MIT HAN LAB

Lecture 16 On-Device-Training and Transfer Learning

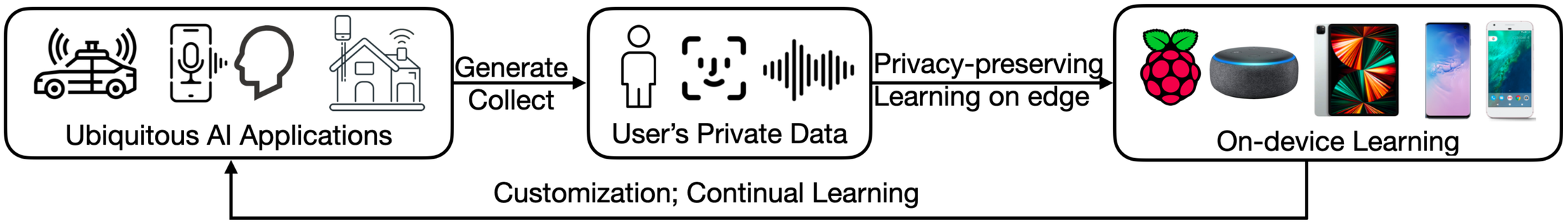
Part II

Song Han
songhan@mit.edu

Can We Learn on the Edge?

From tiny inference to training

A **virtuous cycle**:

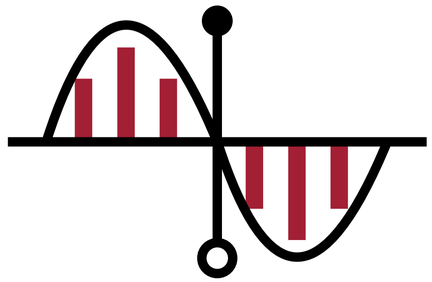


- On-device learning:
  - **customization** by adapting to user data / **life-long** learning
  - better **privacy**, lower **cost**
- Training is more **expensive** than inference
  - For example, store intermediate activation, extra back-propagation, etc.

Section 1: Co-Design for On-Device Training (Part II)

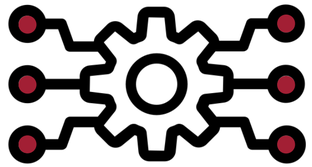
System support for on-device training

On-Device Training Under 256KB Memory



1. Quantization-aware scaling

2. Sparse layer/tensor update

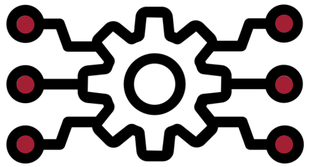


3. Tiny Training Engine

On-Device Training Under 256KB Memory

1. Quantization-aware scaling

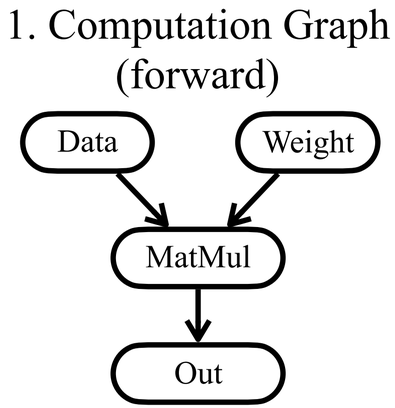
2. Sparse layer/tensor update



3. Tiny Training Engine

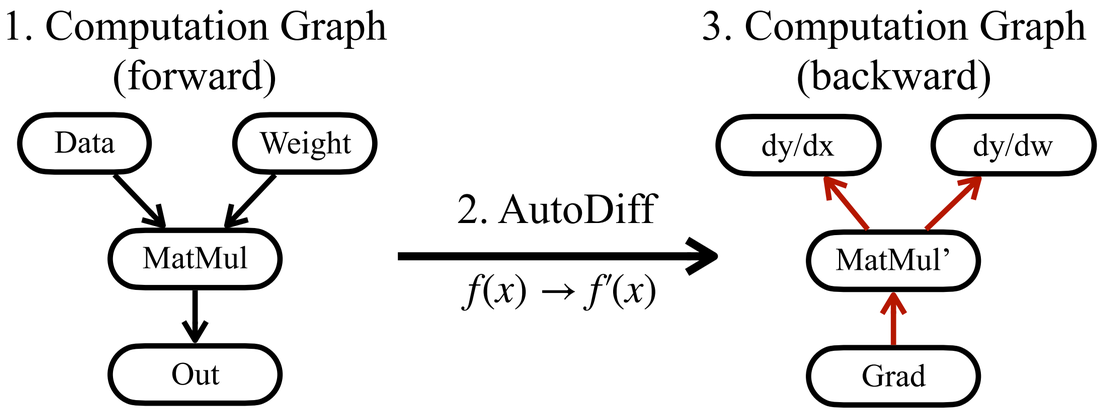
Tiny Training Engine (TTE)

Forward and Backward Computation Graph



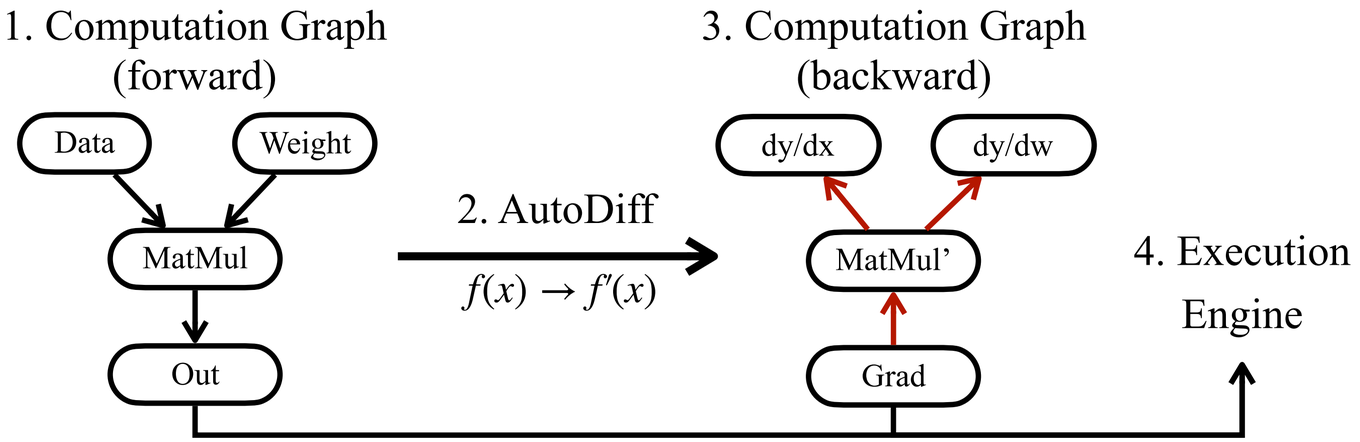
Tiny Training Engine (TTE)

Forward and Backward Computation Graph



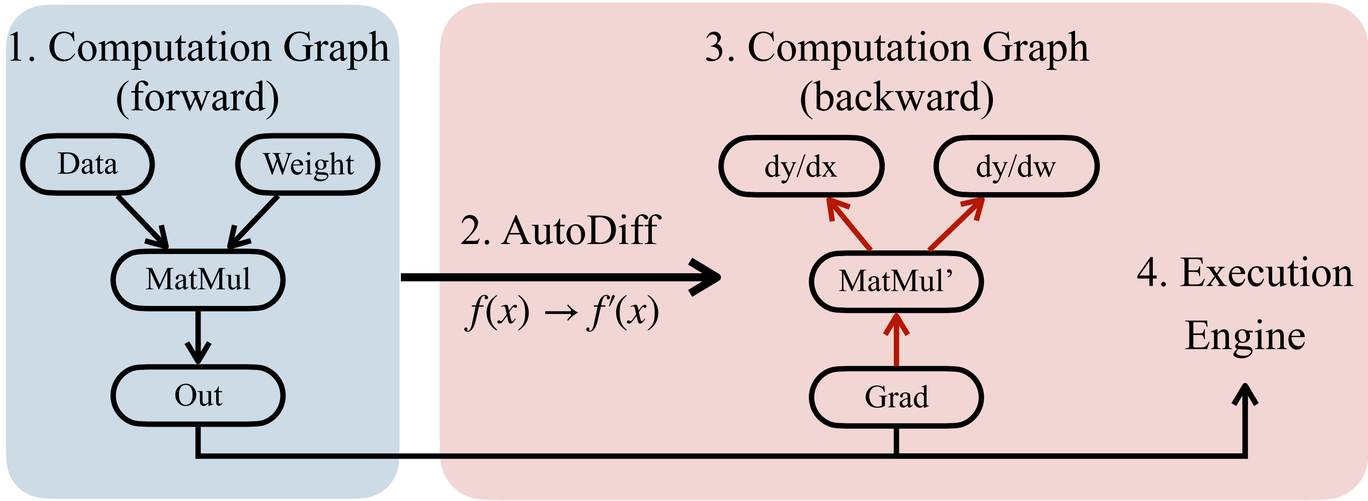
Tiny Training Engine (TTE)

Forward and Backward Computation Graph



Tiny Training Engine (TTE)

Forward and Backward Computation Graph



: Compile-Time

: Runtime

Conventional training framework focus on **flexibility**,
and the auto-diff is performed at **runtime**.
Thus, any optimizations will lead to runtime overhead.

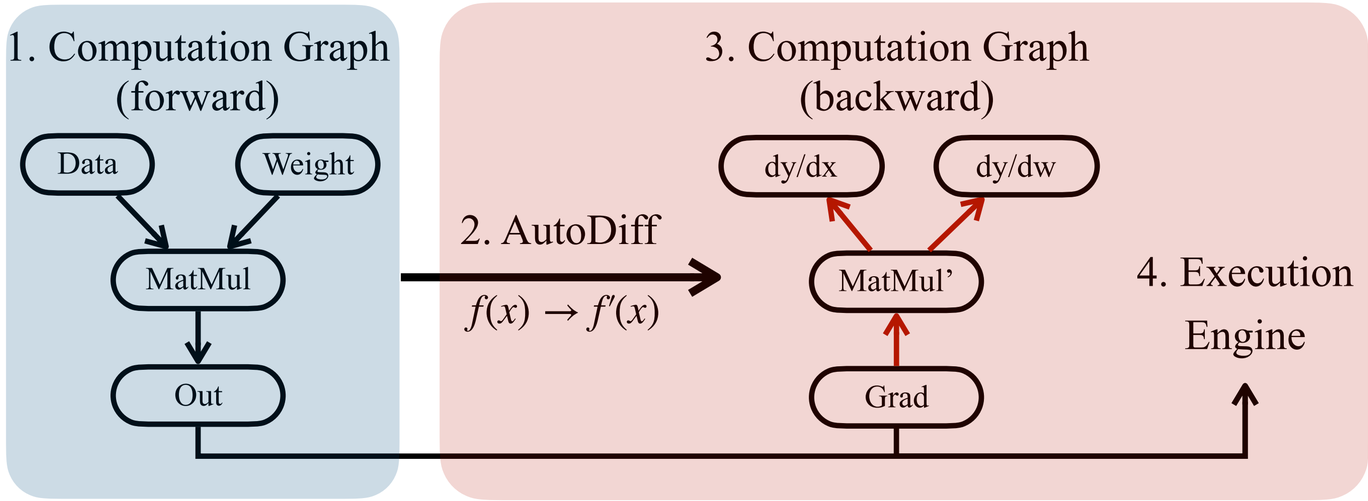
Tiny Training Engine (TTE)

Existing frameworks cannot fit training into tiny devices due to:

- **Runtime** is heavy
  - Heavy dependencies and large binary size (>90MB)
  - Autodiff at runtime
  - Operators optimized for the cloud, not for edge
- **Memory** is heavy
  - A lot of intermediate (and unused) buffers
  - No support for sparse backpropagation

Tiny Training Engine (TTE)

Compile-Time Autodiff



(a) Previous training frameworks

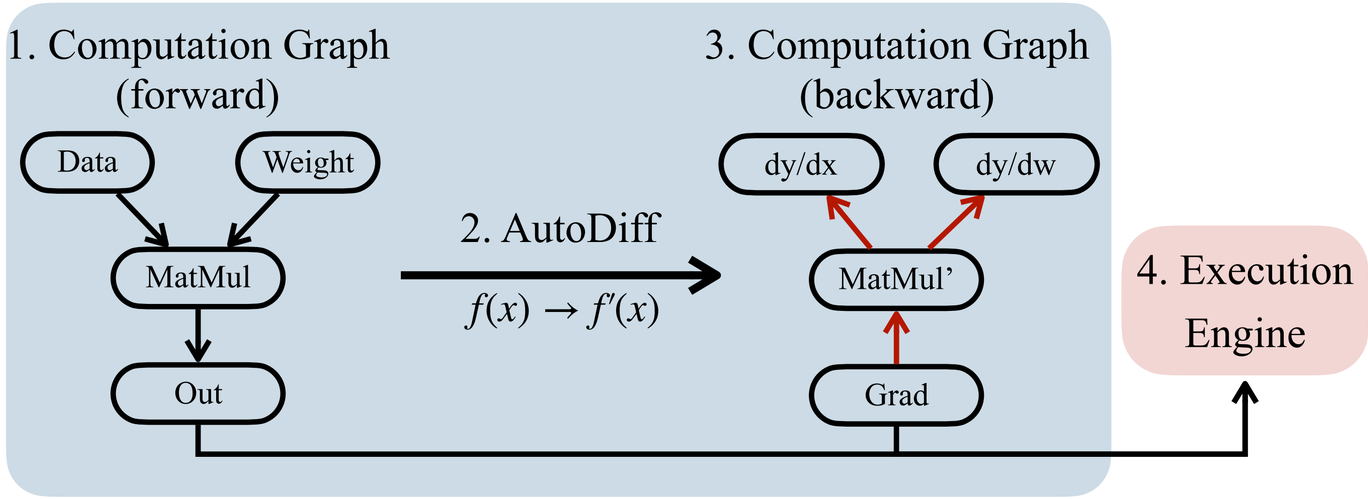
: Compile-Time

: Runtime

Conventional training framework focus on **flexibility**,
and the auto-diff is performed at **runtime**.

Tiny Training Engine (TTE)

Compile-Time Autodiff



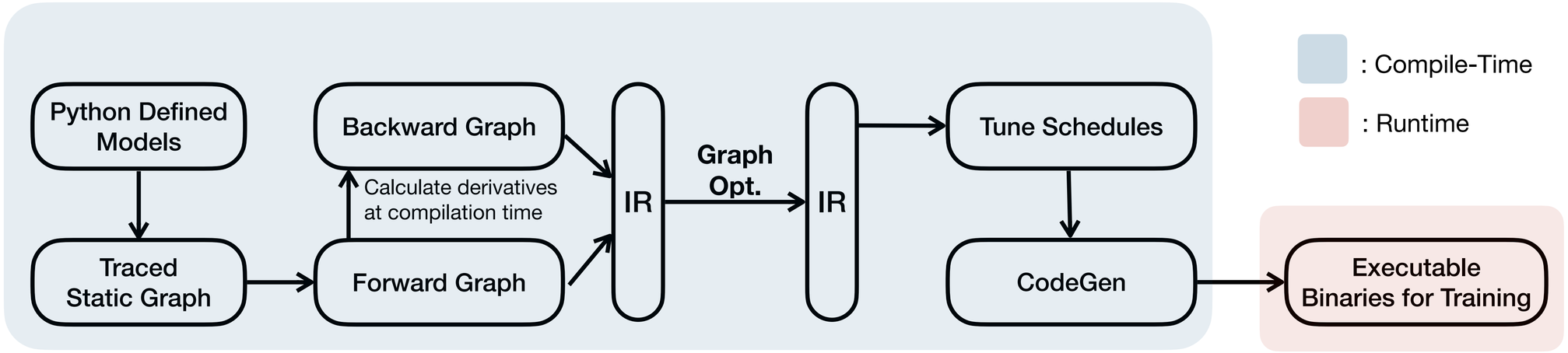
(b) Our Tiny Training Engine

: Compile-Time

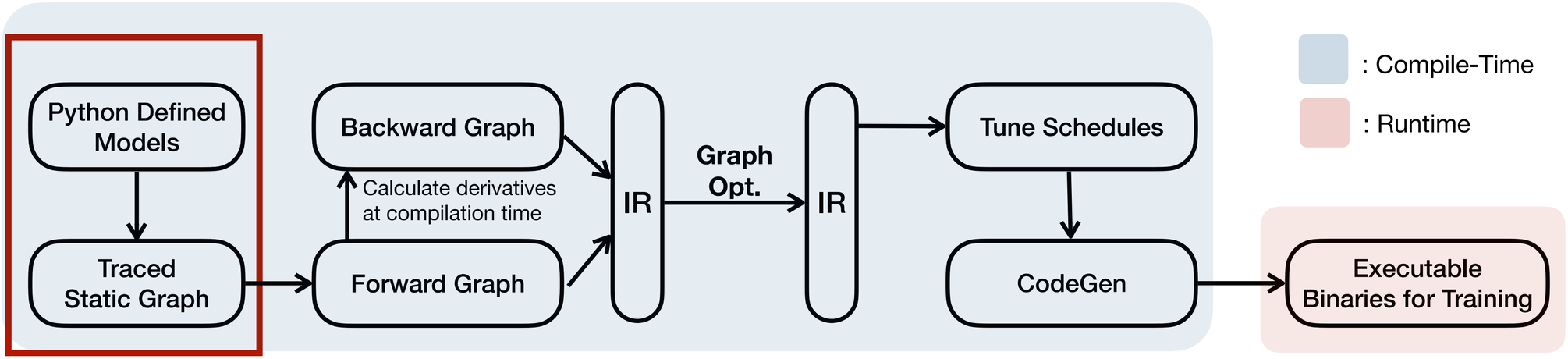
: Runtime

TTE moves most workload from runtime to **compile-time**,
thus minimizes the **runtime overhead**,
also enables opportunities for **extensive graph optimizations**.

Tiny Training Engine (TTE)



Tiny Training Engine (TTE)



Dynamic execution, activation tensor shape / data-type is obtained at run-time. Only layer information (kernel size) is known at compile-time.

Every tensor shape / data-type is obtained at compile-time.

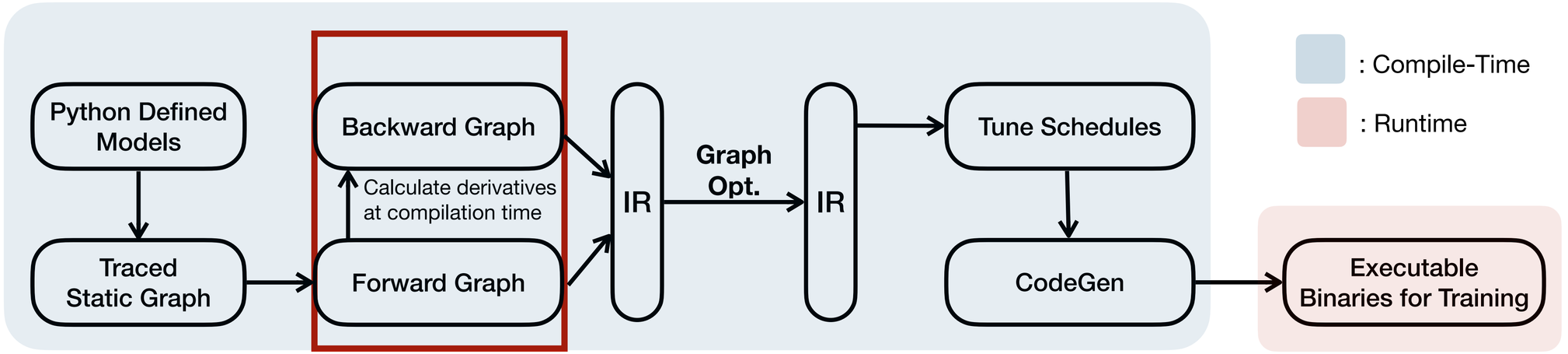
PyTorch

```
net = nn.Sequential(
    nn.Conv2d(3, 3, kernel=3, padding=1),
    nn.ReLU()
)

data = torch.randn(1, 3, 28, 28)
out = net(data)
```

Forward IR

```
fn (%input0: Tensor[(1, 3, 28, 28), float32],
%v0.weight: Tensor[(3, 3, 3, 3), float32]) {
  %0 = nn.conv2d(%input0, %v0.weight, padding=[1, 1,
1, 1], channels=3, kernel_size=[3, 3]);
  nn.relu(%0)
}
```

Tiny Training Engine (TTE)



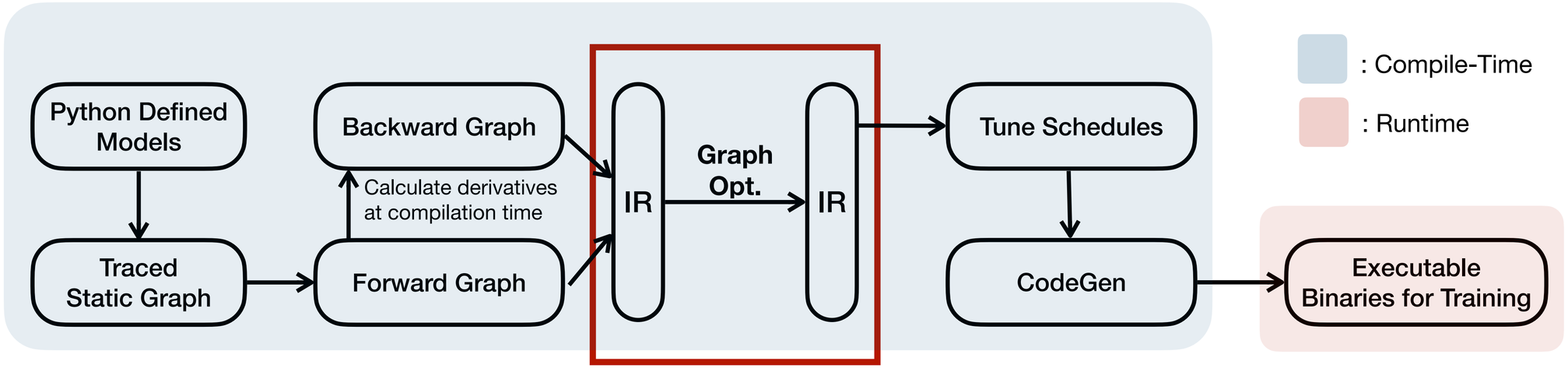
Forward IR

```
fn (%input0: Tensor[(1, 3, 28, 28), float32],
%v0.weight: Tensor[(3, 3, 3, 3), float32]) {
  %0 = nn.conv2d(%input0, %v0.weight,
padding=[1, 1, 1, 1], channels=3,
kernel_size=[3, 3]);
  %0
}
```

Backward IR

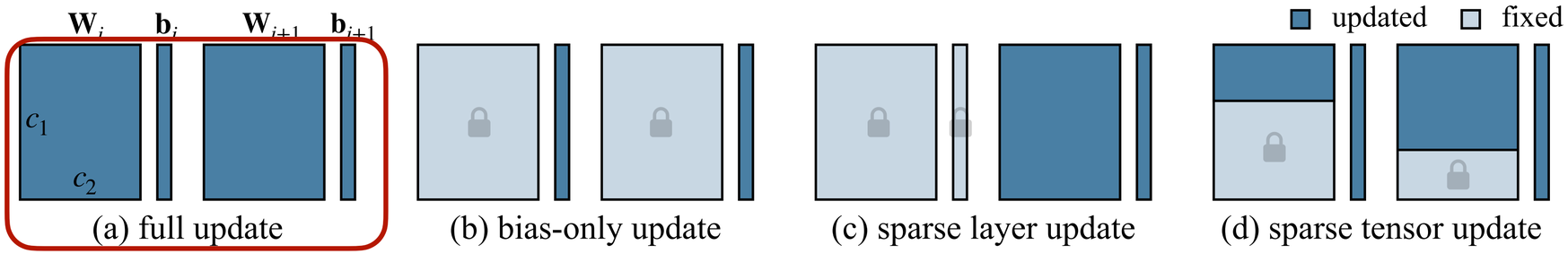
```
fn (%input0: Tensor[(1, 3, 28, 28), float32], %v0.weight: Tensor[(3,
3, 3, 3), float32], %grad_output: Tensor[(1, 3, 28, 28), float32]) {
  # forward
  %0 = nn.conv2d(%input0, %v0.weight, padding=[1, 1, 1, 1],
channels=3, kernel_size=[3, 3]);
  %1 = nn.relu(%0);
  # grad_input
  %2 = padding(%grad_output);
  %3 = nn.conv2d_transpose(%grad_output, %v0.weight, %2, padding=[1,
1, 1, 1], channels=3, kernel_size=[3, 3]);
  # grad_weight
  %4 = reshape_padding(%grad_output);
  %5 = nn.conv2d(%input0, %grad_output, padding=[1, 1, 1, 1],
channels=3, kernel_size=[3, 3]);
  % grad_bias
  %6 = sum(%grad_output, axis=[-1, -2]);
  (%3, %5, %6)
}
```

Tiny Training Engine (TTE)



- Graph-level optimizations:
  - Sparse layer / sparse tensor update
  - Operator reordering and in-place update
  - Constant folding
  - Dead-code elimination

Tiny Training Engine (TTE)



Example from a matrix multiplication with full update

Forward

$$y = XW + b$$

Backward

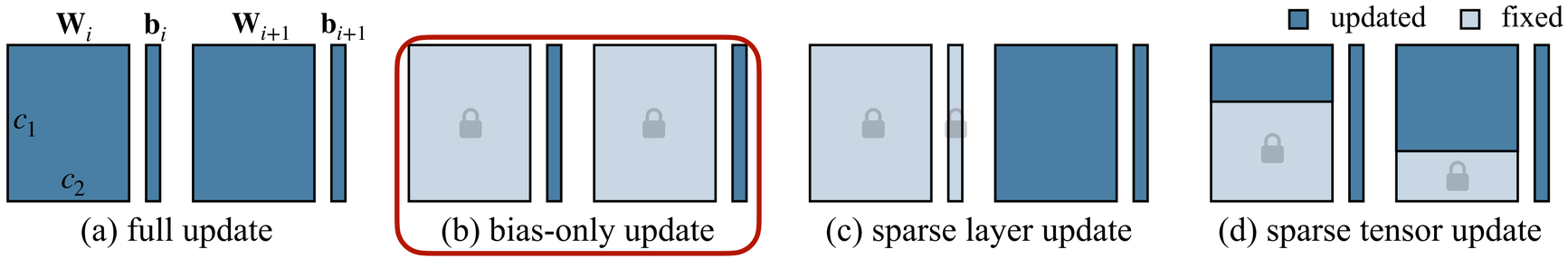
$$dy/dx = GW$$

$$dy/dw = G^T X$$

$$dy/db = sum(G)$$

```
fn (%x: Tensor[(10, 10), float32],
    %weight: Tensor[(10, 10), float32],
    %bias: Tensor[(10), float32]),
    %grad: Tensor[(10), float32]),
{
  # forward
  %0 = multiply(%x, %weight);
  %1 = add(%0, %bias);
  # backward
  %3 = multiply(%grad, %weight); ====> dy / dx
  %4 = transpose(%grad);
  %5 = multiply(%4, %x);         ====> dy / dw
  %6 = sum(%grad, axis=-1);      ====> dy / db
  (%3, %5, %6)
}
```

