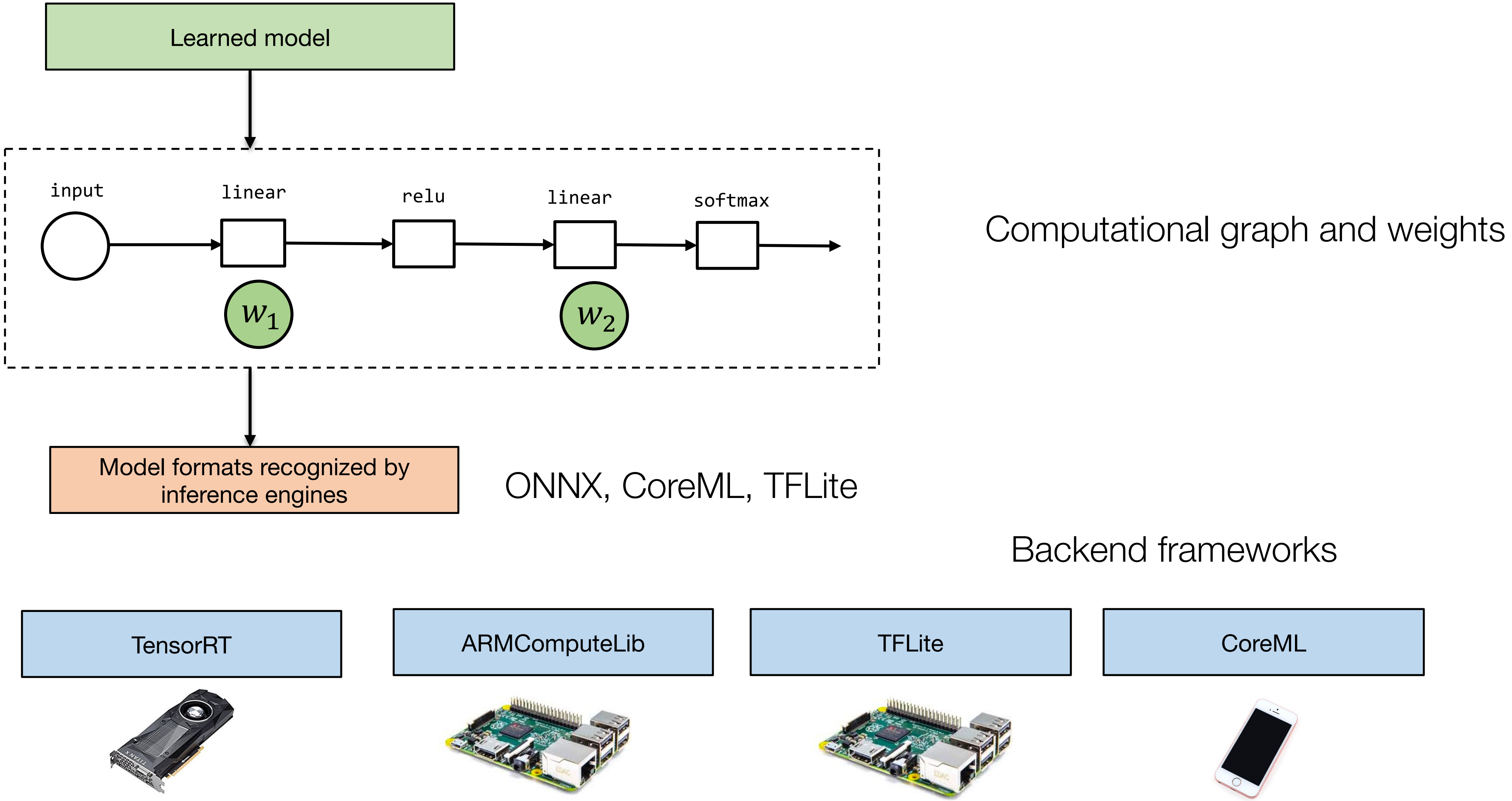
Model deployment considerations

Application environment may bring restrictions (model size, no-python)

