Compiler Optimizations for Machine Learning Workloads

Bojian Zheng

CSCD70 Compiler Optimization

2023/3/20

Announcements

- The lecture & tutorial next week (i.e., 2023/03/27) will be **cancelled**.

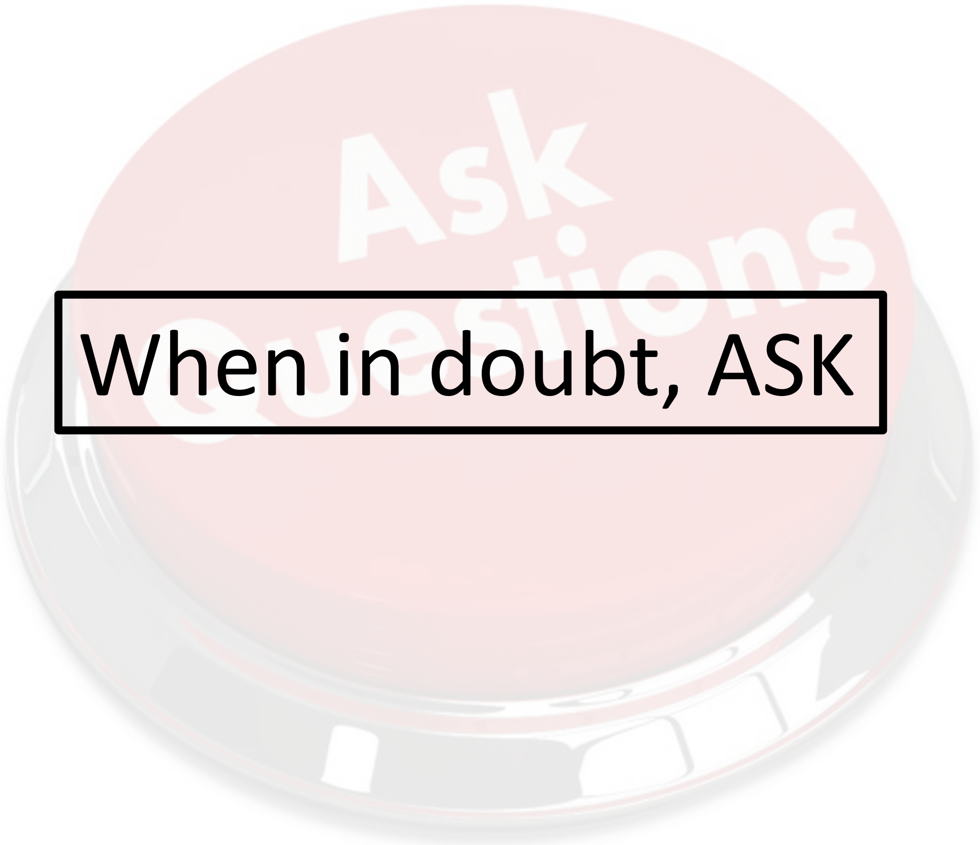
- Assignment 3 will be released this Friday (i.e., 2023/03/24).
  - 2 weeks will be given.
  - Covers loop invariant code motion and register allocation.

Agenda

0. Background: Deep Neural Networks

1. Machine Learning Systems

2. Memory Optimizations



When in doubt, ASK

Hypes in Machine Learning

Image Synthesis

https://stablediffusionweb.com/

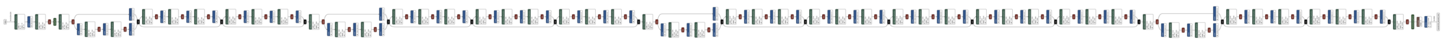
Chat Bot

https://openai.com/blog/chatgpt

Deep Neural Networks

- An important class of machine learning algorithms, usually interpreted as **graphs**.

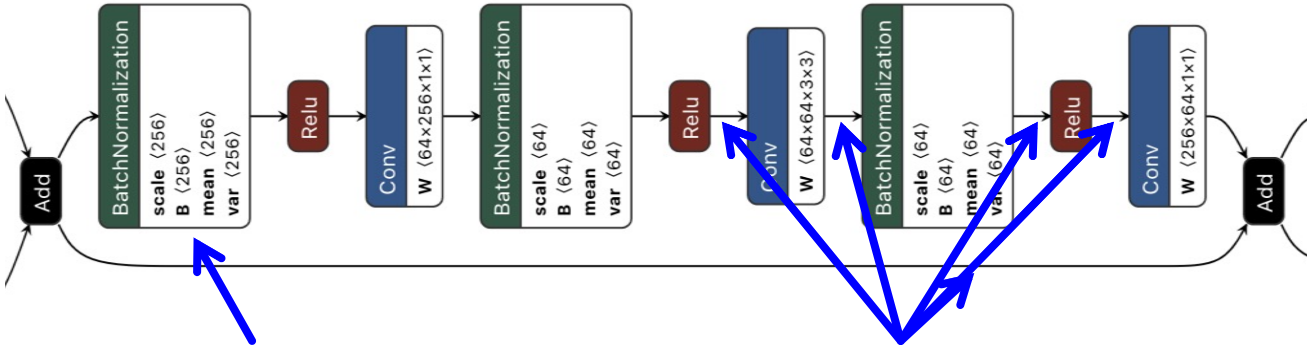
Graph visualization of ResNet-50, an image classification model



Deep Neural Networks

- An important class of machine learning algorithms, usually interpreted as **<u>graphs</u>**.

Graph visualization of ResNet-50, an image classification model

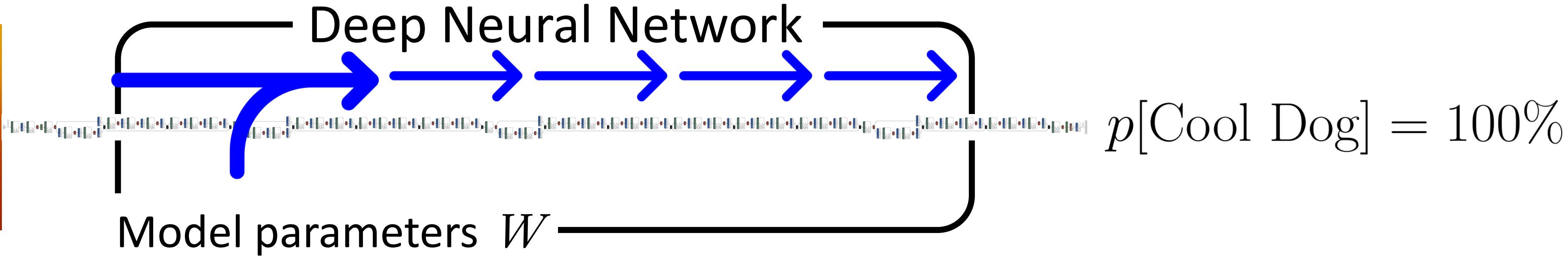


Graph = Nodes (i.e., **Operators**) + Edges (i.e., **Tensors**)

Tensor = NDArray = Multi-dimensional array

Deep Neural Networks

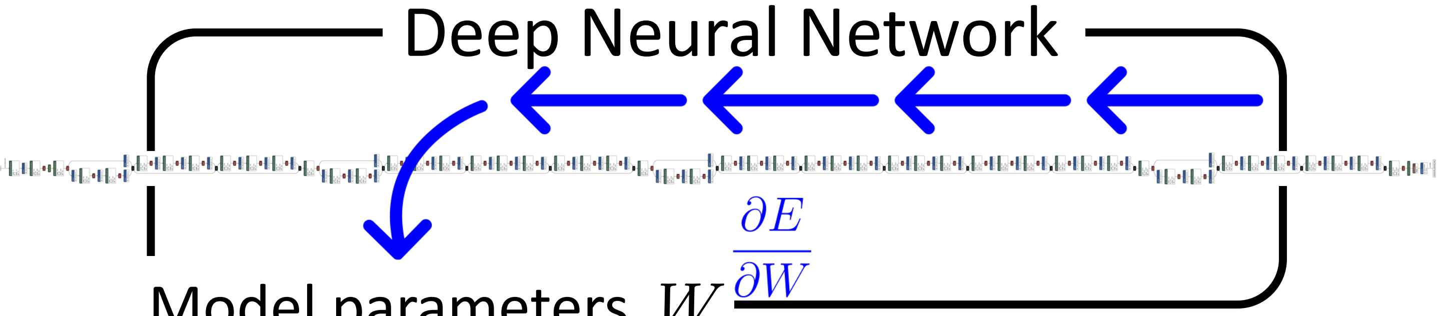
- 3 phases:



❶ Forward Pass

Deep Neural Networks

- 3 phases:



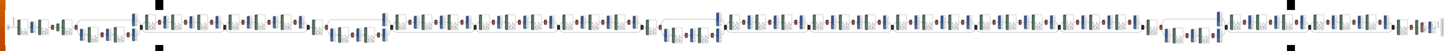
$$\frac{\partial E}{\partial y}$$ Training loss

❶ Forward Pass, ❷ Backward Pass

Deep Neural Networks

- 3 phases:

Deep Neural Network



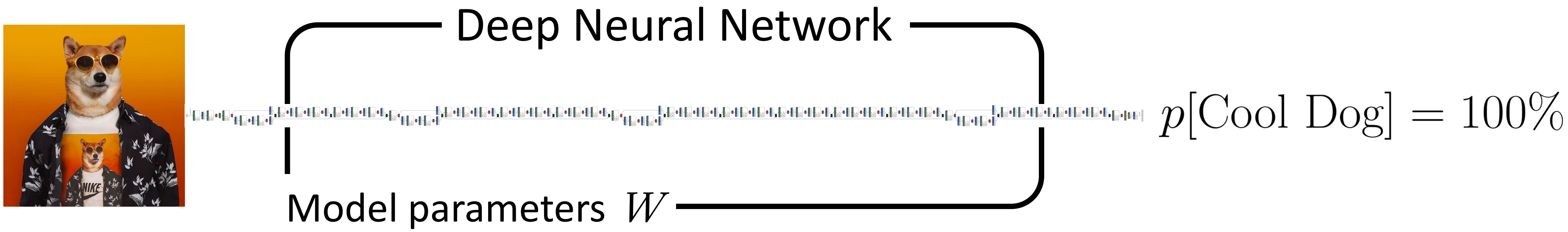
Model parameters W

Learning rate

❶ Forward Pass, ❷ Backward Pass, and ❸ Weight Update $W = W - \alpha \frac{\partial E}{\partial W}$

Deep Neural Networks

- 3 phases:

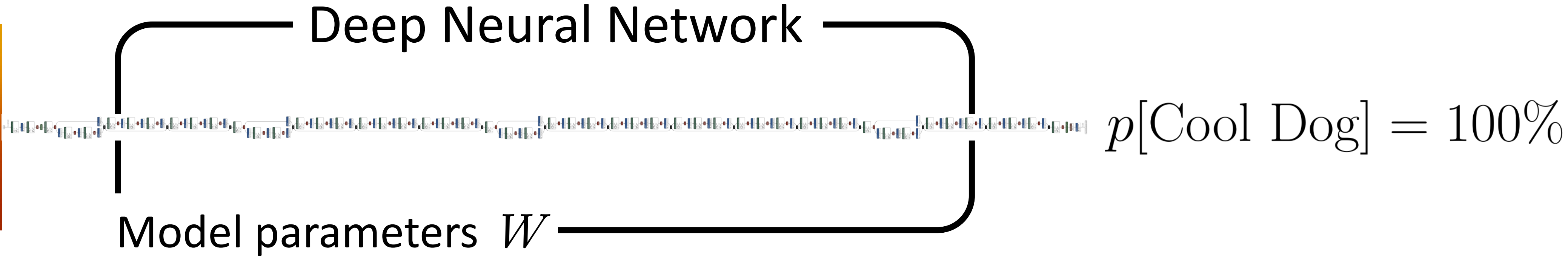


❶ Forward Pass, ❷ Backward Pass, and ❸ Weight Update

- **Training**: Learn the model parameters.