Tiny Training Engine (TTE)




```
fn (%x: Tensor[(10, 10), float32, needs_grad=True],
    %weight: Tensor[(10, 10), float32, needs_grad=False],
    %bias: Tensor[(10), float32, needs_grad=True],
    %grad: Tensor[(10), float32]),
{
  # forward
  %0 = multiply(%x, %weight);
  %1 = add(%0, %bias);
  # backward
  %3 = multiply(%grad, %weight); ====> dy / dx
  %4 = transpose(%grad);
  %5 = multiply(%4, %x);         ====> dy / dw
  %6 = sum(%grad, axis=-1);      ====> dy / db
  (%3, %5, %6)
}
```

Annotate whether a tensor requires gradient or not

Forward

$$y = XW + b$$

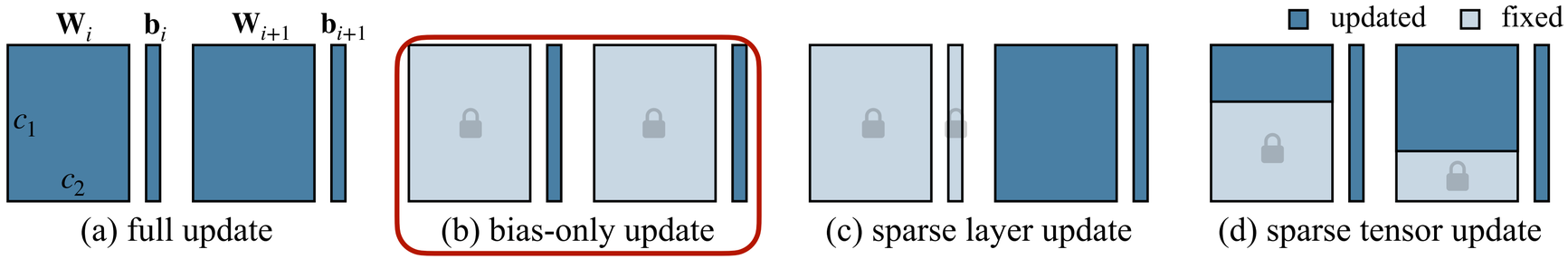
Backward

$$dy/dx = GW$$

$$dy/dw = G^T X$$

$$dy/db = sum(G)$$

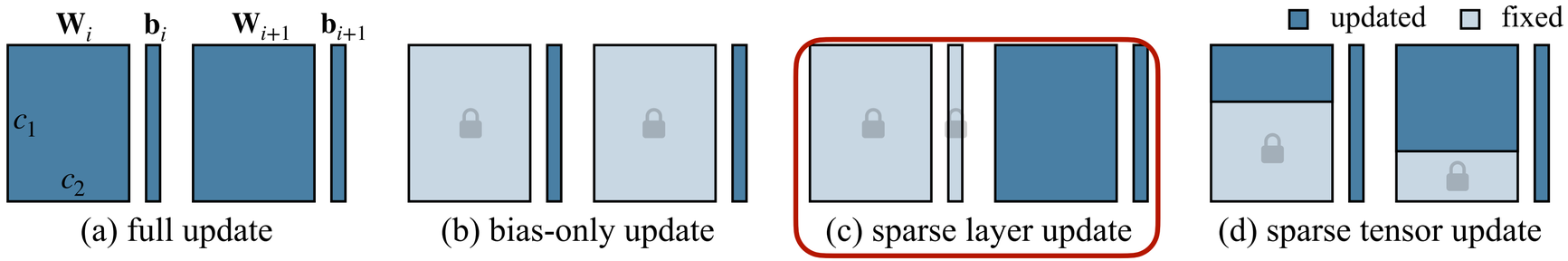
Tiny Training Engine (TTE)




```
fn (%x: Tensor[(10, 10), float32, needs_grad=True],
    %weight: Tensor[(10, 10), float32, needs_grad=False],
    %bias: Tensor[(10), float32, needs_grad=True],
    %grad: Tensor[(10), float32]),
{
  # forward
  %0 = multiply(%x, %weight);
  %1 = add(%0, %bias);
  # backward
  %3 = multiply(%grad, %weight); ====> dy / dx
  %4 = transpose(%grad);
  %5 = multiply(%4, %x);          ====> dy / dw
  %6 = sum(%grad, axis=-1);       ====> dy / db
  (%3, %5, %6)
}
```

Remove unnecessary computations from DAG via dependency analysis and dead-code elimination.

Tiny Training Engine (TTE)

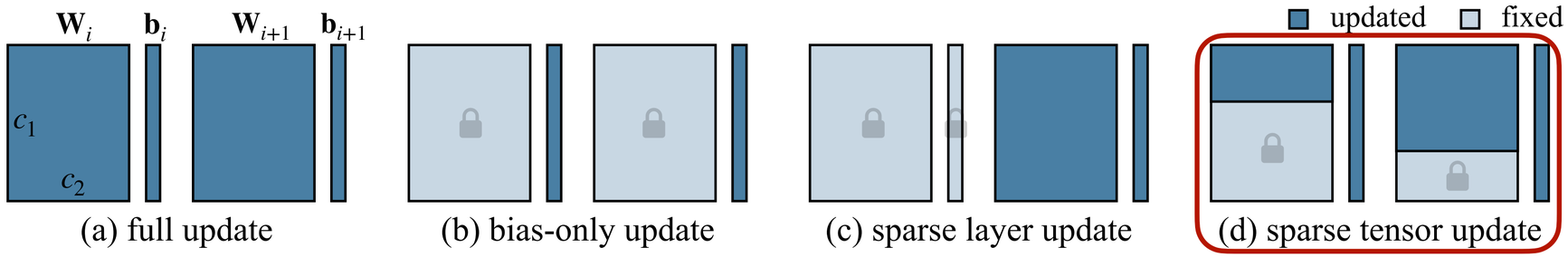



```
fn (%x: Tensor[(10, 10), float32, needs_grad=False],
    %weight1: needs_grad=False],
    %bias1: needs_grad=False],
    %weight2: needs_grad=True],
    %bias2: needs_grad=True],
    ..........
    %grad: .., float32]),
{
  # ...
}
```

Freely annotate **ANY** parameters

TTE will trim the computation accordingly.

Tiny Training Engine (TTE)



(a) full update (b) bias-only update (c) sparse layer update (d) sparse tensor update

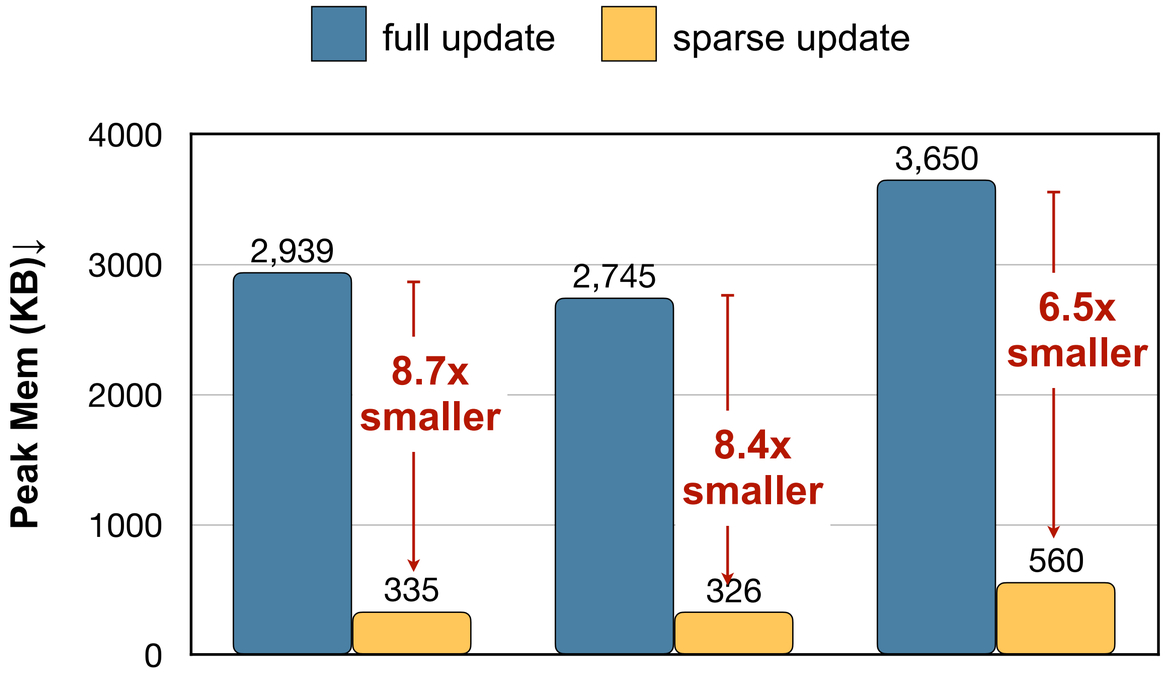
```
fn (%x: Tensor[(10, 10), float32],
    %weight: Tensor[(10, 10), float32],
    %bias: Tensor[(10), float32]),
    %grad: Tensor[(10), float32]),
{
  # forward
  %0 = multiply(%x, %weight);
  %1 = add(%0, %bias);
  # backward
  %3 = multiply(%grad, %weight);
  %4 = transpose(%grad)
  %5 = multiply(%4, %x);
  %6 = sum(%grad, axis=-1);
  (%3, %5, %6)
}
```

Automatically remove the buffers of pruned gradients from the computation graph.

```
fn (%x: Tensor[(10, 10), float32, needs_grad=True],
    %weight: Tensor[(20, 10), float32, needs_grad=0.5],
    %bias: Tensor[(20), float32, needs_grad=True],
    %grad: Tensor[(10, 20), float32]),
{
  # forward
  %0 = multiply(%x, %weight);
  %0.1 = slice(%x, begin=[0, 0], ends=[10, 10]);
  %1 = add(%0, %bias);
  # backward
  %3 = multiply(%grad, %weight);
  %4 = transpose(%grad)
  %5 = multiply(%4, %0.1);
  %6 = sum(%grad, axis=-1);
  (%3, %5, %6)
}
```

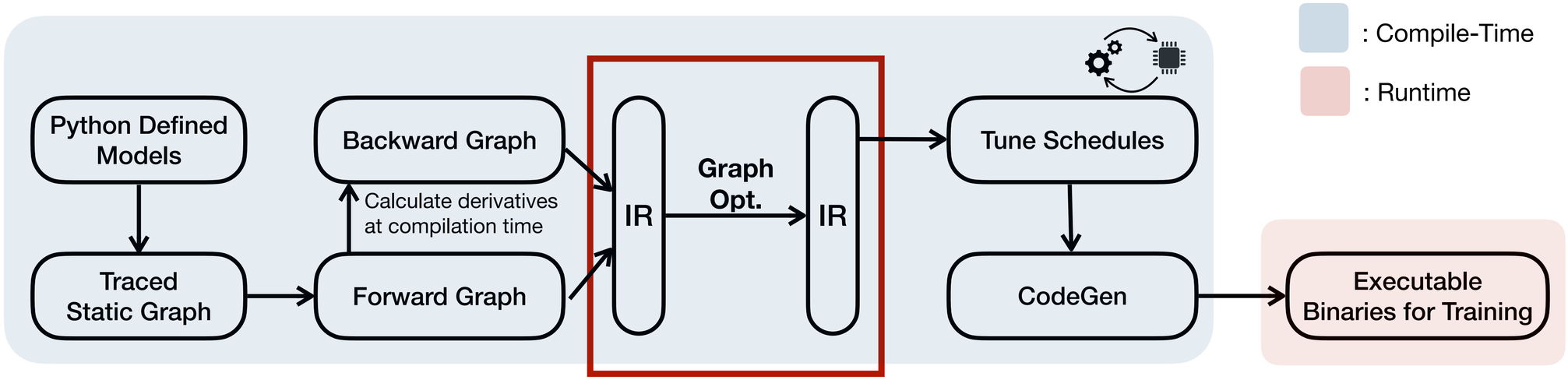
Tiny Training Engine (TTE)

Sparse update results



- Tiny Training Engine supports backward graph pruning and sparse update at IR-level.
- After graph pruning, un-used weights and sub-tensors are pruned from DAG => 6.5-8.7x memory saving

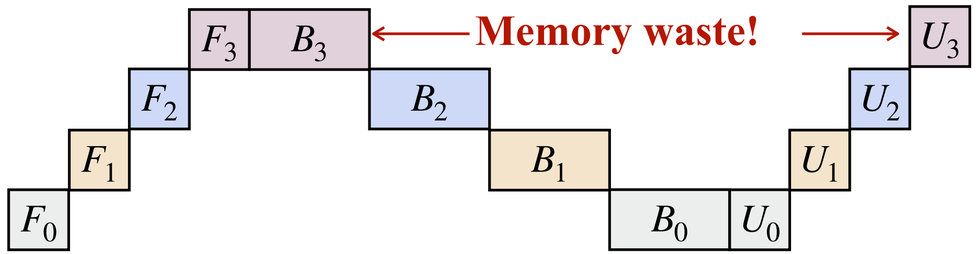
Tiny Training Engine (TTE)



- Graph-level optimizations:
  - Sparse layer / sparse tensor update
  - Operator reordering and in-place update
  - Constant folding
  - Dead-code elimination

Tiny Training Engine (TTE)

Re-ordering reduces memory footprint



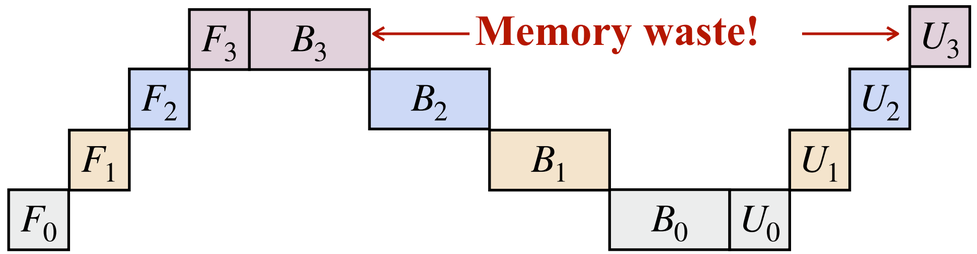
(a) Conventional way to update parameters

Operator life-cycle analysis reveals the **memory redundancy** in the optimization step.

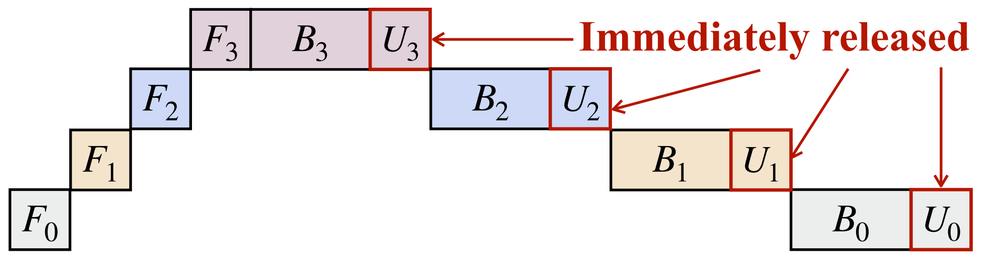
F: Forward, *B*: Backward, *U*: Update

Tiny Training Engine (TTE)

Re-ordering reduces memory footprint



(a) Conventional way to update parameters



(b) Operator re-ordering

Operator life-cycle analysis reveals the **memory redundancy** in the optimization step.

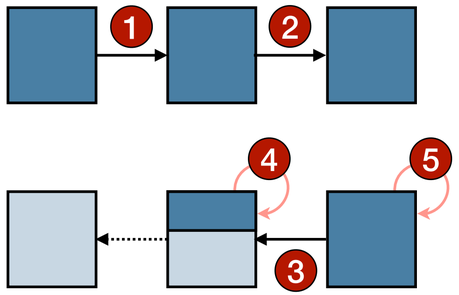
After re-ordering, the **redundant memory usage is eliminated** from training.

F: Forward, *B*: Backward, *U*: Update

Tiny Training Engine (TTE)

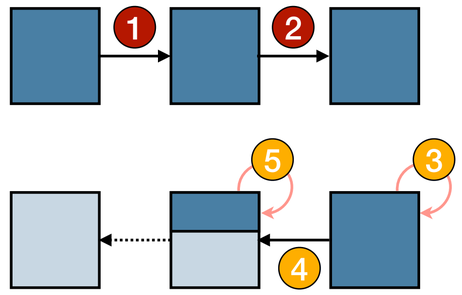
Re-ordering reduces memory footprint

```
out = model(data)
loss = criterion(out, label)
gradients = loss.backward()
optim.update(model, gradients)
```



Calculate **all gradients first**,
then **apply one-by-one**.
Intermediate buffers consume a lot of spaces.

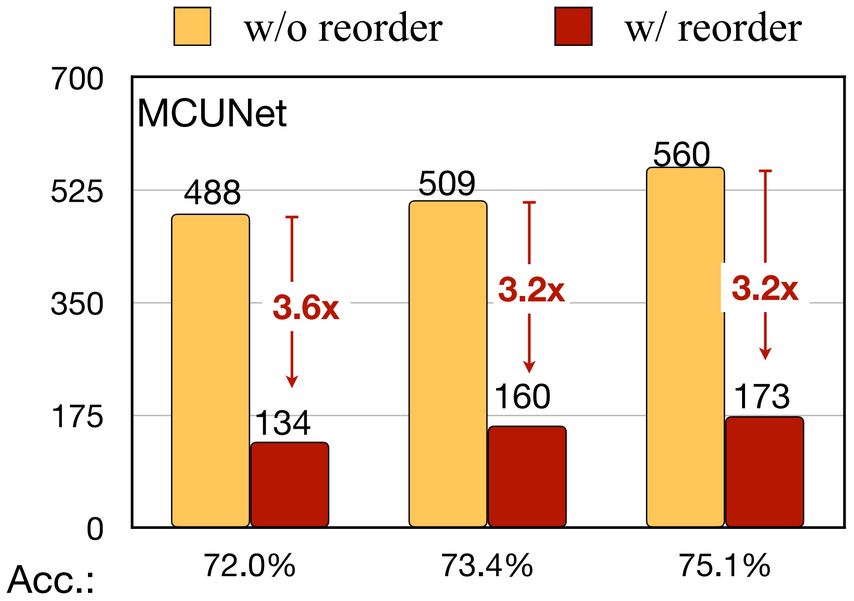
```
out = model(data)
loss = criterion(out, label)
for layers in model:
    dydx, grad = layers.backward(loss)
    optim.update(layers, grad)
    loss = dydx
```



Gradient updates are **immediately applied** once calculated.
Intermediate buffers can be released.

Tiny Training Engine (TTE)

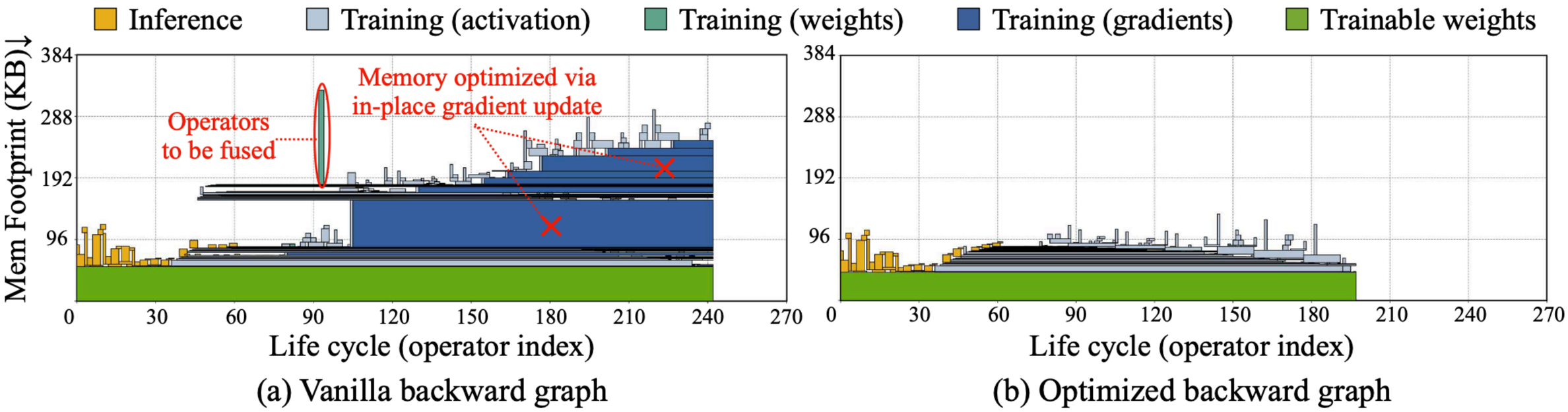
Re-ordering reduces memory footprint



By reordering, the gradient update can be immediately applied.Gradients buffer can be released earlier before before back-propagating to earlier layers, leading to **2.7x ~ 3.1x** peak memory reduction.

Tiny Training Engine (TTE)

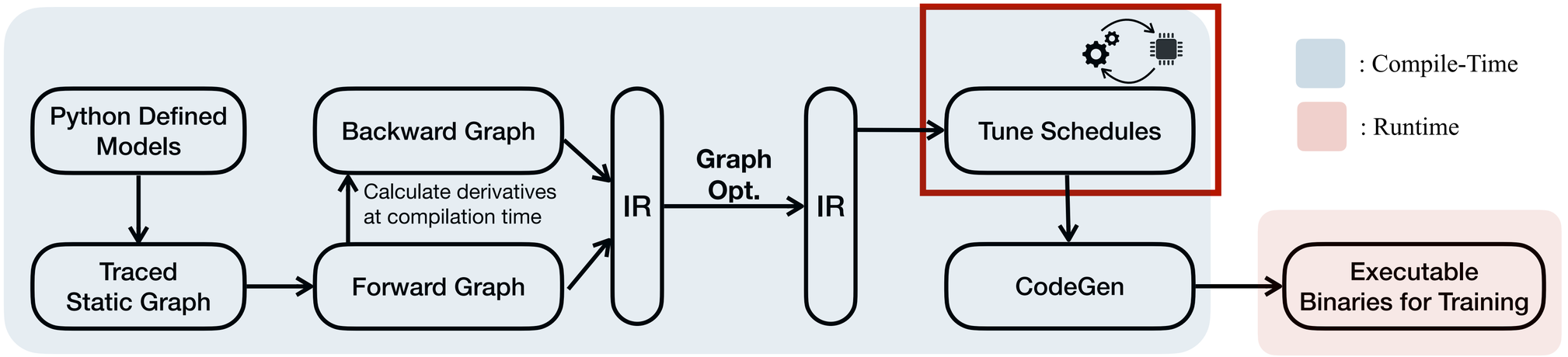
Re-ordering reduces memory footprint

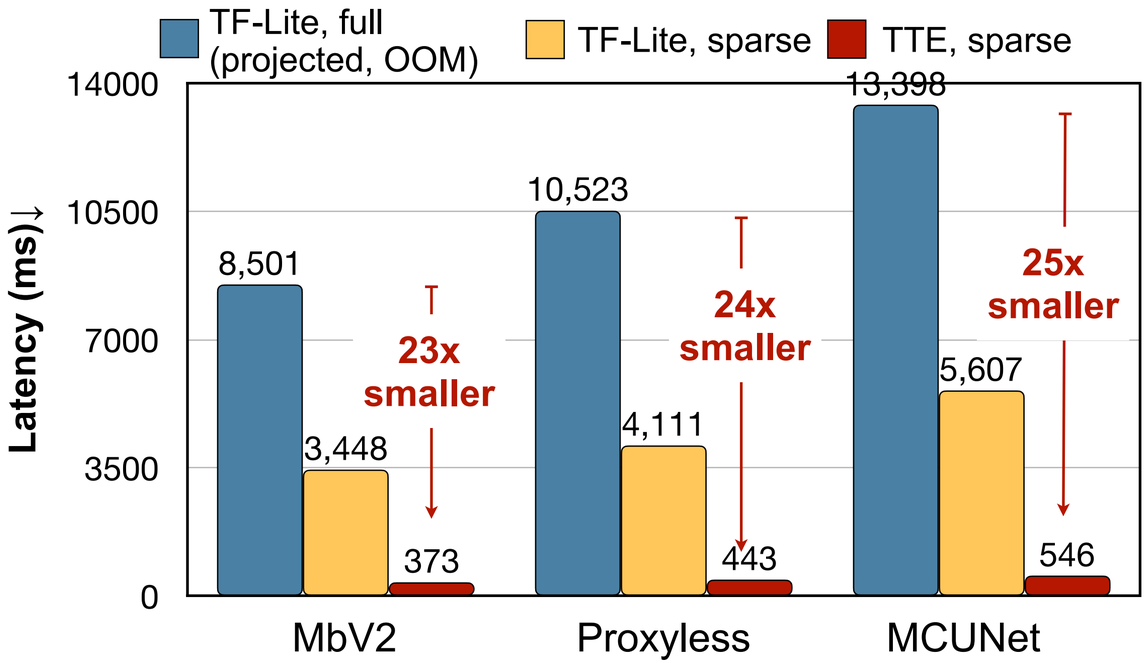


Operator Fusion: perform tiling/reshape operators in-place with convolution

Operator life-cycle analysis shows memory footprint can be greatly reduced by operator re-ordering.

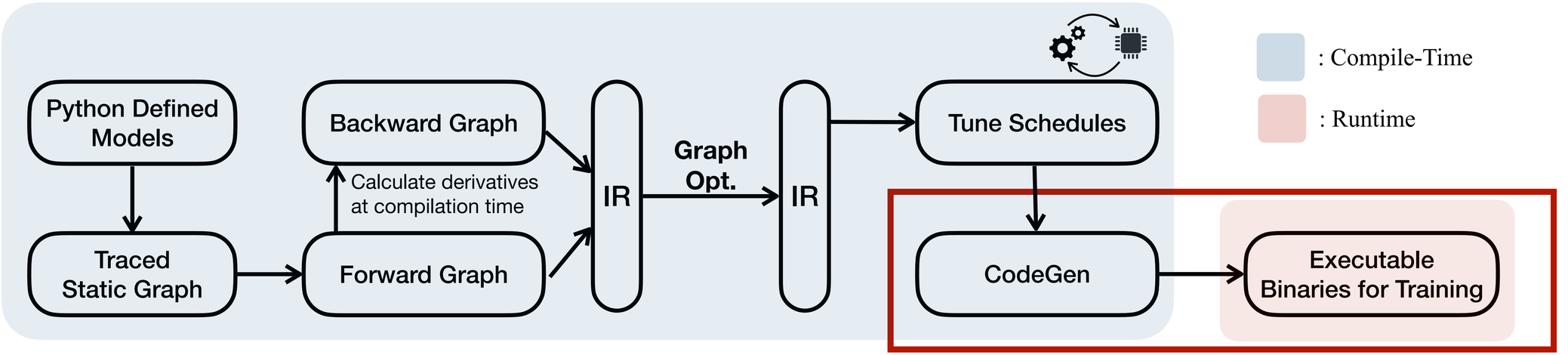
Tiny Training Engine (TTE)





Our optimized operators demonstrate **23x ~ 25x** speedup over TensorFlow-Lite.