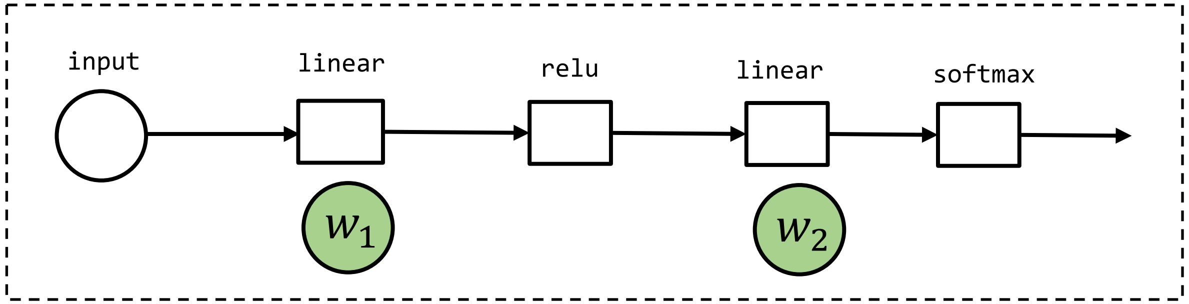
Leverage local hardware acceleration (mobile GPUs, accelerated CPU instructions, NPUs)

Integration with the applications (data preprocessing, post processing)

Model exportation and deploy to inference engines



Inference engine internals



Computational graph

Many inference engines are structured as computational graph interpreters

Allocate memories for intermediate activations
Traverse the graph and execute each of the operators

Usually only support a limited set of operators and programming models (e.g. dynamism)

Outline

Model deployment overview

Machine learning compilation

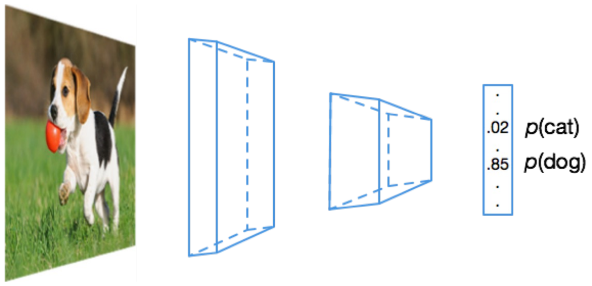
Limitation of library driven inference engine deployments