Deep Neural Networks

- 3 phases:



❶ Forward Pass, ❷ Backward Pass, and ❸ Weight Update

- **Inference**: Forward only to obtain the output labels.

Section Summary

- Deep neural networks: graphs of operators and tensors.

- 3 phases & 2 modes of operation:
  - Training: Forward, Backward, and Weight Update
  - Inference: Forward only

- These special properties call for domain-specific system design.

Machine Learning Systems

- Machine Learning Systems Overview
- TensorFlow & PyTorch: Declarative vs. Imperative
- Evolution of PyTorch Compiler Design

Machine Learning Systems Overview

Application

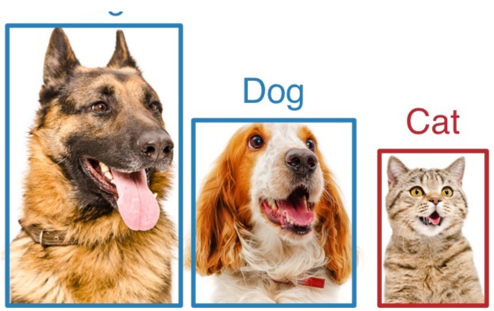


Image Classification

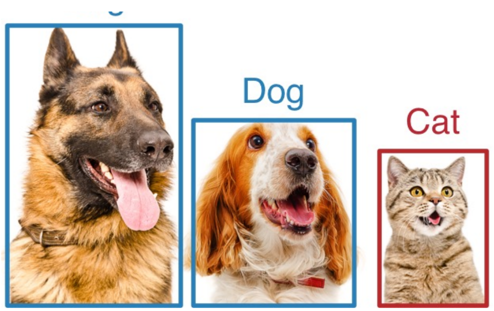
Machine Translation

Speech Recognition

…

Machine Learning Systems Overview

Application



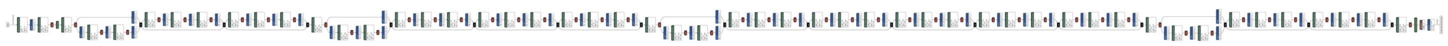
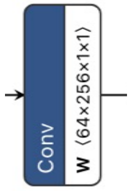


Image Classification

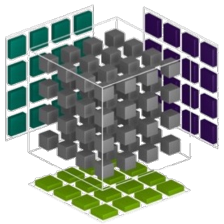
Machine Learning Systems Overview



Operator
(e.g., Convolution)

Invoke

Vendor Libraries



cuBLAS

cuDNN

•••

Hardware

NVIDIA GPU

•••

Machine Learning Systems Overview

Python Programming Front-end

C++ Framework Core

Vendor APIs & Libraries

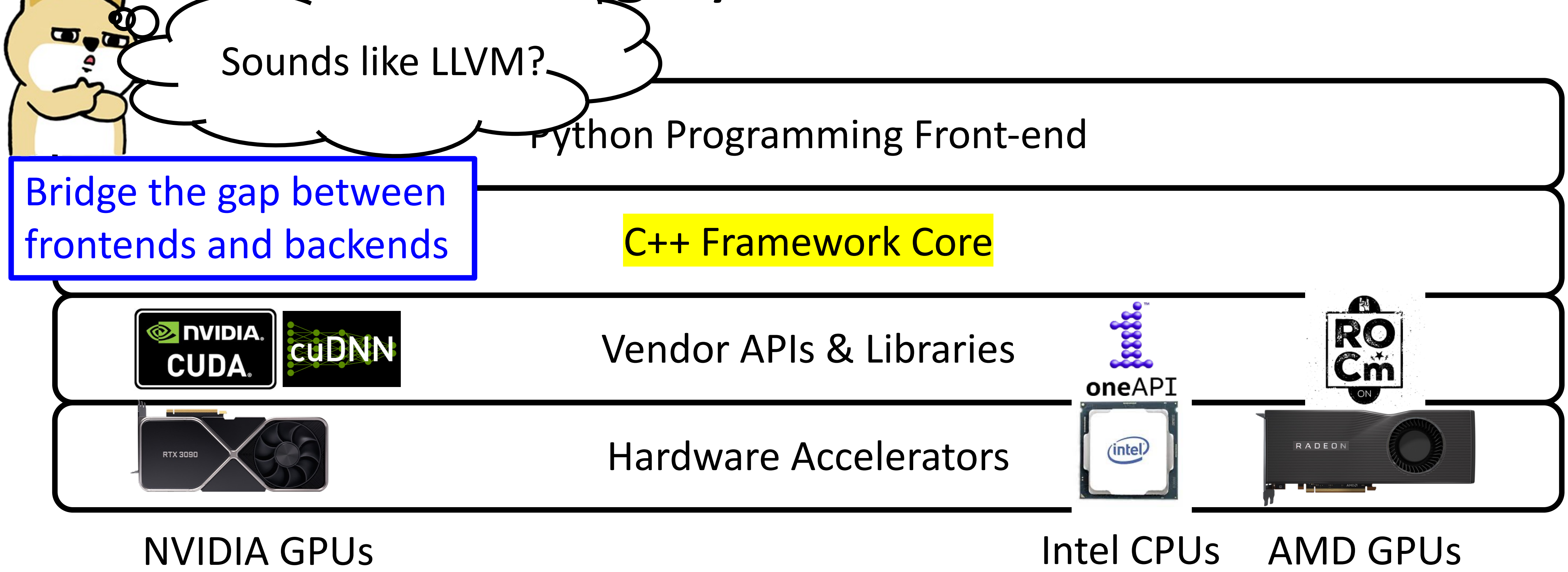
NVIDIA CUDA | cuDNN | oneAPI | ROCm

Hardware Accelerators

NVIDIA GPUs | Intel CPUs | AMD GPUs

- Apply generically to many state-of-the-art systems.

Machine Learning Systems Overview



- Apply generically to many state-of-the-art systems.

TensorFlow (V1)

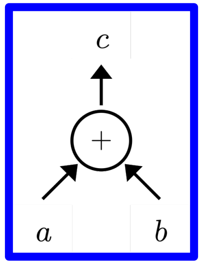
- One of the first mature machine learning frameworks that support GPUs.
- **Declarative** programming paradigm:

```
import tensorflow as tf

a = tf.placeholder()
b = tf.placeholder()
c = a + b
with tf.Session() as sess:
  sess.run(
    c, feed_dict={a: 10, b: 32}
  )
```

TensorFlow (V1)

- One of the first mature machine learning frameworks that support GPUs.
- **Declarative** programming paradigm:

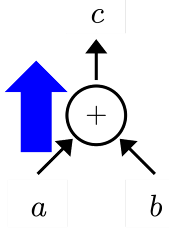
```
import tensorflow as tf

a = tf.placeholder()
b = tf.placeholder()
c = a + b
with tf.Session() as sess:
  sess.run(
    c, feed_dict={a: 10, b: 32}
  )
```



TensorFlow (V1)

- One of the first mature machine learning frameworks that support GPUs.
- **Declarative** programming paradigm:

```
import tensorflow as tf

a = tf.placeholder()
b = tf.placeholder()
c = a + b
with tf.Session() as sess:
  sess.run(
    c, feed_dict={a: 10, b: 32}
  )
```



TensorFlow (V1)

- One of the first mature machine learning frameworks that support GPUs.
- **<u>Declarative</u>** programming paradigm
  - Key Idea: Create a **<u>compilable</u>** graph object in Python, an interpreter environment.

(+) **<u>Holistic view</u>** of the model makes many optimizations easy to implement.

- TensorFlow **Grappler** Optimizer, responsible for
  - Arithmetic optimizations, e.g., constant folding, common subexpression elimination, dead *node* elimination, …
  - Memory allocations
  - …

TensorFlow (V1)

- One of the first mature machine learning frameworks that support GPUs.
- **Declarative** programming paradigm
  - Key Idea: Create a **compilable** graph object in Python, an interpreter environment.

(+) **Holistic view** of the model makes many optimizations easy to implement.

(–) **Hard to program** models with dynamic control flows.

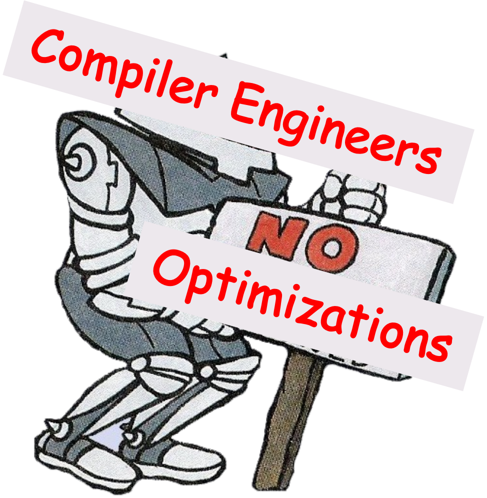
(–) **Hard to debug** intermediate tensor values.

PyTorch

- One of the prevalent machine learning frameworks that adopts **imperative** programming.

(+) Easy to program and debug.

