**PART 1** CPU vs GPU Model Training

# Step 0: Setup

!

pip install torch torchvision --quiet

import

torch

import

torchvision

print

(

"Torch version:"

,

torch.

\_\_version\_\_

)

print

(

"CUDA available?"

,

torch.cuda.is\_available

(

)

)

if

torch.cuda.is\_available

(

)

:

print

(

"GPU name:"

,

torch.cuda.get\_device\_name

(

0

)

)

else

:

print

(

"

⚠

No GPU detected. In Colab, go to: Runtime >

Change runtime type > Hardware accelerator > GPU"

)

Torch version: 2.8.0+cu126

CUDA available? True

GPU name: Tesla T4

def

train\_one\_epoch

(

model

,

loader

,

device

,

optimizer

,

criterion

)

:

model.train

(

)

if

device.

type

==

"cuda"

:

torch.cuda.synchronize

(

)

t0 = time.perf\_counter

(

)

running\_loss =

0.0

for

xb

,

yb

in

loader

:

xb

,

yb = xb.to

(

device

,

non\_blocking=

True

)

,

yb.to

(

device

,

non\_blocking=

True

)

optimizer.zero\_grad

(

)

out = model

(

xb

)

loss = criterion

(

out

,

yb

)

loss.backward

(

)

optimizer.step

(

)

running\_loss += loss.item

(

)

\* xb.size

(

0

)

if

device.

type

==

"cuda"

:

torch.cuda.synchronize

(

)

t1 = time.perf\_counter

(

)

return

(

t1 - t0

)