Need to build specialized libraries for each hardware backend

A lot of engineering efforts to optimization

Machine learning compilation

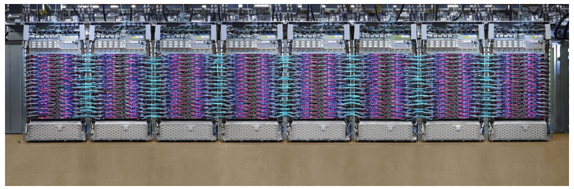
ML Models



High-level IR Optimizations and Transformations

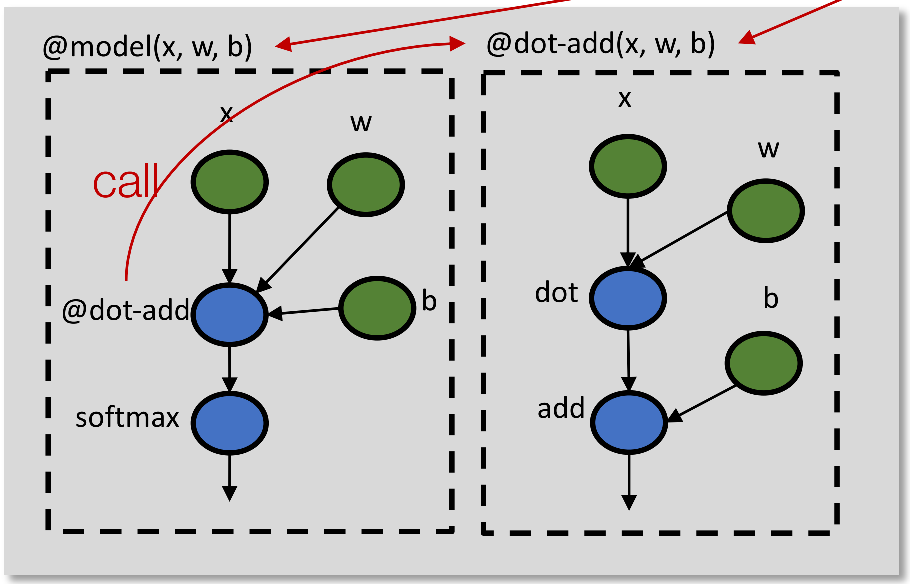
Tensor Operator Level Optimization

Direct code generation



Compiler representation of a model

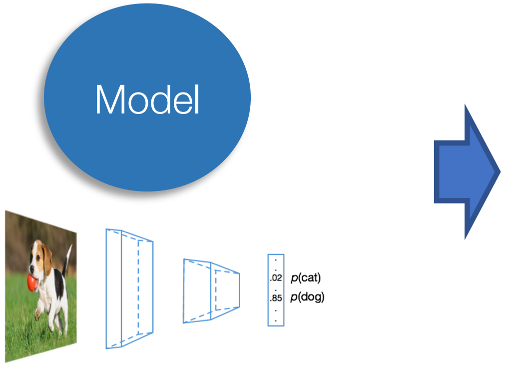
functions



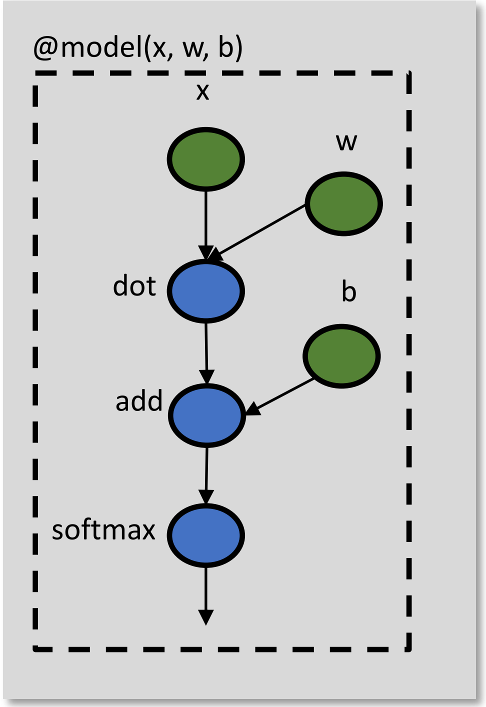
Also called intermediate representation (IR)

