Efficient ConvBN Blocks for Transfer Learning and Beyond

Kaichao You, Guo Qin, Anchang Bao, Meng Cao, Ping Huang, Jiulong Shan, Mingsheng Long

International Conference on Learning Representations 2024 (Spotlight)

Tsinghua University and Apple · May 7, 2024

TL;DR;

- This method is a builtin functionality of PyTorch since 2.2.
- Try it out if you use torch.compile!
    - Save up to 40% GPU memory without modifying your model.
    - No accuracy loss.

```
from torch._inductor import config as inductor_config

inductor_config.efficient_conv_bn_eval_fx_passes = True

model = torch.compile(model)
```

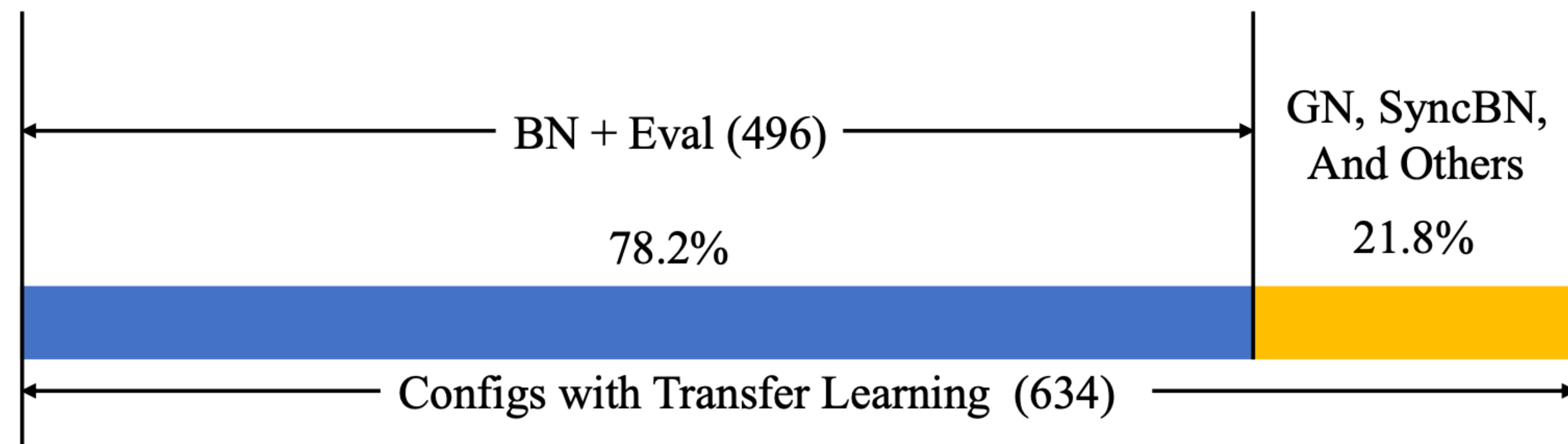
Agenda

- Introduction and background
- Method and theoretical analyses
- Application and experimental results
- Integration as PyTorch builtin functionality

Introduction and background

ConvBN blocks are popular in ConvNets

- When the model is deployed, BN will be merged into Conv
- When the model is fine-tuned, BN layers are often trained in Eval mode
  - e.g. 78.2% object detection training configs use this recipe



- Can we exploit such optimization in fine-tuning?

