lab2_report

March 5, 2026

1 Lab 2 Report — ResNet-18 on CIFAR10

ECE-GY 9143 HPML — Spring 2026

- Bharath Kumar Kakumani
- BK2991

Run command: `source venv/bin/activate && python lab2.py --cuda --task all && python lab2_torchscript.py --cuda`

---

2 Part A: Training and Profiling

2.1 C2: Time Measurement (10 pts)

Per-epoch breakdown: data-loading time (C2.1), training/mini-batch time (C2.2), total time (C2.3).

```
Device: cuda
ResNet-18 params: 11,173,962

============================================================
 C1/C2: Training - 5 epochs
 optimizer=sgd  workers=4  bn=on  device=cuda
============================================================
 Ep     Loss    Acc%  Data(s)  Train(s)  Total(s)
  1   1.8894  31.12%     0.65     13.41     15.77
  2   1.3499  50.39%     1.01     10.92     13.18
  3   1.0827  61.14%     0.87     11.63     13.88
  4   0.8919  68.36%     0.61     12.90     15.23
  5   0.7512  73.83%     0.65     12.94     15.28

[Q3] Trainable params : 11,173,962
[Q3] Params w/ grads  : 11,173,962
[Q3] Optimizer states : 62
```

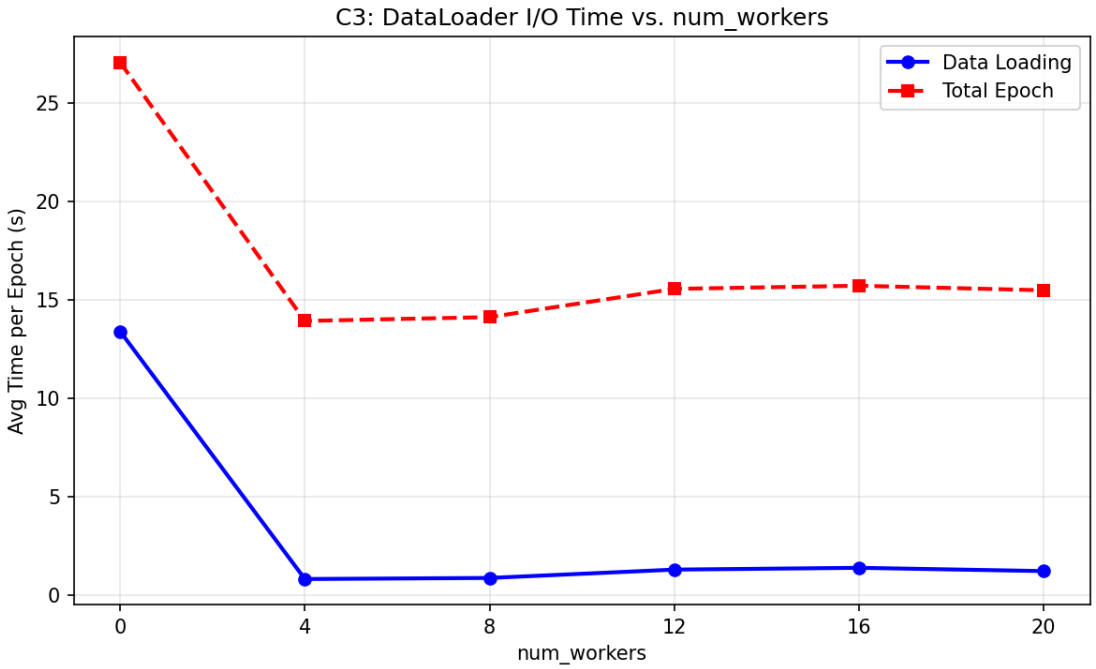
2.2 C3: I/O Optimization (10 pts)

Sweep `num_workers` from 0 to 20 in steps of 4, measuring avg data-loading and total time per epoch over 5 epochs.

```
============================================================
 C3: I/O Optimization - Worker Sweep (5 epochs each)
============================================================
 Workers  AvgData(s)  AvgTrain(s)  AvgTotal(s)
       0      13.396       12.060       27.086
       4       0.797       11.638       13.928
       8       0.852       11.728       14.123
      12       1.276       12.501       15.560
      16       1.367       12.548       15.719
      20       1.201       12.477       15.488

C3.2 => Best num_workers = 4 (avg total: 13.928s, avg data: 0.797s)
```

C3.1 — Graph:



C3.2 — Best num_workers = 4. Going from 0→4 workers cuts data-loading from 13.4s to 0.8s (prefetching overlaps with GPU compute). Beyond 4, overhead from extra worker processes outweighs the benefit.