(–) No graphs, …



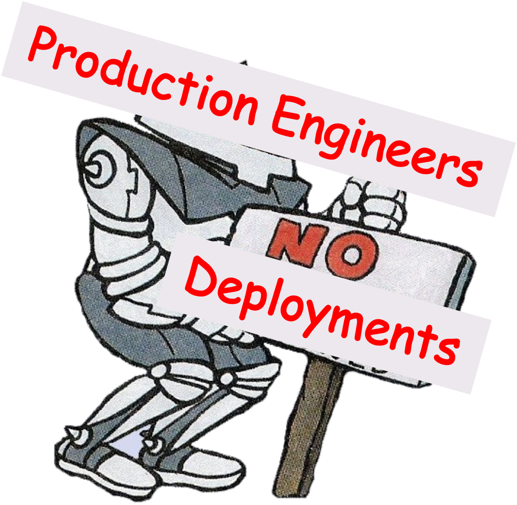
https://chips-compilers-mlsys-22.github.io/assets/slides/PyTorch%20Compilers%20(Compiler%20&%20Chips%20Symposium%202022).pdf

PyTorch

- One of the prevalent machine learning frameworks that adopts **<u>imperative</u>** programming.

(+) Easy to program and debug.

(–) No graphs, …



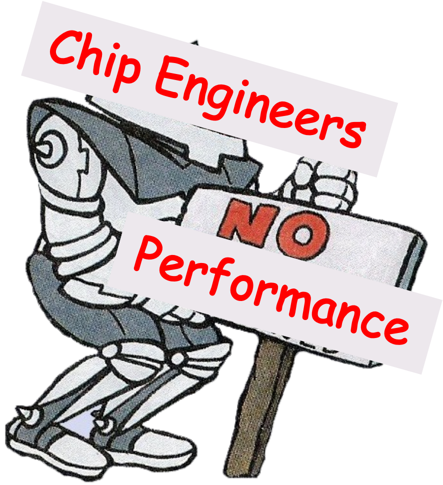
https://chips-compilers-mlsys-22.github.io/assets/slides/PyTorch%20Compilers%20(Compiler%20&%20Chips%20Symposium%202022).pdf

PyTorch

- One of the prevalent machine learning frameworks that adopts **imperative** programming.

(+) Easy to program and debug.

(–) No graphs, …



https://chips-compilers-mlsys-22.github.io/assets/slides/PyTorch%20Compilers%20(Compiler%20&%20Chips%20Symposium%202022).pdf

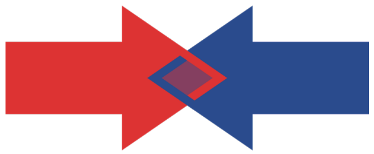
TensorFlow & PyTorch

TensorFlow

- TensorFlow.**Eager** switched to imperative execution in 2018.

PyTorch

- Gen1: torch.jit.**script/trace**
- Gen2: torch.**fx**
- Gen3: torch.**Dynamo**



PyTorch Gen1 Compiler

torch.jit.script

- An embedded language that moves outside of Python.

torch.jit.trace

- A tracer that records all the evaluated operators.

```
import torch
from torch.nn import Module

class MyModel(Module):
  ...

model = MyModel()
```

```
scripted_model = \
    torch.jit.script(model)
```

```
traced_model = torch.jit.trace(
  model, (sample_input,)
)
```

PyTorch Gen1 Compiler

torch.jit.**script**

- An embedded language that moves outside of Python.

(+) Easy to deploy and convert to other formats.

(–) Limited operator coverage.

```
scripted_model = \
    torch.jit.script(model)
```

torch.jit.**trace**

- A tracer that records all the evaluated operators

(–) Specialized to the provided sample input.

```
traced_model = torch.jit.trace(
  model, (sample_input,)
)
```

PyTorch Gen2 Compiler torch.fx

- Key Idea: Python-to-Python transformation.
- 3 main components:
  - Symbolic tracing

```python
import torch
from torch.nn import Module

class MyModel(Module):
  def forward(self, x, y):
    return x + y


model = MyModel()

traced_model = \
    torch.fx.symbolic_trace(module)
```

Feed in proxy inputs and record operations on them

PyTorch Gen2 Compiler torch.fx

- Key Idea: Python-to-Python transformation.
- 3 main components:
  - Symbolic tracing
  - Duck 🦆-typed Python IR

Operate on this representation

```
import torch
from torch.nn import Module

class MyModel(Module):
  def forward(self, x, y):
    return x + y


model = MyModel()

fx_model = torch.fx.symbolic_trace(module)

print(fx_model.graph)
"""
graph():
  %x : = placeholder[target=x]
  %y : = placeholder[target=y]
  %ret : = call_function[target=op.add](
    args=(%x, %y), kwargs={}
  )
"""
```

PyTorch Gen2 Compiler torch.fx

- Key Idea: Python-to-Python transformation.
- 3 main components:
  - Symbolic tracing
  - Duck 🦆-typed Python IR
  - Python code generation

```
import torch
from torch.nn import Module

class MyModel(Module):
  def forward(self, x, y):
    return x + y


model = MyModel()
fx_model = torch.fx.symbolic_trace(module)
print(fx_model.graph)
fx_model.recompile()
print(fx.code)
"""
def forward(self, x, y):
  return x + y
"""
```

PyTorch Gen2 Compiler torch.fx

- Key Idea: Python-to-Python transformation.

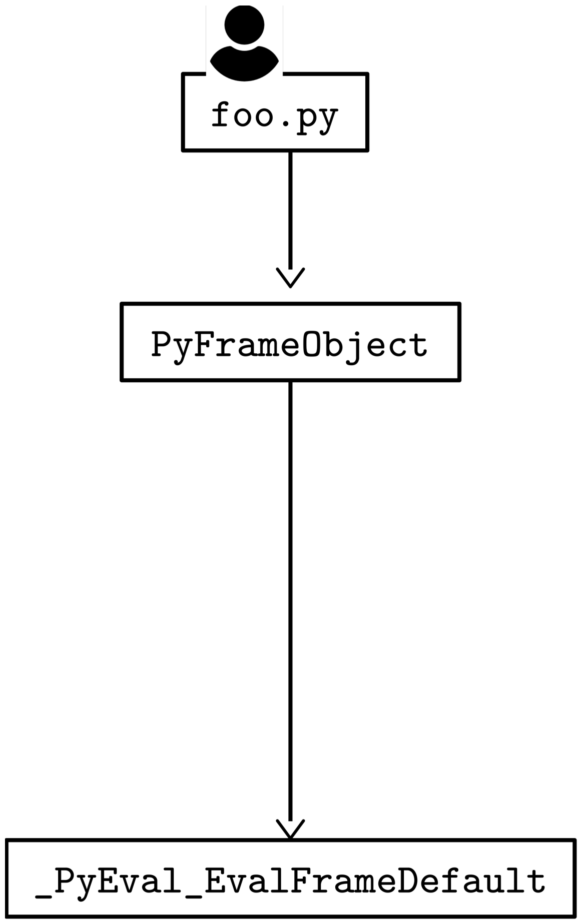
- 3 main components:
  - Symbolic tracing
  - Duck 🦆-typed Python IR
  - Python code generation

(+) The Python-like IR is easy to understand and manipulate.

```python
import torch
from torch.nn import Module

class MyModel(Module):
  def forward(self, x, y):
    return x + y


model = MyModel()
fx_model = torch.fx.symbolic_trace(module)
print(fx_model.graph)
fx_model.recompile()
print(fx.code)
```

PyTorch Gen3 Compiler torch.Dynamo

- Key Idea: torch.fx but supports **partial** capture.

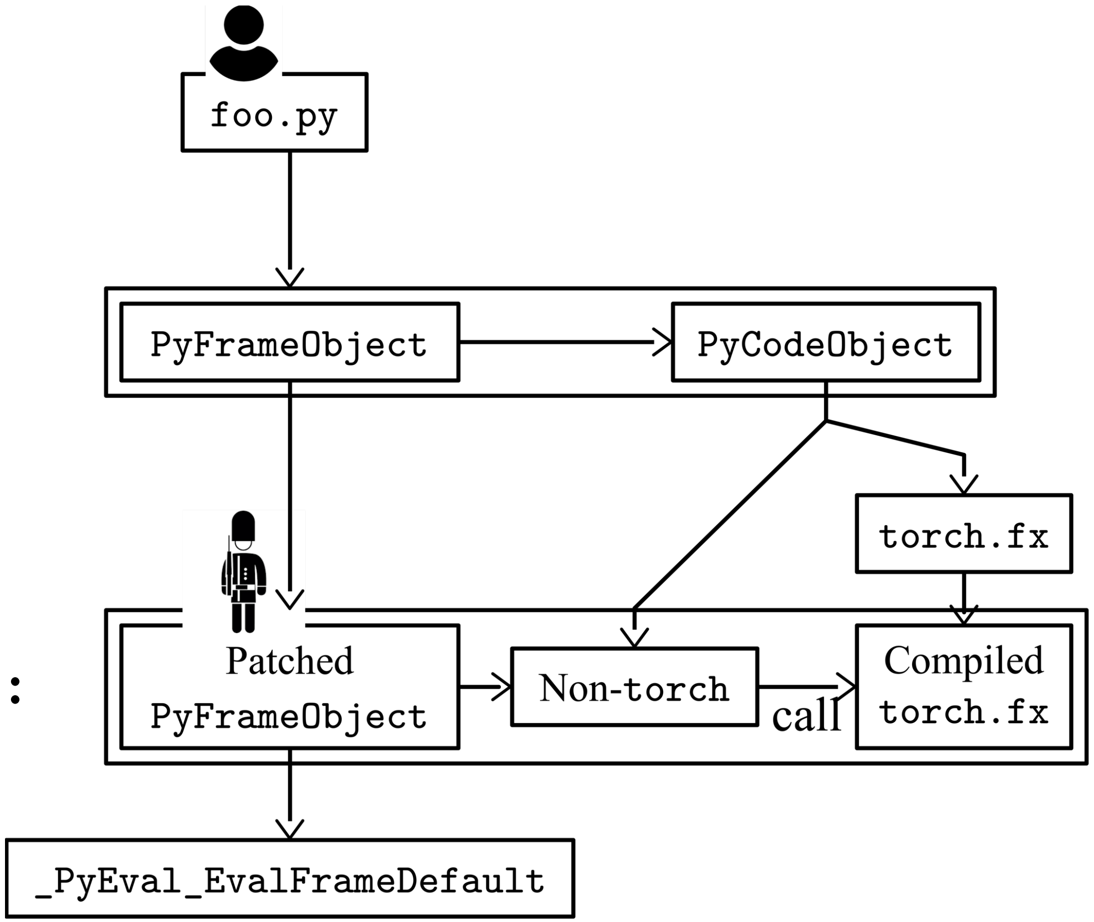


https://chips-compilers-mlsys-22.github.io/assets/slides/PyTorch%20Compilers%20(Compiler%20&%20Chips%20Symposium%202022).pdf

PyTorch Gen3 Compiler torch.Dynamo

- Key Idea: torch.fx but supports **partial** capture.