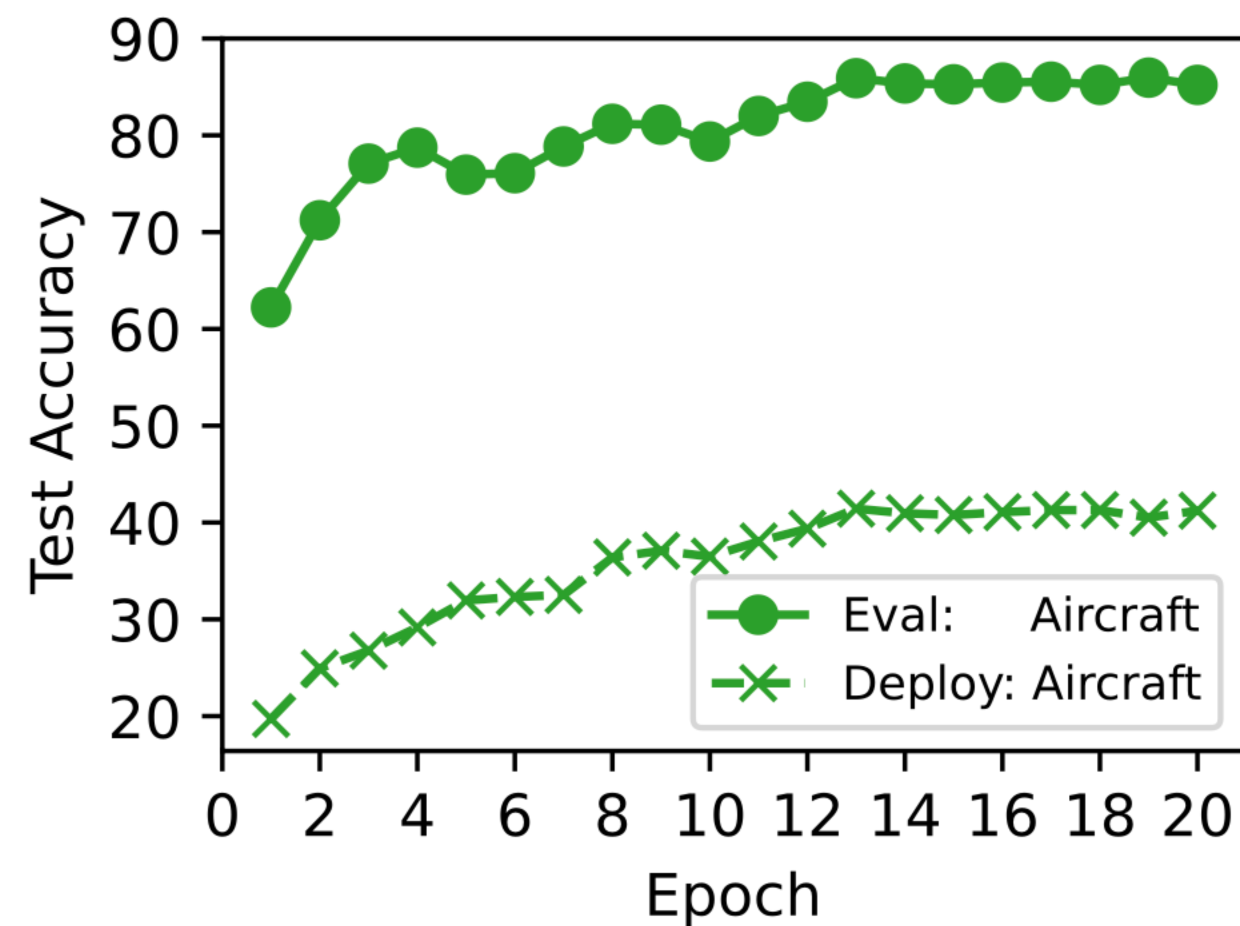
Method and theoretical analyses

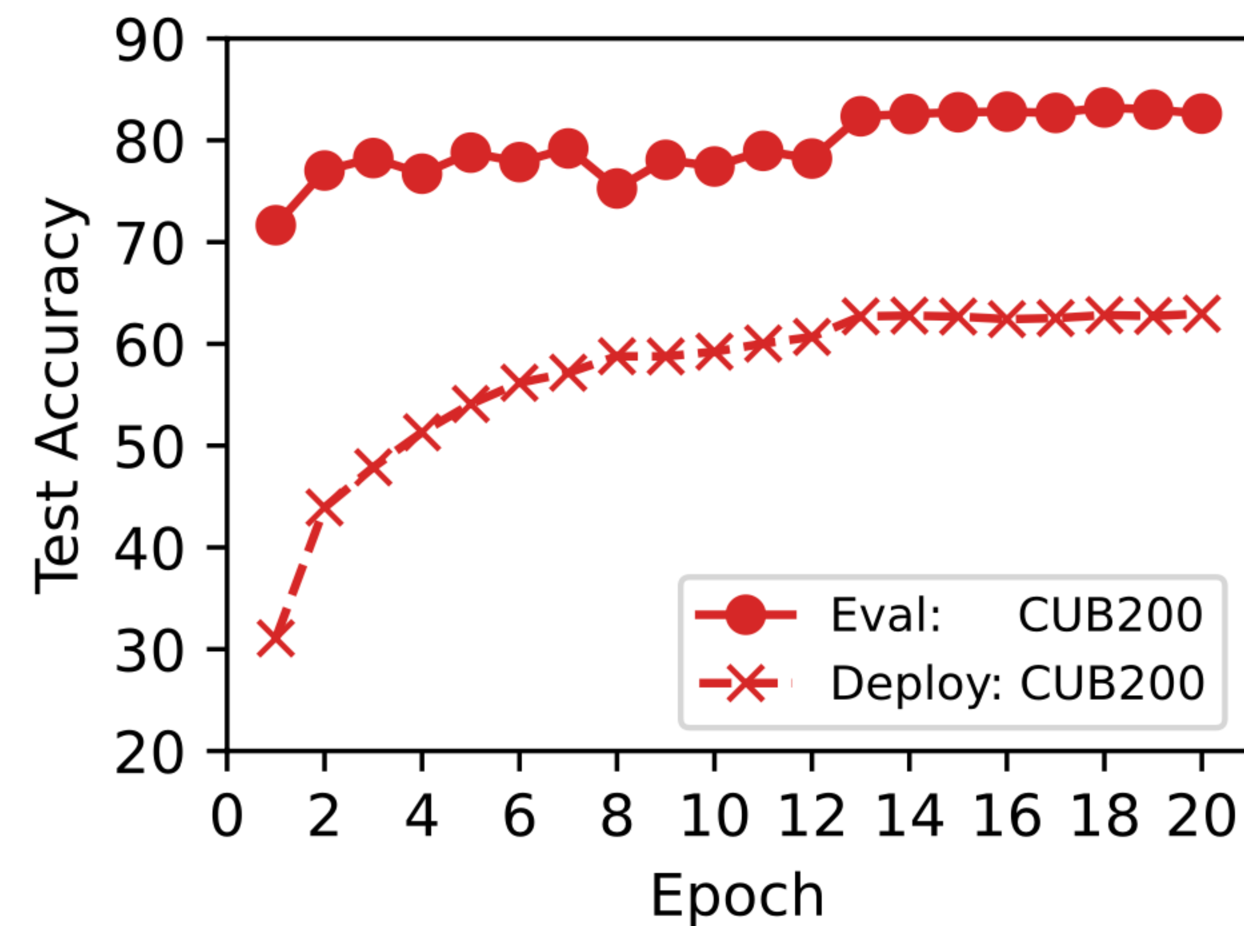