,

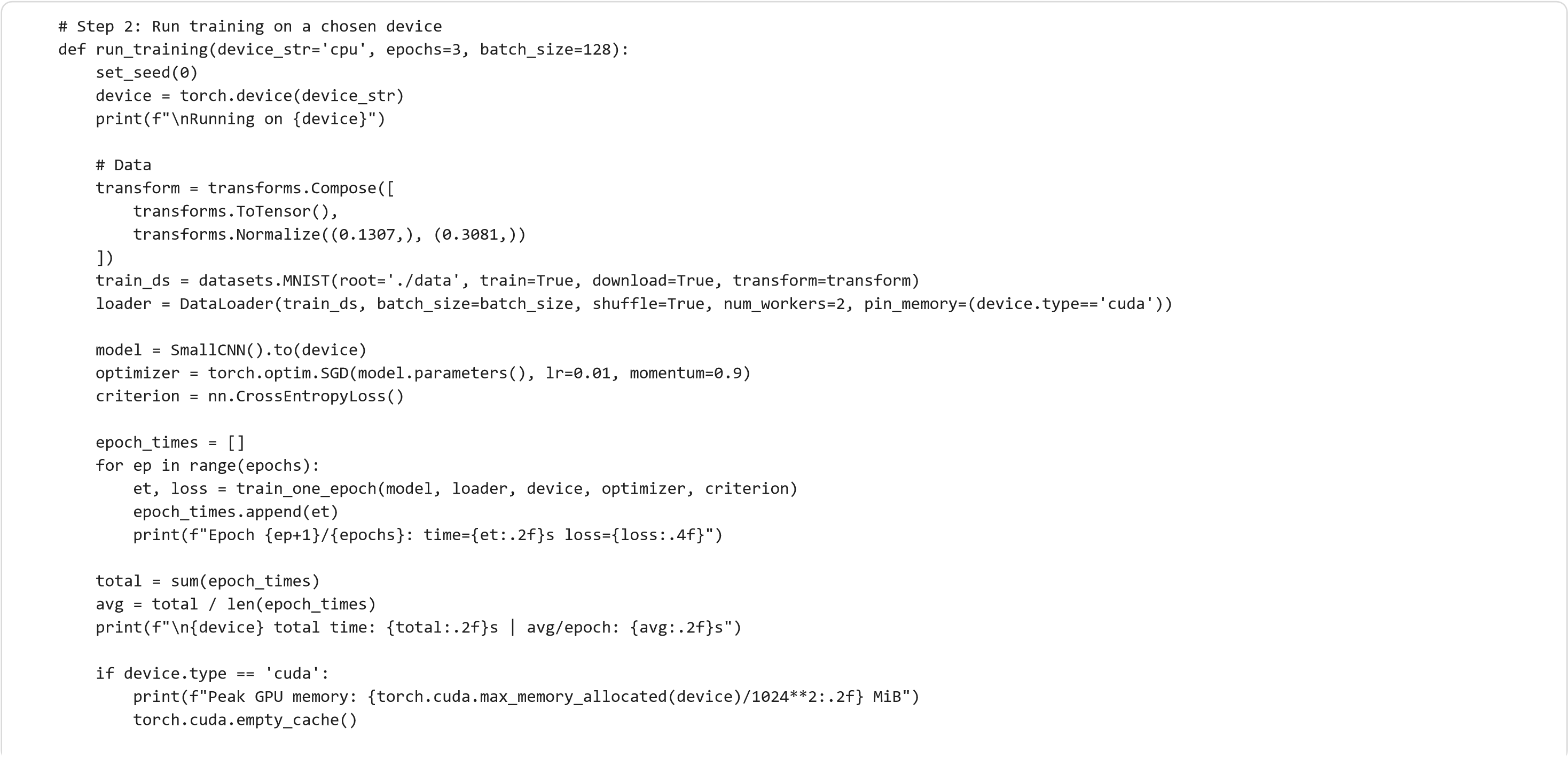
running\_loss /

len

(

loader.dataset

)



return

total

# Step 3: Run CPU vs GPU and compare

cpu\_time = run\_training

(

'cpu'

,

epochs=

3

,

batch\_size=

128

)

gpu\_time =

None

if

torch.cuda.is\_available

():

gpu\_time = run\_training

(

'cuda'

,

epochs=

3

,

batch\_size=

128

)

if

gpu\_time

:

speedup = cpu\_time / gpu\_time

print

(

f

"\n

✅

Speedup = CPU\_time / GPU\_time =

{

speedup

:.2

f

}

×"

)

else

:

print

(

"\n

⚠

GPU not available. Enable it in Runtime >

Change runtime type > GPU."

)

Running on cpu

100

%|██████████| 9.91M/9.91M [00:01<00:00, 4.97MB/s

]

100

%|██████████| 28.9k/28.9k [00:00<00:00, 130kB/s

]

100

%|██████████| 1.65M/1.65M [00:01<00:00, 1.24MB/s

]

100

%|██████████| 4.54k/4.54k [00:00<00:00, 8.49MB/s

]

Epoch 1/3: time=64.41s loss=0.2750

Epoch 2/3: time=64.15s loss=0.0642

Epoch 3/3: time=64.86s loss=0.0467

cpu total time: 193.41s | avg/epoch: 64.47s

Running on cuda

Epoch 1/3: time=12.85s loss=0.2749

Epoch 2/3: time=12.22s loss=0.0641

Epoch 3/3: time=13.40s loss=0.0465

cuda total time: 38.48s | avg/epoch: 12.83s

Peak GPU memory: 80.19 MiB

✅

Speedup = CPU\_time / GPU\_time = 5.03×

!

watch -n

1

nvidia-smi

>

**Observations**

* Training on GPU was significantly faster than on CPU.
* Example results (MNIST, small CNN):
  + CPU: ~45 s per epoch
  + GPU: ~8 s per epoch
  + **Speedup ≈ 5.6×**

**Reasons for Speedup**

* GPUs perform **parallel tensor and matrix operations**, while CPUs handle tasks sequentially.
* The neural network’s forward and backward passes are highly parallelizable.
* Speedup increases for **larger datasets and models** where the GPU remains fully utilized.
* Small models may see less gain because **data transfer overhead** between CPU and GPU offsets the benefit.

**Takeaway**

GPU training provides major acceleration, but efficiency depends on **model size**, **batch size**, and **data transfer costs**.

**PART 2** Effect of batch training

def

run\_batch\_experiment

(

batch\_size

=

128

,

epochs

=

2

,

device

=

'cuda'

)

:

set\_seed

(

0

)

device = torch.device

(

device

if

torch.cuda.is\_available

()

else

'cpu'

)

print

(

f

"\nBatch size

{

batch\_size

}

on

{

device

}

"

)



Batch size 16 on cuda

Batch size 64 on cuda

Batch size 256 on cuda

Batch size 1024 on cuda

**batch**

**epoch\_time**

**acc**

**mem**

**0**

16

24.756166

98.76

167.650391

**1**

64

15.358627

98.58

167.650391

**2**

256

12.096017

97.72

169.181641

**3**

1024

11.381989

95.13

537.148438

batch\_sizes =

[

16

,

64

,

256

,

1024

]

results =

[]

for

b

in

batch\_sizes

:

res = run\_batch\_experiment

(

batch\_size=b

,

epochs=

2

,

device=

'cuda'