Tiny Training Engine (TTE)



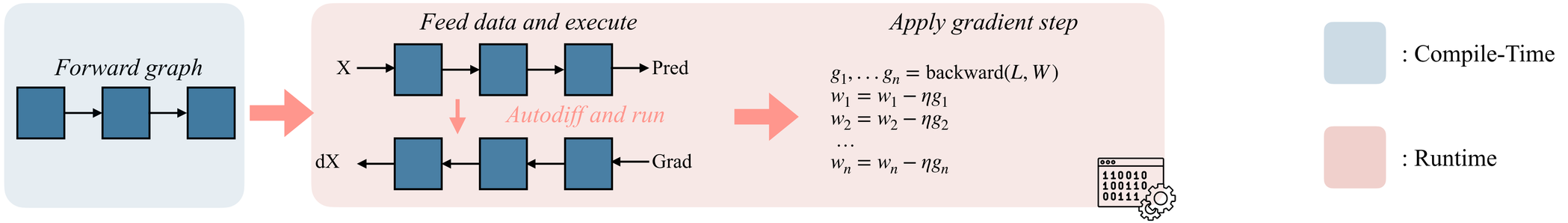

```
runtime::Module fwd_mod = runtime::Module::LoadFromFile("fwd.so");
runtime::Module bwd_mod = runtime::Module::LoadFromFile("bwd.so");

auto data = tensor::randn(1, 3, 128, 128);
auto out = fwd_mod(data);
auto gradients = bwd_mod(data);
```

TTE only generate binaries for the operators that's used by the workload

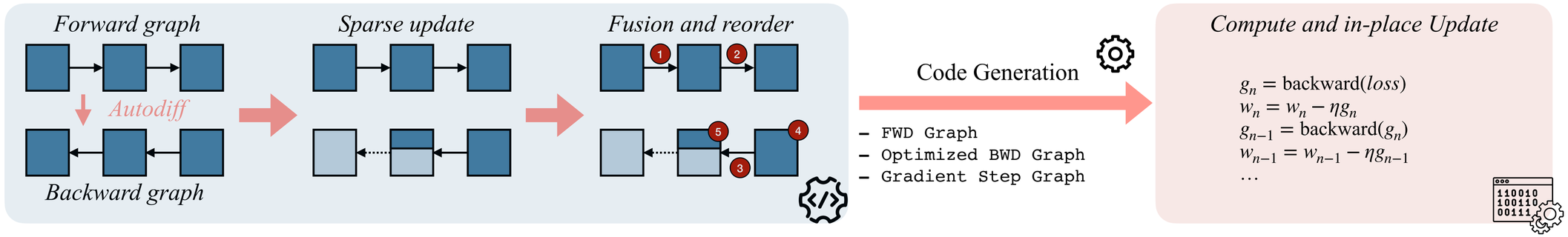
TTE delivers a light-weight, portable, and efficient binary.

Tiny Training Engine (TTE)

Comparison of Previous Infra and TTE



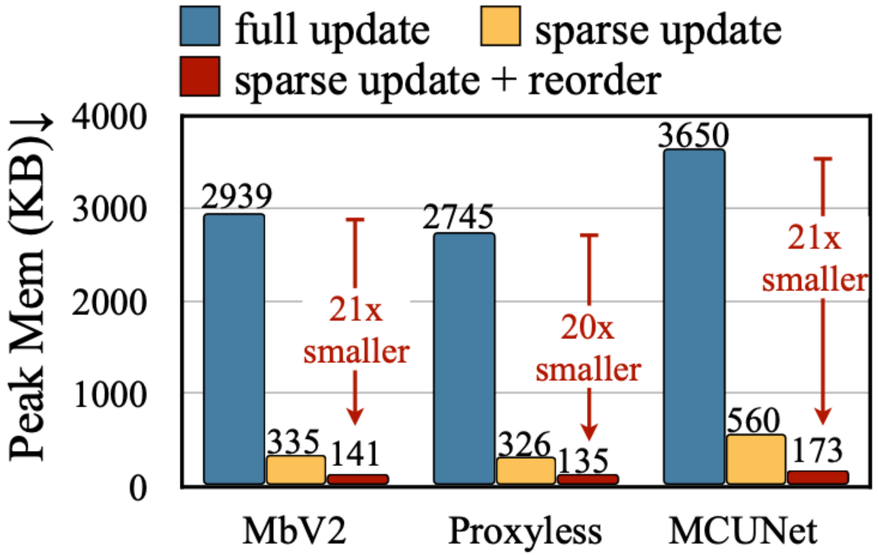
Conventional training framework performs most tasks at runtime.



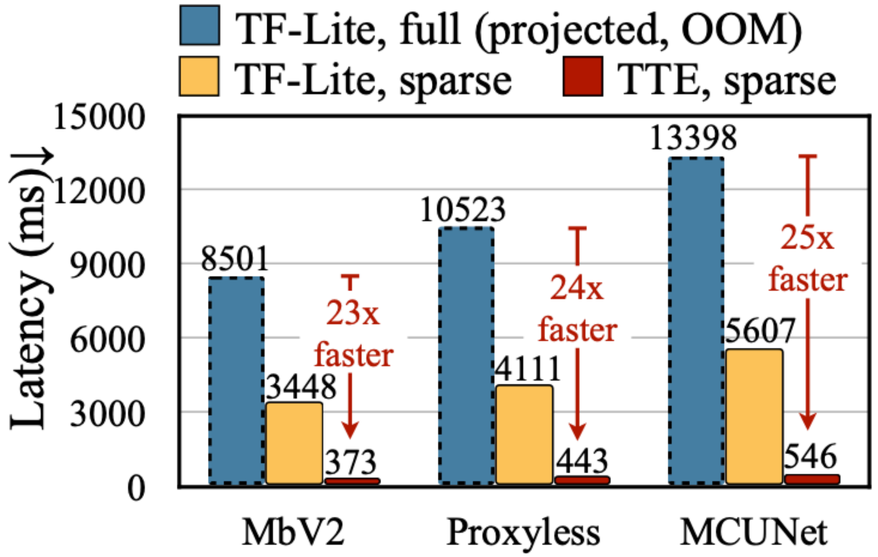
Tiny Training Engine (ours) **separate** the environment of runtime and compile time.

Tiny Training Engine (TTE)

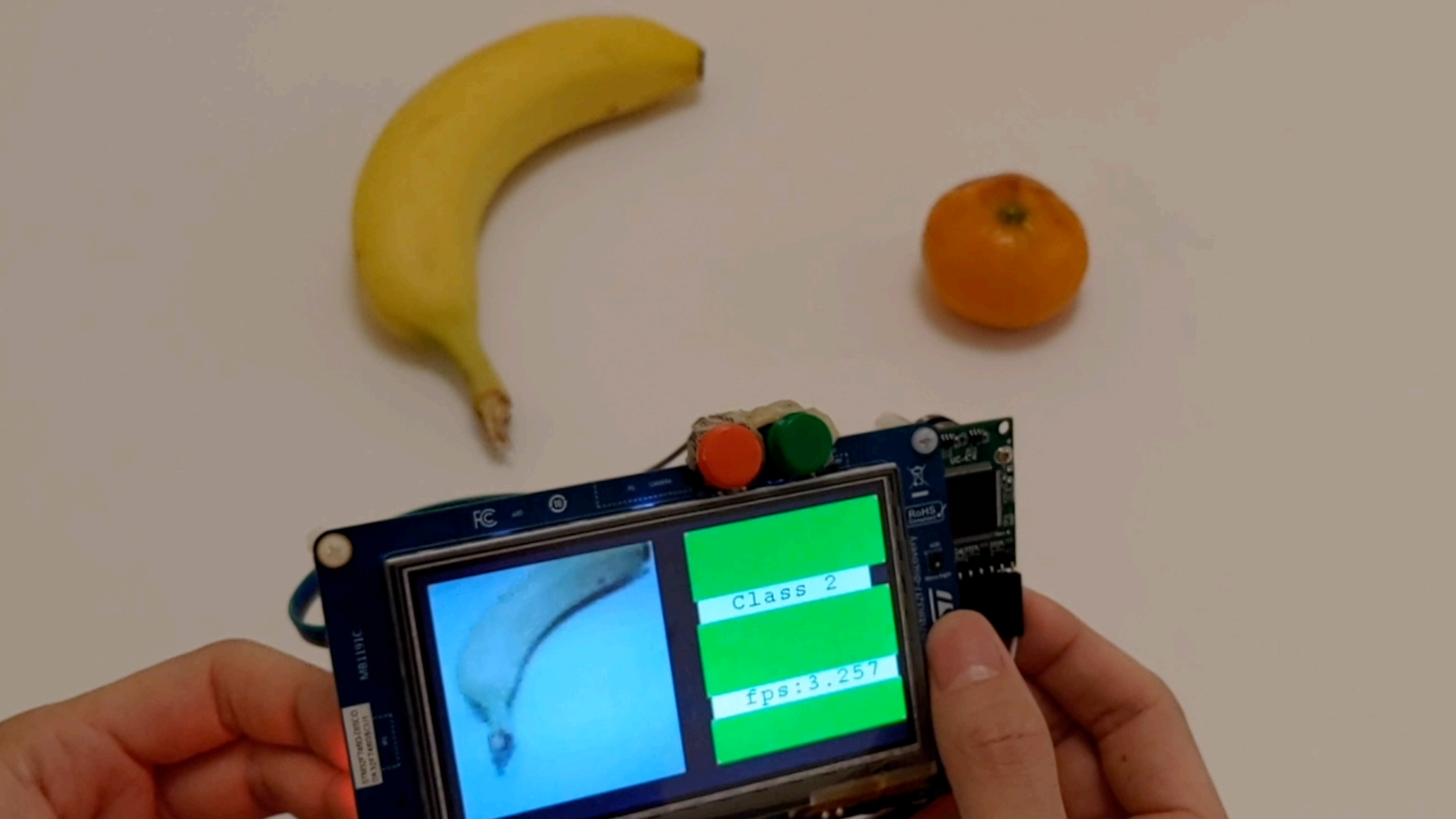
Smaller memory usage, faster training speed



20x smaller memory



23x faster speed

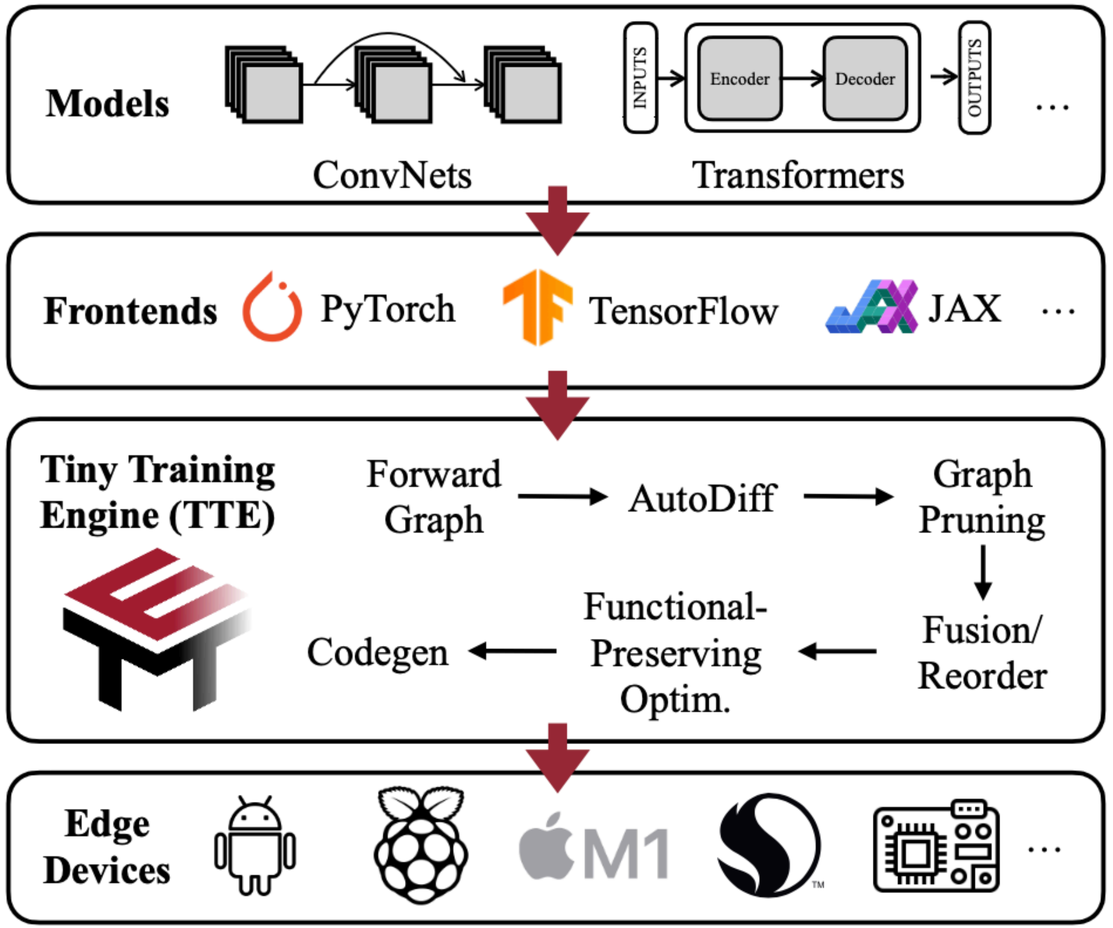

Class 2
fps:3.257

2. On-device training

Extending TTE to More Platforms

Enable on-device training on diverse edge hardware

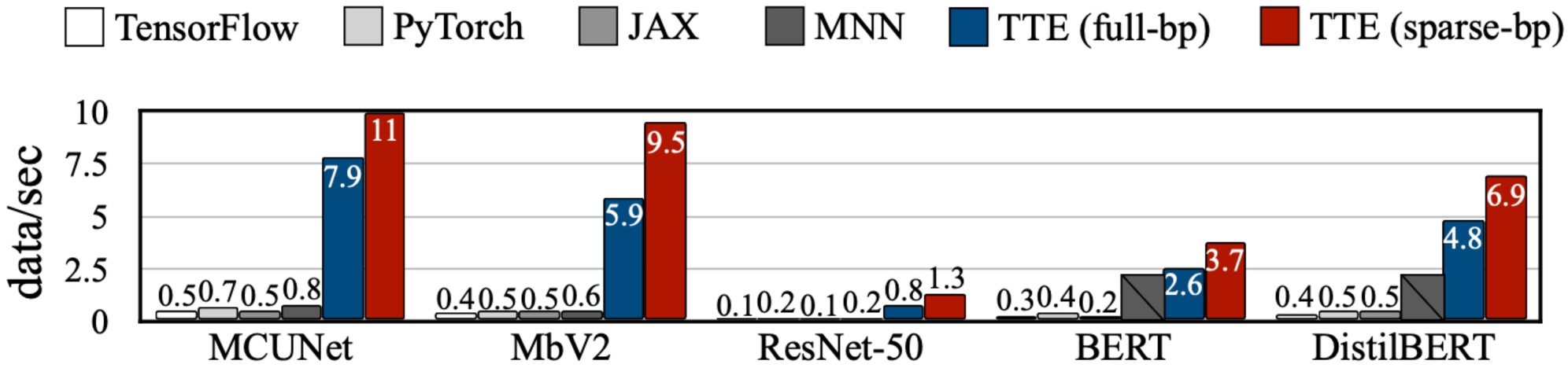
- We extend TTE to support:
  - Diverse models (CNN + Transformers)
  - Diverse frontends
    - PyTorch
    - TensorFlow
    - Jax
  - Diverse hardware backends
    - Apple M1
    - Raspberry Pi
    - Smartphones
    - …



Extending TTE to More Platforms

Consistently speed up training on diverse platforms

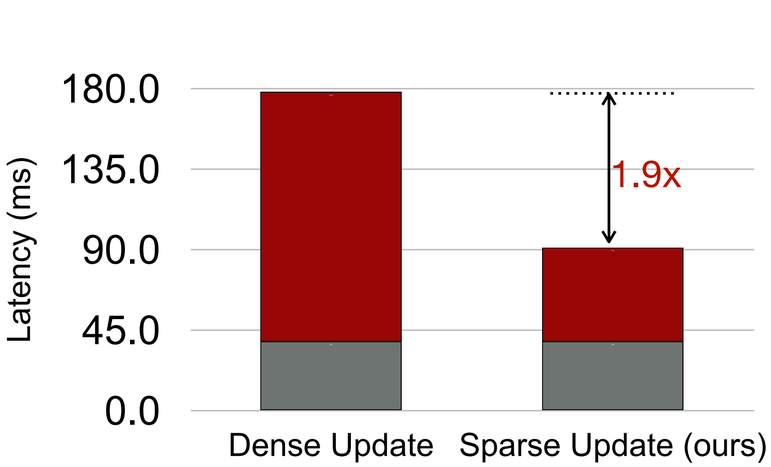
- TTE provides a systematic support for sparse update schemes for vision and NLP models, leading to consistent memory saving at the same training accuracy



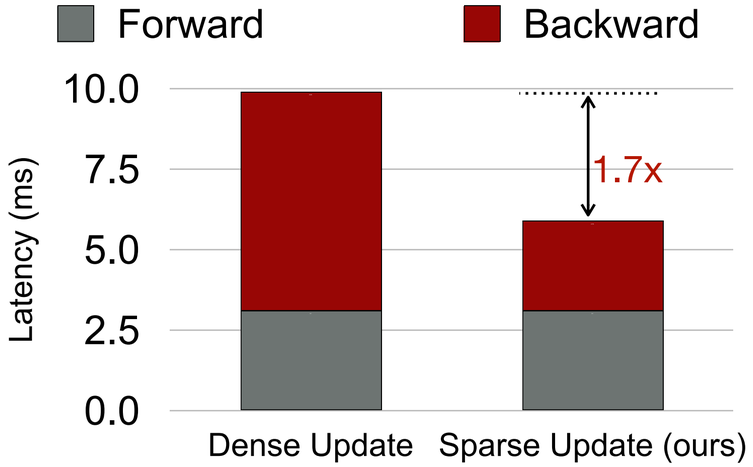
Results measured on Raspberry Pi 4B+.

Extending TTE to More Platforms

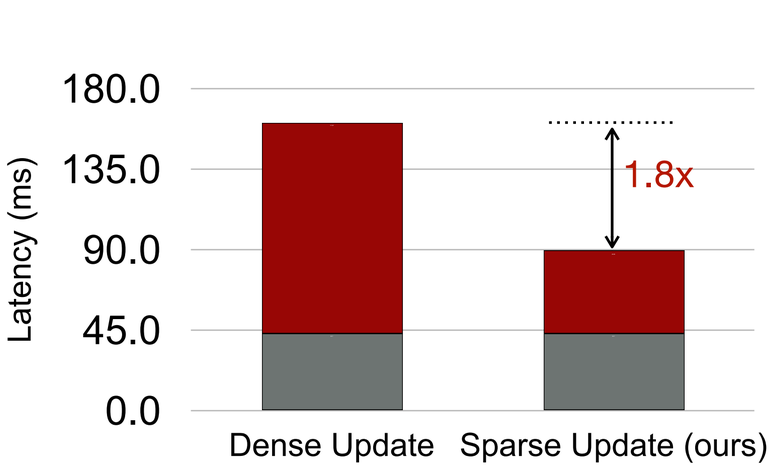
Consistently speed up training with sparse update



Qualcomm S8Gen1 CPU



Jetson Nano GPU



Raspberry Pi 4B+ CPU

The measured timed includes the **complete forward + backward**.

The benchmark model is MobilenetV2 with input resolution 224x224.

Our engine **supports various platforms** and our sparse update shows consistent speedup **1.7 to 1.9x**.

On-Device training was highlighted on MIT home page

Massachusetts Institute of Technology

Education Research Innovation Admissions + Aid Campus Life News Alumni About MIT

Explore websites, people, and locations

amy finkelstein

Top resources for

_ prospective students
_ current students
_ faculty & staff
_ alumni
_ parents
_ Covid-19 and MIT
_ all resources

Join us in building a better world.

A new technique enables AI models to continually learn from new data on intelligent “edge devices” like smartphones and sensors, reducing energy costs and privacy risks. The advance “makes deep learning more accessible,” says Song Han.

Oct 4, 2022 Full story Share: Explore more spotlights

On-Device Training Take-home

- **Quantized Training**
  - Fake quantization does not save memory
  - Real quantized graph achieves saving, however, is hard to optimize.
  - With quantization-aware scaling (QAS), we can optimize quantized graph.
- **Sparse Update**
  - Sparse learning happens in human brain
  - Update only important layers and parameters to save memory.
- **System Support**
  - Move workloads to compile-time (such as auto-diff) to minimize runtime cost.
  - Optimized schedules and kernels to improve throughput.

Section 2: Privacy Leakage in Federated Learning

The background of federated learning and why it may not be safe.

Background of Federated Learning

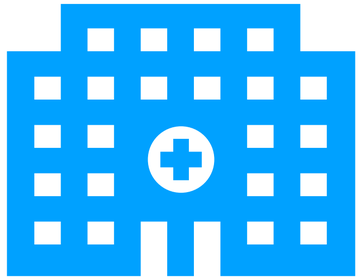
- Customization
  - I.e., Different users will have a different accent for speech recognition

Amazon
Alexa

Apple
Home Pod

Google
Home

- Security
  - Data cannot leave device because of security and regularization.



Background of Federated Learning

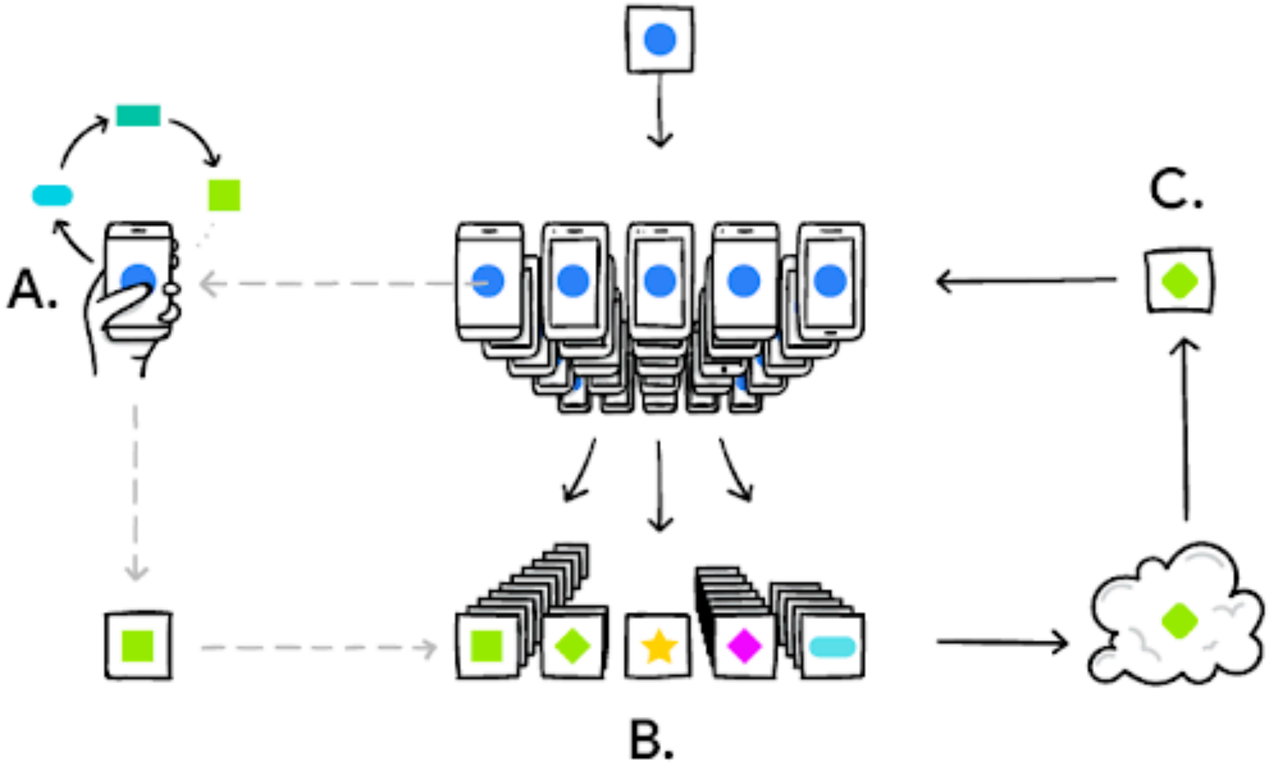
- Why we need federated learning?
  - Because of data.

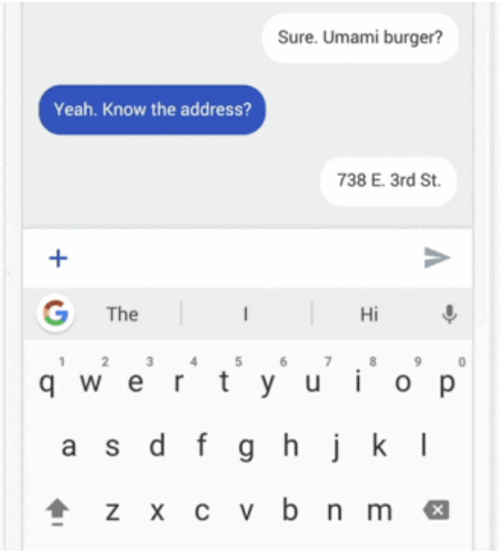
- Situation of data
  - Data is everywhere (sensors, devices, databases)
  - But isolated (due to privacy and legal issues)

- Federated learning provides a way to utilize them without centralization.