Eval Mode:

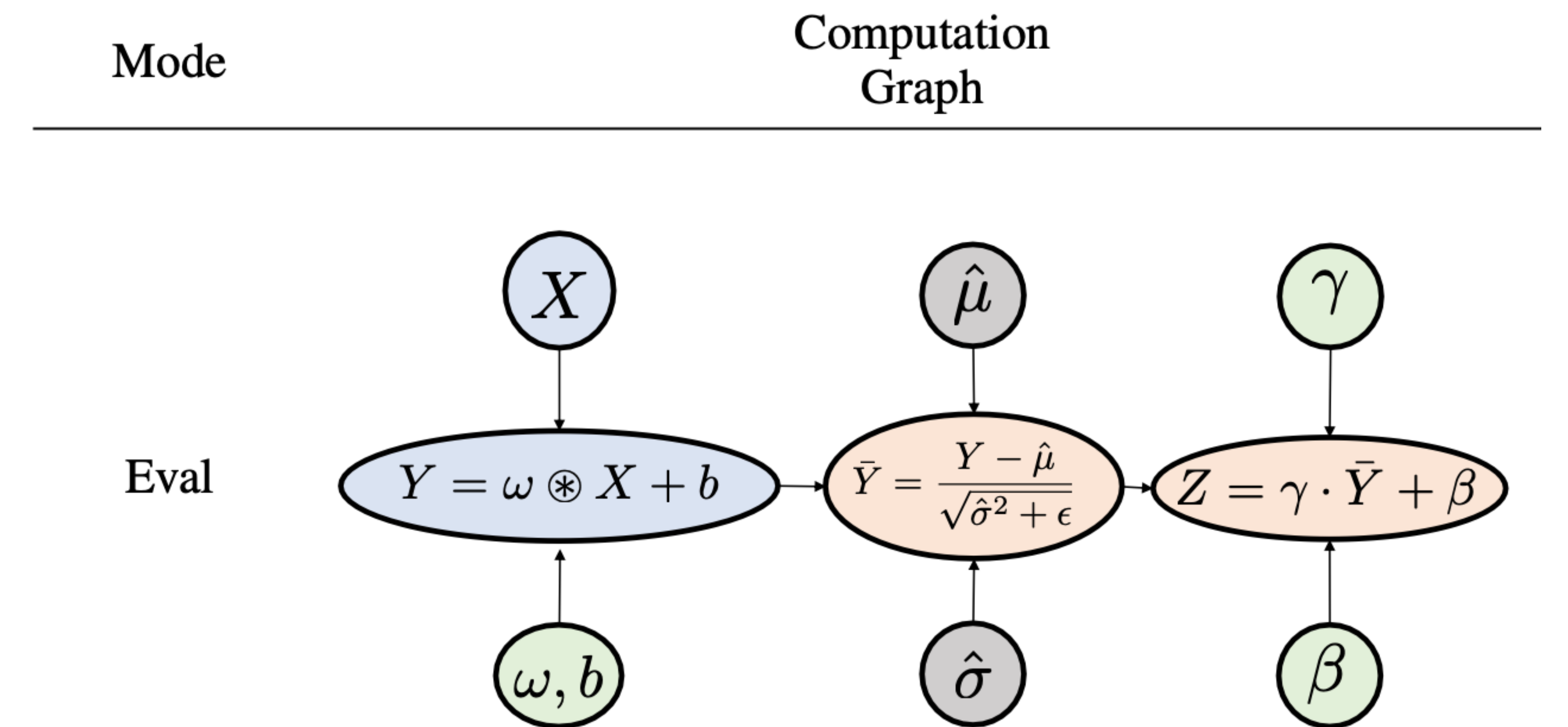
- Convolution + Normalization + Affine transformation