IRModule: a collection if interdependent functions

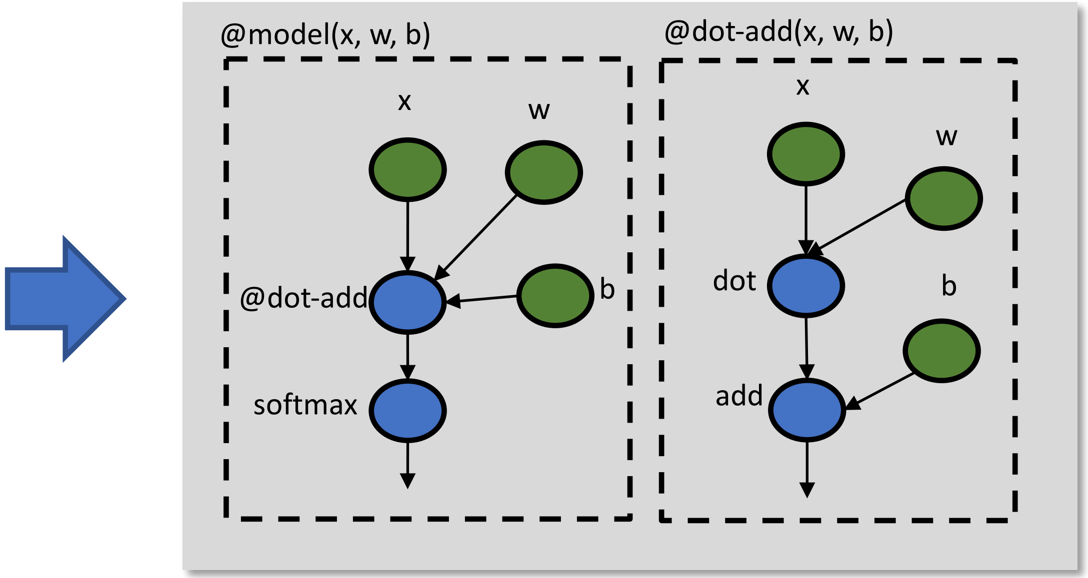
Example compilation flow: high-level transformations