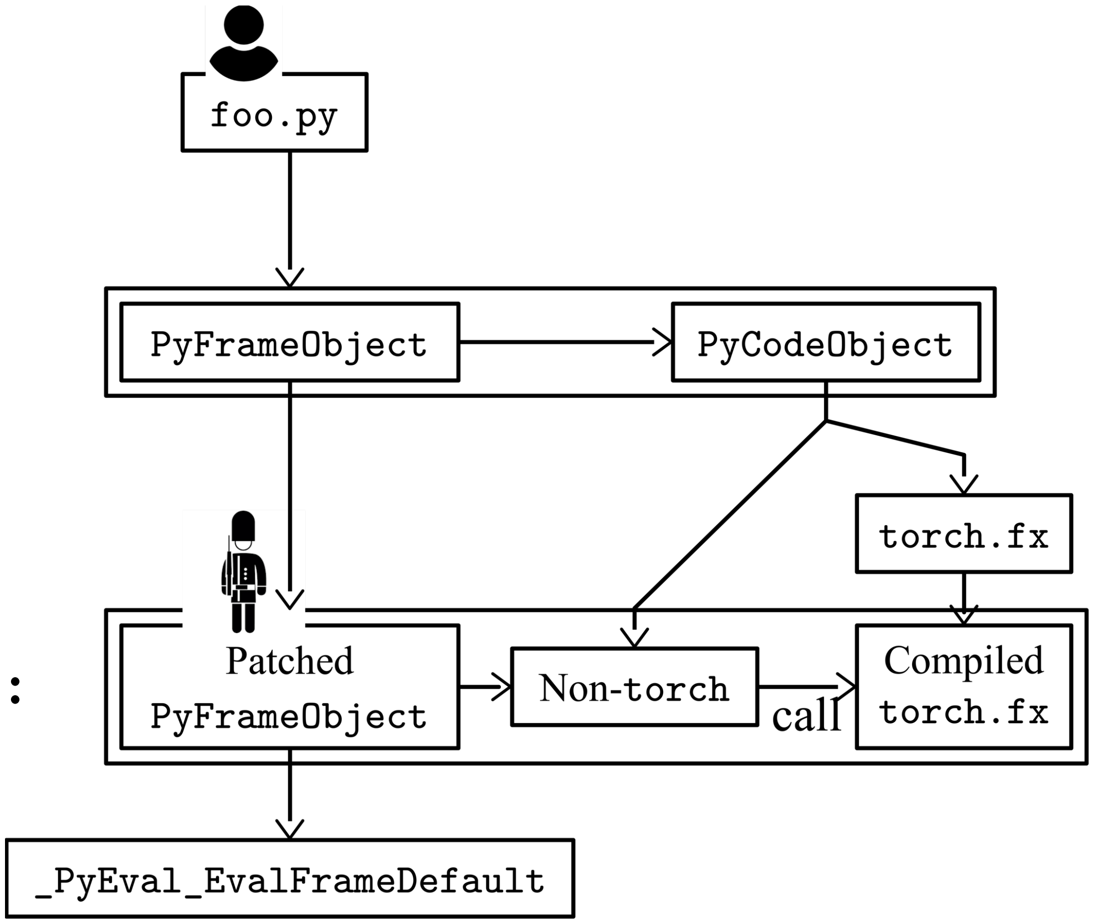
```
import torch

def toy_example(a, b):
  x = a / (torch.abs(a) + 1)
  if b.sum() > 0:
    b = b * -1
  return x * b

def my_pass(fx_module, sample_inputs):
  pass

with torch.dynamo.optimize(my_pass):
  toy_example(
    torch.randn(10), torch.randn(10)
  )
```



PyTorch Gen3 Compiler torch.Dynamo

- Key Idea: torch.fx but supports **partial** capture.

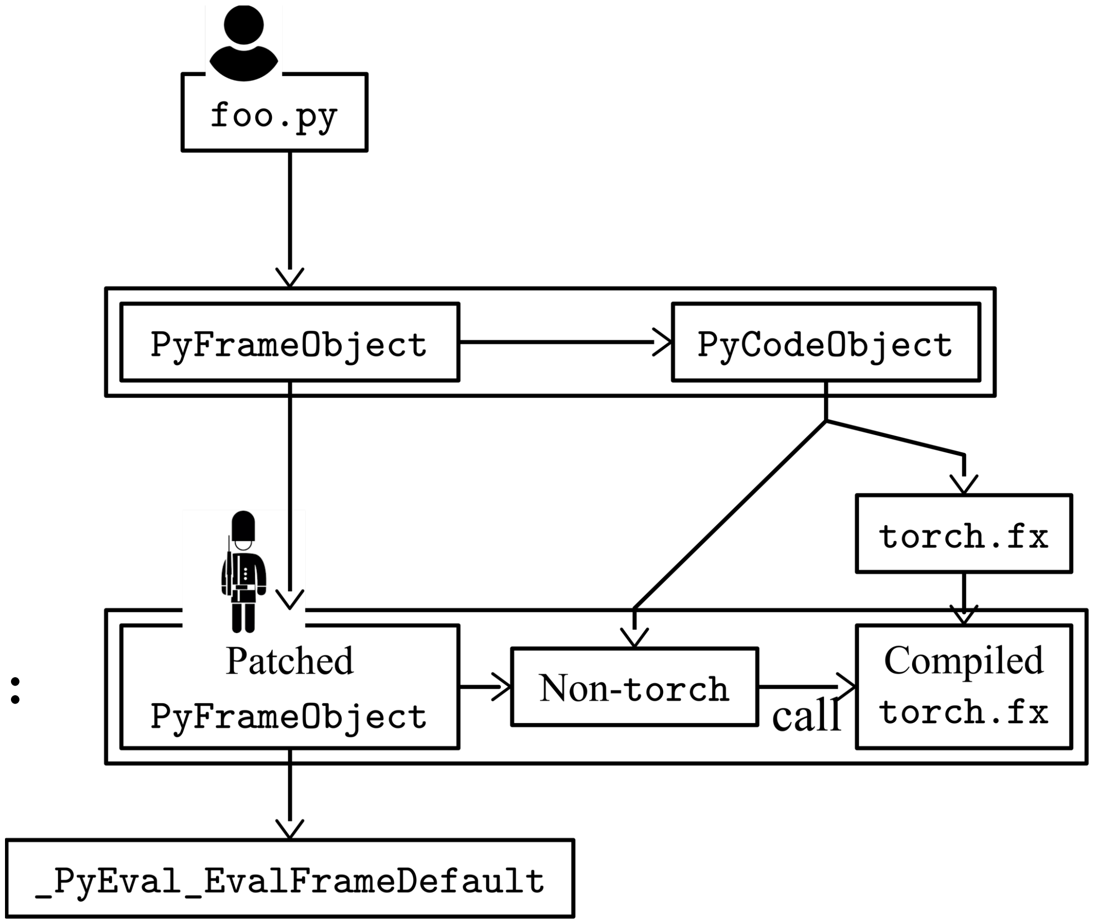
```
import torch

def toy_example(a, b):
  x = a / (torch.abs(a) + 1)
  if b.sum() > 0:
    b = b * -1
  return x * b

def my_pass(fx_module, sample_inputs):
  pass

with torch.dynamo.optimize(my_pass):
  toy_example(
    torch.randn(10), torch.randn(10)
  )
```



PyTorch Gen3 Compiler torch.Dynamo

- Key Idea: torch.fx but supports **partial** capture.

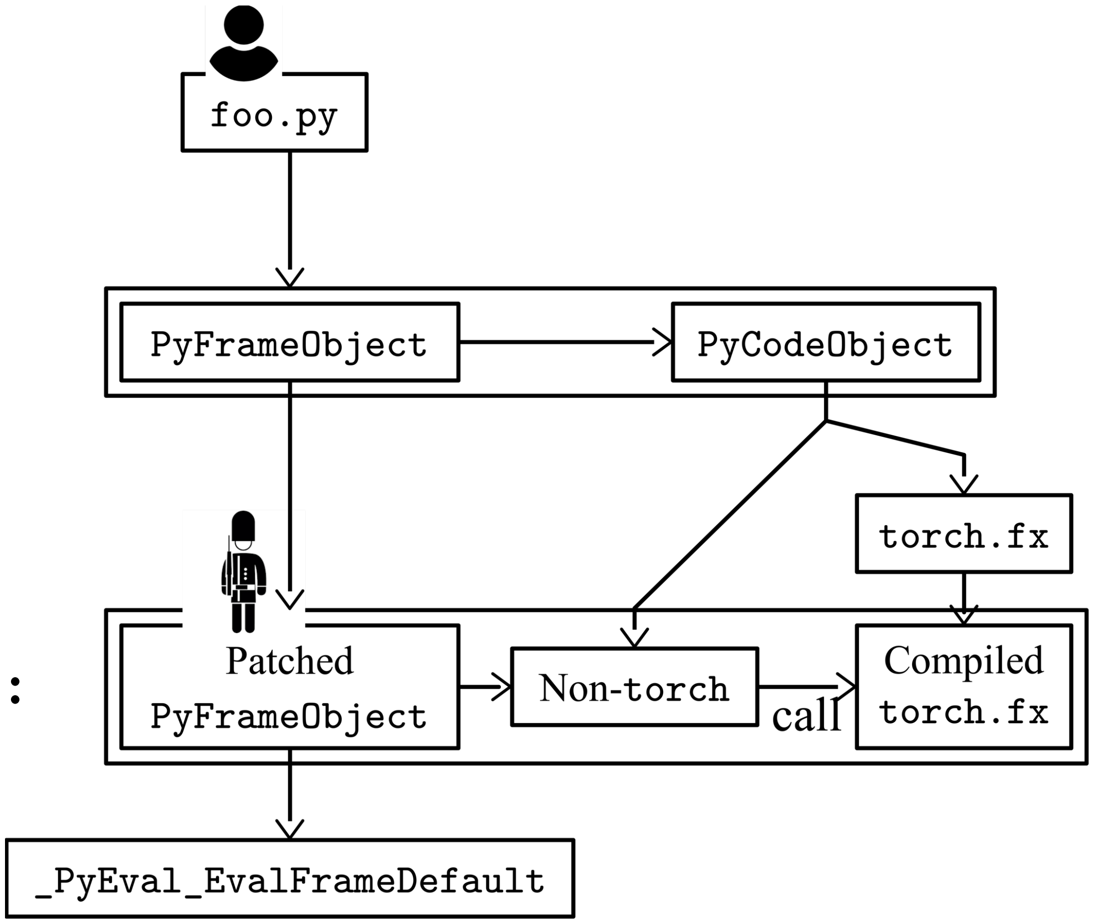
```
import torch

def toy_example(a, b):
  x = a / (torch.abs(a) + 1)
  if b.sum() > 0:
    b = b * -1
  return x * b

def my_pass(fx_module, sample_inputs):
  pass

with torch.dynamo.optimize(my_pass):
  toy_example(
    torch.randn(10), torch.randn(10)
  )
```