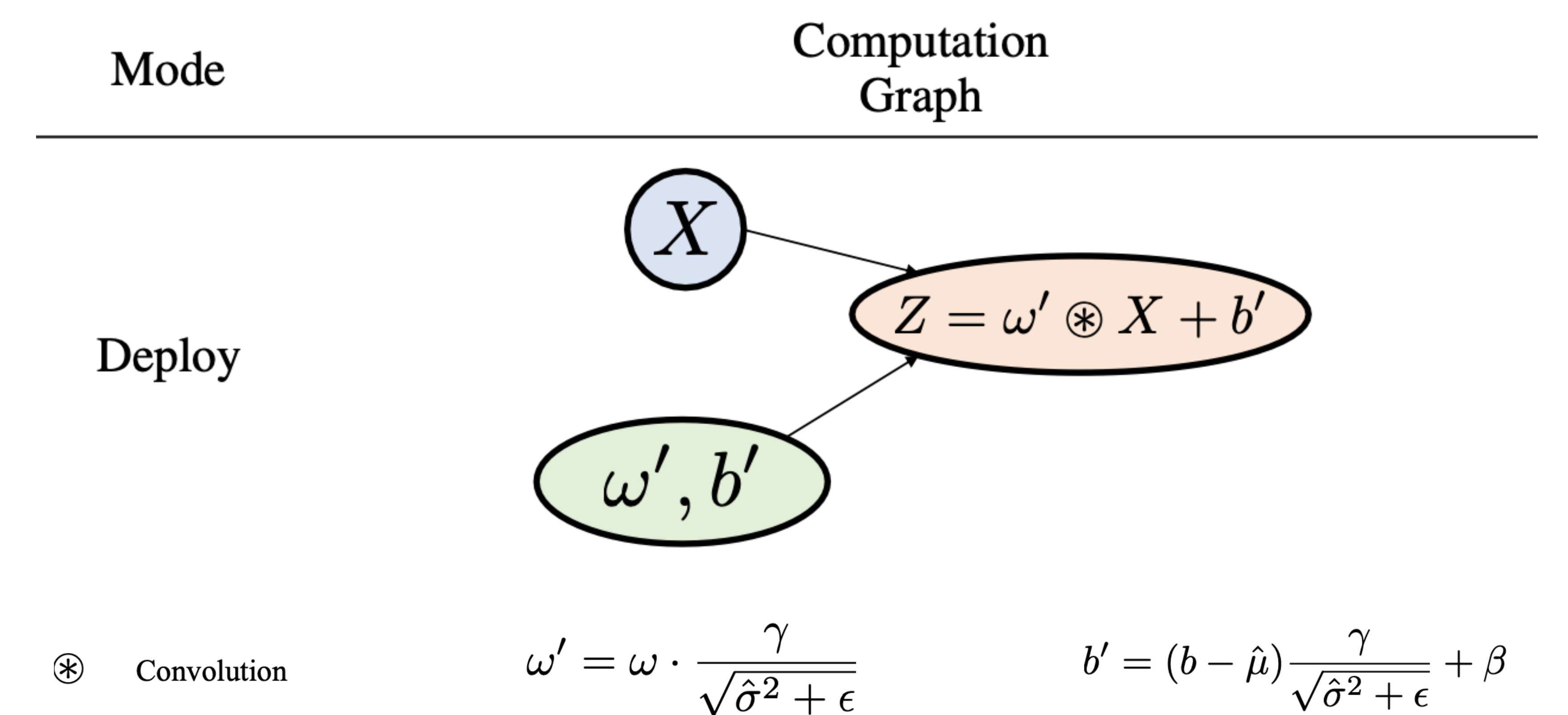
Deploy Mode:

- Only convolution, very fast and efficient
- ... but not stable for training





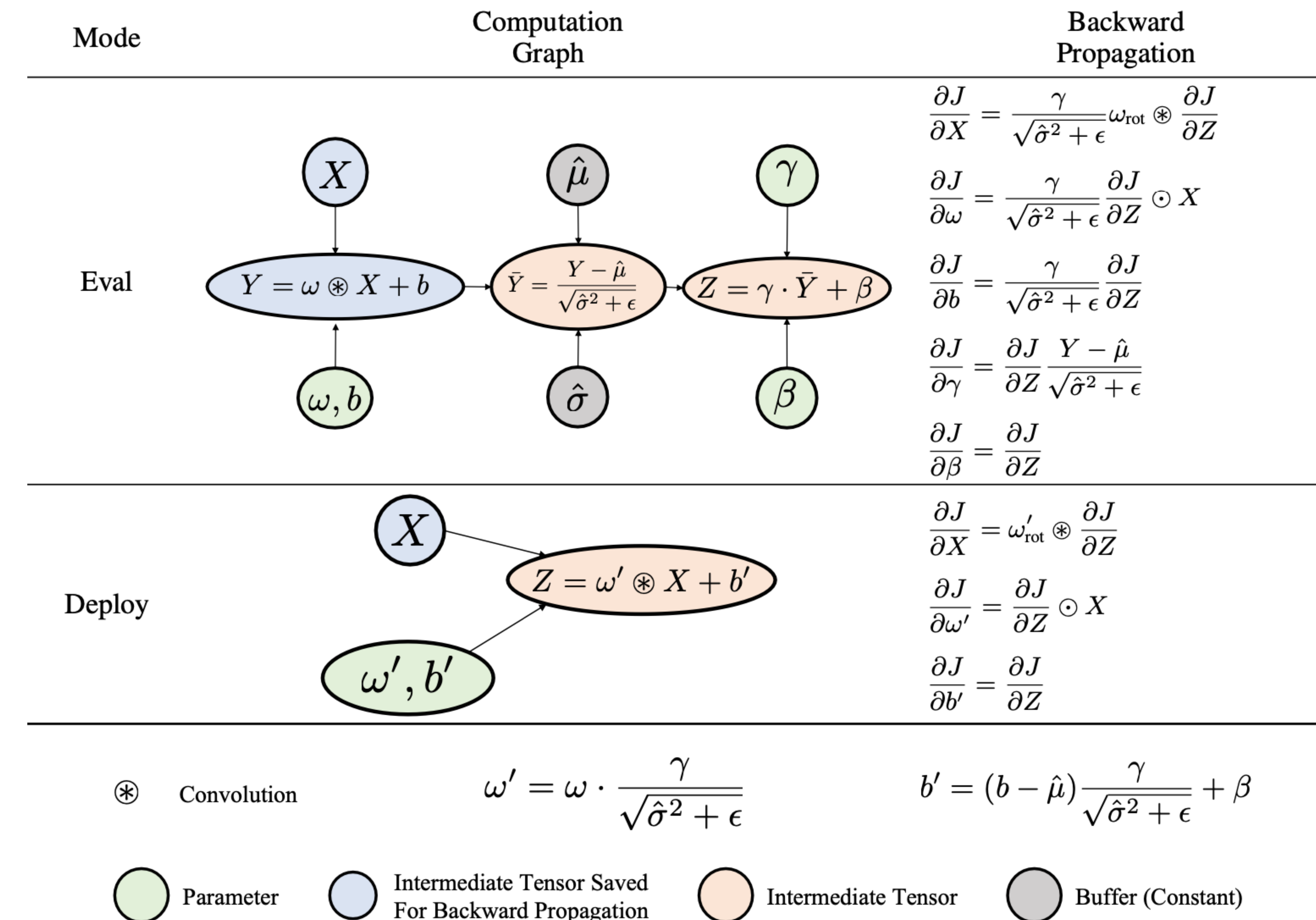




Method and theoretical analyses

What's the problem of training in Deploy mode?

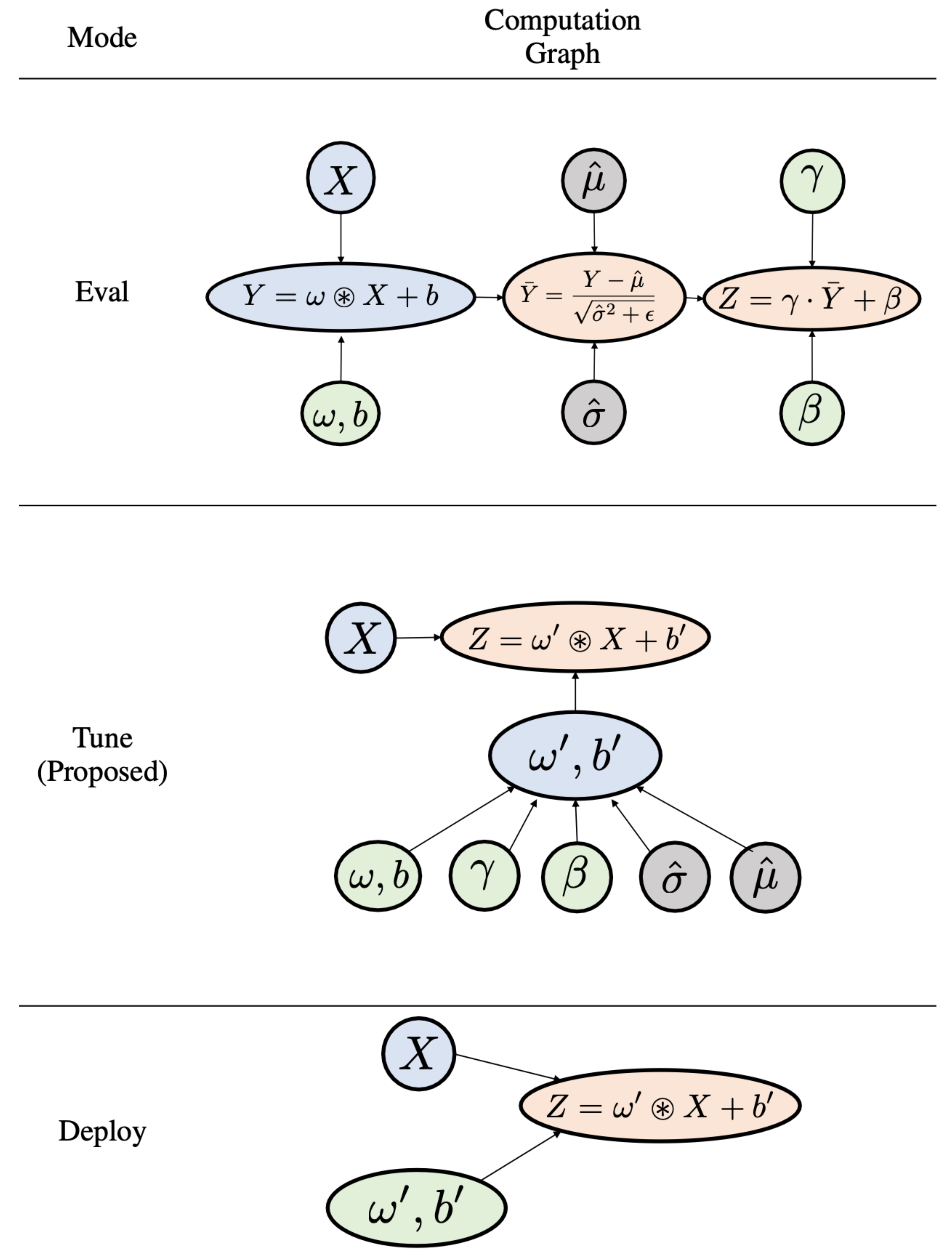
- Forward computation is equivalent with Eval
- But backward computation is different from Eval
- The inverse-scaling problem:
  - weight scaling $\omega' = \omega \cdot \frac{\gamma}{\sqrt{\hat{\sigma}^2 + \epsilon}}$
  - gradient inverse scaling $\frac{\partial J}{\partial \omega'} = \frac{\sqrt{\hat{\sigma}^2+\epsilon}}{\gamma} \frac{\partial J}{\partial \omega}$
- It really hurts the stability of training!
  - Weight scales to 0.1x, gradient scales to 10x