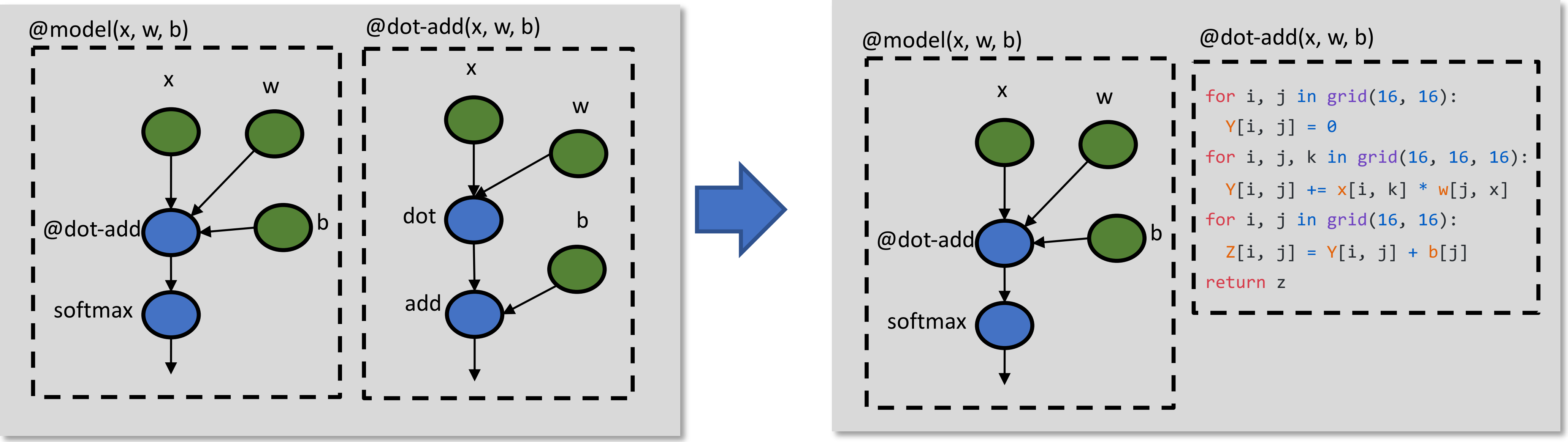
import



High-level
transformations

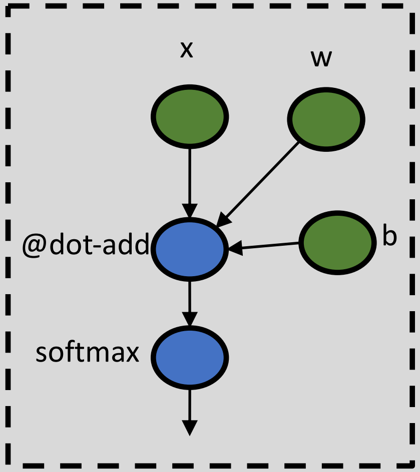


Example compilation flow: lowering to loop IR



Example compilation flow: low-level transformations

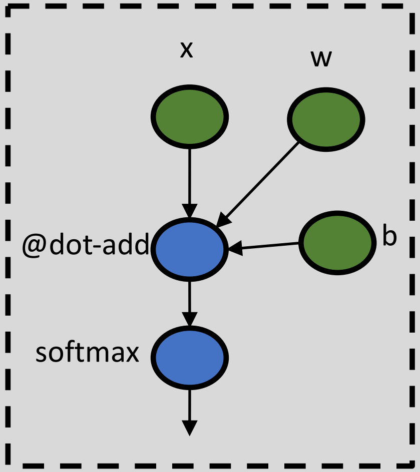
@model(x, w, b)



@dot-add(x, w, b)

```
for i, j in grid(16, 16):
  Y[i, j] = 0
for i, j, k in grid(16, 16, 16):
  Y[i, j] += x[i, k] * w[j, x]
for i, j in grid(16, 16):
  Z[i, j] = Y[i, j] + b[j]
return Z
```

@model(x, w, b)

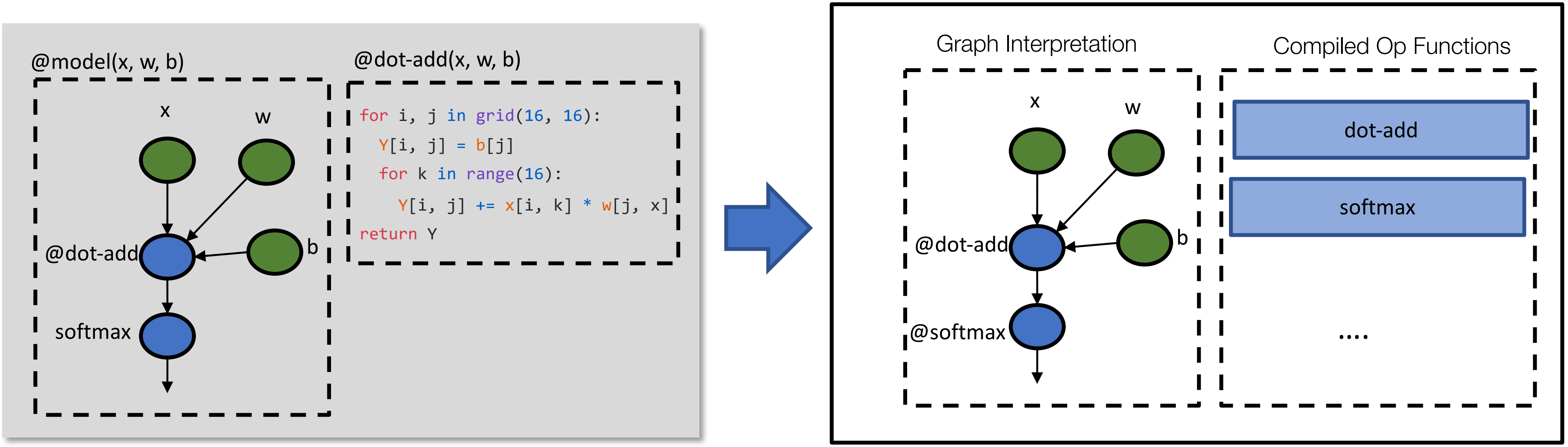


@dot-add(x, w, b)

```
for i, j in grid(16, 16):
  Y[i, j] = b[j]
  for k in range(16):
    Y[i, j] += x[i, k] * w[j, x]
return Y
```