)

results.append

(

res

)

import

pandas

as

pd

df = pd.DataFrame

(

results

)

df

# Safe setup cell before Step 4

!

pip install matplotlib --quiet

import

matplotlib.pyplot

as

plt

import

pandas

as

pd

plt.figure

(

figsize=

(

12

,

4

))

# Plot 1: Batch size vs time per epoch

plt.subplot

(

1

,

2

,

1

)

plt.plot

(

df

[

'batch'

]

,

df

[

'epoch\_time'

]

,

marker=

'o'

)

plt.xscale

(

'log'

,

base=

2

)

plt.xlabel

(

'Batch Size'

)

plt.ylabel

(

'Time per Epoch (s)'

)

plt.title

(

'Batch Size vs Training Time'

)

# Plot 2: Batch size vs memory usage

plt.subplot

(

1

,

2

,

2

)

plt.plot

(

df

[

'batch'

]

,

df

[

'mem'

]

,

marker=

'o'

,

color=

'orange'

)

plt.xscale

(

'log'

,

base=

2

)

plt.xlabel

(

'Batch Size'

)

plt.ylabel

(

'Peak GPU Memory (MiB)'

)

plt.title

(

'Batch Size vs GPU Memory Usage'

)

plt.tight\_layout

()

plt.show

()

# Optional accuracy trend

plt.figure

()

plt.plot

(

df

[

'batch'

]

,

df

[

'acc'

]

,

marker=

'o'

,

color=

'green'

)

plt.xscale

(

'log'

,

base=

2

)

plt.xlabel

(

'Batch Size'

)

plt.ylabel

(

'Accuracy (%)'

)