Method and theoretical analyses

Solution: A new Tune mode

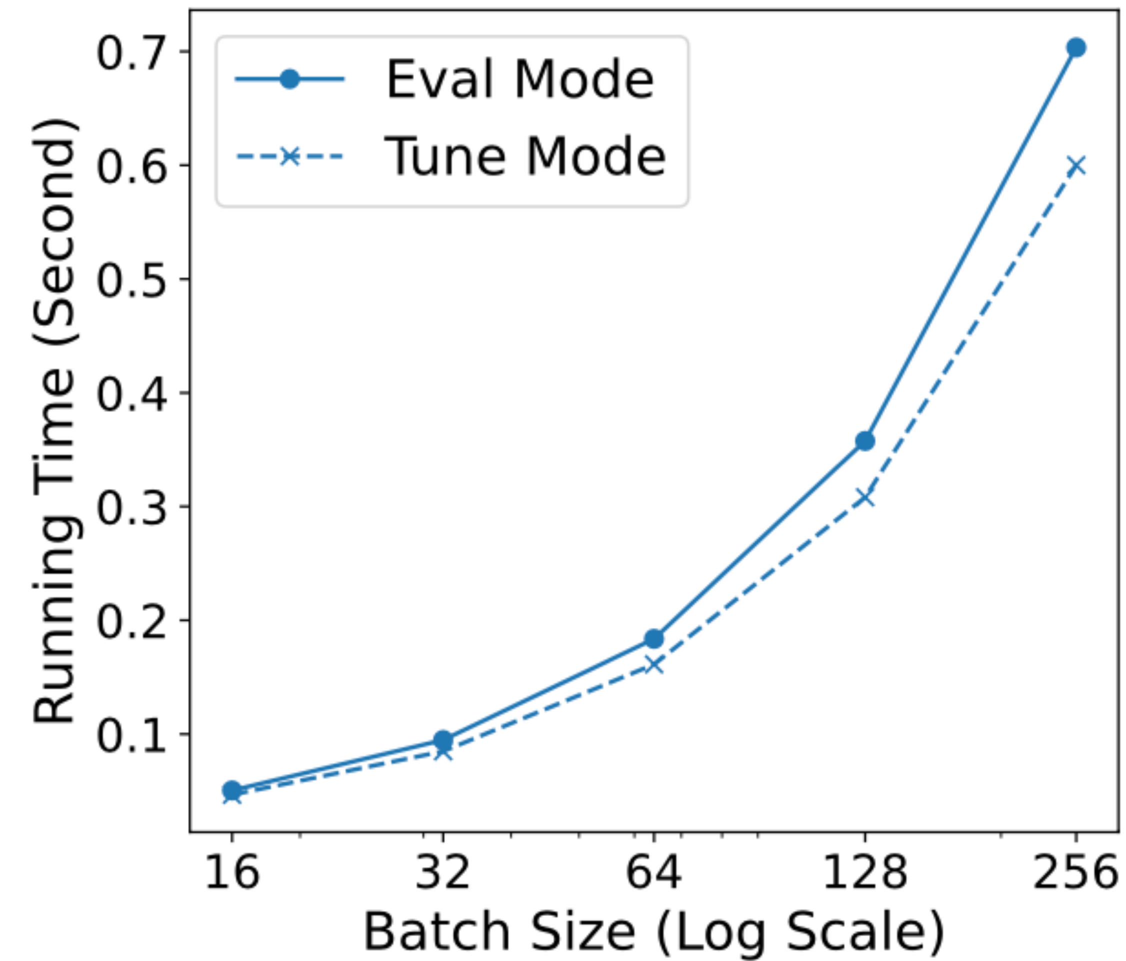
- Normalize and transform weight on the fly
- Equivalent with Eval mode
  - The same forward computation
  - The same backward computation
- More efficient than Eval mode
  - Less computation time and memory footprint
- More stable than Deploy mode
  - No more training stability problem