2.3 C4: Training on GPUs vs CPUs (10 pts)

Average running time over 5 epochs with `num_workers=4`.

```
============================================================
```

```
 C4: GPU vs CPU
============================================================

  [CPU] Training 5 epochs...
    Ep 1: loss=1.9320  acc=30.70%  time=579.23s
    Ep 2: loss=1.4355  acc=47.51%  time=431.86s
    Ep 3: loss=1.1419  acc=59.25%  time=432.04s
    Ep 4: loss=0.9589  acc=66.19%  time=429.39s
    Ep 5: loss=0.8158  acc=71.12%  time=467.04s
  [CPU] Avg epoch time: 467.91s

  [CUDA] Training 5 epochs...
    Ep 1: loss=1.9007  acc=30.89%  time=7.07s
    Ep 2: loss=1.4480  acc=46.87%  time=6.49s
    Ep 3: loss=1.2061  acc=56.13%  time=6.50s
    Ep 4: loss=0.9932  acc=64.87%  time=6.56s
    Ep 5: loss=0.8537  acc=69.84%  time=6.52s
  [CUDA] Avg epoch time: 6.63s

============================================================
 C4 Summary: GPU vs CPU (workers=4)
============================================================
  CPU avg: 467.91s/epoch
  GPU avg: 6.63s/epoch
  GPU speedup: 70.6x
```

Device	Avg Epoch Time	Speedup
CPU	467.91 s	1x
GPU	6.63 s	**70.6x**

2.4 C5: Experimenting with Different Optimizers (10 pts)

GPU-enabled, `num_workers=4`, same hyperparams: `lr=0.1, weight_decay=5e-4, momentum=0.9`.

```
  [SGD]
   Ep     Loss    Acc%  Train(s)  Total(s)
    1   1.9082  31.52%      5.80      6.82
    2   1.4078  48.34%      5.37      6.42
    3   1.1673  57.93%      5.37      6.40
    4   0.9827  65.60%      5.38      6.38
    5   0.8566  69.93%      5.38      6.39

  [SGD_NESTEROV]
   Ep     Loss    Acc%  Train(s)  Total(s)
    1   1.9148  31.59%      5.42      6.46
    2   1.3372  51.10%      5.41      6.44
    3   1.0420  62.62%      5.42      6.51
```

```
   4   0.8773  69.03%      5.43      6.50
   5   0.7533  73.69%      5.42      6.47

  [ADAM]
   Ep     Loss    Acc%  Train(s)  Total(s)
    1   2.2094  20.82%      5.55      6.62
    2   1.8813  27.24%      5.52      6.61
    3   1.8430  28.43%      5.52      6.60
    4   1.8157  30.26%      5.52      6.59
    5   1.8078  30.47%      5.54      6.60

============================================================
 C5 Summary: Optimizer Comparison (workers=4)
============================================================
  Optimizer        AvgLoss  AvgAcc%  AvgTrain(s)
  sgd               1.2645   54.66%         5.46
  sgd_nesterov      1.1849   57.61%         5.42
  adam              1.9114   27.44%         5.53
```

Optimizer	Avg Loss	Avg Acc	Avg Train (s)
SGD	1.2645	54.66%	5.46
SGD + Nesterov	1.1849	57.61%	5.42
Adam	1.9114	27.44%	5.53

SGD+Nesterov converges fastest thanks to its look-ahead gradient. Adam performs poorly because `lr=0.1` is far too high for its adaptive step sizes (its default is 1e-3).

2.5 C6: Experimenting without Batch Norm (5 pts)

GPU-enabled, `num_workers=4`, SGD optimizer, all BatchNorm layers replaced by `nn.Identity()`.

```
============================================================
 C6: Without Batch Norm - 5 epochs (SGD, workers=4)
============================================================
 Ep     Loss    Acc%  Train(s)  Total(s)
  1   1.9335  26.70%      4.63      5.62
  2   1.5540  42.90%      4.21      5.28
  3   1.3556  51.21%      4.22      5.26
  4   1.1640  58.84%      4.21      5.18
  5   1.0096  64.68%      4.21      5.19

C6 Summary => avg loss: 1.4034, avg acc: 48.87%
```

Metric	With BN (C5 SGD)	Without BN
Avg Loss	1.2645	1.4034
Avg Acc	54.66%	48.87%

Without batch normalization, convergence is slower and accuracy drops ~6%. Training is slightly faster per-epoch (fewer ops) but less effective.