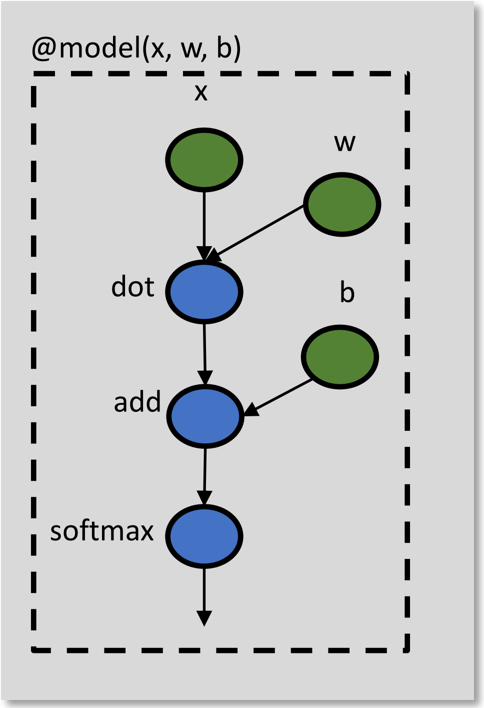
Low-level transformations

Example compilation flow: code generation and execution



Runtime Execution

High-level IR and optimizations



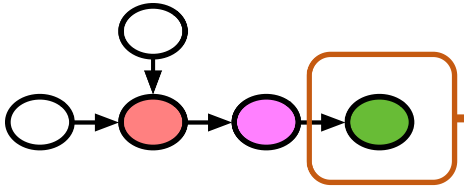
Computation graph(or graph-like) representation

Each node is a tensor operator(e.g. convolution)

Can be transformed (e.g. fusion) and annotated (e.g. device placement)

Most ML frameworks have this layer

Low-level code optimizations



Specification

```
C = tvm.compute((m, n),
  lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Search Space of Possible Program Optimizations

Low-level Program Variants

```
inp_buffer AL[8][8], BL[8][8]
acc_buffer CL[8][8]
for yo in range(128):
  for xo in range(128):
    vdla.fill_zero(CL)
    for ko in range(128):
      vdla.dma_copy2d(AL, A[ko*8:ko*8+8][yo*8:yo*8+8])
      vdla.dma_copy2d(BL, B[ko*8:ko*8+8][xo*8:xo*8+8])
      vdla.fused_gemm8x8_add(CL, AL, BL)
    vdla.dma_copy2d(C[yo*8:yo*8+8,xo*8:xo*8+8], CL)
```

```
for yo in range(128):
  for xo in range(128):
    C[yo*8:yo*8+8][xo*8:xo*8+8] = 0
    for ko in range(128):
      for yi in range(8):
        for xi in range(8):
          for ki in range(8):
            C[yo*8+yi][xo*8+xi] +=
               A[ko*8+ki][yo*8+yi] * B[ko*8+ki][xo*8+xi]
```

```
for y in range(1024):
  for x in range(1024):
    C[y][x] = 0
    for k in range(1024):
      C[y][x] += A[k][y] * B[k][x]
```

Elements of low-level loop representations

@dot-add(x, w, b)

```
for i, j in grid(16, 16):
  Y[i, j] = 0
for i, j, k in grid(16, 16, 16):
  Y[i, j] += x[i, k] * w[j, x]
for i, j in grid(16, 16):
  Z[i, j] = Y[i, j] + b[j]
```

Multi-dimensional buffer

Loop nests

Array computation

Transforming loops: splitting

Code

```
for x in range(128):
  C[x] = A[x] + B[x]
```

```
for xo in range(32):
  for xi in range(4):
     C[xo * 4 + xi]
      = A[xo * 4 + xi] + B[xo * 4 + xi]
```

Transformation

```
x = get_loop("x")
xo, xi = split(x, 4)
```

Transforming loops: reorder

Code

```
for xo in range(32):
  for xi in range(4):
     C[xo * 4 + xi]
      = A[xo * 4 + xi] + B[xo * 4 + xi]
```