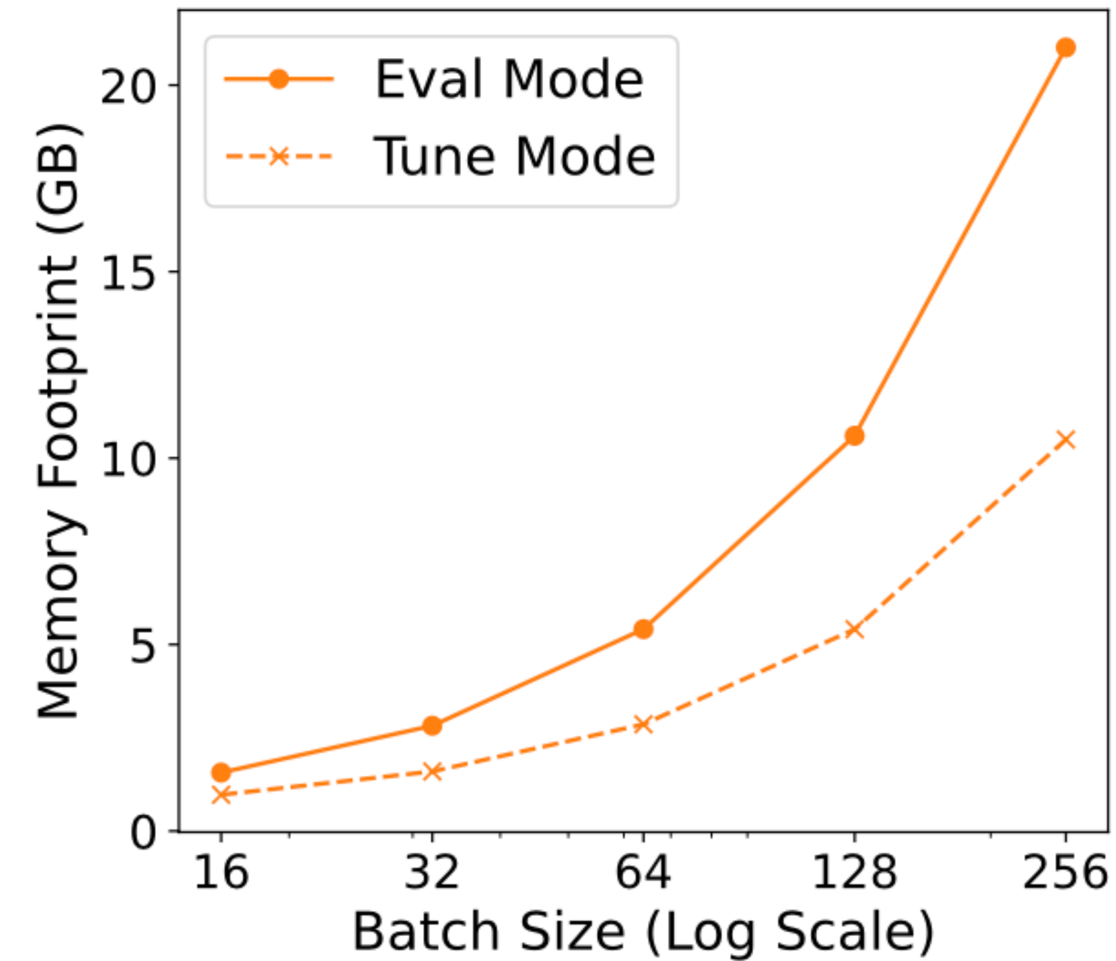


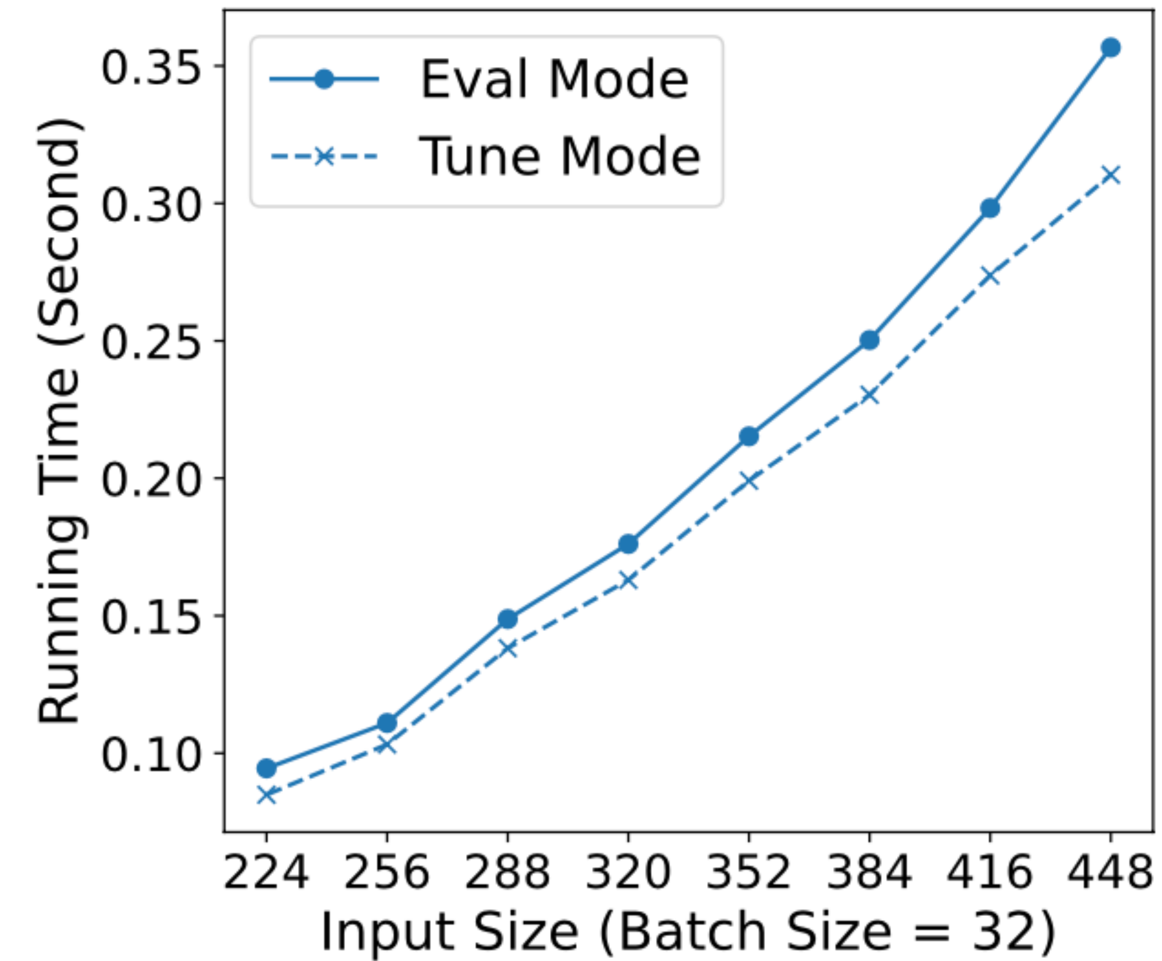
Application and experimental results

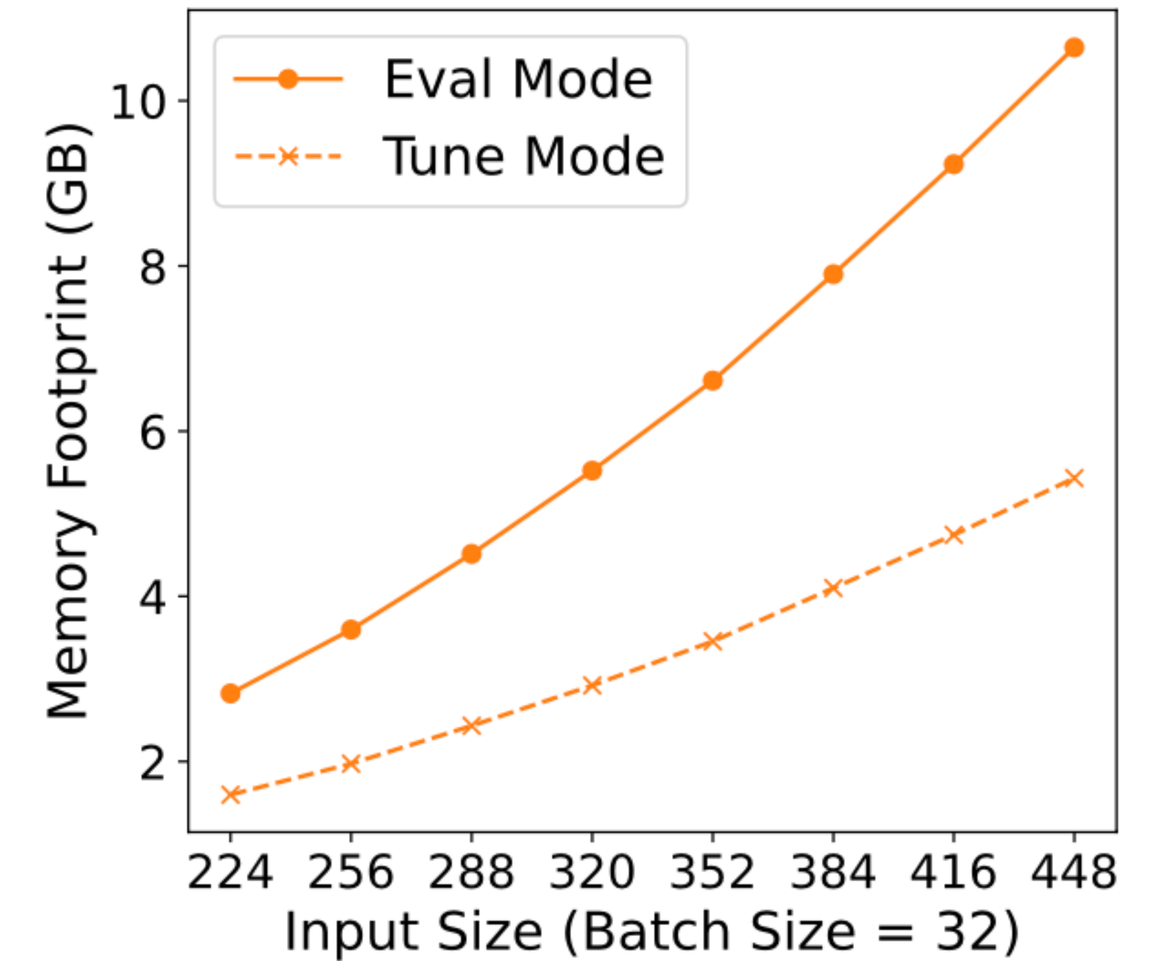
Efficiency comparison with Eval Mode:

- Tune mode uses 40% less memory footprint
- Tune mode takes 10% less computation time