PyTorch Gen3 Compiler torch.Dynamo

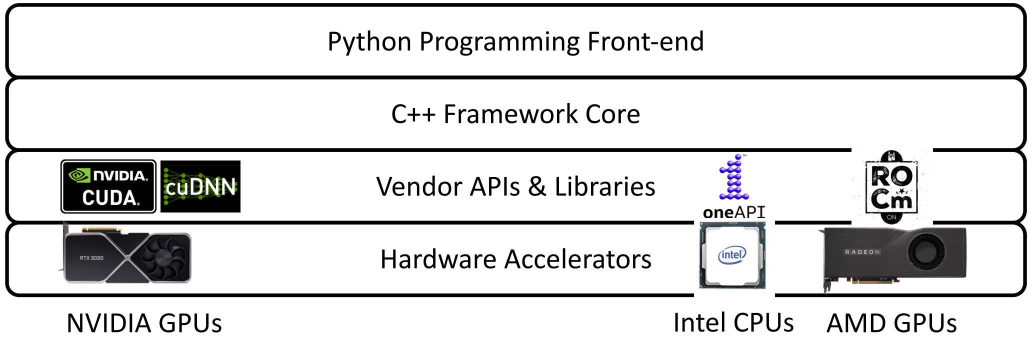
- Key Idea: torch.fx but supports **partial** capture.

```
import torch

def toy_example(a, b):
  x = a / (torch.abs(a) + 1)
  if b.sum() > 0:
    b = b * -1
  return x * b

def my_pass(fx_module, sample_inputs):
  pass

with torch.dynamo.optimize(my_pass):
  toy_example(
    torch.randn(10), torch.randn(10)
  )
```



Section Summary

- MLSys Overview



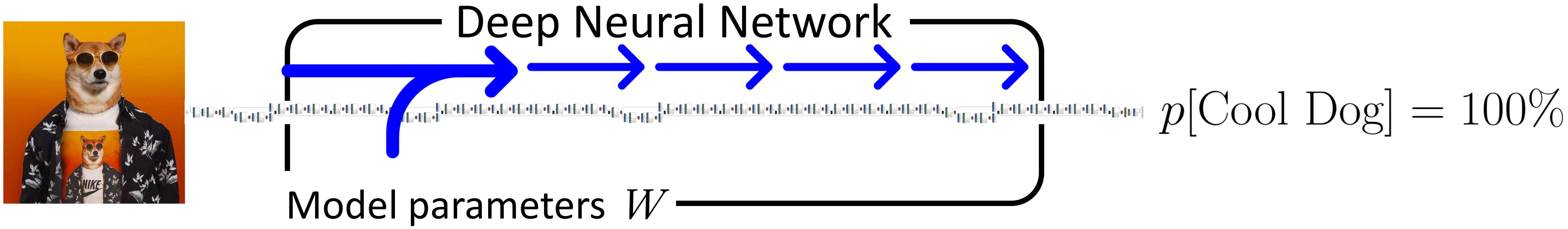
- TensorFlow and PyTorch
  - Declarative vs. Imperative
- Evolution of PyTorch Compilers
  - Gen1: Scripting and tracing
  - Gen2: Ducked-type Python IR
  - Gen3: Partial capture
- Can jump out of those existing systems and create much more **powerful** wheels!
- Please support the research work **Hidet** from my colleague Yaoyao: www.github.com/hidet-org/hidet by staring the repository.

Memory Optimizations

- Background: Feature Maps
- Why memory matters?
- 3 optimization strategies ⇒ Selective Recomputation
- Impact of memory optimizations

Deep Neural Networks

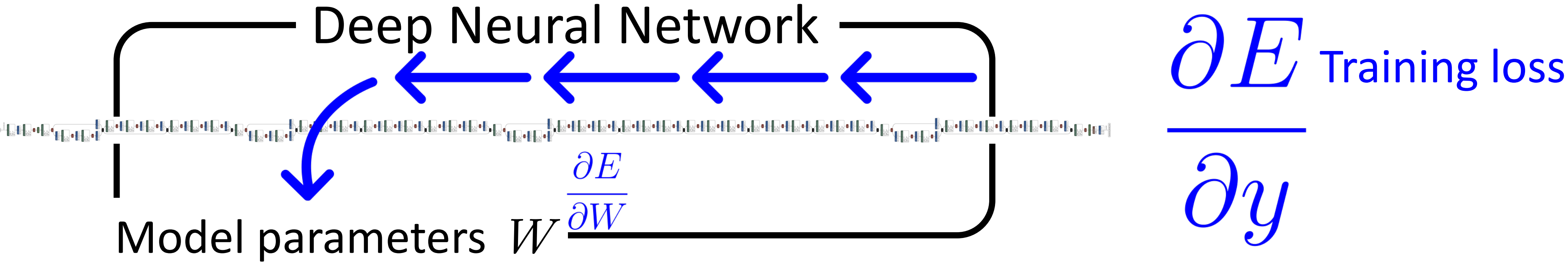
- 3 phases:



❶ Forward Pass

Deep Neural Networks

- 3 phases:



❶ Forward Pass, ❷ Backward Pass

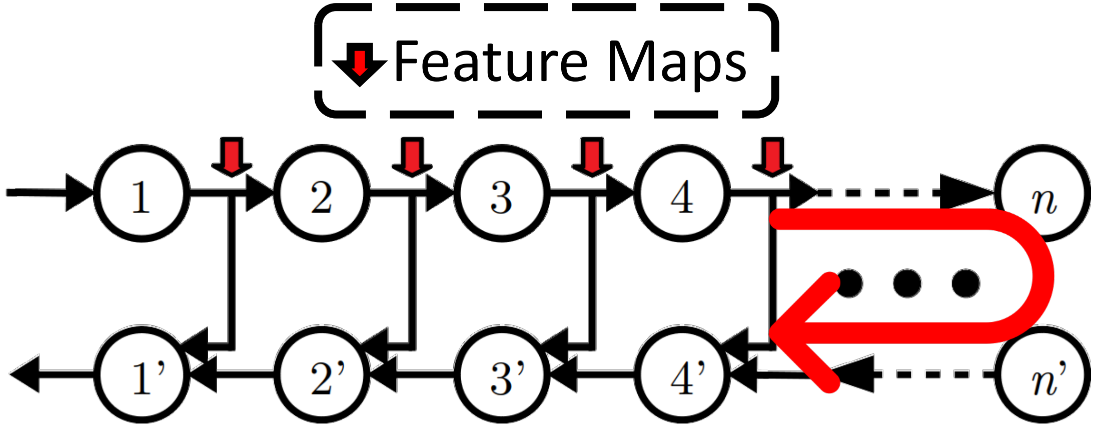
Feature Maps

- Data entries that are stashed by the forward pass to compute the backward gradients.

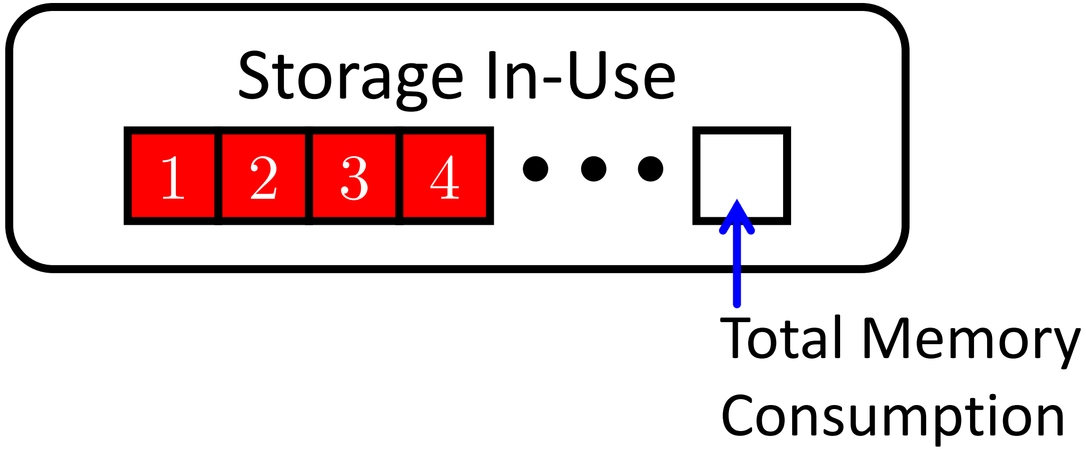
$$
\begin{aligned}
y = \tanh x \Rightarrow \frac{\partial E}{\partial x} &= \frac{\partial E}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}x} \\
&= \frac{\partial E}{\partial y}(1 - \tanh^2 x) \\
&= \frac{\partial E}{\partial y}(1 - y^2)
\end{aligned}
$$