Background of Federated Learning

FedAvg Algorithm

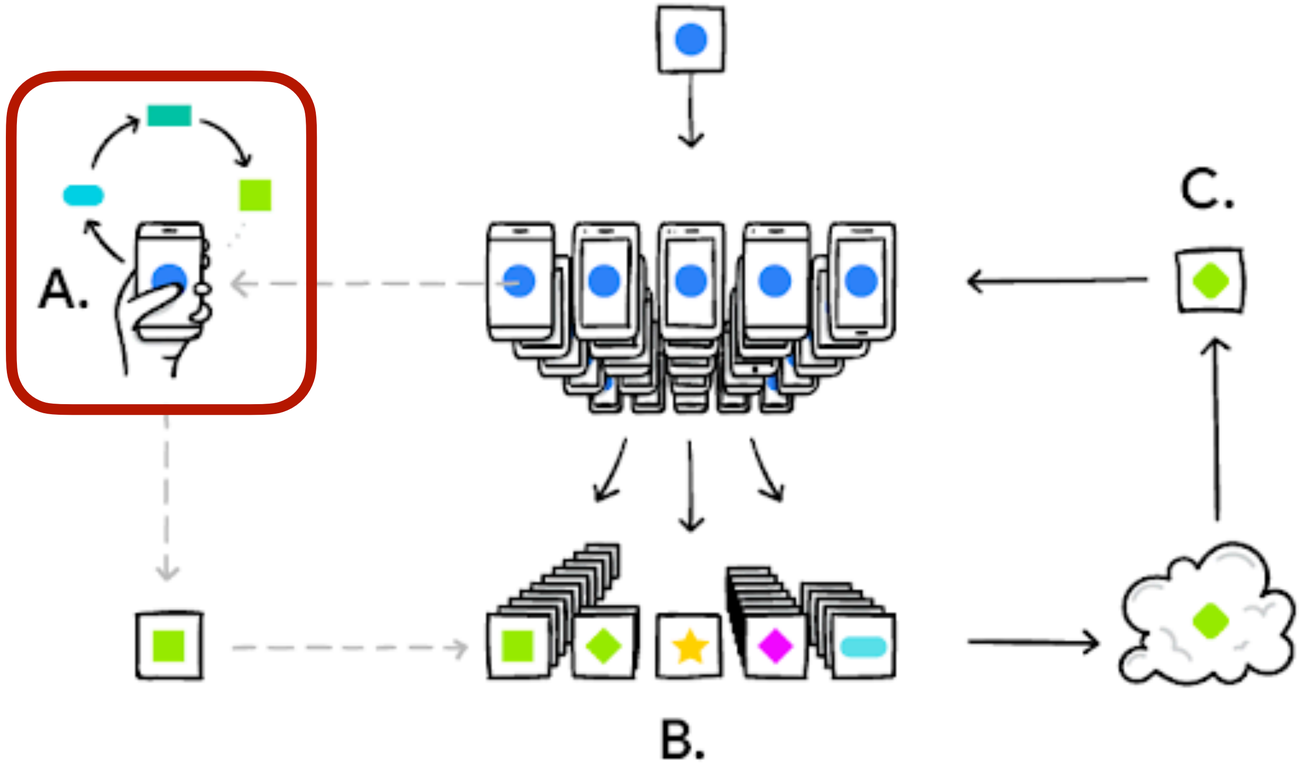




Communication-Efficient Learning of Deep Networks from Decentralized Data. [McMahan, 2016]

Background of Federated Learning

FedAvg Algorithm

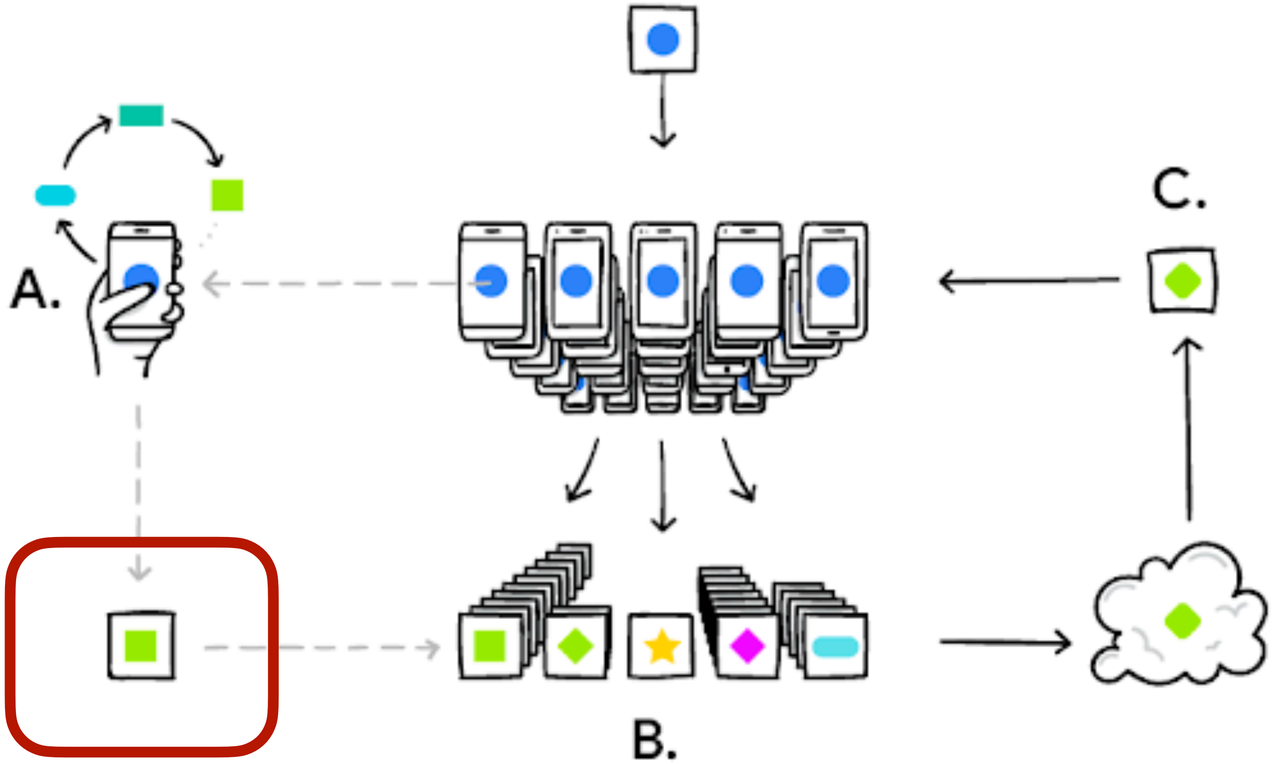


1. Users generate personal data on device and perform local training.

Communication-Efficient Learning of Deep Networks from Decentralized Data. [McMahan, 2016]

Background of Federated Learning

FedAvg Algorithm

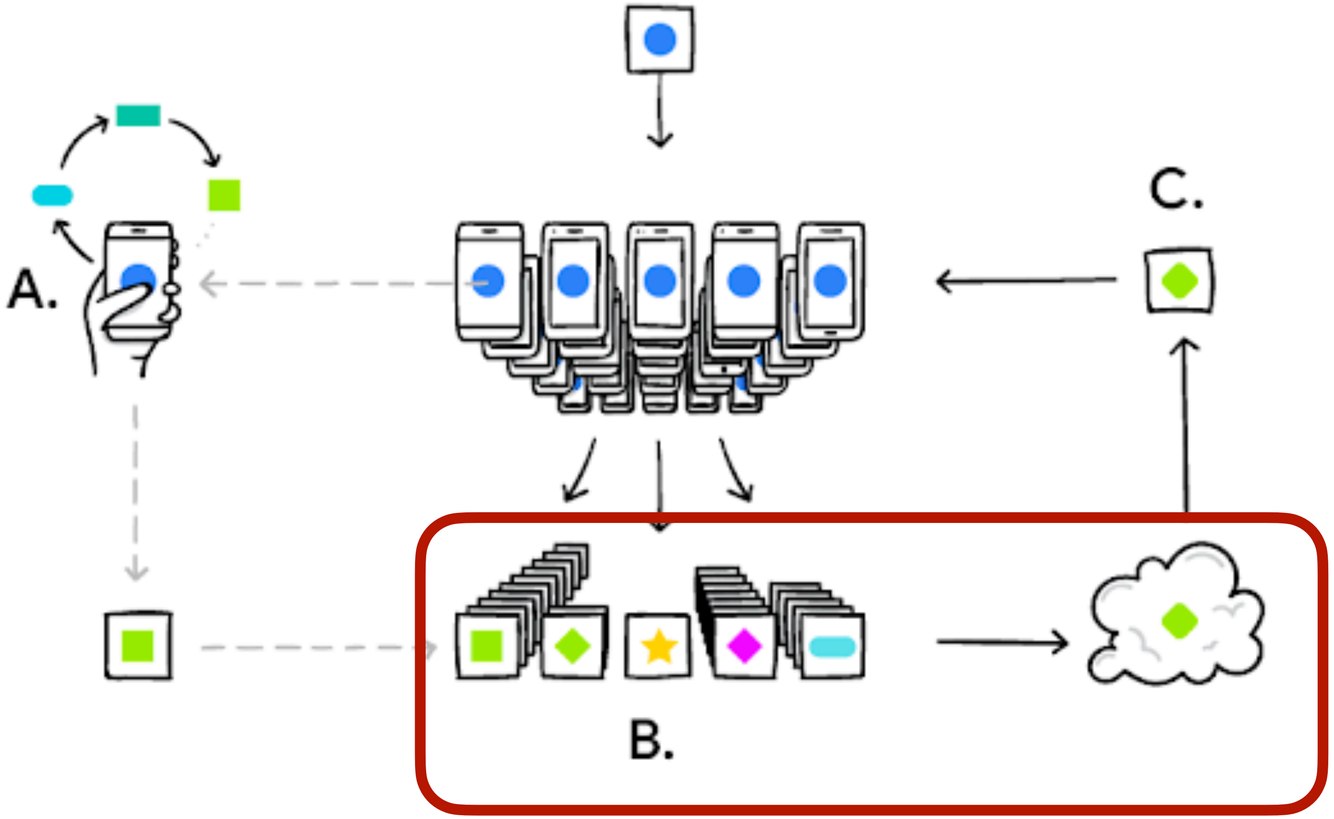


1. Users generate personal data on device and perform local training.
2. Each device update its model using local data for N iterations.

Communication-Efficient Learning of Deep Networks from Decentralized Data. [McMahan, 2016]

Background of Federated Learning

FedAvg Algorithm

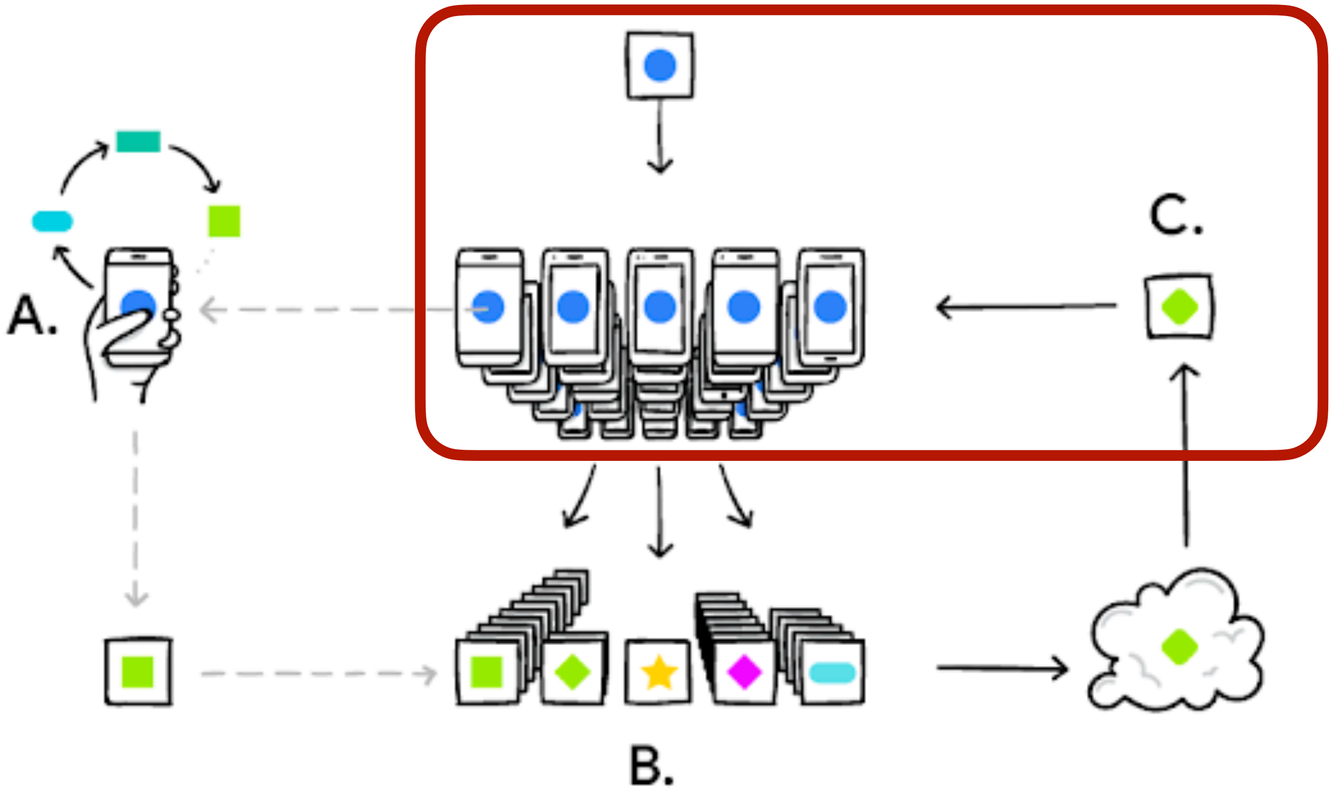


1. Users generate personal data on device and perform local training.
2. Each device update its model using local data for N iterations.
3. Updated models are sent to the server.

Communication-Efficient Learning of Deep Networks from Decentralized Data. [McMahan, 2016]

Background of Federated Learning

FedAvg Algorithm

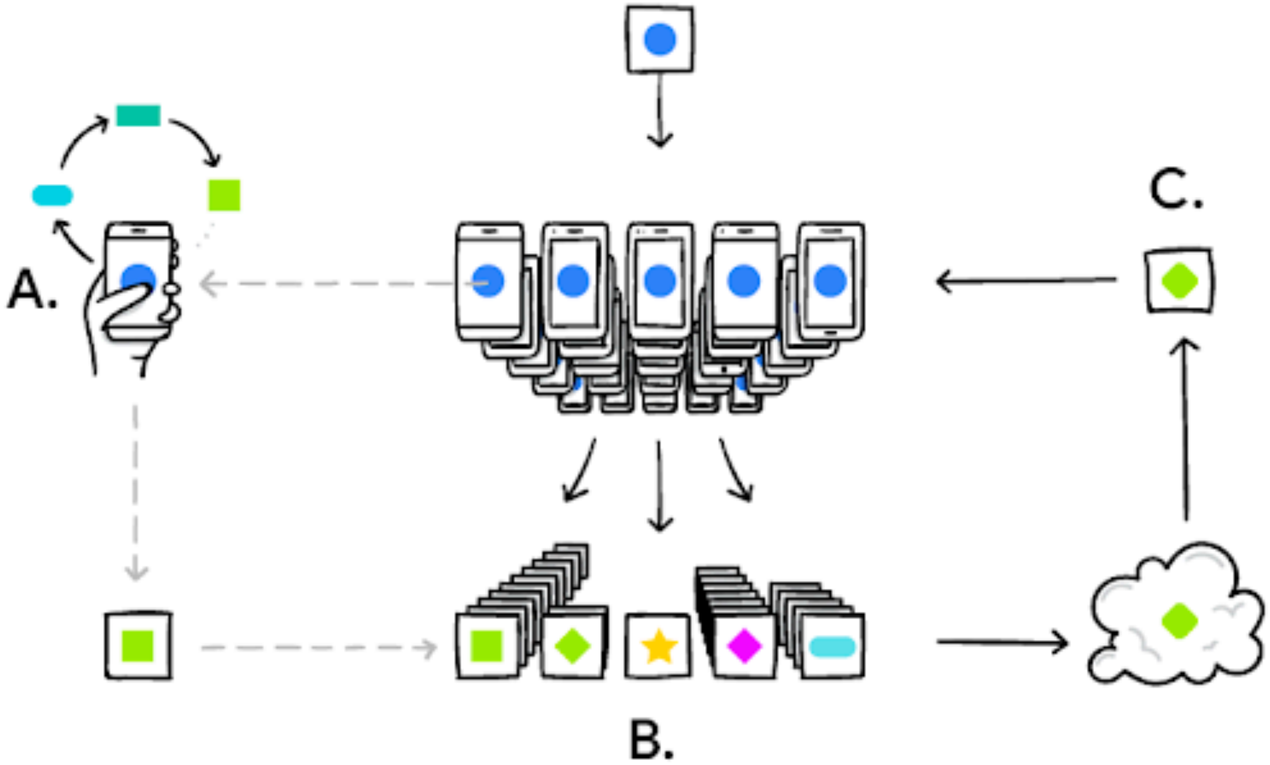


1. Users generate personal data on device and perform local training.
2. Each device update its model using local data for N iterations.
3. Updated models are sent to the server.
4. Models are averaged on the server and sent back to devices.

Communication-Efficient Learning of Deep Networks from Decentralized Data. [McMahan, 2016]

Background of Federated Learning

FedAvg Algorithm



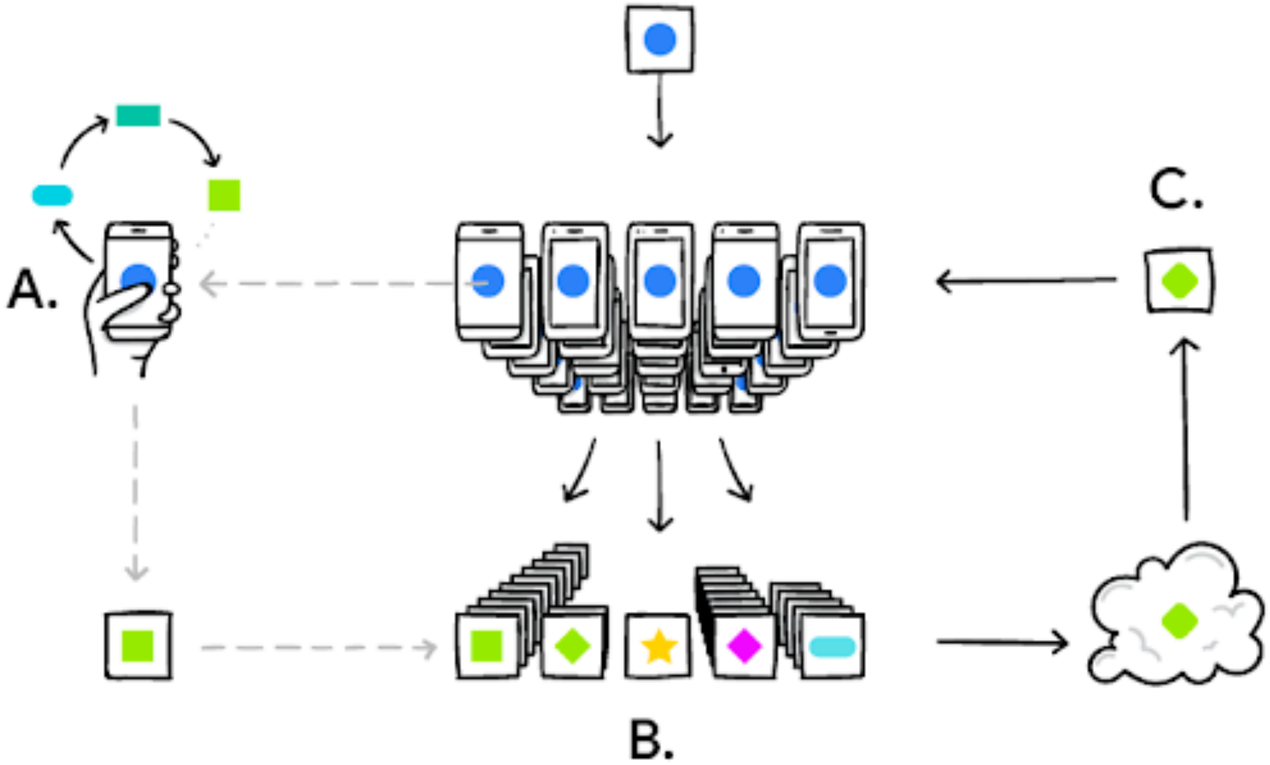
1. Users generate personal data on device and perform local training.
2. Each device update its model using local data for N iterations.
3. Updated models are sent to the server.
4. Models are averaged on the server and sent back to devices.

The important & private user data **NEVER** leaves local devices.

Communication-Efficient Learning of Deep Networks from Decentralized Data. [McMahan, 2016]

Background of Federated Learning

FedAvg Algorithm



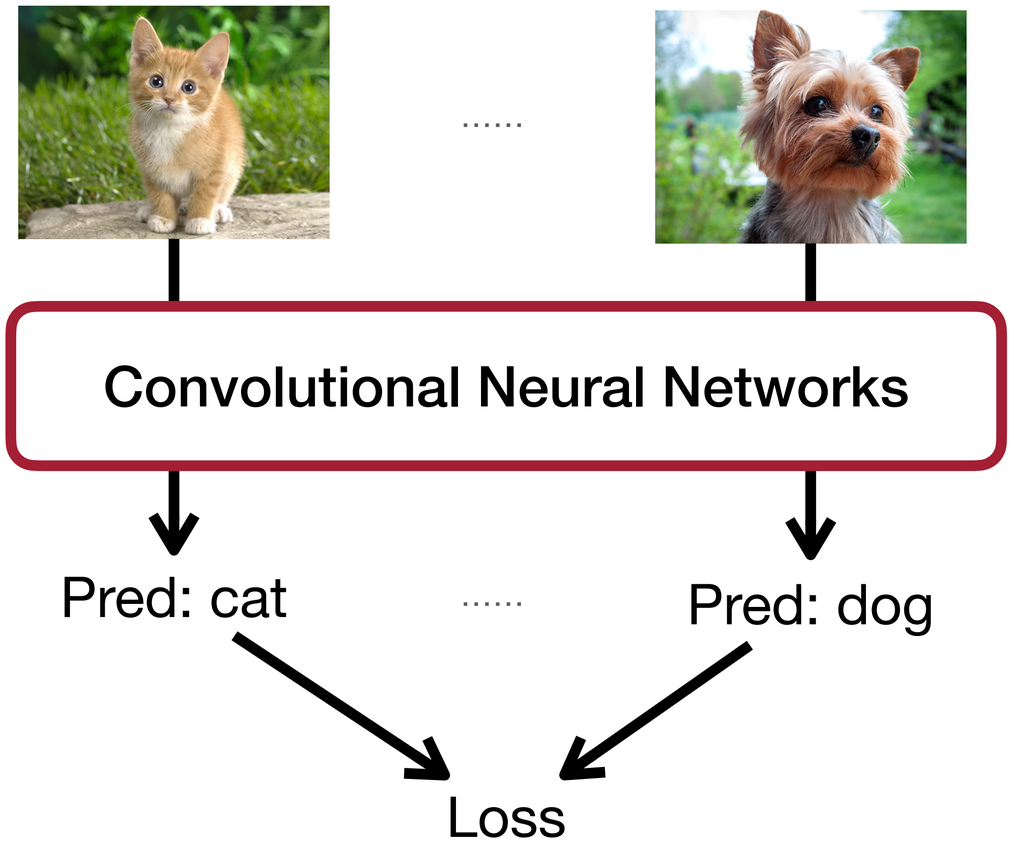
1. Users generate personal data on device and perform local training.
2. Each device update its model using local data for N iterations.
3. Updated models are sent to the server.
4. Models are averaged on the server and sent back to devices.

The important & private user data **NEVER** leaves local devices.

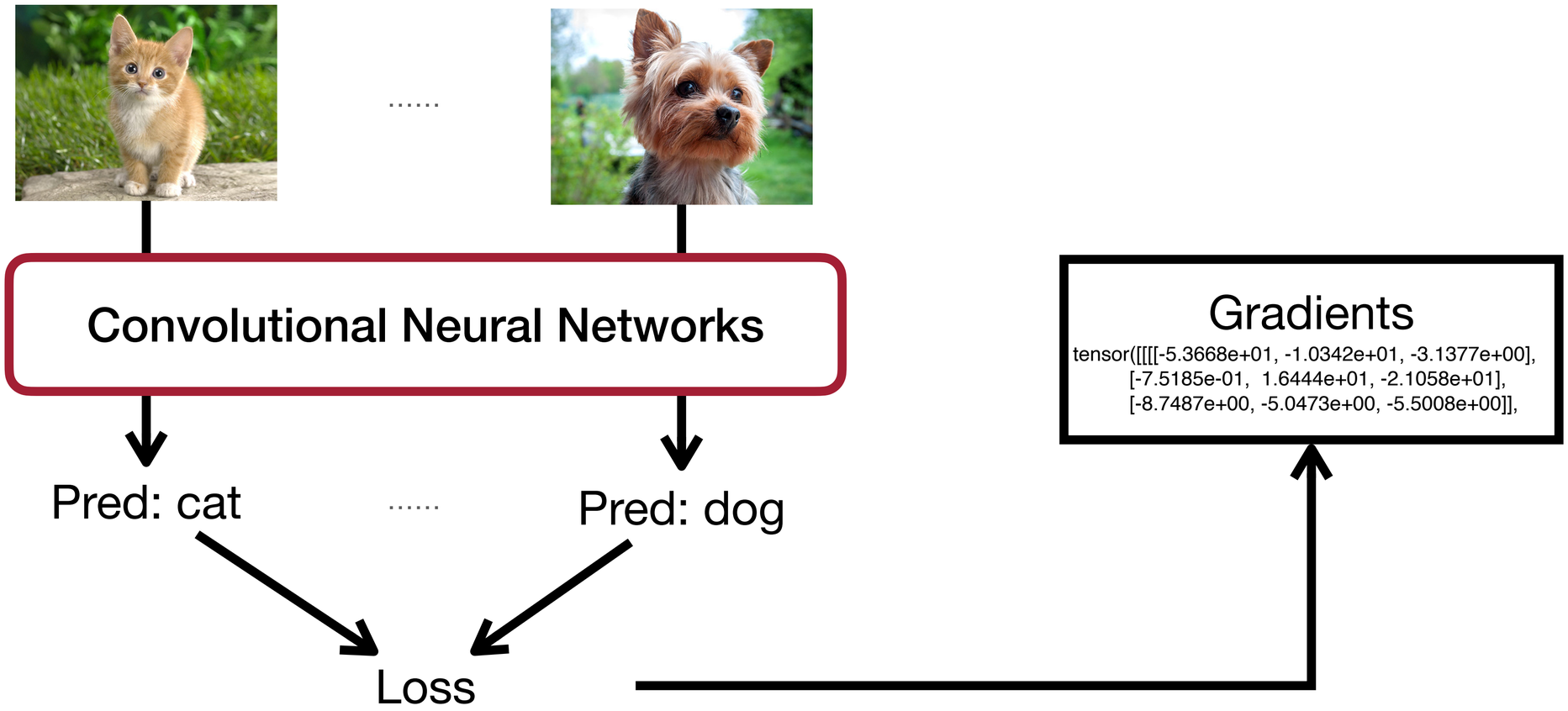
Does this **really** protect user information?