---

3 Part B: TorchScript (C7–C10)

3.1 C7: Convert to TorchScript (5 pts)

Trained ResNet-18 for 5 epochs, converted via `torch.jit.script`, saved and reloaded.

```
Training 5 epochs before scripting...
  Ep 1: loss=1.8304  acc=32.97%
  Ep 2: loss=1.3699  acc=49.58%
  Ep 3: loss=1.0921  acc=60.89%
  Ep 4: loss=0.9062  acc=67.85%
  Ep 5: loss=0.7824  acc=72.45%

C7: Scripted model saved -> resnet18_scripted.pt
C7: Save/load verification - max diff: 0.0e+00
```

Model successfully scripted, saved to `resnet18_scripted.pt`, reloaded with `torch.jit.load`, and verified (zero output difference).

3.2 C8: Print Model Graph (5 pts)

```
graph(%self.1 : __torch__.lab2.ResNet,
      %x.1 : Tensor):
  %42 : int = prim::Constant[value=-1]()
  %13 : Function = prim::Constant[name="relu"]()
  %12 : bool = prim::Constant[value=0]()
  %41 : int = prim::Constant[value=1]()
  %bn1.1  = prim::GetAttr[name="bn1"](%self.1)
  %conv1.1 = prim::GetAttr[name="conv1"](%self.1)
  %9  : Tensor = prim::CallMethod[name="forward"](%conv1.1, %x.1)
  %10 : Tensor = prim::CallMethod[name="forward"](%bn1.1, %9)
  %x0.1 : Tensor = prim::CallFunction(%13, %10, %12)      # relu
  %layer1.1 = prim::GetAttr[name="layer1"](%self.1)
  %x1.1 : Tensor = prim::CallMethod[name="forward"](%layer1.1, %x0.1)
  %layer2.1 = prim::GetAttr[name="layer2"](%self.1)
  %x2.1 : Tensor = prim::CallMethod[name="forward"](%layer2.1, %x1.1)
  %layer3.1 = prim::GetAttr[name="layer3"](%self.1)
  %x3.1 : Tensor = prim::CallMethod[name="forward"](%layer3.1, %x2.1)
  %layer4.1 = prim::GetAttr[name="layer4"](%self.1)
  %x4.1 : Tensor = prim::CallMethod[name="forward"](%layer4.1, %x3.1)
  %avgpool.1 = prim::GetAttr[name="avgpool"](%self.1)
  %x5.1 : Tensor = prim::CallMethod[name="forward"](%avgpool.1, %x4.1)
  %x6.1 : Tensor = aten::flatten(%x5.1, %41, %42)
  %fc.1 = prim::GetAttr[name="fc"](%self.1)
  %48 : Tensor = prim::CallMethod[name="forward"](%fc.1, %x6.1)
```

```
  return (%48)
```

The graph shows the sequential flow: `conv1 → bn1 → relu → layer1 → layer2 → layer3 → layer4 → avgpool → flatten → fc`.

3.3 C9: Evaluate the TorchScript Model (5 pts)

Evaluated on CIFAR10 test set (10,000 images).

```
============================================================
 C9: Test Set Accuracy
============================================================
  PyTorch model:     70.94%
  TorchScript model: 70.94%
```

Both models produce **identical accuracy (70.94%)**, confirming scripting preserves model semantics exactly.

3.4 C10: Latency Comparison (10 pts)

Single-image inference latency (averaged over 200 runs with 50 warmup iterations).

```
============================================================
 C10: Latency Comparison (single image, ms)
============================================================
                    CPU (ms)   GPU (ms)
  PyTorch              12.17       2.03
  TorchScript          11.22       1.26
  CPU speedup: 1.08x
  CUDA speedup: 1.61x
```

	Latency on CPU (ms)	Latency on GPU (ms)
PyTorch	12.17	2.03
TorchScript	11.22	1.26
Speedup	**1.08x**	**1.61x**

TorchScript provides a **1.61x GPU speedup** through graph-level optimizations like operator fusion and memory planning. CPU speedup is more modest (1.08x) since CPU inference is already compute-bound with less room for kernel fusion gains.

---

4 Questions (Q1–Q6)

4.1 Q1: How many convolutional layers are in the ResNet-18 model? (4 pts)

20 convolutional layers total.

Breakdown: - **1** initial conv layer (`conv1`: 3→64, 3×3) - **16** convs inside 8 BasicBlocks (4 subgroups × 2 blocks × 2 convs each) - **3** shortcut 1×1 convs in subgroups 2, 3, 4 (needed when stride=2 changes spatial dims or channel count changes)

Component	Count
Initial conv	1
layer1 (64ch, stride=1) — 2 blocks, no shortcuts needed	4
layer2 (128ch, stride=2) — 2 blocks + 1 shortcut	5
layer3 (256ch, stride=2) — 2 blocks + 1 shortcut	5
layer4 (512ch, stride=2) — 2 blocks + 1 shortcut	5
Total	**20**

4.2 Q2: What is the input dimension of the last linear layer? (4 pts)

512.