```
for xi in range(4):
  for xo in range(32):
    C[xo * 4 + xi]
      = A[xo * 4 + xi] + B[xo * 4 + xi]
```

Transformation

```
x = get_loop("x")
xo, xi = split(x, 4)
reorder(xi, xo)
```

Transforming loops: thread binding

Code

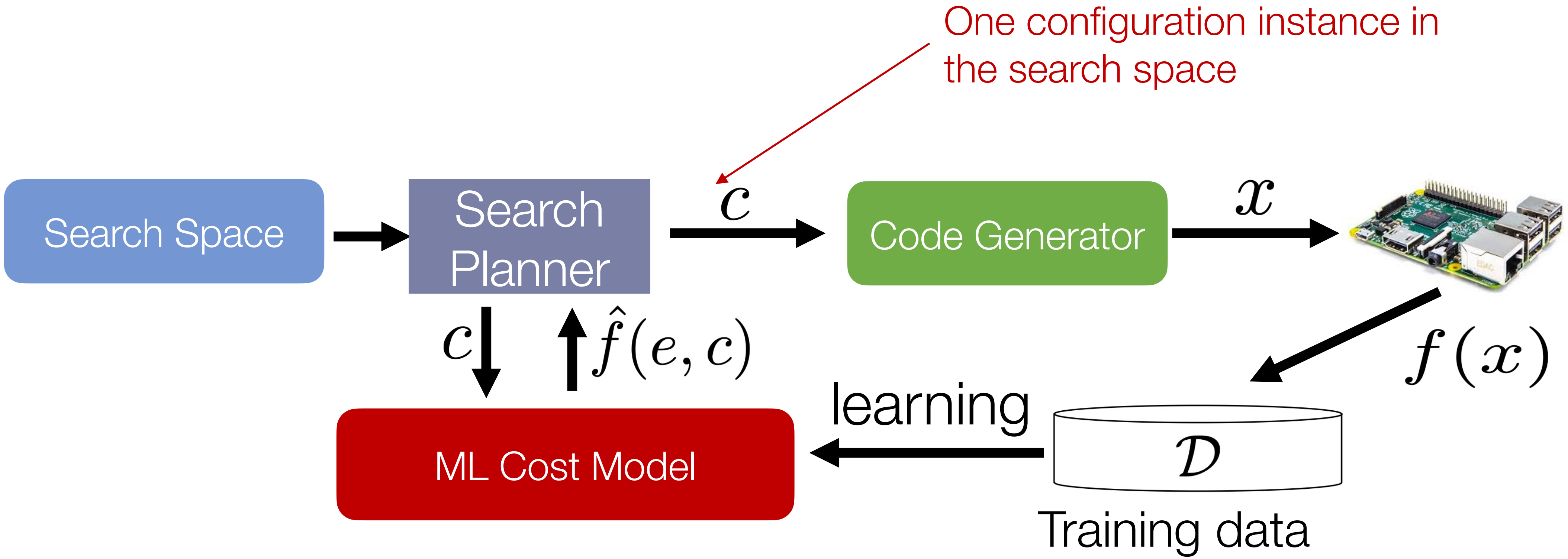
```
for xi in range(4):
  for xo in range(32):
    C[xo * 4 + xi]
      = A[xo * 4 + xi] + B[xo * 4 + xi]
```

```
def gpu_kernel():
  C[threadId.x * 4 + blockIdx.x] = . . .
```

Transformation

```
x = get_loop("x")
xo, xi = split(x, 4)
reorder(xi, xo)
bind_thread(xo, "threadIdx.x")
bind_thread(xi, "blockIdx.x")
```

Search via learned cost model



Summary: elements of an automated ML compiler

Program representation

- Represent the program/optimization of interest, (e.g. dense tensor linear algebra, data structures)

Build search space through a set of transformations

- Cover common optimizations
- Find ways for domain experts to provide input

Effective search

- Cost models, transferability

Still an open research area!

Outline

Deploying models to different backends

Machine learning compilation