Communication-Efficient Learning of Deep Networks from Decentralized Data. [McMahan, 2016]

Rethink the Safety of Gradients

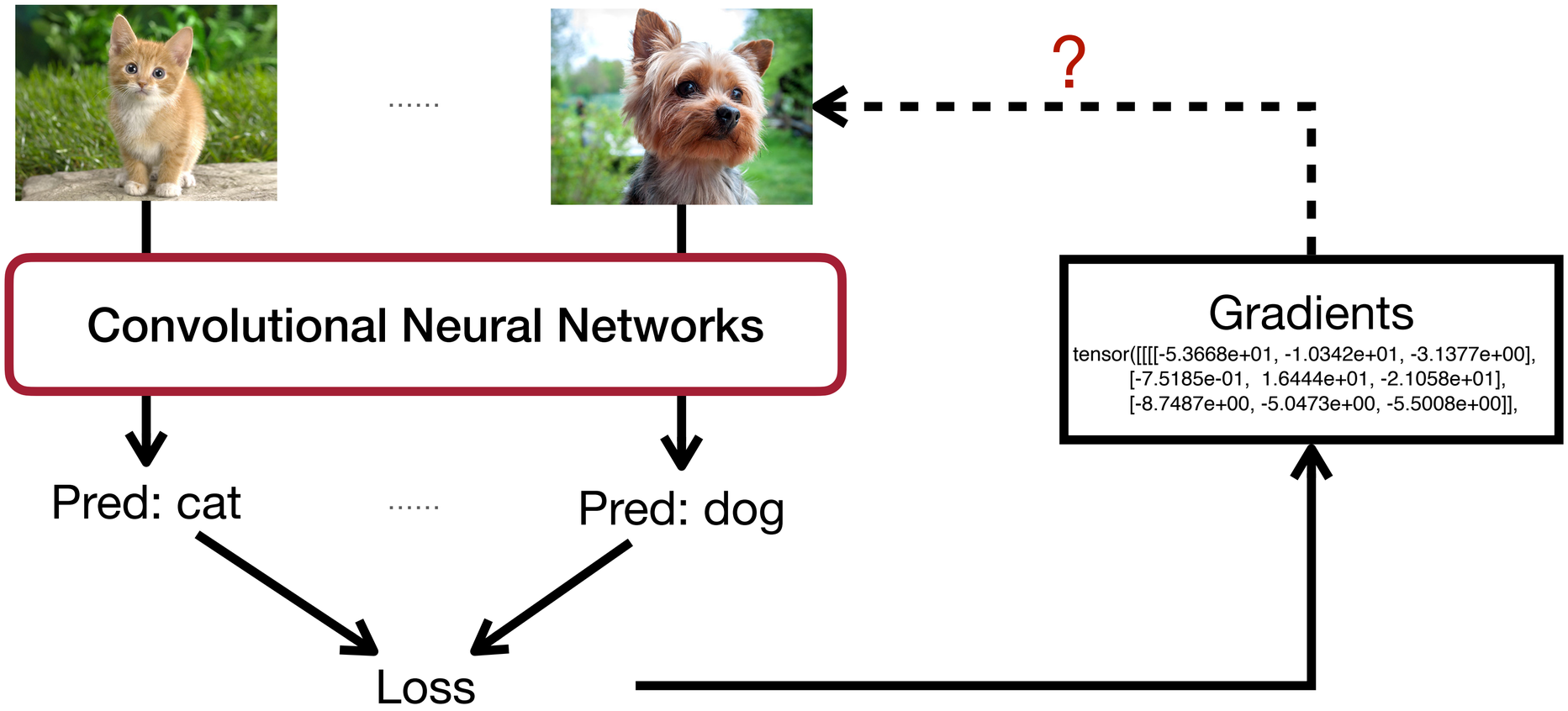


Rethink the Safety of Gradients



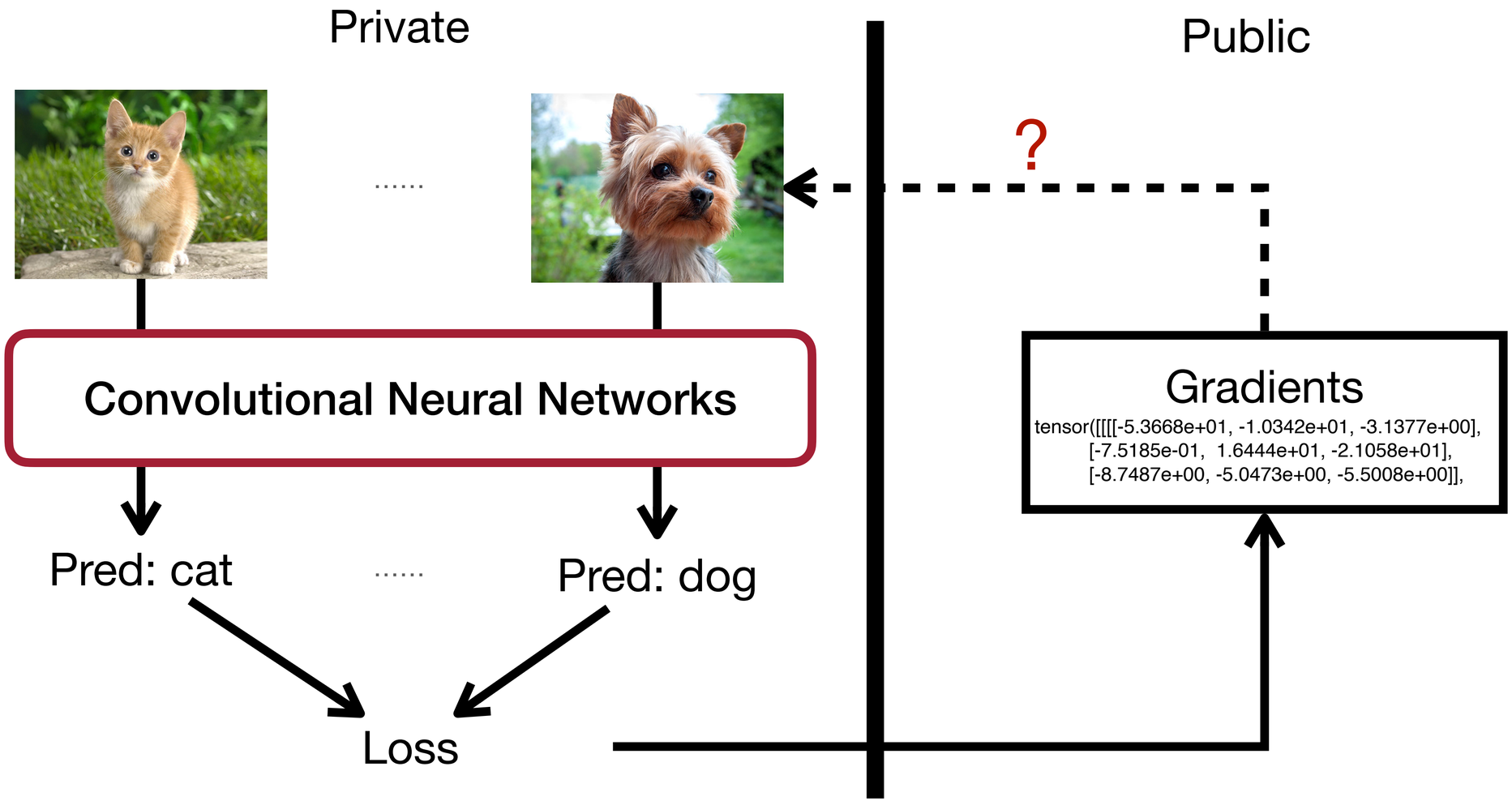
derive gradients from model and training data.

Rethink the Safety of Gradients



Can we derive the training data from gradients?

Rethink the Safety of Gradients



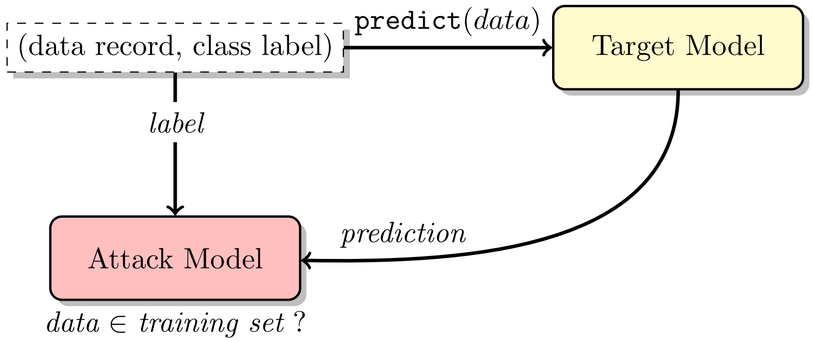
If that is possible, then sharing the gradient is not safe!

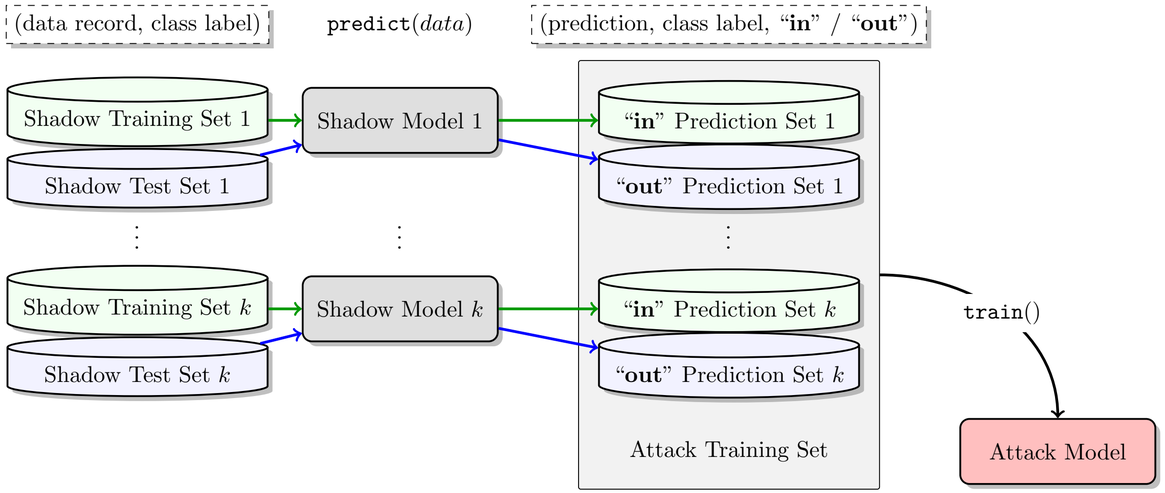
Rethink the Safety of Gradients

Existing Work of Gradient Inversion

Membership Inference [Shokri 2016]

- Given gradients, it's possible to find whether a data point belongs to the batch.





Main idea: Use ML API to construct "shadow" training sets for "in/out" pairs.

Then feed into Attack Model to learn the pattern.

Exploiting unintended feature leakage in collaborative learning. [Melis 2018]
Membership inference attacks against machine learning models. [Shokri 2016]

Rethink the Safety of Gradients

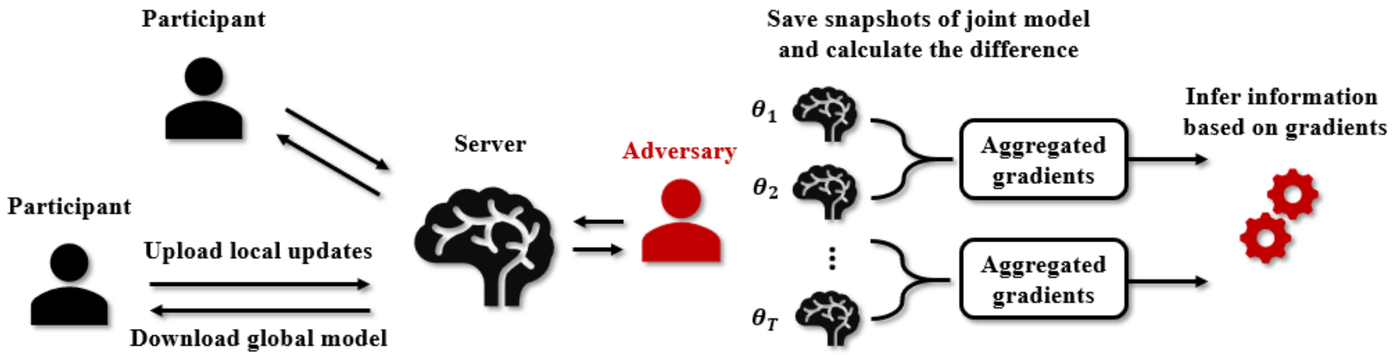
Existing Work of Gradient Inversion

Membership Inference [Shokri 2016]

- Given gradients, it's possible to find whether a data point belongs to the batch.

Property Inference [Melis 2018]

- Given gradients, it's possible to find whether a data point with certain property is in the batch.



Main idea: Use gradients as the latent vectors to train a GAN to generate images.

Exploiting unintended feature leakage in collaborative learning. [Melis 2018]
Membership inference attacks against machine learning models. [Shokri 2016]

Rethink the Safety of Gradients

Existing Work of Gradient Inversion

Membership Inference [Shokri 2016]

- Given gradients, it's possible to find whether a data point belongs to the batch.

Property Inference [Melis 2018]

- Given gradients, it's possible to find whether a data point with certain property is in the batch.

Exploiting unintended feature leakage in collaborative learning. [Melis 2018]
Membership inference attacks against machine learning models. [Shokri 2016]

Rethink the Safety of Gradients

Existing Work of Gradient Inversion

Membership Inference [Shokri 2016]

- Given gradients, it's possible to find whether a data point belongs to the batch.

Property Inference [Melis 2018]

- Given gradients, it's possible to find whether a data point with certain property is in the batch.

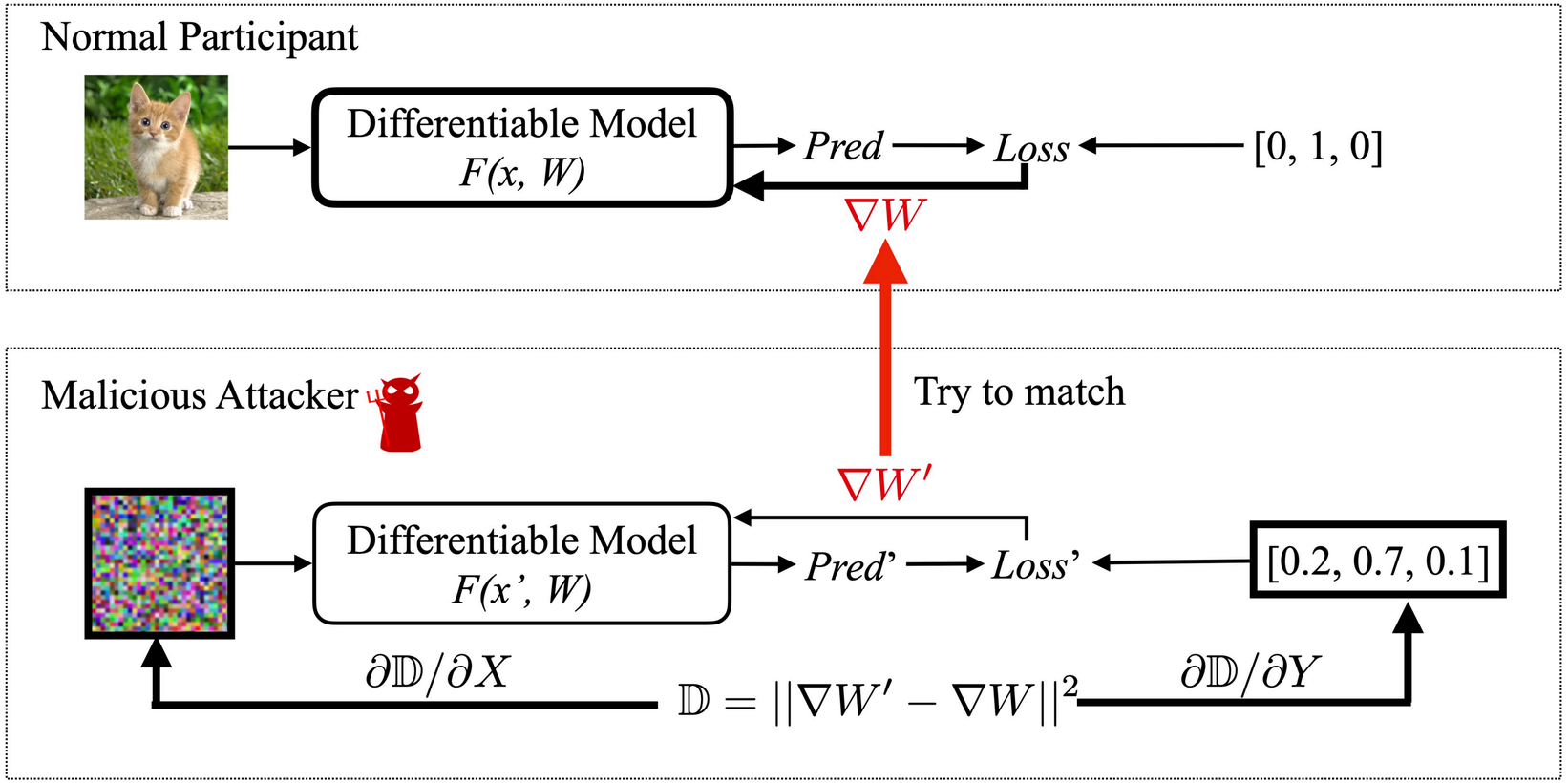
Gradients contain certain information about the training data.

Can we obtain the **raw training data** from **gradient**?

Exploiting unintended feature leakage in collaborative learning. [Melis 2018]
Membership inference attacks against machine learning models. [Shokri 2016]

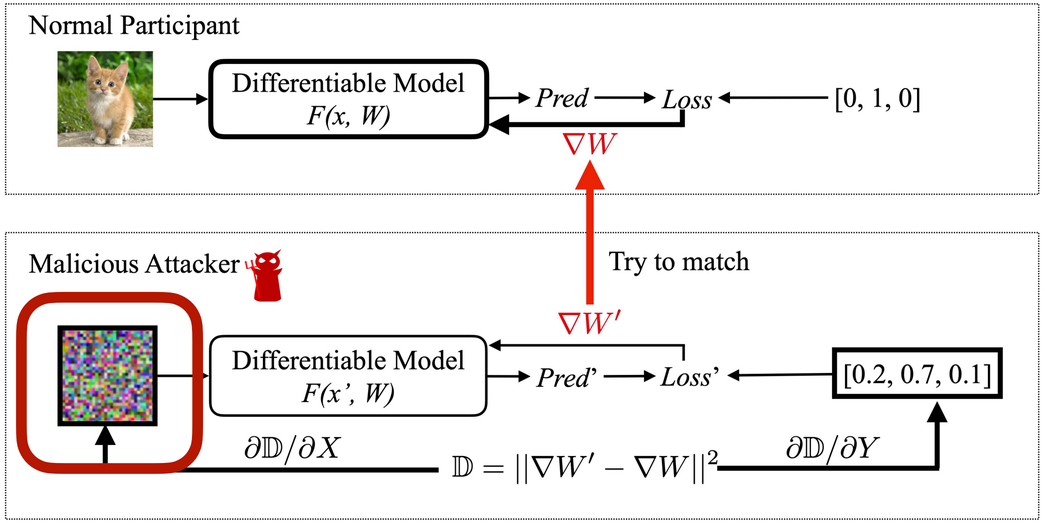
Deep Leakage by Gradient Matching

Match the Gradients to Leak Training Data



Deep Gradient Leakage from Gradients. [Zhu 2019]

Deep Leakage by Gradient Matching



Algorithm 1 Deep Leakage from Gradients.

Input: $F(\mathbf{x}; W)$: Differentiable machine learning model; W: parameter weights; ∇W: gradients calculated by training data

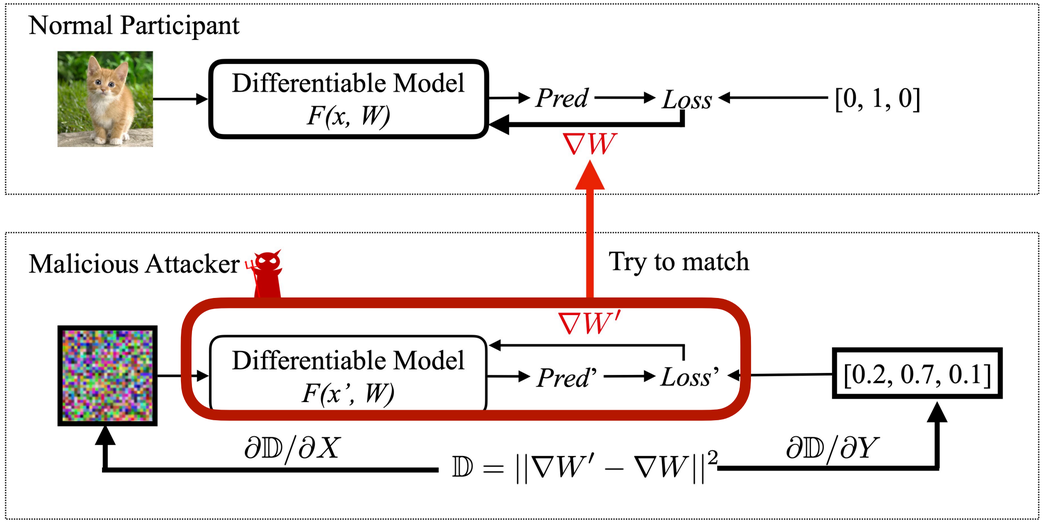
Output: private training data $\mathbf{x}, \mathbf{y}$

procedure DLG(F, W, ∇W)
  $\mathbf{x'}_1 \leftarrow \mathcal{N}(0,1)$, $\mathbf{y'}_1 \leftarrow \mathcal{N}(0,1)$ ▷ Initialize dummy inputs and labels.
  for $i \leftarrow 1$ to n **do**
    $\nabla W'_i \leftarrow \partial \ell(F(\mathbf{x'}_i, W_t), \mathbf{y'}_i)/\partial W_t$ ▷ Compute dummy gradients.
    $\mathbb{D}_i \leftarrow ||\nabla W'_i - \nabla W||^2$
    $\mathbf{x}'_{i+1} \leftarrow \mathbf{x}'_i - \eta \nabla_{\mathbf{x}'_i} \mathbb{D}_i$, $\mathbf{y}'_{i+1} \leftarrow \mathbf{y}'_i - \eta \nabla_{\mathbf{y}'_i} \mathbb{D}_i$ ▷ Update data to match gradients.
  end for
  return $\mathbf{x}'_{n+1}, \mathbf{y}'_{n+1}$
end procedure

1. Initialize a dummy (data, label) pair

Deep Gradient Leakage from Gradients. [Zhu 2019]

Deep Leakage by Gradient Matching



Algorithm 1 Deep Leakage from Gradients.

Input: $F(\mathbf{x}; W)$: Differentiable machine learning model; W: parameter weights; ∇W: gradients calculated by training data

Output: private training data $\mathbf{x}, \mathbf{y}$

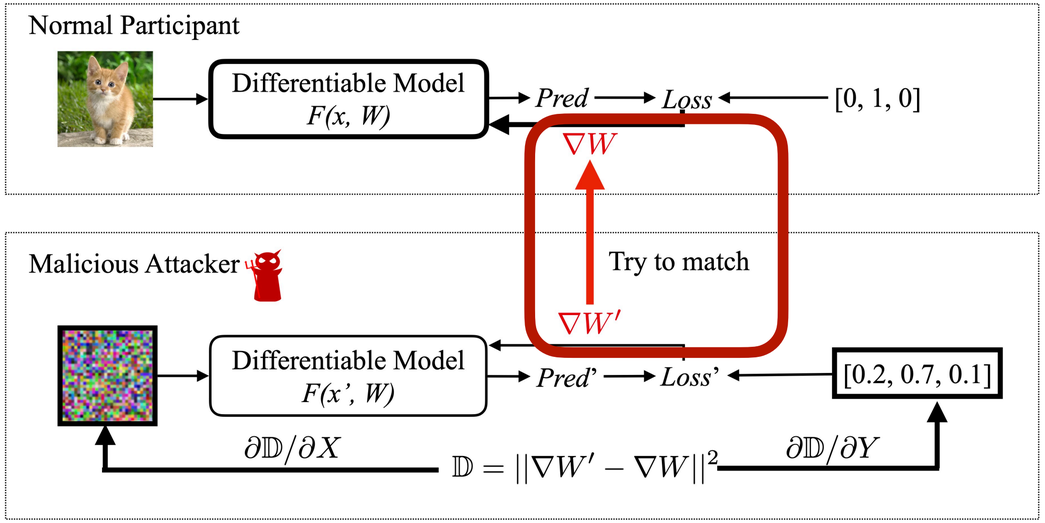
procedure DLG(F, W, ∇W)
  $\mathbf{x'}_1 \leftarrow \mathcal{N}(0,1)$, $\mathbf{y'}_1 \leftarrow \mathcal{N}(0,1)$ ▷ Initialize dummy inputs and labels.
  for $i \leftarrow 1$ to n **do**
    $\nabla W'_i \leftarrow \partial \ell(F(\mathbf{x'}_i, W_t), \mathbf{y'}_i)/\partial W$ ▷ Compute dummy gradients.
    $\mathbb{D}_i \leftarrow ||\nabla W'_i - \nabla W||^2$
    $\mathbf{x}'_{i+1} \leftarrow \mathbf{x}'_i - \eta \nabla_{\mathbf{x}'_i} \mathbb{D}_i$, $\mathbf{y}'_{i+1} \leftarrow \mathbf{y}'_i - \eta \nabla_{\mathbf{y}'_i} \mathbb{D}_i$ ▷ Update data to match gradients.
  end for
  return $\mathbf{x}'_{n+1}, \mathbf{y}'_{n+1}$
end procedure

1. Initialize a dummy (data, label) pair.

2. Feed the (fake) dummy data and label and calculate the dummy loss

Deep Gradient Leakage from Gradients. [Zhu 2019]

Deep Leakage by Gradient Matching



Algorithm 1 Deep Leakage from Gradients.