Feature Maps

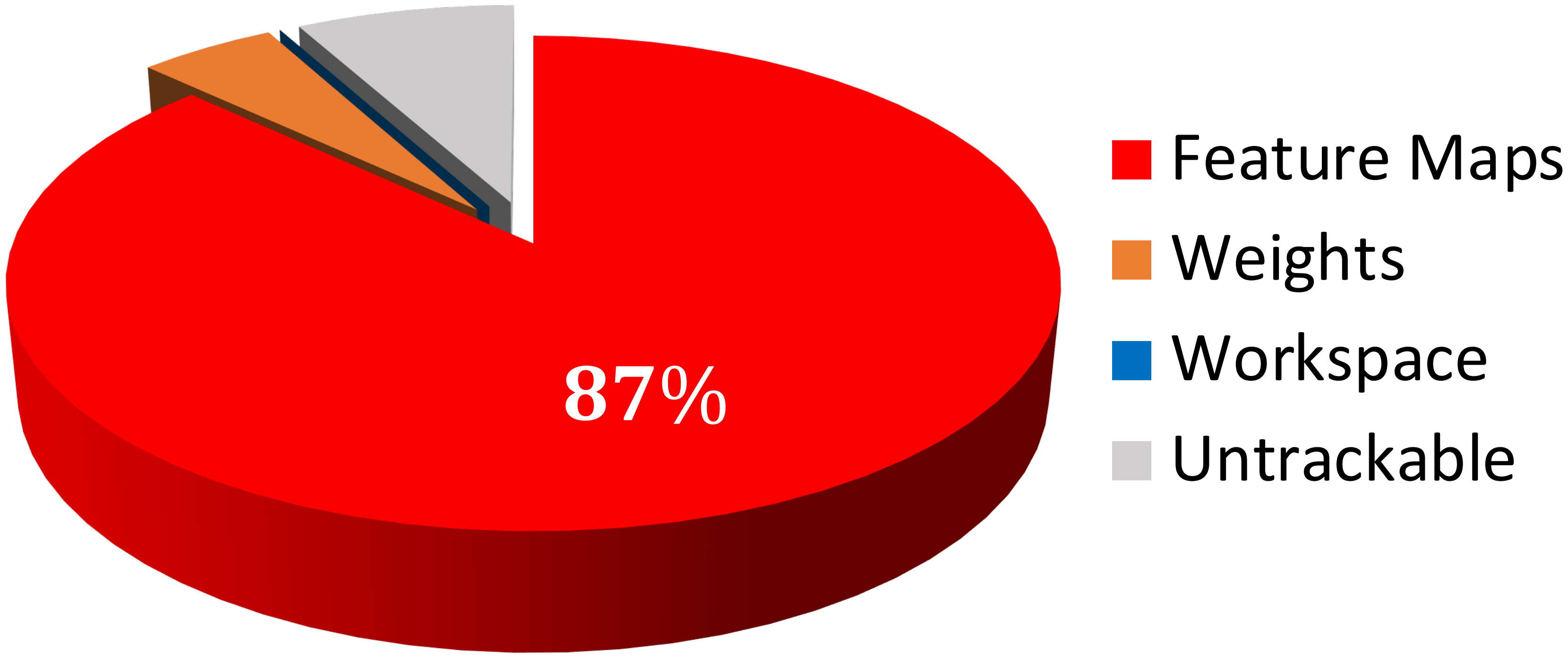
- Data entries that are stashed by the forward pass to compute the backward gradients.



Large **temporal gap** between usage



GPU Memory Consumption Profile of
A Machine Translation Workload



Feature maps dominate the GPU memory consumption

Why memory matters?

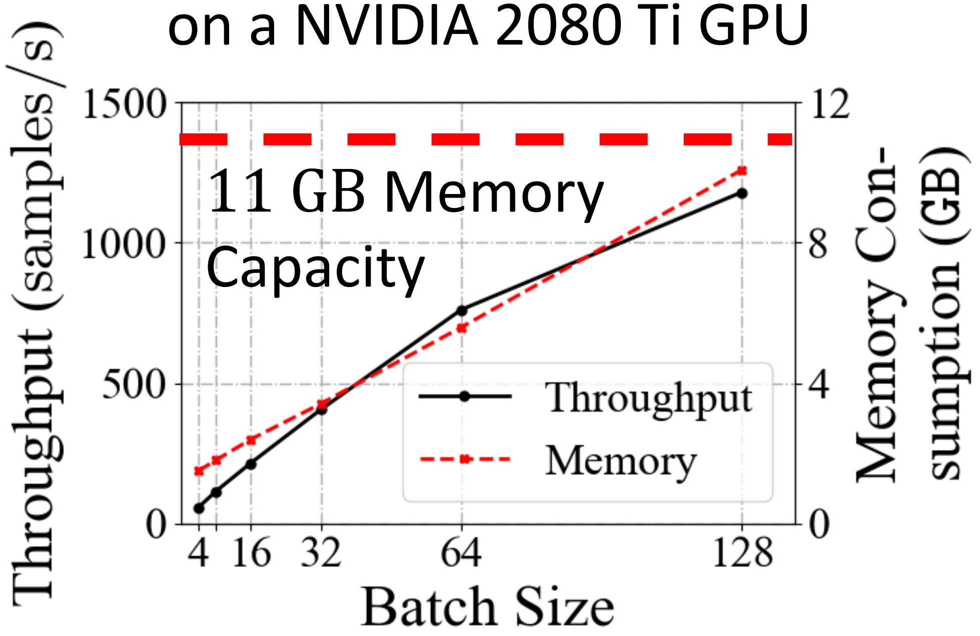
- Hardware accelerators (e.g., NVIDIA GPUs) usually have limited memory capacity (10-40 GB).

- Memory optimizations allow for
  - Training for **deeper** neural networks ($\approx$ better training quality).
  - Higher training **throughputs**.

Memory → Training Throughputs

- When training, data is usually **batched** for higher throughput and faster convergence.

Throughput and memory consumption of English-Vietnamese translation on a NVIDIA 2080 Ti GPU



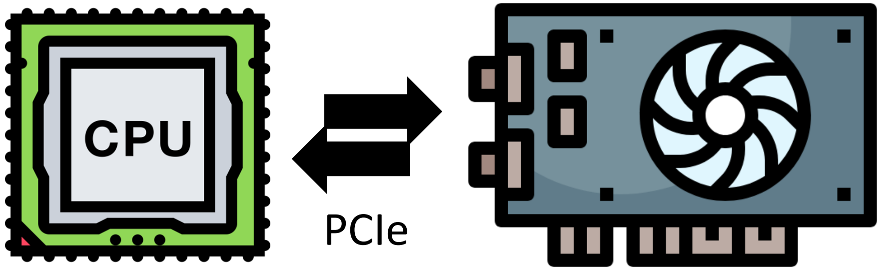
Strategy #1. Virtualization

- Key Idea: Temporarily **offload** data entries to the CPU side.

(+) Generic

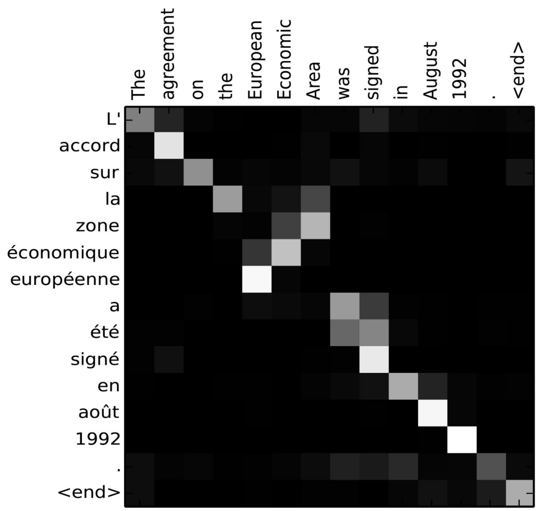
(–) Intensive use of interconnect (a valuable resource in distributed systems)

- Hard to control the **timing**.
  - Significant performance overhead if data is not fetched back on time.
  - Graph + System Information ⇒ What data to offload & When to issue the prefetch.



Strategy #2. Data Encoding

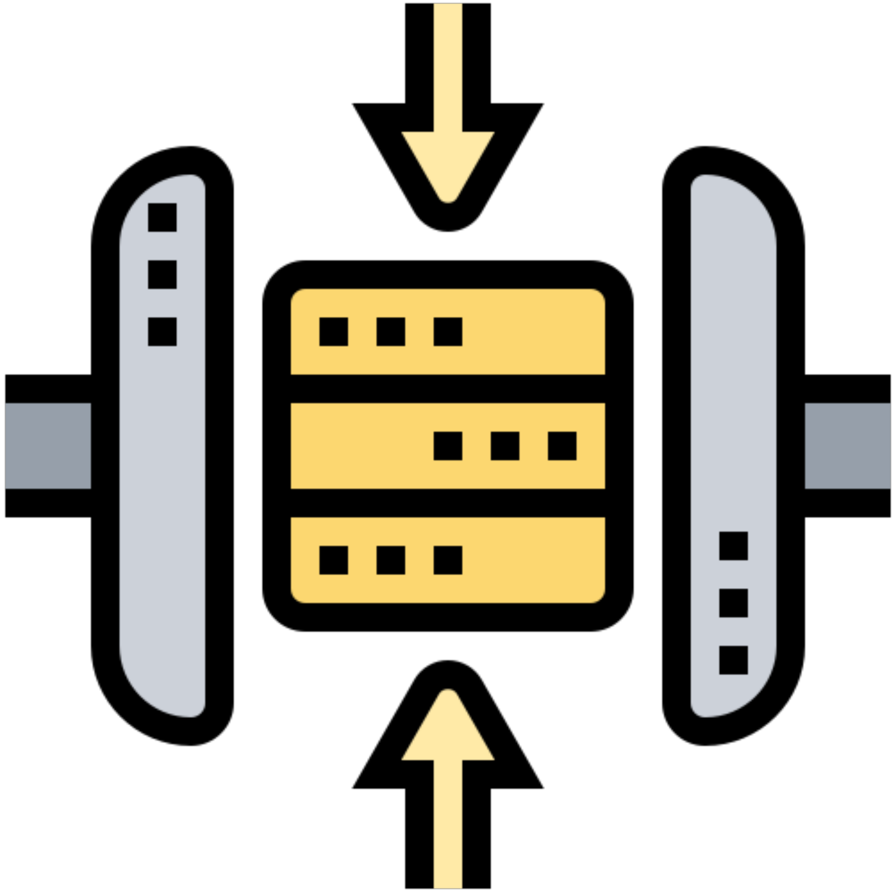
- Key Idea: Compress (usually eliminate zeros).