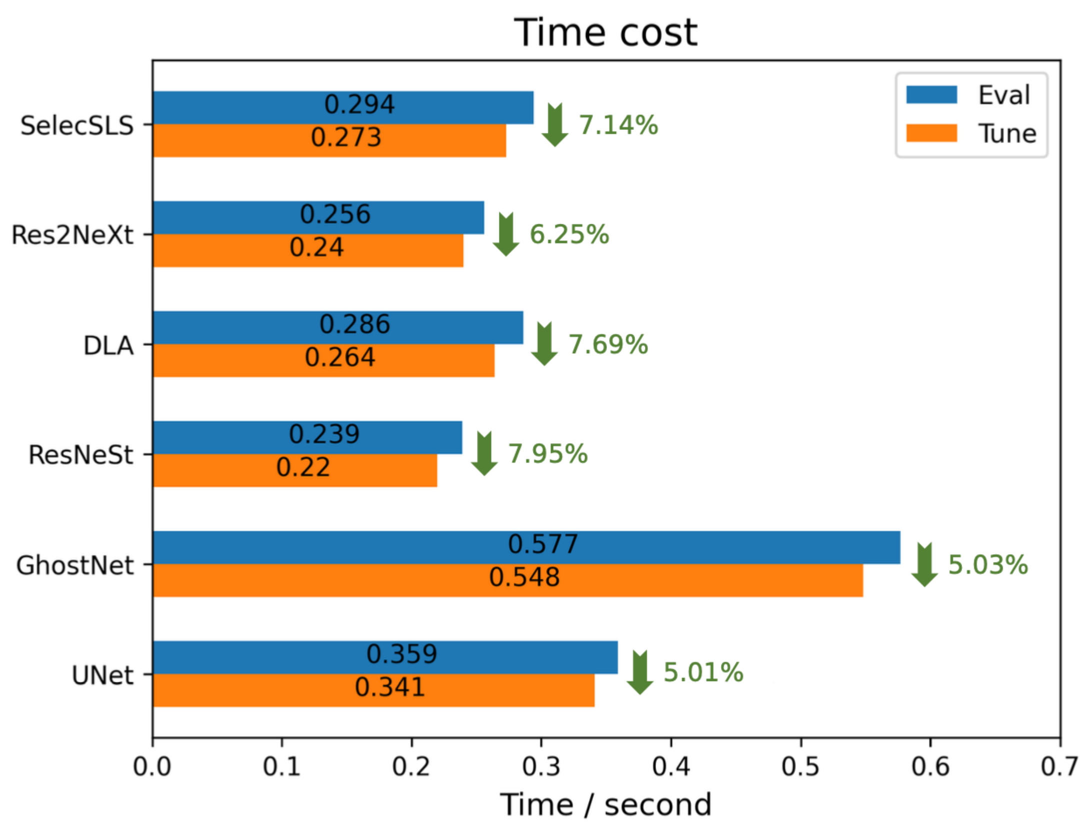


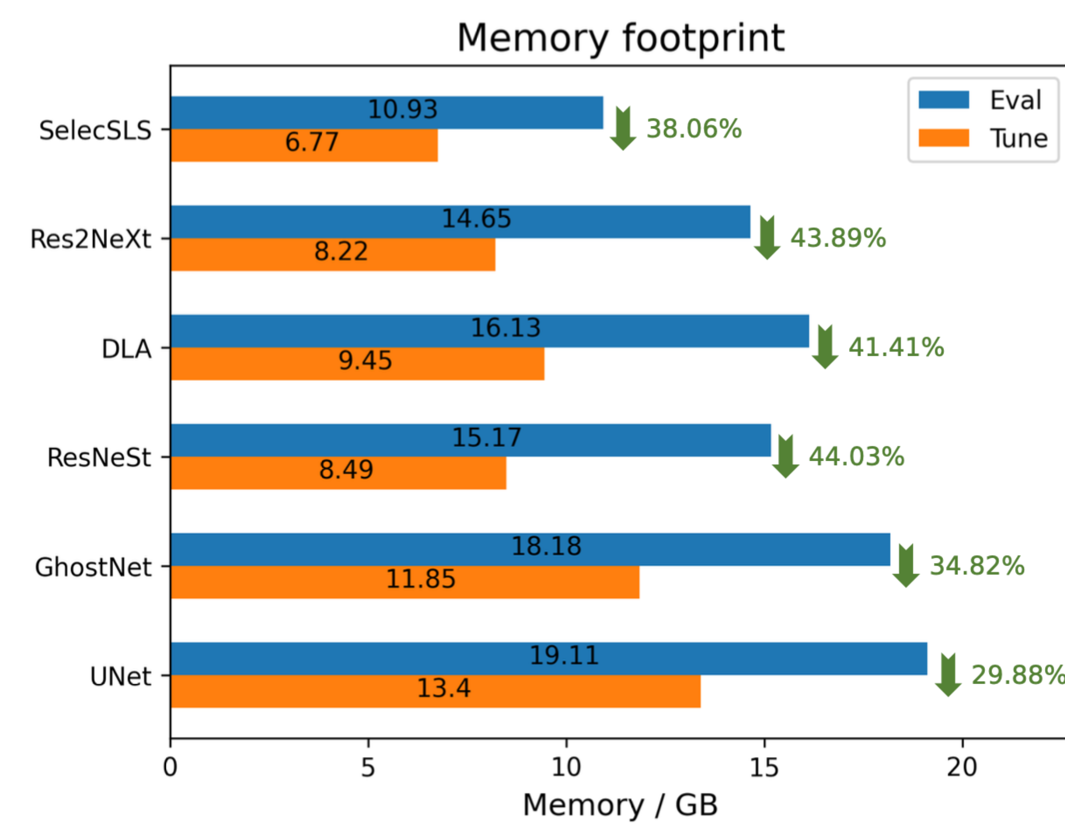
Application and experimental results

Application in classification, detection, and adversarial example generation

- Better efficiency without hurting accuracy of Eval mode

Dataset	mode	Accuracy	Memory (GB)	Time (second/iteration)
CUB-200	Eval Tune	82.62 (± 0.14) 83.20 (± 0.00)	19.499 12.323 (**36.80%**↓)	0.549 0.501 (**8.74%**↓)
Aircrafts	Eval Tune	85.21 (± 0.22) 85.90 (± 0.26)	19.497 12.321 (**36.81%**↓)	0.548 0.505 (**7.85%**↓)
Stanford Cars	Eval Tune	90.11 (± 0.03) 90.13 (± 0.12)	19.499 12.321 (**36.81%**↓)	0.541 0.491 (**9.24%**↓)





Detector	Backbone	BatchSize	Precision	mode	mAP	Memory (GB)
Faster RCNN	ResNet50	2	FP32	Eval Tune	0.3739 0.3728 (-0.0011)	3.857 3.003 (**22.15%**↓)
Mask RCNN	ResNet50	2	FP32	Eval Tune	0.3824 0.3825 (+0.0001)	4.329 3.470 (**19.85%**↓)
Mask RCNN	ResNet101	16	FP16	Eval Tune	0.3755 0.3756 (+0.0001)	13.687 9.980 (**27.08%**↓)
Retina Net	ResNet50	2	FP32	Eval Tune	0.3675 0.3647 (-0.0028)	3.631 2.774 (**23.59%**↓)
Faster RCNN	ResNet101	2	FP32	Eval Tune	0.3944 0.3921 (-0.0023)	5.781 4.183 (**27.65%**↓)
Faster RCNN	ResNext101	2	FP32	Eval Tune	0.4126 0.4131 (+0.0005)	6.980 4.773 (**31.62%**↓)
Faster RCNN	RegNet	2	FP32	Eval Tune	0.3985 0.3995 (+0.0010)	4.361 3.138 (**28.06%**↓)
Faster RCNN	HRNet	2	FP32	Eval Tune	0.4017 0.4031 (+0.0014)	8.504 5.463 (**35.76%**↓)
Faster RCNN	RepVGG	16	FP16	Eval Tune	0.3350 0.3350 (+0.0000)	15.794 8.996 (**43.04%**↓)

Integration as PyTorch builtin functionality

The idea is simple and works great, but how to implement it?

Not easy to find ConvBN blocks because PyTorch and Python are so dynamic

- Finding ConvBN blocks are labor-intensive
  - e.g. self.conv1 + self.bn1
  - e.g. self.conv2 + self.nested.0
  - e.g. self.nested.1 + self.wrapped.mod

```
class WrappedBatchNorm(nn.Module):
    def __init__(self):
        super().__init__()
        self.mod = nn.BatchNorm2d(1)
    def forward(self, x):
        return self.mod(x)

class M(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 1, 1)
        self.bn1 = nn.BatchNorm2d(1)
        self.conv2 = nn.Conv2d(1, 1, 1)
        self.nested = nn.Sequential(
            nn.BatchNorm2d(1),
            nn.Conv2d(1, 1, 1),
        )
        self.wrapped = WrappedBatchNorm()
```

```
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.conv2(x)
        x = self.nested(x)
        x = self.wrapped(x)
        return x

model = M()
```

Integration as PyTorch builtin functionality

The idea is simple and works great, but how to implement it?

Use deep learning compiler to automatically find ConvBN blocks!

- Now it is a PyTorch builtin functionality
- If you are using PyTorch≥2.2:
  - Use torch.compile
  - Turn on a config switch
  - ConvBN blocks will be automatically optimized!

```python
from torch._inductor import config as inductor_config
inductor_config.efficient_conv_bn_eval_fx_passes = True

model = torch.compile(model)
```

```python
class WrappedBatchNorm(nn.Module):
    def __init__(self):
        super().__init__()
        self.mod = nn.BatchNorm2d(1)
    def forward(self, x):
        return self.mod(x)

class M(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 1, 1)
        self.bn1 = nn.BatchNorm2d(1)
        self.conv2 = nn.Conv2d(1, 1, 1)
        self.nested = nn.Sequential(
            nn.BatchNorm2d(1),
            nn.Conv2d(1, 1, 1),
        )
        self.wrapped = WrappedBatchNorm()
```

```python
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.conv2(x)
        x = self.nested(x)
        x = self.wrapped(x)
        return x

model = M()
```

Conclusions

- Out-of-the-box acceleration within torch.compile
- Better efficiency without hurting accuracy of Eval mode.

More testimonials:

https://github.com/open-mmlab/mmengine/discussions/1252

I used the maskrcnn model as a baseline to test the difference of turning on this function. When the model first started training, it could indeed save 13% of the memory.

testing shows that it can save 33.8% memory

It reduced the memory from 4576Mb to 2623Mb

Questions or comments?

youkaichao@gmail.com