Input: $F(\mathbf{x}; W)$: Differentiable machine learning model; W: parameter weights; ∇W: gradients calculated by training data

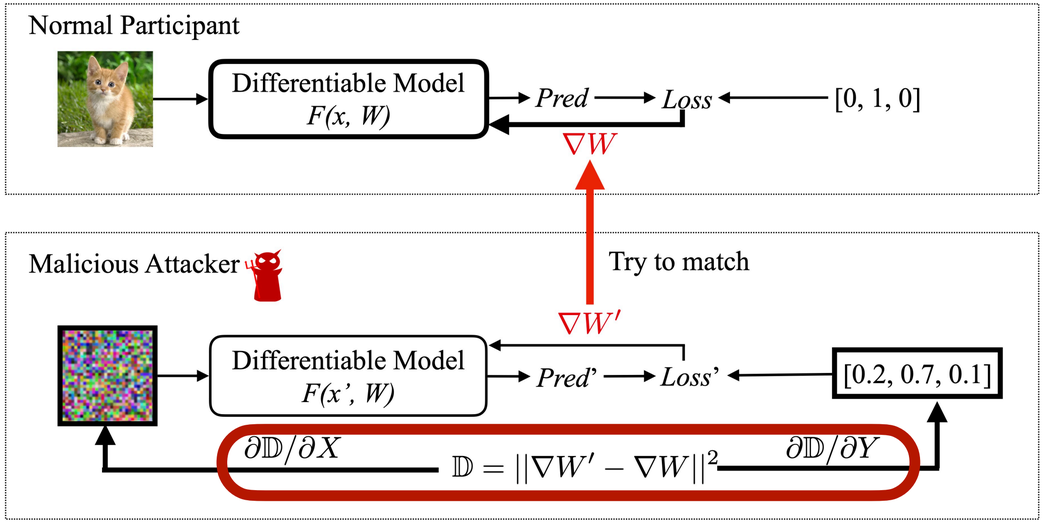
Output: private training data $\mathbf{x}, \mathbf{y}$

procedure DLG(F, W, ∇W)
  $\mathbf{x'}_1 \leftarrow \mathcal{N}(0,1)$, $\mathbf{y'}_1 \leftarrow \mathcal{N}(0,1)$ ▷ Initialize dummy inputs and labels.
  for $i \leftarrow 1$ to n **do**
    $\nabla W'_i \leftarrow \partial \ell(F(\mathbf{x'}_i, W_t), \mathbf{y'}_i)/\partial W_t$ ▷ Compute dummy gradients.
    $\mathbb{D}_i \leftarrow ||\nabla W'_i - \nabla W||^2$
    $\mathbf{x}'_{i+1} \leftarrow \mathbf{x}'_i - \eta \nabla_{\mathbf{x}'_i} \mathbb{D}_i$, $\mathbf{y}'_{i+1} \leftarrow \mathbf{y}'_i - \eta \nabla_{\mathbf{y}'_i} \mathbb{D}_i$ ▷ Update data to match gradients.
  end for
  return $\mathbf{x}'_{n+1}, \mathbf{y}'_{n+1}$
end procedure

1. Initialize a dummy (data, label) pair.
2. Feed the (fake) dummy data and label and calculate the dummy loss
3. Match the distance between dummy loss and real loss.

Deep Gradient Leakage from Gradients. [Zhu 2019]

Deep Leakage by Gradient Matching



Algorithm 1 Deep Leakage from Gradients.

Input: $F(\mathbf{x}; W)$: Differentiable machine learning model; W: parameter weights; ∇W: gradients calculated by training data

Output: private training data $\mathbf{x}, \mathbf{y}$

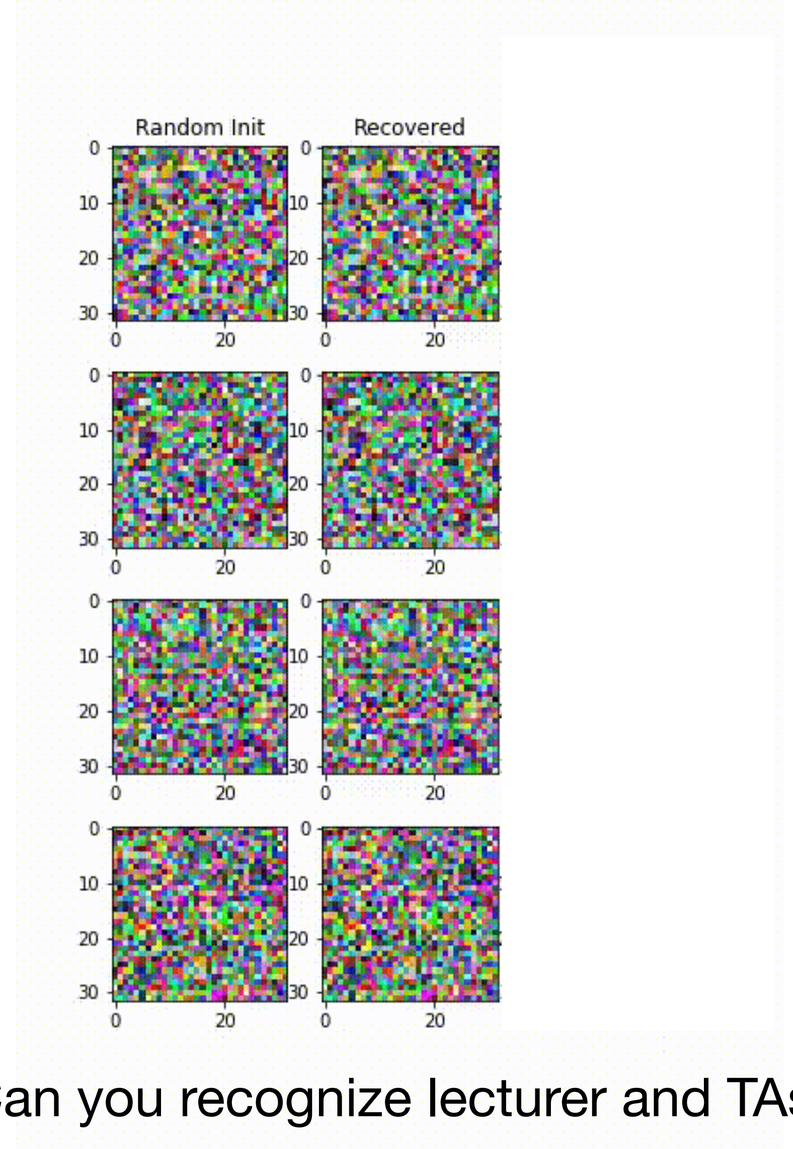
procedure DLG(F, W, ∇W)
  $\mathbf{x'}_1 \leftarrow \mathcal{N}(0,1)$, $\mathbf{y'}_1 \leftarrow \mathcal{N}(0,1)$ ▷ Initialize dummy inputs and labels.
  for $i \leftarrow 1$ to n **do**
    $\nabla W'_i \leftarrow \partial \ell(F(\mathbf{x'}_i, W_t), \mathbf{y'}_i)/\partial W_t$ ▷ Compute dummy gradients.
    $\mathbb{D}_i \leftarrow ||\nabla W'_i - \nabla W||^2$
    $\mathbf{x}'_{i+1} \leftarrow \mathbf{x}'_i - \eta \nabla_{\mathbf{x}'_i} \mathbb{D}_i$, $\mathbf{y}'_{i+1} \leftarrow \mathbf{y}'_i - \eta \nabla_{\mathbf{y}'_i} \mathbb{D}_i$ ▷ Update data to match gradients.
  end for
  return $\mathbf{x}'_{n+1}, \mathbf{y}'_{n+1}$
end procedure

1. Initialize a dummy (data, label) pair.
2. Feed the (fake) dummy data and label and calculate the dummy loss
3. Match the distance between dummy loss and real loss.
4. Update the dummy training data and label using chain rules.

Deep Gradient Leakage from Gradients. [Zhu 2019]

Deep Leakage by Gradient Matching

Evaluation on Vision



Can you recognize lecturer and TAs here?

Deep Gradient Leakage from Gradients. [Zhu 2019]

Deep Leakage by Gradient Matching

Evaluation on NLP

Iters=0: tilting fill given **less word **itude fine **nton overheard living vegas **vac **vation *f forte **dis cerambycidae ellison **don yards marne **kali

Iters=10: tilting fill given **less full solicitor other ligue shrill living vegas rider treatment carry played sculptures lifelong ellison net yards marne **kali

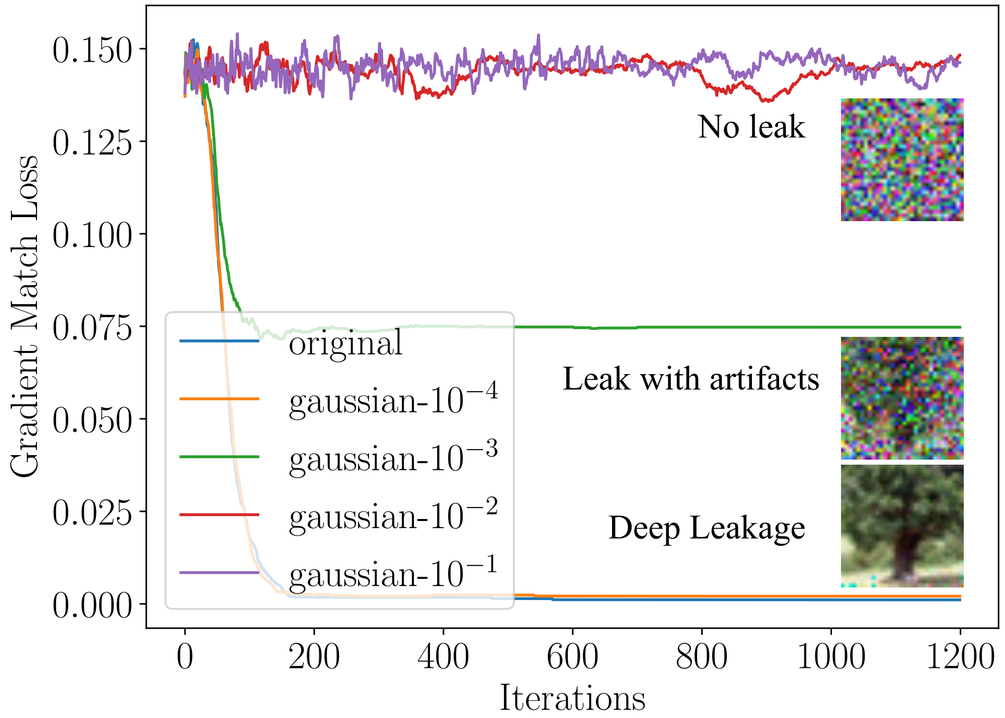
Iters=20: registration , volunteer applications , at student travel application open the ; week of played ; child care will be glare .

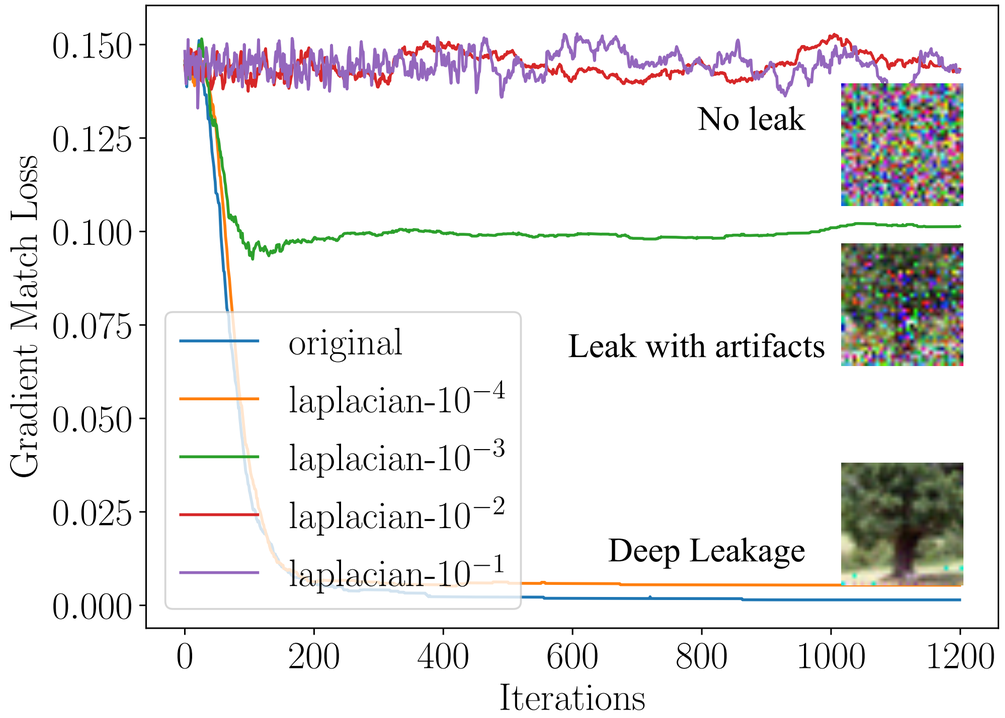
Iters=30: registration, volunteer applications, and student travel application open the first week of september . child care will be available

Original text: Registration, volunteer applications, and student travel application open the first week of September. Child care will be available.

Deep Leakage by Gradient Matching

Defense Strategy

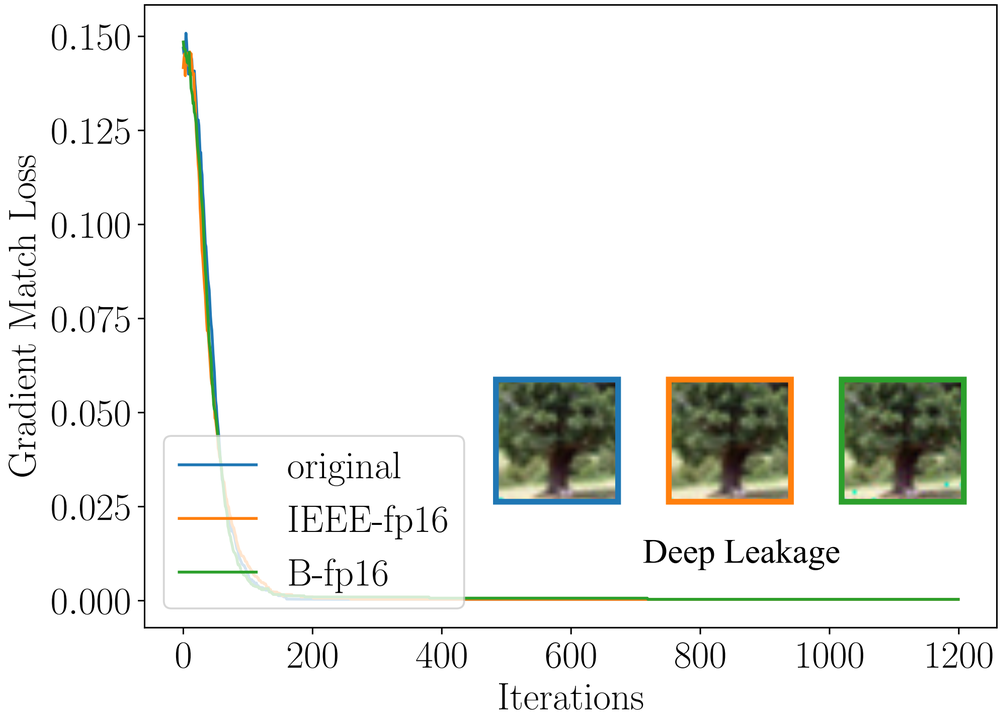




Neither gaussian nor laplacian noise can defend the leakage.
unless serious accuracy drop is acceptable.

Deep Leakage by Gradient Matching

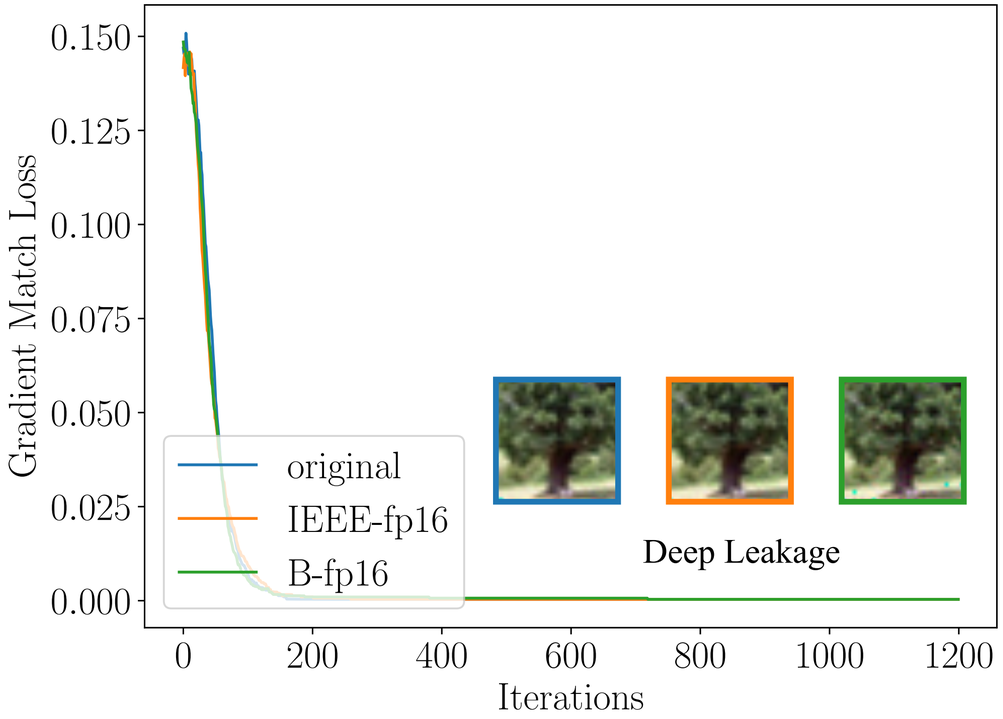
Defense Strategy

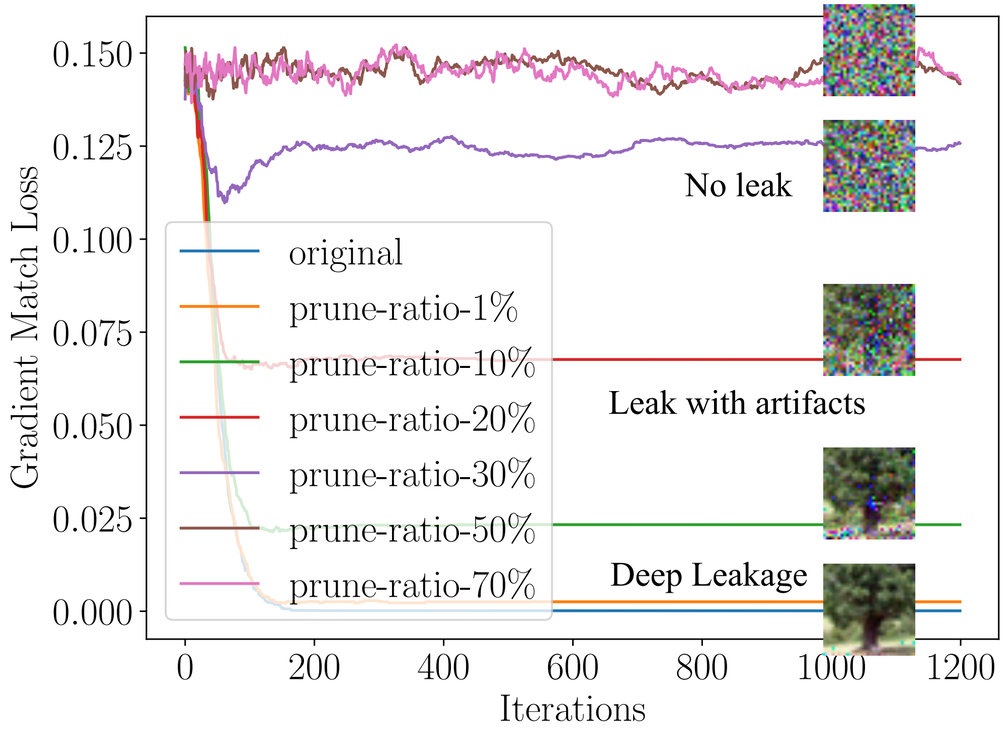


Half precision (FP16 / BFP16) **cannot protect** the privacy as well.

Deep Leakage by Gradient Matching

Defense Strategy

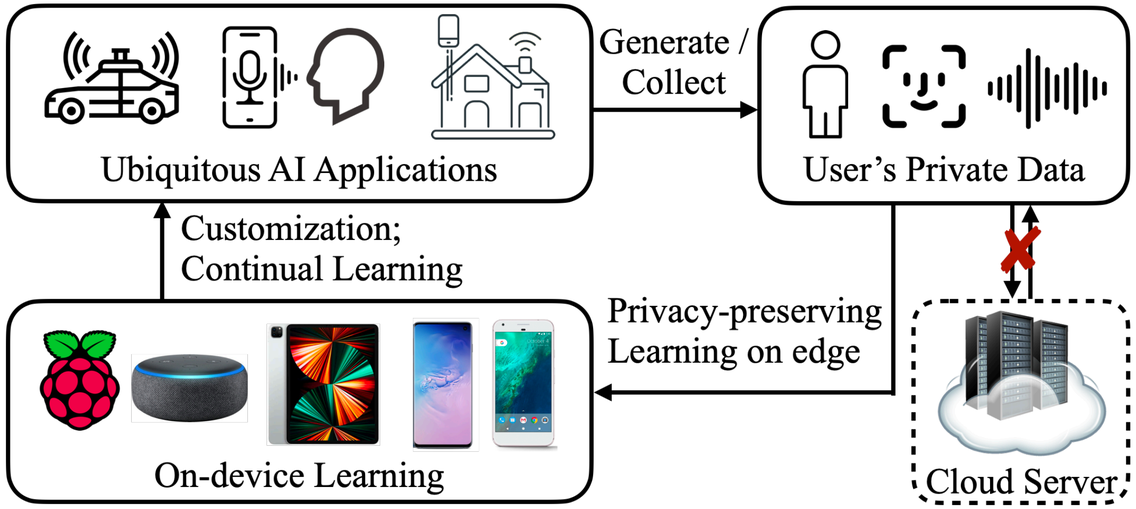




Gradient compression can **effectively protect** the privacy (sparsity > 30%).

Summary

- Algorithm-system co-design for on-device training
  - Why training a quantized model is difficult and how to improve.
  - Full-update is too expensive => using sparse update.
  - System support for efficient on-device training.



- Why federated learning
  - There are vast amount of isolated data
  - Federated learning allows joint training without exchanging data.

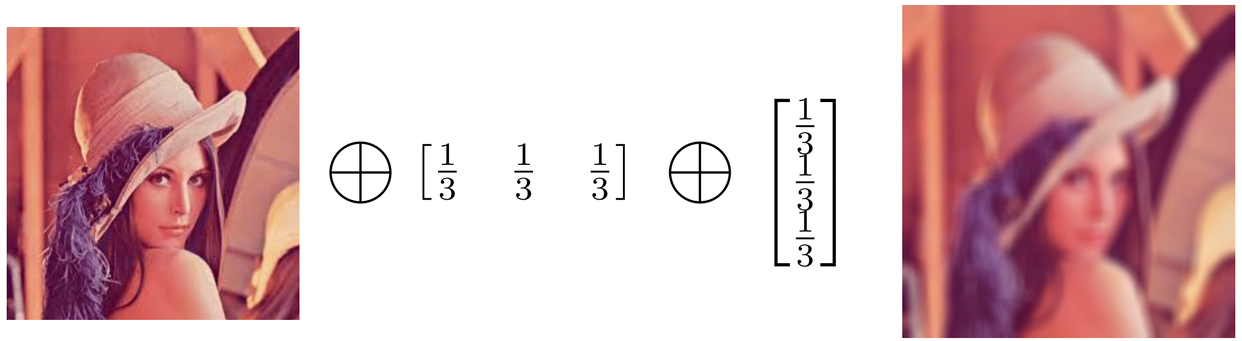
- Rethink the safe of gradients
  - Conventional studies show "shallow" leakage of gradients (membership, property)
  - DLG shows deep leakage from gradients (exact original data)
  - The most effective method to defend leakage is gradient compression.

Section 3:
Compilers, Languages and Optimizations for Deep Learning

How do we implement DL algorithms properly, quickly, while keeping the runtime efficient?

Problem: Simple Convolution to Process the Image

Gaussian Blue Example



How would you to implement this algorithm?
C++? Python? Assembly?

Figure credits from Halide

Simple Convolution - Lena Example

100x speedup with low-level instructions (9.94ms -> 0.90ms)

Hard to program and low readability (10 LoC -> 40 LoC)

Tensor programing is not easy!

Decouple hardware related **schedules** and **algorithms**

(a) Clean C++ : 9.94 ms per megapixel

```
void blur(const Image &in, Image &blurred) {
 Image tmp(in.width(), in.height());

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```

(b) Fast C++ (for x86) : 0.90 ms per megapixel

```
void fast_blur(const Image &in, Image &blurred) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *tmpPtr = tmp;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_loadu_si128((__m128i*)(inPtr-1));
     b = _mm_loadu_si128((__m128i*)(inPtr+1));
     c = _mm_load_si128((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(tmpPtr++, avg);
     inPtr += 8;
   }}
   tmpPtr = tmp;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
     a = _mm_load_si128(tmpPtr+(2*256)/8);
     b = _mm_load_si128(tmpPtr+256/8);
     c = _mm_load_si128(tmpPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

Figure credits from Halide

Halide: a language for fast, portable computation

Decouple Computation and Scheduling

(c) Halide : 0.90 ms per megapixel

```
Func halide_blur(Func in) {
 Func tmp, blurred;
 Var x, y, xi, yi;

// The algorithm
tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;

// The schedule
blurred.tile(x, y, xi, yi, 256, 32)
       .vectorize(xi, 8).parallel(y);
tmp.chunk(x).vectorize(x, 8);

return blurred;
}
```

Algorithm: **What** to compute

Schedule: **How** to compute

- Easy build pipelines
- Easy to specify & explore optimizations
- Leave the compiler to generate fast code

Ragan-Kelley et al, Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines

Compare Halide and Conventional Implementations

Conventional implementation

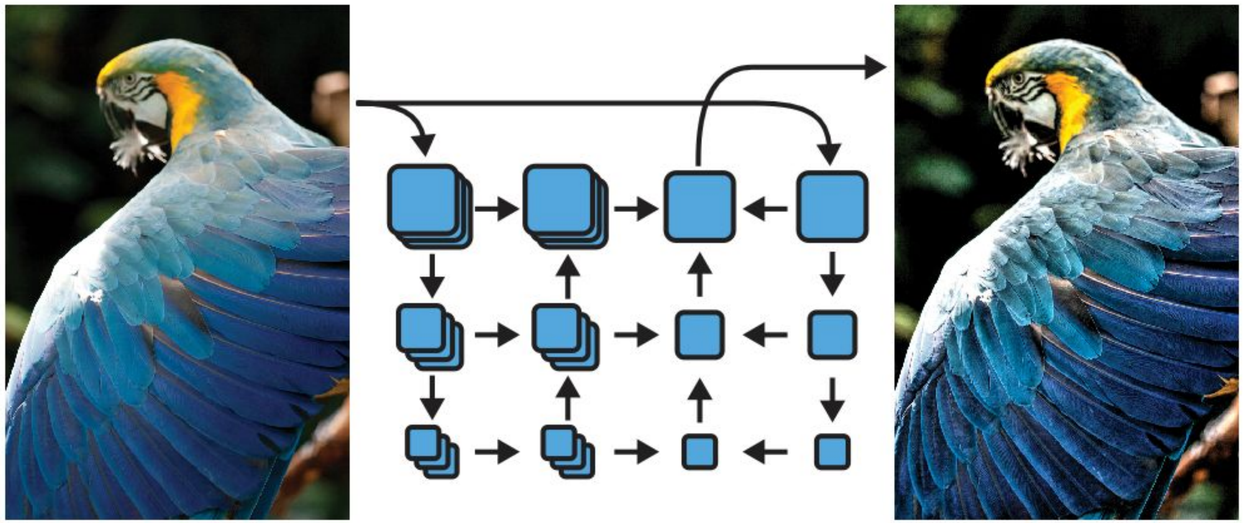
- Reference: 300 lines of C++

Adobe Optimized:

- 1500 lines of expert-optimized C++
- 3 months of work
- 10x faster (with Multi-threaded, SSE SIMD intrinsics)

Halide implementation

- **60 lines**
- **1 intern-day**
- 20x faster compared v.s. reference
- 2x faster v.s. Adobe ver
- GPU adaptation, 70x faster than reference.