Example feature maps (darker means → 0)

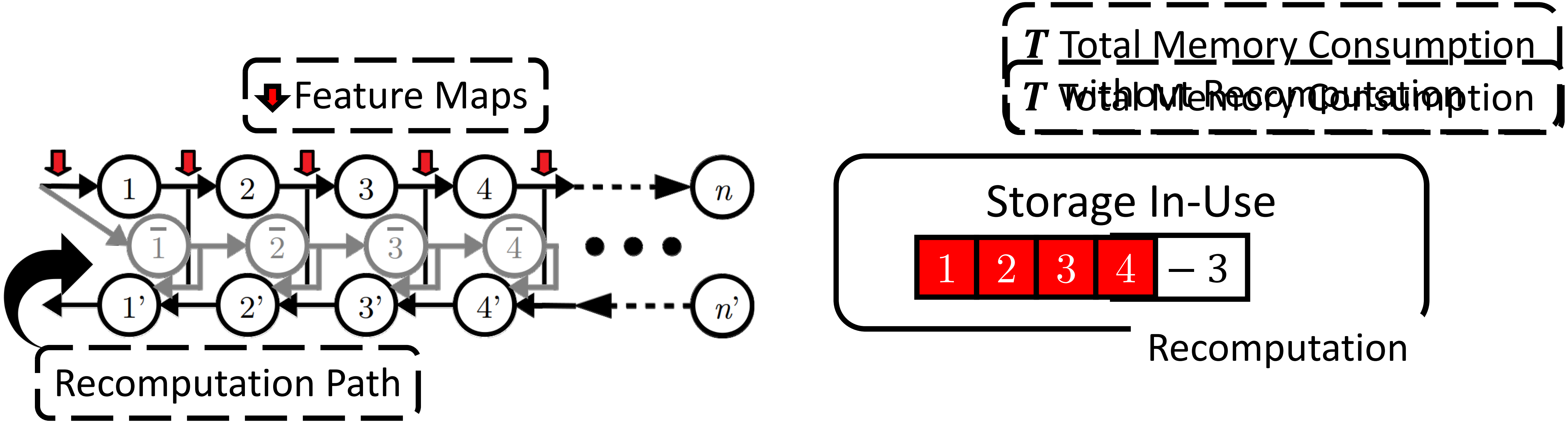
(+) Low performance overhead

(–) Model/layer specific



Strategy #3. Selective Recomputation

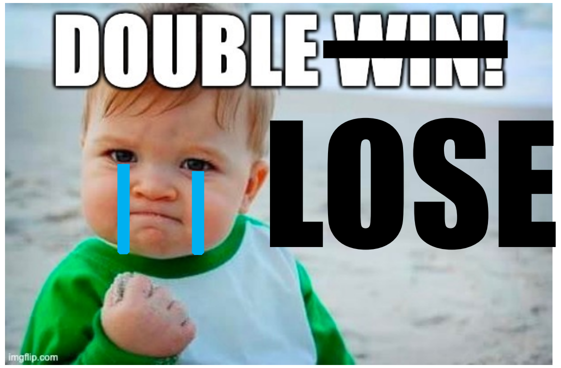
- Key Idea: Trade **runtime** with **memory capacity**.



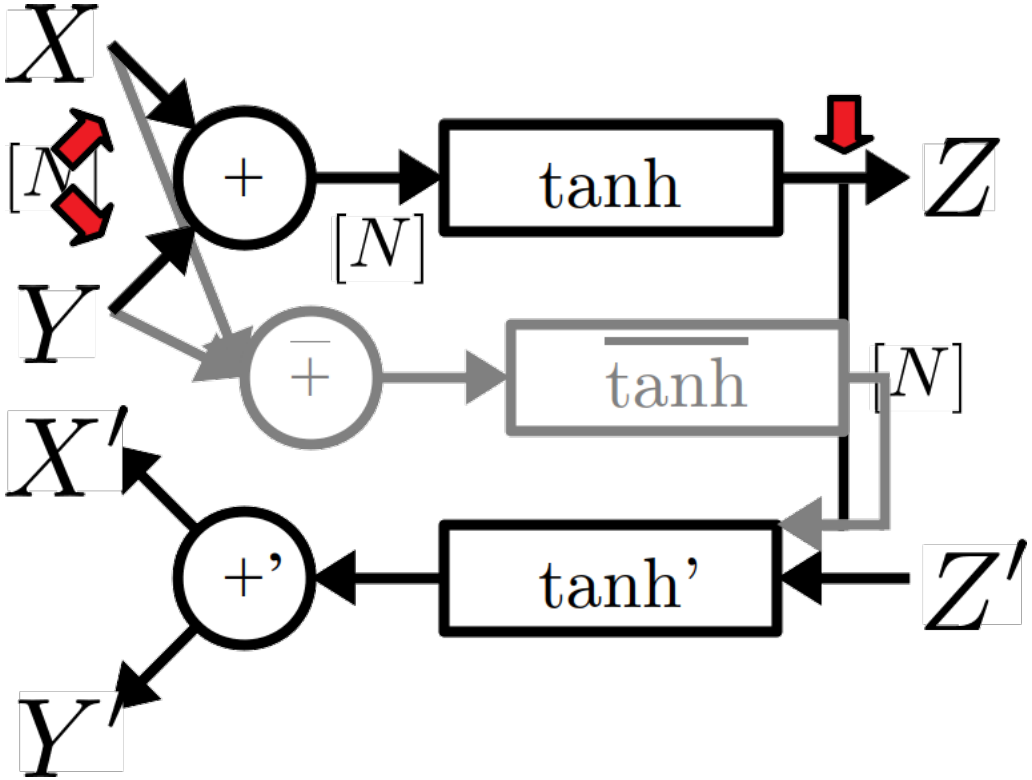
- The recomputation path should only involve **lightweight** operators.

Strategy #3. Selective Recomputation

- Recomputing naively can end up with **more** memory.
  - (–) Feature maps ↑ ($N \rightarrow 2N$)
  - (–) Performance ↓



$$Z = \tanh(X + Y)$$

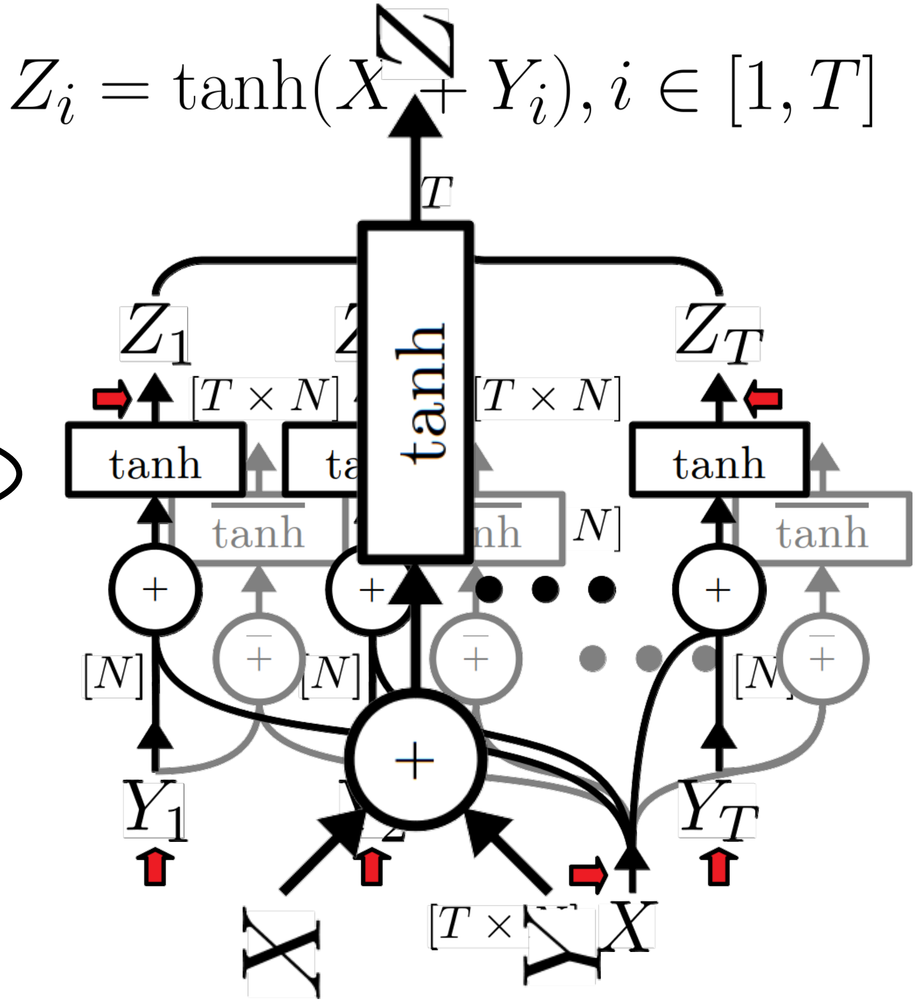


Strategy #3. Selective Recomputation

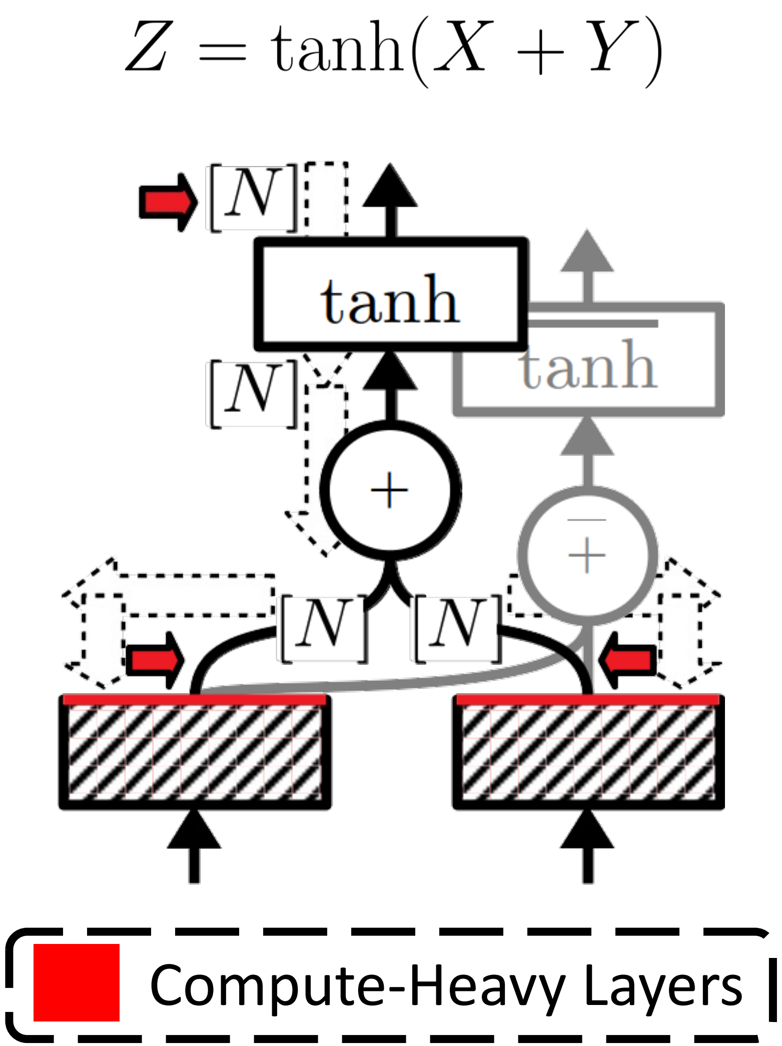
- Not considering the **global** graph structure could have us miss key optimization opportunities.
  - E.g., $T^{2}N \rightarrow 2TN$