After `layer4`, the feature map is 512 channels × 4×4 spatial. The `AdaptiveAvgPool2d(1,1)` collapses the spatial dims to 1×1, giving a 512-dim vector after flattening. So `fc = Linear(512, 10)`.

4.3 Q3: Trainable parameters and gradients — SGD optimizer (8 pts)

Trainable parameters: 11,173,962 Gradients: 11,173,962

Every trainable parameter gets a gradient after one backward pass. The counts match because all parameters in the model have `requires_grad=True` and none are frozen.

Code below (also in `lab2.py` under `count_parameters`):

[1]:
```
import torch
import torch.nn as nn
from lab2 import resnet18, get_optimizer

model = resnet18(use_bn=True)

# need one fwd+bwd pass so .grad tensors get populated
optimizer = get_optimizer('sgd', model.parameters())
criterion = nn.CrossEntropyLoss()
dummy_input = torch.randn(1, 3, 32, 32)
loss = criterion(model(dummy_input), torch.tensor([0]))
loss.backward()

trainable = sum(p.numel() for p in model.parameters() if p.requires_grad)
grad_count = sum(p.numel() for p in model.parameters() if p.grad is not None)

print(f"[Q3 - SGD]")
print(f"Trainable parameters: {trainable:,}")
print(f"Gradients:            {grad_count:,}")
```

```
[W305 23:05:54.050181651 NNPACK.cpp:61] Could not initialize NNPACK! Reason:
Unsupported hardware.
```

```
[Q3 - SGD]
Trainable parameters: 11,173,962
Gradients:            11,173,962
```

4.4 Q4: Trainable parameters and gradients — Adam optimizer (4 pts)

Trainable parameters: 11,173,962 Gradients: 11,173,962

Same as SGD — the optimizer choice doesn't change the model architecture or which parameters are trainable. The gradient count is also identical since every parameter still gets a gradient via backprop.

The key difference is in **optimizer state memory**: Adam keeps 2 running averages per parameter (first moment `m` and second moment `v`), so it uses ~3× the parameter memory compared to SGD's single momentum buffer. For our 11.17M params, Adam stores ~186 state entries (93 parameter groups × 2 states each) vs SGD's 62 momentum buffers.

4.5 Q5: Differences between tracing and scripting in TorchScript (3 pts)

Tracing (`torch.jit.trace`) runs the model once with a concrete sample input and records every operation that fires. The result is a static graph that faithfully replays that exact execution path. The upside is it works with almost any PyTorch code (even non-standard Python); the downside is it *cannot* capture data-dependent control flow — any `if/else` or variable-length loop that depends on runtime tensor values gets baked to whatever the sample triggered.

Scripting (`torch.jit.script`) statically analyzes the Python source code and compiles it into TorchScript's intermediate representation (IR). It preserves control flow — `if`, `for`, `while` based on tensor values all survive into the graph. The tradeoff is the code must obey TorchScript's subset of Python: explicit type annotations, no unsupported builtins, no dynamic features like `eval()` or arbitrary Python objects.

	Tracing	Scripting
Mechanism	Records ops from a sample run	Parses & compiles source code
Control flow	Flattened (only one path captured)	Fully preserved
Code requirements	Almost anything that runs	Must be TorchScript-compatible
Typical use	Models with no/minimal branching	Models with conditionals or loops

4.6 Q6: Changes needed in ResNet-18 for scripting (2 pts)

For our CIFAR10 ResNet-18, **no changes are needed**. The model scripts successfully with `torch.jit.script` as-is because:

1. The `forward()` methods in both `BasicBlock` and `ResNet` only use standard PyTorch operations (convolutions, batch norm, ReLU, pooling, flatten) with no data-dependent control flow.
2. The conditional logic (`if use_bn`, `if stride != 1`) only runs during `__init__` (construction time), not during `forward`. TorchScript only compiles `forward` and methods called from it, so construction-time branching is irrelevant.

3. `nn.Sequential`, `nn.Identity`, and `torch.flatten` are all supported in TorchScript.

In general, changes you *might* need for other models include: - Adding type annotations to helper methods called during forward - Replacing unsupported Python constructs (e.g., `dict` iteration, `zip`, certain list comprehensions) - Using `torch.jit.export` to expose additional methods beyond `forward` - Avoiding `None` return types without explicit `Optional` annotations