Local Laplacian Filter

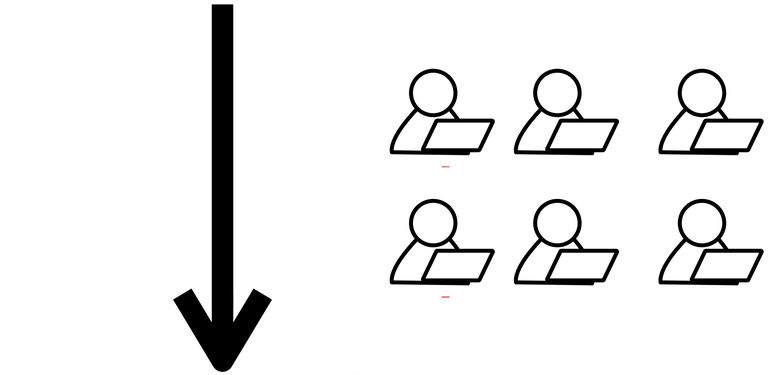
Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines [Ragan-Kelley et al. 2013]

Deep Learning Compiler Stack

Why deep learning compiler?

TensorFlow PYTORCH Caffe2 mxnet

Framework diversity: each infra has its pipeline and op definition.



Engineer Expensive: each adaptation requires experienced engineers.

Hardware diversity: each platform has its own intrinsics and optimization policy.

TVM: Learning-based DL Compiler

- Specialized for deep learning applications and support diverse hardware backends (CPU, GPU, DSP, NPUs …)
- Following Halide's idea about DSL to decouple computation and scheduling.
- The schedules in TVM can be automatically optimized by machine learning algorithms.

Figure from *TVM Conference Overview Presentation*

TVM: Learning-based DL Compiler

Tensor Expression Language

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

Example of Matrix Multiplication (Matmul)

Vanilla Code Generation

```
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]
```

Loop Tiling for Locality

```
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]
```

Map to Hardware Instructions

```
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)
```

TVM: Learning-based DL Compiler

- TVM allows us to separate the algorithm and schedules

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

- But writing schedules is still challenging and engineering expensive:

e compute expression

```
A = t.placeholder((1024, 1024))
B = t.placeholder((1024, 1024))
k = t.reduce_axis((0, 1024))
C = t.compute((1024, 1024),
   lambda y, x:
   t.sum(A[k, y] * B[k, x], axis=k))
```

x_0 default code

```
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]
```

s_1 loop tiling

```
yo, xo, yi, xi = s[C].title(y, x, ty, tx)
s[C].reorder(yo, xo, k, yi, xi)
```

$x_1 = g(e, s_1)$

```
for yo in range(1024 / ty):
  for xo in range(1024 / tx):
    C[yo*ty:yo*ty+ty][xo*tx:xo*tx+tx] = 0
    for k in range(1024):
      for yi in range(ty):
        for xi in range(tx):
          C[yo*ty+yi][xo*tx+xi] +=
             A[k][yo*ty+yi] * B[k][xo*tx+xi]
```

s_2 tiling, map to micro kernel intrinsics

```
yo,xo,ko,yi,xi,ki = s[C].title(y,x,k,8,8,8)
s[C].tensorize(yi, intrin.gemm8x8)
```

$x_2 = g(e, s_2)$

```
for yo in range(128):
  for xo in range(128):
    intrin.fill_zero(C[yo*8:yo*8+8][xo*8:xo*8+8])
    for ko in range(128):
      intrin.fused_gemm8x8_add(
        C[yo*8:yo*8+8][xo*8:xo*8+8],
        A[ko*8:ko*8+8][yo*8:yo*8+8],
        B[ko*8:ko*8+8][xo*8:xo*8+8])
```

.....

Tons of possible schedules to explore**!**

How to automatically select the best schedules of possible programs?

AutoTVM: Learning to Optimize Tensor Programs

Use machine learning to optimize machine learning models

e compute expression

```
A = t.placeholder((1024, 1024))
B = t.placeholder((1024, 1024))
k = t.reduce_axis((0, 1024))
C = t.compute((1024, 1024),
    lambda y, x:
    t.sum(A[k, y] * B[k, x], axis=k))
```

x_0 default code

```
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]
```

s_1 loop tiling

```
yo, xo, yi, xi = s[C].title(y, x, ty, tx)
s[C].reorder(yo, xo, k, yi, xi)
```

$x_1 = g(e, s_1)$

```
for yo in range(1024 / ty):
  for xo in range(1024 / tx):
    C[yo*ty:yo*ty+ty][xo*tx:xo*tx+tx] = 0
    for k in range(1024):
      for yi in range(ty):
        for xi in range(tx):
          C[yo*ty+yi][xo*tx+xi] +=
            A[k][yo*ty+yi] * B[k][xo*tx+xi]
```

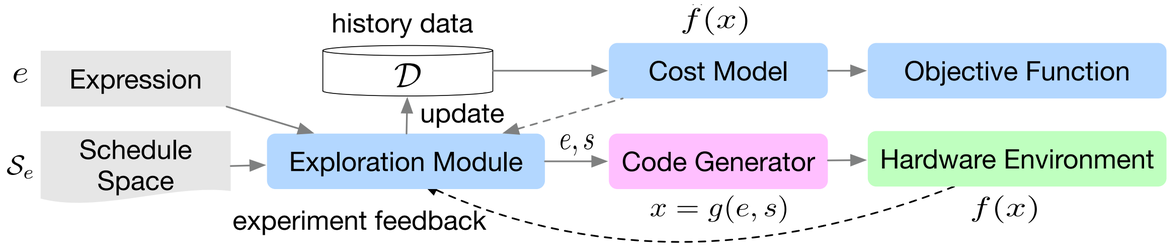
s_2 tiling, map to micro kernel intrinsics

```
yo,xo,ko,yi,xi,ki = s[C].title(y,x,k,8,8,8)
s[C].tensorize(yi, intrin.gemm8x8)
```

$x_2 = g(e, s_2)$

```
for yo in range(128):
  for xo in range(128):
    intrin.fill_zero(C[yo*8:yo*8+8][xo*8:xo*8+8])
    for ko in range(128):
      intrin.fused_gemm8x8_add(
        C[yo*8:yo*8+8][xo*8:xo*8+8],
        A[ko*8:ko*8+8][yo*8:yo*8+8],
        B[ko*8:ko*8+8][xo*8:xo*8+8])
```

- The loop tiling size (tx ty) can viewed as hyper parameters of code generation.



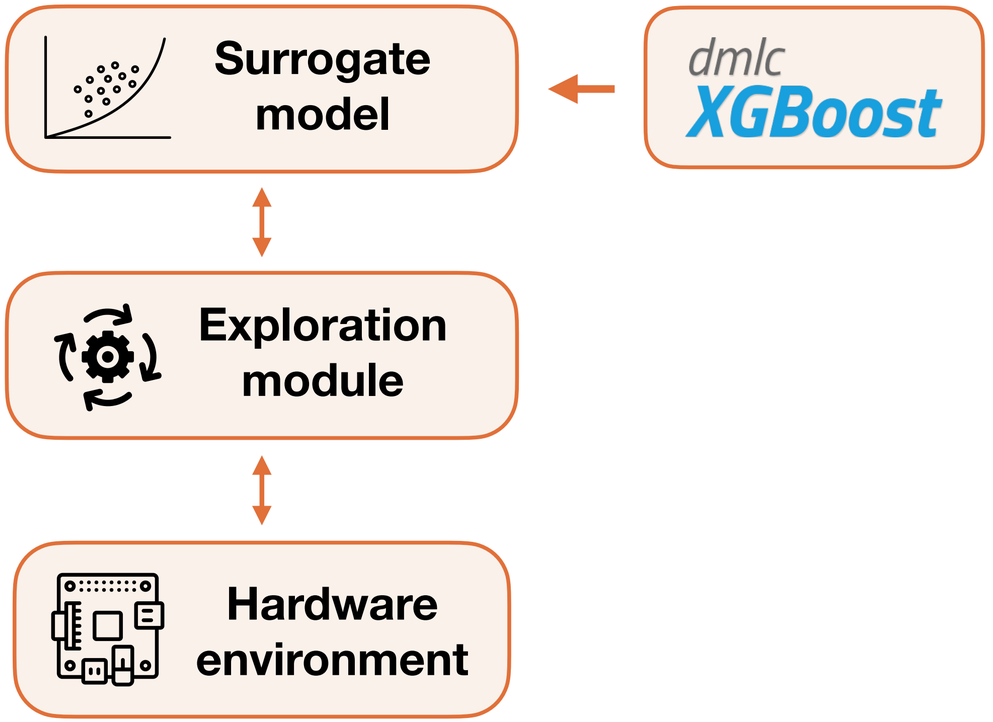
- Learning based methods can help us automatically find the best configs.

Learning to Optimize Tensor Programs [Chen et al 2018]

AutoTVM: Learning to Optimize Tensor Programs

Template Based Search Algorithms

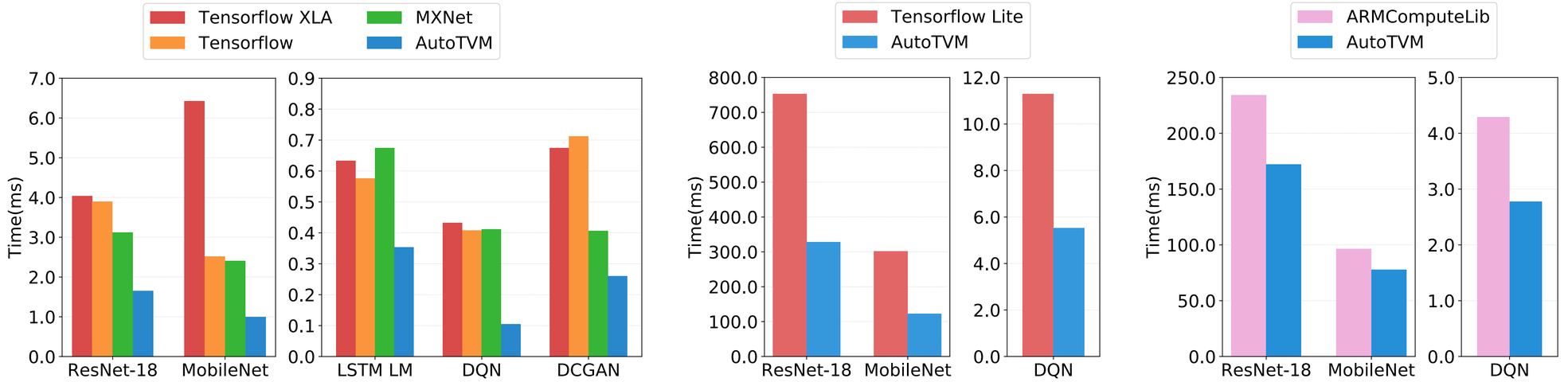
1. Choose schedule x^*according from a surrogate model (e.g., Gaussian Process)
2. Evaluate $f(x^*)$
3. Add $(x^*, f(x^*))$ to the training data
4. Fit data to model
5. Go to step 1 and sample new



Source from CMU 15-849 [Jia 2022]

AutoTVM: Learning to Optimize Tensor Programs

Performance Evaluation



(a) NVIDIA TITAN X End2End (b) ARM Cortex-A53 End2End (c) ARM Mali-T860 End2End

End-to-end performance across back-ends. AutoTVM **outperforms the manually written** baseline methods

Learning to Optimize Tensor Programs [Chen et al 2018]

Learn to Optimize Tensor Programs

Summary

- Encode program AST into features
- Propose a config from template zoos and evaluate
- Compile and evaluate on real hardware, then update the model
- Automate the process of exploring schedules by a learning-based model

Ansor: Generating High-Performance Tensor Programs for Deep Learning [Zheng 2020]

Summary of Today's Lecture

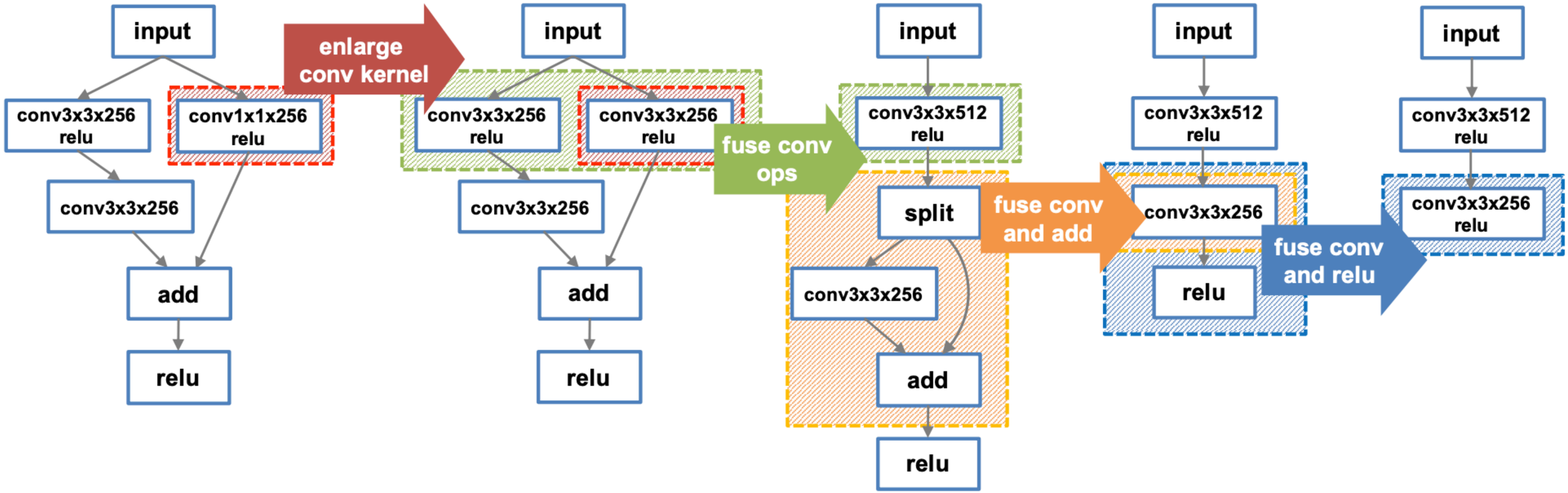
Today we learned:

1. The background of federated learning.
2. Why gradient exchange is not always safe and rethink the safety of gradients.
3. The difficulty of developing efficient tensor programs.
    1. Halide, a domain specific language for parallel computing.
    2. TVM, a domain specific compiler for deep learning.
    3. AutoTVM, a learning-based optimizer to generate efficient schedules

Section 4: Graph-level Optimization

What is graph-level optimizations and why use it?

MetaFlow: Optimizing DNN Computation

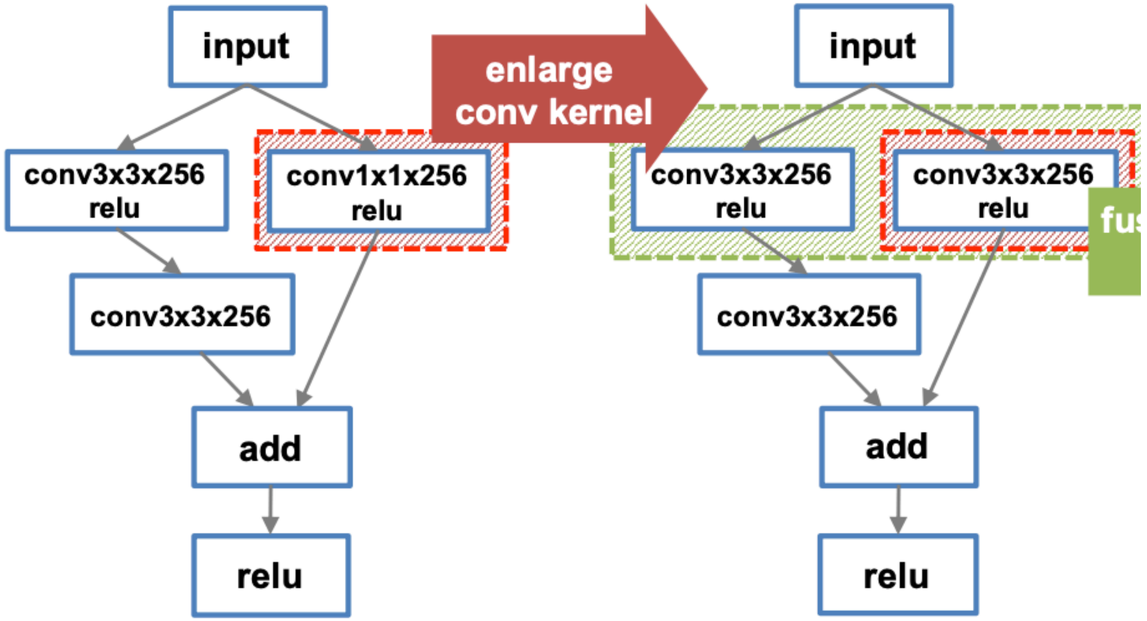


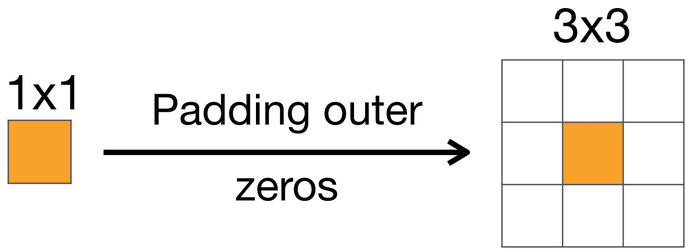
The final graph is 30% faster on V100.

MetaFlow: Optimizing DNN Computation with Relaxed Graph Substitutions [Jia 2019]

MetaFlow: Optimizing DNN Computation

PiEnlarge conv kernels



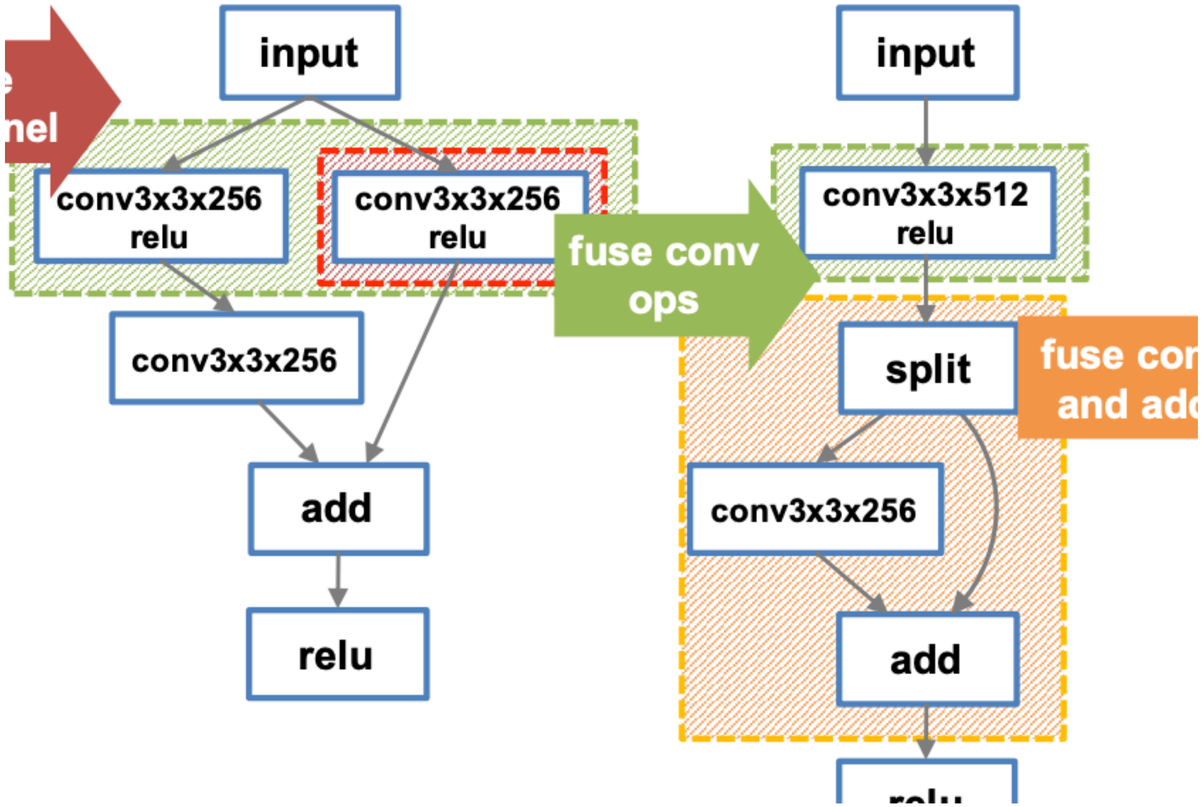


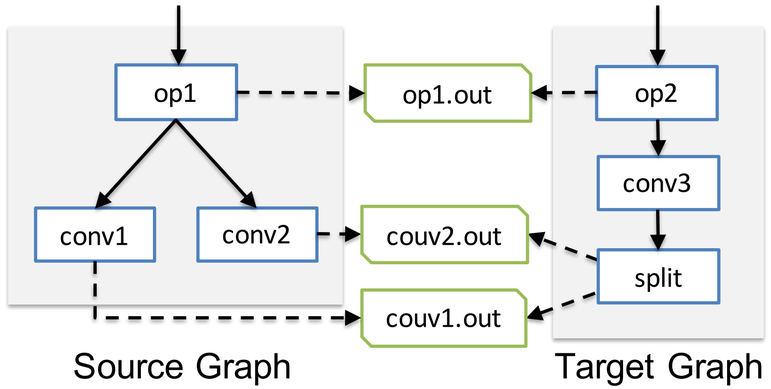
- Mathematically equivalent to previous graph.
- More computation / FLOPs
- But less latency because of fewer kernel calls.

MetaFlow: Optimizing DNN Computation with Relaxed Graph Substitutions [Jia 2019]

MetaFlow: Optimizing DNN Computation

Fuse parallel convolutions






```
# Constraints on the source graph:
conv1.kernel == conv2.kernel
conv1.stride == conv2.stride
conv1.padding == conv2.padding
```

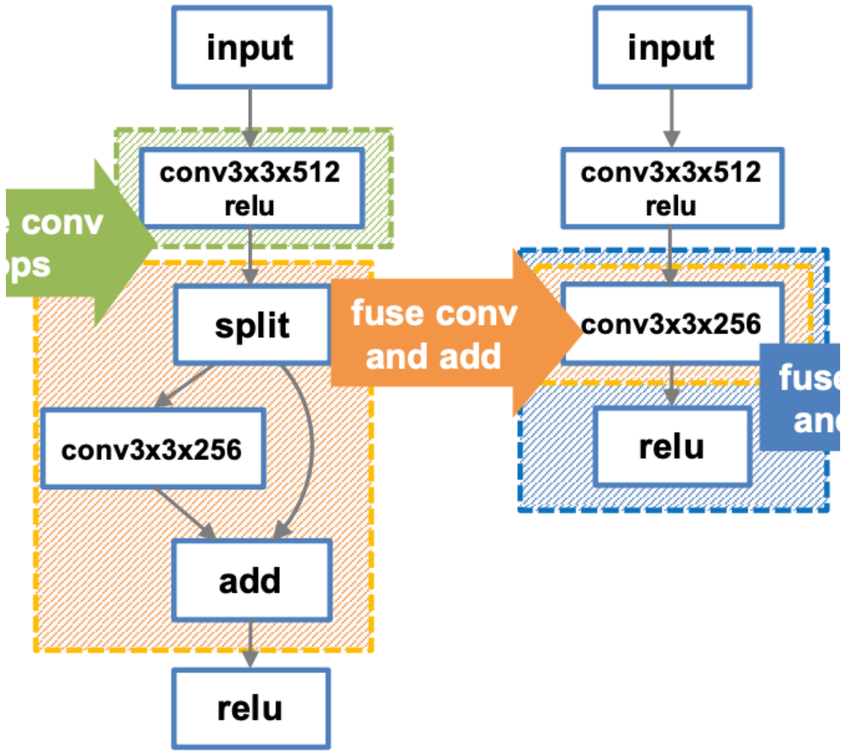
```
# Construct the target graph:
op2._ = op1._
conv3._ = conv1._
conv3.outChannels = conv1.outChannels + conv2.outChannels
conv3.weights = concat(conv1.weights, conv2.weights)
split.sizes = [conv1.outChannels, conv2.outChannels]
```

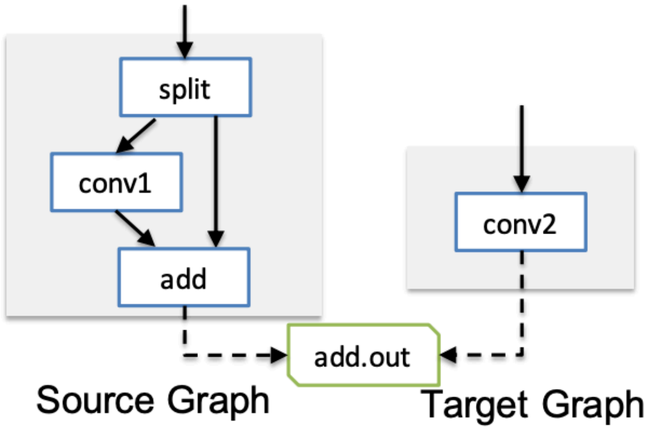
- Merge two convolution into one grouped convolution.
- Reduce #num of kernel calls.

MetaFlow: Optimizing DNN Computation with Relaxed Graph Substitutions [Jia 2019]

MetaFlow: Optimizing DNN Computation

Fuse conv and add






```
# Constraints on the source graph:
conv1.stride == (1, 1)
```

```
# Construct the target graph:
conv2.inChannels = conv1.inChannels + conv1.outChannels
conv2.outChannels = conv1.outChannels
# I is an identity matrix
conv2.weights = concat(conv1.weights, I)
```

- Fuse a convolution and an add.
- Reduce #num of kernel calls.