Want to know the **sweet spot** where doing recomputation is optimal

Our approach:
Bidirectional Analysis



Bidirectional Analysis

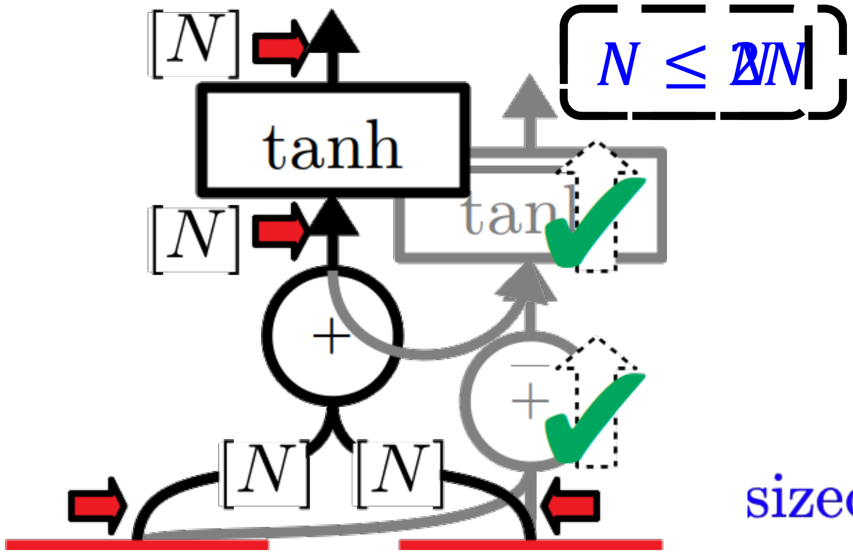


▼ **Backward Pass**

- Breaks at compute-heavy layers to **partition** the graph
- Constructs a recomputation path that consists of nodes visited

Bidirectional Analysis

$$Z = \tanh(X + Y)$$



▼ Backward Pass

- Breaks at compute-heavy layers to **partition** the graph.
- Constructs a recomputation path that consists of nodes visited.

▲ Forward Pass

- Remove operator nodes from the recomputation path if

$$\text{sizeof}\left(\text{FeatureMaps}_{\text{New}}\right) \leq \text{sizeof}\left(\text{FeatureMaps}_{\text{Old}}\right)$$

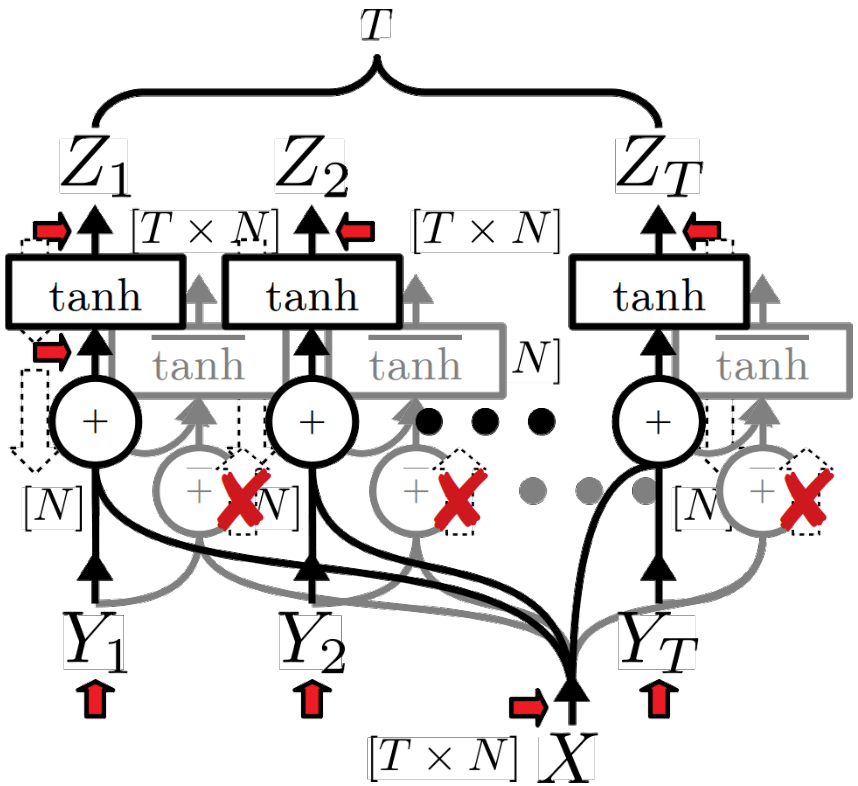
Bidirectional Analysis

- Tensor sharing causes all the correlated operators to forward propagate simultaneously:

$$\text{sizeof}\left(\sum \text{FeatureMaps}_{\text{New}}\right) \leq \text{sizeof}\left(\sum \text{FeatureMaps}_{\text{Old}}\right)$$

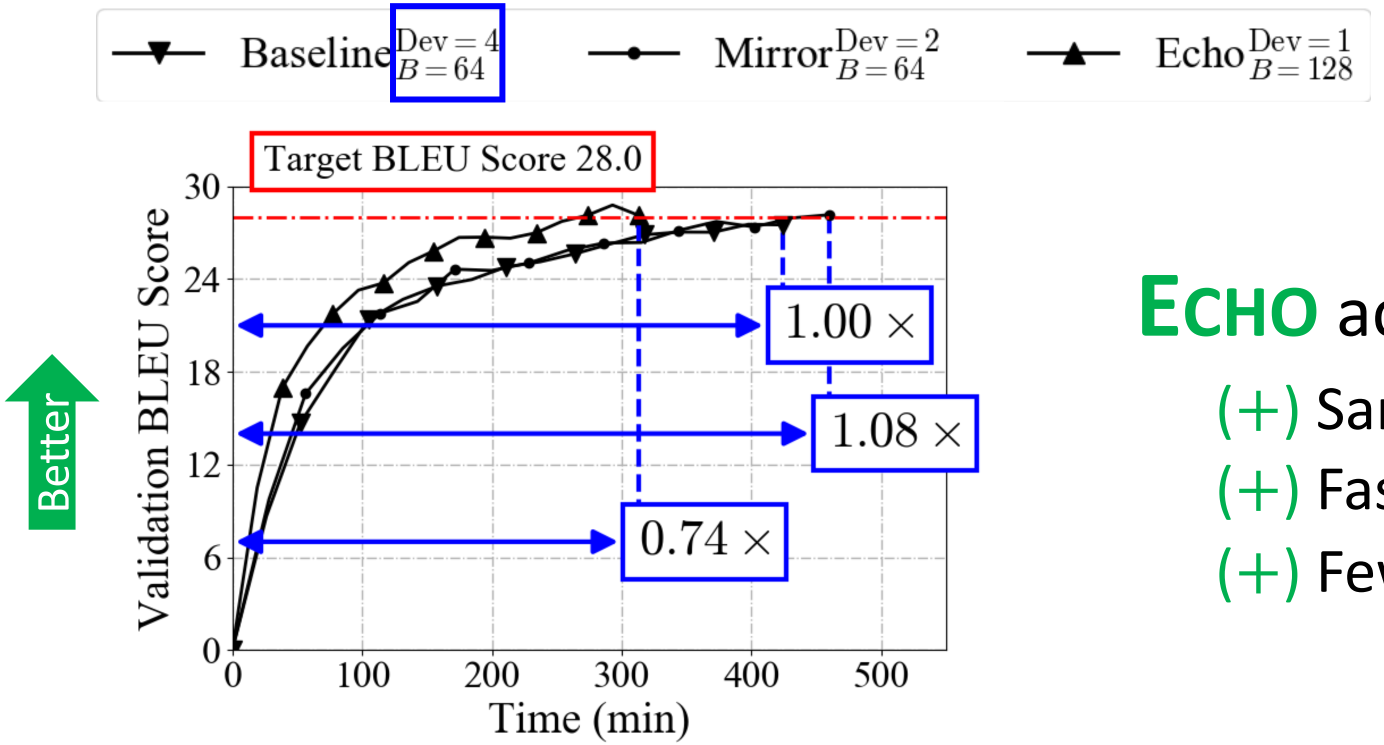
$$\boxed{T^2N \nleq 2TN}$$

$$Z_i = \tanh(X + Y_i), i \in [1, T]$$



Evaluation

English-German translation with the same number of training steps



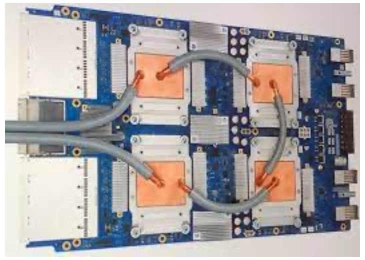
ECHO achieves:

- (+) Same training quality
- (+) Faster convergence
- (+) Fewer compute devices

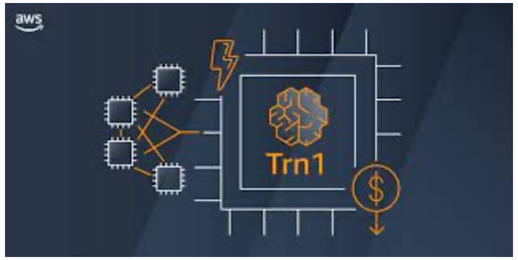
Section Summary

- Why memory matters?
  - Deeper neural networks
  - Higher training throughputs
- 3 Optimization Strategies:
  - Virtualization
  - Data Encoding
  - Selective Recomputation (formulated as a bidirectional analysis)
- Impact of memory optimizations
  - Same training quality
  - Faster convergence
  - Fewer compute devices

Future Vision

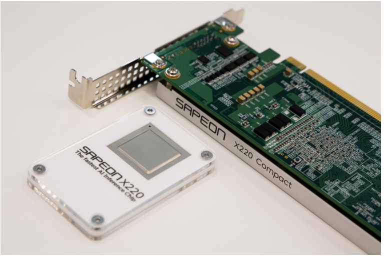


Google TPU



AWS Trainium

Qualcomm
Cloud AI 100



SAPEON X220

● ● ●

Compute power cannot be exploited without a mighty compiler stack.

Compiler Optimizations for Machine Learning Workloads

Bojian Zheng

CSCD70 Compiler Optimization

2023/3/20