MetaFlow: Optimizing DNN Computation with Relaxed Graph Substitutions [Jia 2019]

MetaFlow: Optimizing DNN Computation

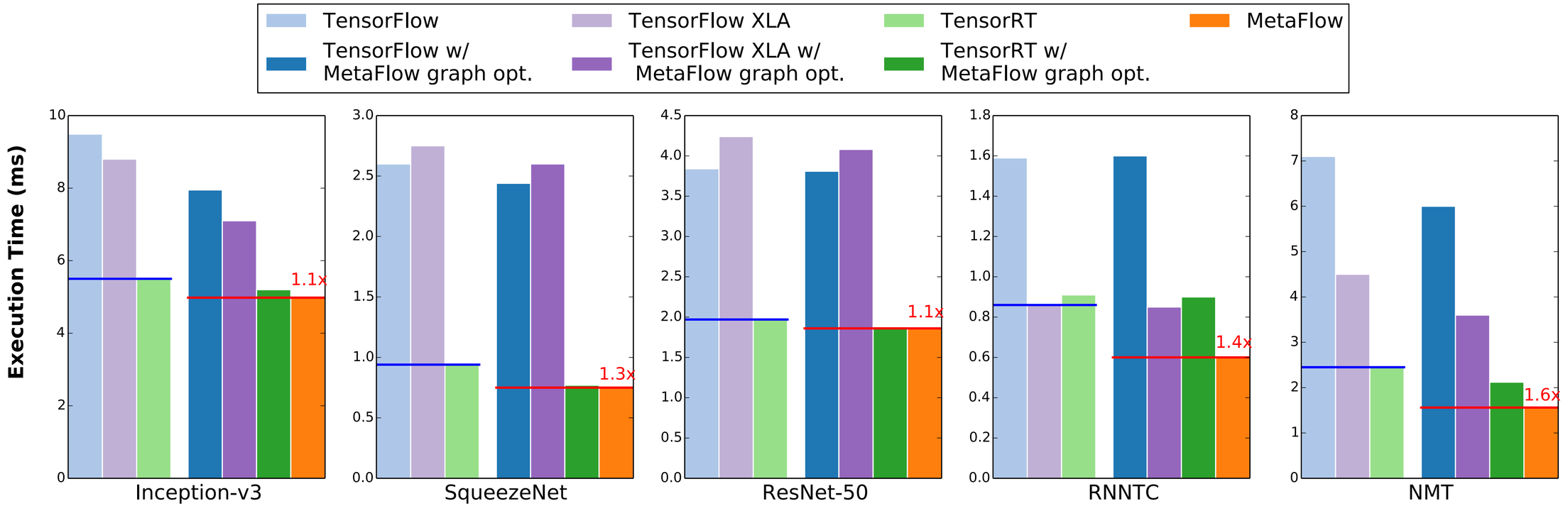
Automating the search process — Cost Model

- Model dependent:
  - FLOPs, memory footprint, number of kernel launches
- Hardware dependent:
  - Inference time (obtained from measurement)
- Record measured subgraph/hardware into a lookup table to speed up search

MetaFlow: Optimizing DNN Computation with Relaxed Graph Substitutions [Jia 2019]

MetaFlow: Optimizing DNN Computation

Performance Evaluation

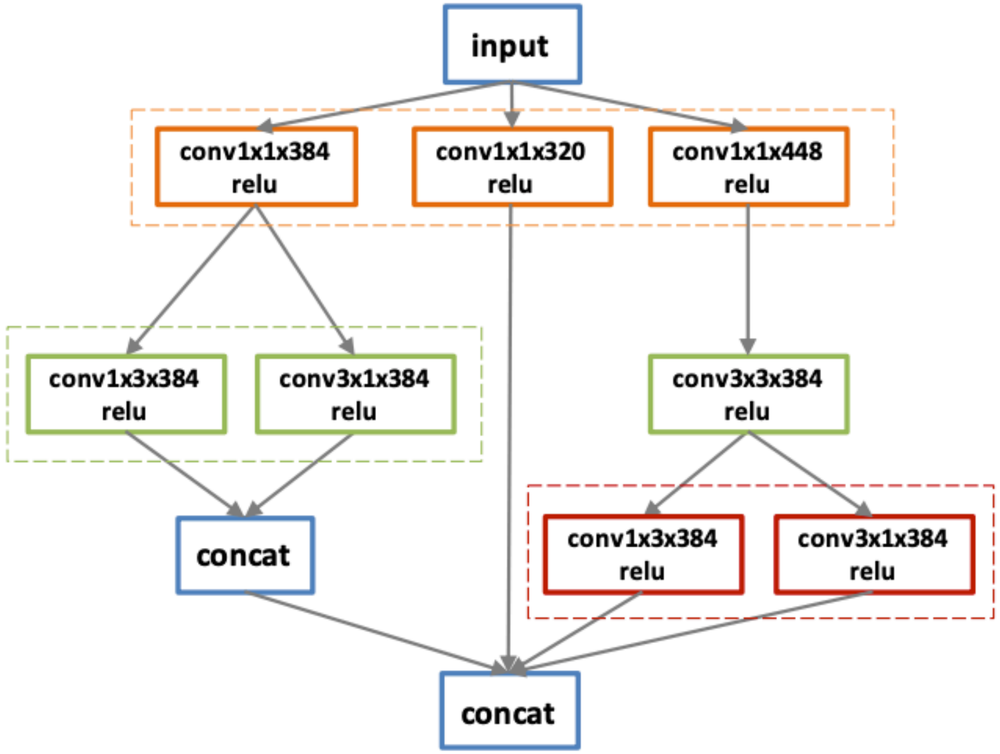


MetaFlow achieves 1.1x to 1.6x speedup over previous SOTAs.

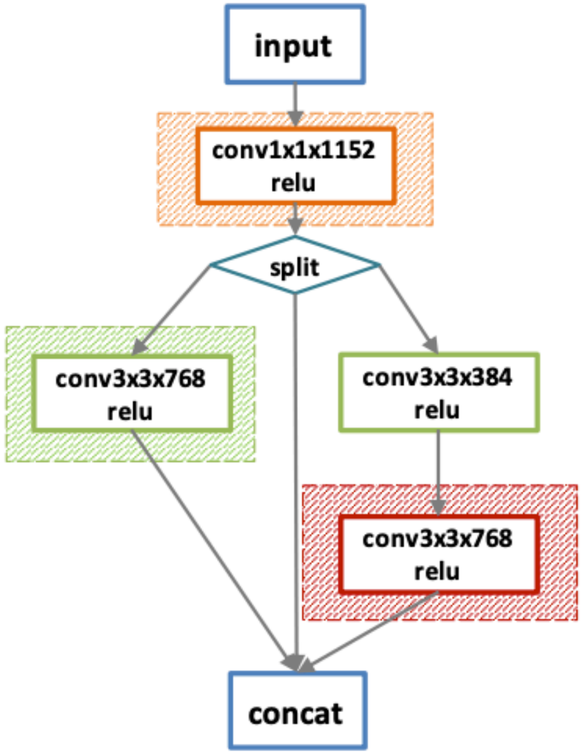
MetaFlow: Optimizing DNN Computation with Relaxed Graph Substitutions [Jia 2019]

MetaFlow: Optimizing DNN Computation

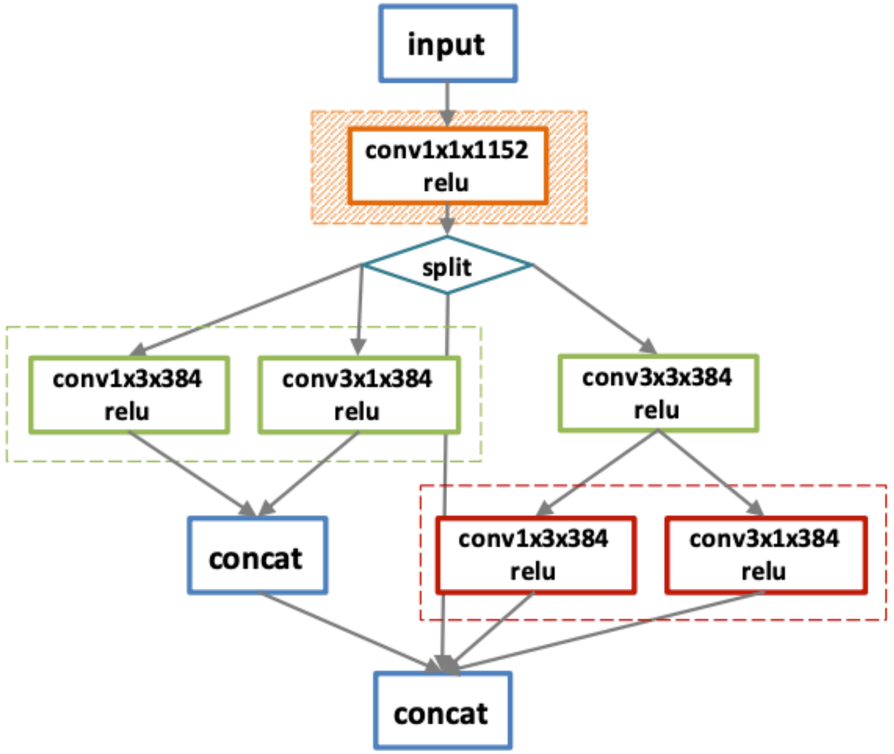
Hardware Adaptation



(a) Original graph.



(b) Optimized graph for V100.



(c) Optimized graph for K80.

Different hardware has different optimal transformed graphs.

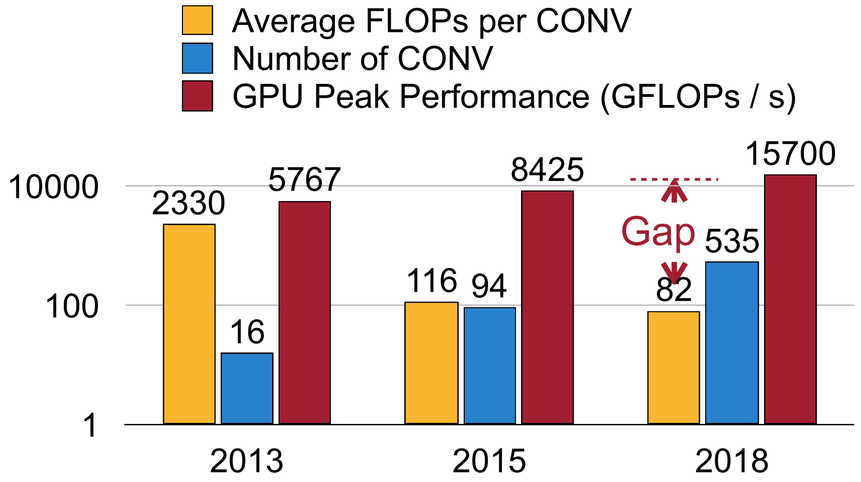
MetaFlow: Optimizing DNN Computation with Relaxed Graph Substitutions [Jia 2019]

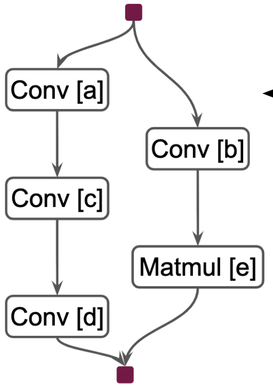
Motivations for Inter-Operator Parallelization

- With recent advanced in GPU and model design, single operator cannot fully utilize the hardware

- Current implementations focus on optimizing intra-operator parallelism, leading to low utilization

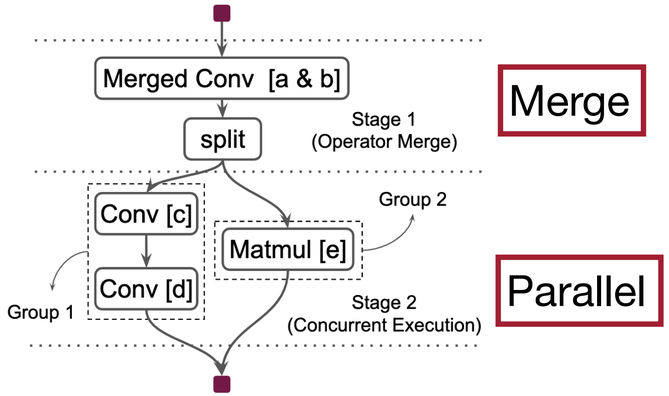
- **Solution:** Parallelly execute and merge operators to improve utilization





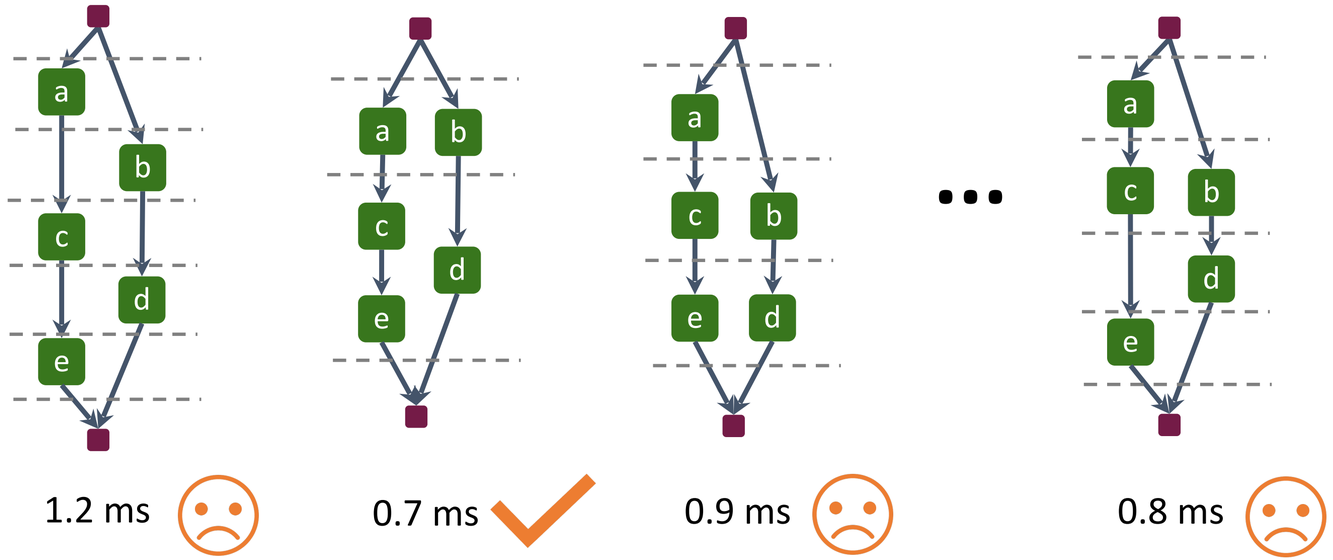
Sequential execution:
[a] -> [b] -> [c] -> [d] -> [e]

Parallel and Merged execution:
[a & b] => [c] -> [d]



IOS: Inter-Operator Scheduler for CNN Acceleration

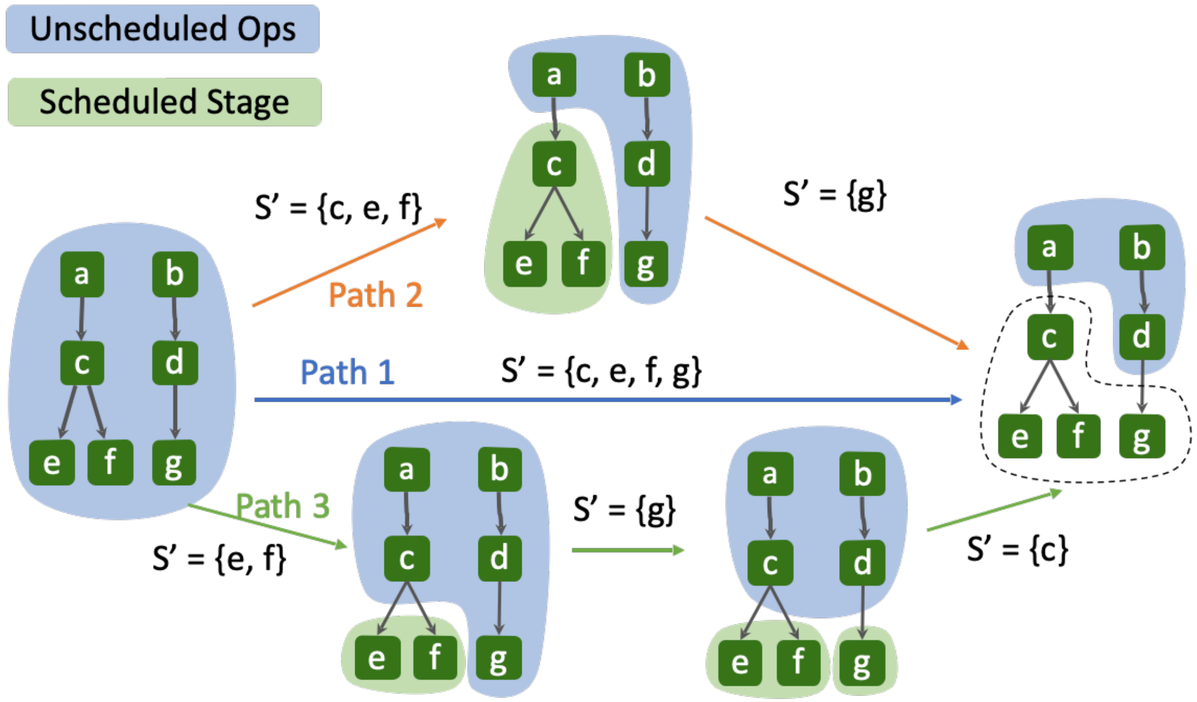
- **Main idea:** find the best inter-operator schedule plans for a given architecture



IOS: Inter-Operator Scheduler for CNN Acceleration

Optimal Schedule Finding Algorithms

- Use Dynamic Programming (DP) find the best schedule policy.
- **Key idea**: recursive + memorization
- The time complexity of the dynamic programming: $O((\frac{n}{d}+1)^{2d})$

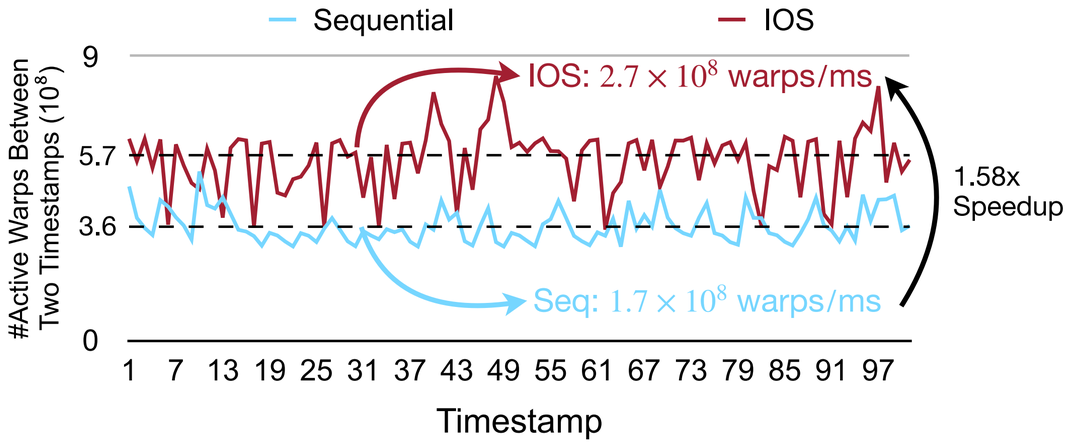


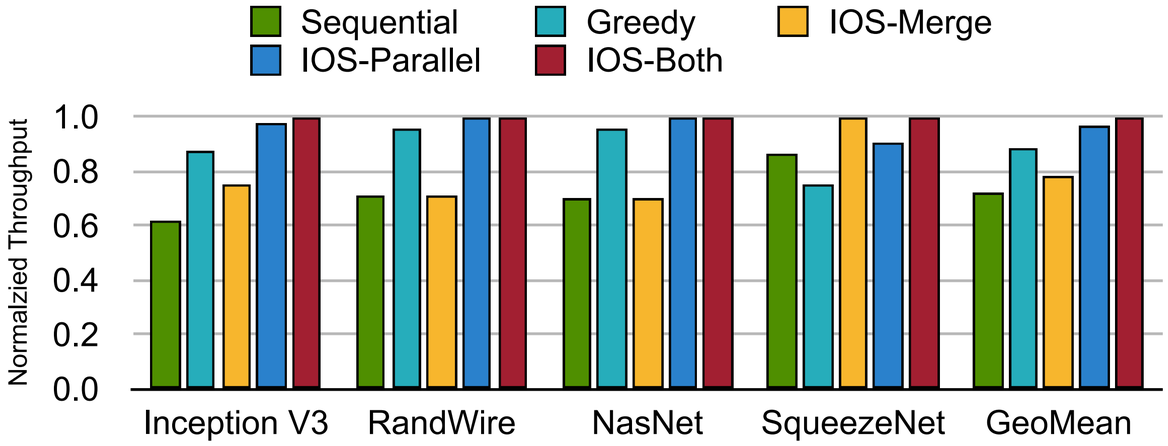
$$Latency[S] = \min_{\text{S' is an ending of S}} (Latency[S - S'] + StageLatency[S'])$$

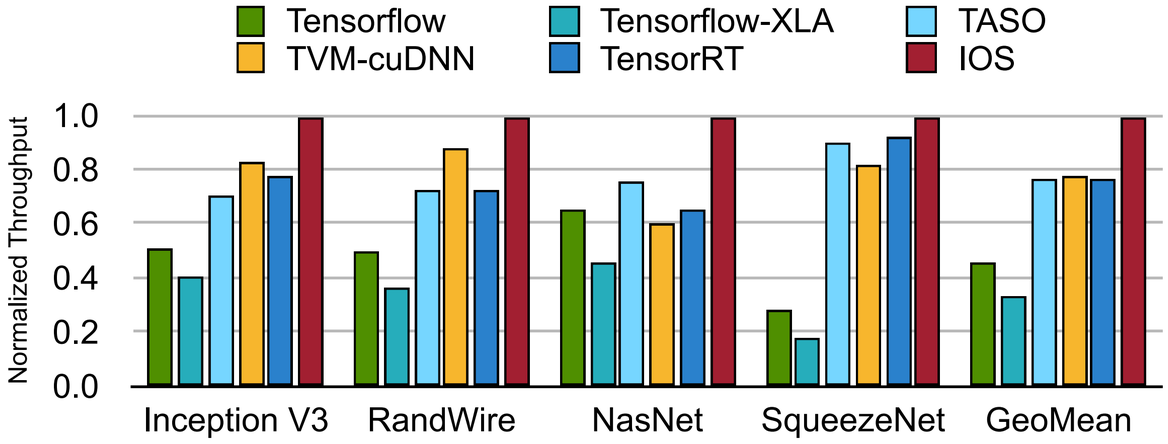
IOS: Inter-Operator Scheduler for CNN Acceleration

Performance Evaluation

- Use Dynamic Programming (DP) find the best schedule policy.
- Greatly improve device utilization and achieve up to 1.4x speedup over existing frameworks.





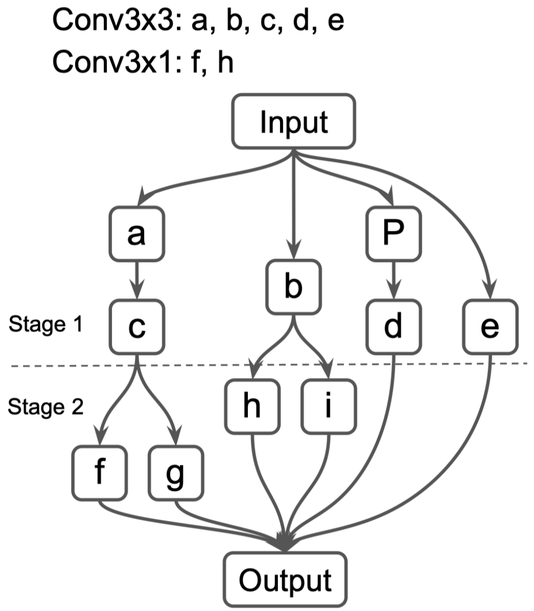


IOS: Inter-Operator Scheduler for CNN Acceleration

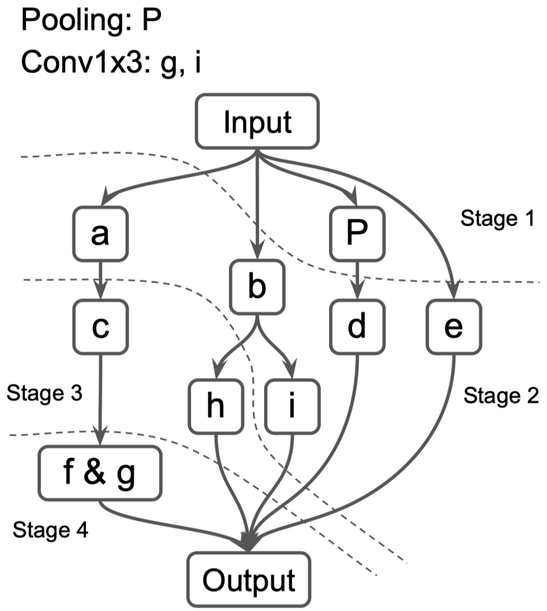
IOS Specializes for Different Settings

Specialization for Different Devices		Optimized for	
		K80	V100
Execute on	K80	**13.87**	14.65
	V100	4.49	**4.03**

Specialization for Different Batch Sizes		Optimized for		
		1	32	128
Execute on	1	**4.03**	4.50	4.63
	32	29.21	**27.44**	27.93
	128	105.98	103.74	**103.29**



(1) Schedule optimized for BS 1



(2) Schedule optimized for BS 32

Summary of Today's Lecture

Today we learned:

1. The background of federated learning.
2. Why gradient exchange is not always safe and rethink the safety of gradients.
3. The difficulty of developing efficient tensor programs.
4. Halide, a domain specific language for parallel computing.
5. TVM, a domain specific compiler for deep learning.
6. AutoTVM and Ansor to automatically learn an efficient schedule for tensor programs.
7. Graph optimization (Metaflow, IOS) to improve device utilization.

References

- MCUNet: Tiny Deep Learning on IoT Devices [Lin et al 2020]
- On-Device Training Under 256KB Memory [Lin et al. 2022]
- Communication-Efficient Learning of Deep Networks from Decentralized Data. [McMahan et al. 2016]
- Exploiting unintended feature leakage in collaborative learning. [Melis et al. 2018]
- Membership inference attacks against machine learning models. [Shokri et al. 2016]
- Deep Gradient Leakage from Gradients. [Zhu et al. 2019]
- Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines[Ragan-Kelley et al, 2012]
- Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines [Ragan-Kelley et al, 2013]