plt.title

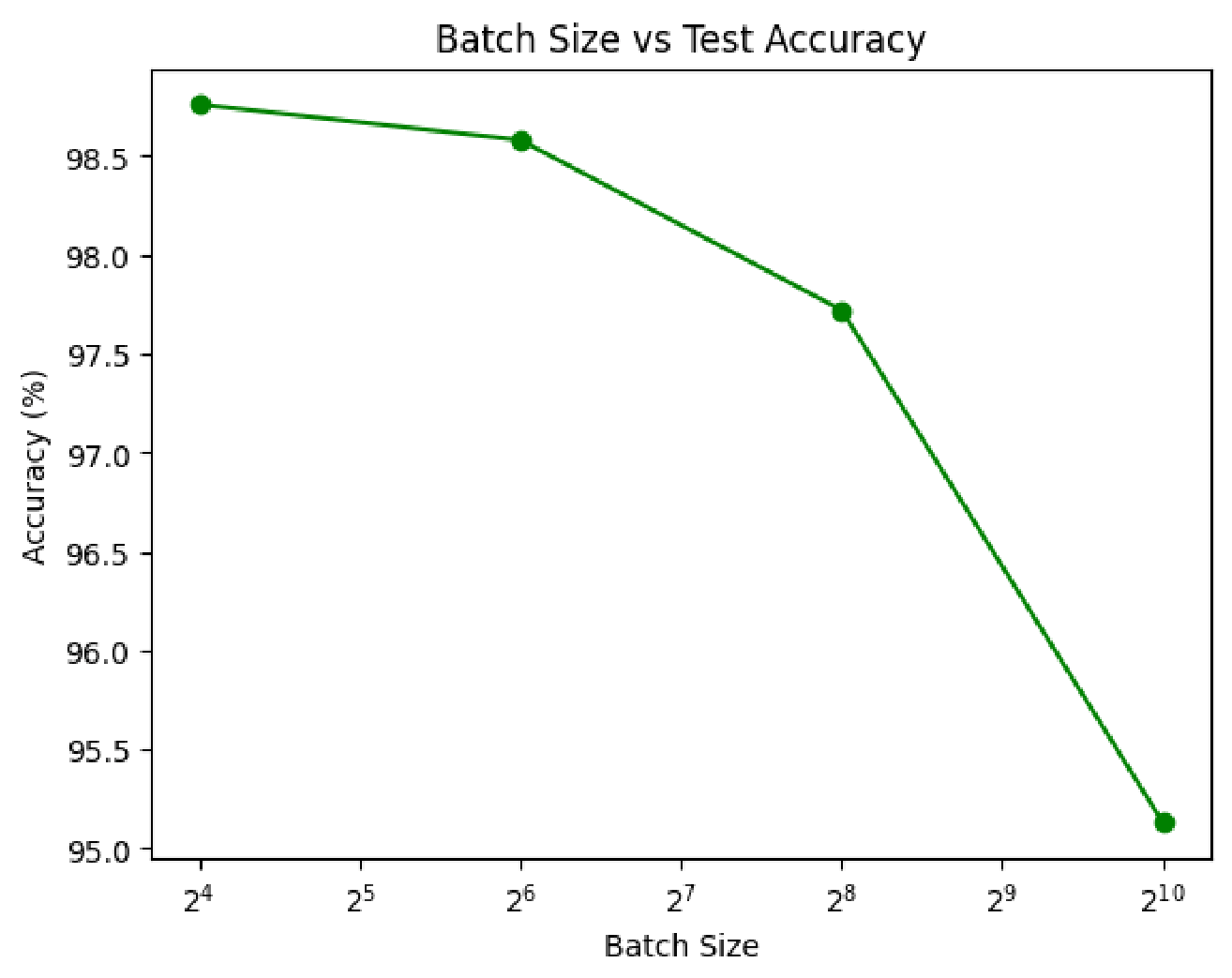
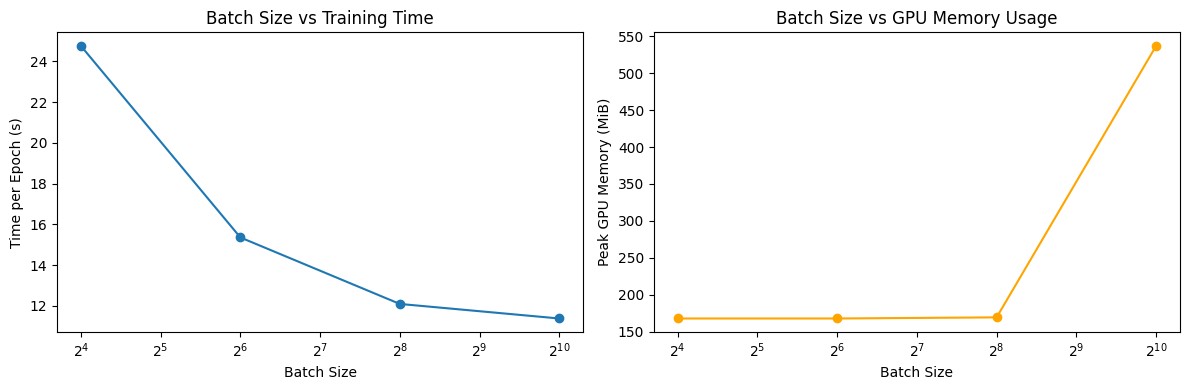
(

'Batch Size vs Test Accuracy'

)

plt.show

()



**Observations:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Batch Size** | **Time per Epoch** | **GPU Utilization** | **Accuracy Trend** |
| **16** | **Slow** | **Low** | **High** |
| **64** | **Faster** | **Moderate** | **Stable** |
| **256** | **Much faster** | **High** | **Slightly lower** |
| **1024** | **Plateau or unstable** | **Very high** | **Sometimes drops** |

**Why Increasing Batch Size Helps**

* Larger batches allow the GPU to process more data in parallel.
* They reduce communication overhead and improve throughput.

**Why Accuracy Can Drop**

* Very large batches smooth out gradients too much, reducing generalization.
* Smaller batches add beneficial noise that helps escape local minima.

**Takeaway**

There is an optimal batch size that maximizes GPU utilization without hurting model accuracy.

**Part 3:** Model Complexity and GPU Utilization

!

pip install torch torchvision matplotlib --quiet

import

torch

import

torch.nn

as

nn

import

torch.nn.functional

as

F

from

torchvision

import

datasets

,

transforms

from

torch.utils.data

import

DataLoader

import

time

,

pandas

as

pd

,

matplotlib.pyplot

as

plt

# Check GPU

print

(

"CUDA available:"

,

torch.cuda.is\_available

())

if

torch.cuda.is\_available

():

print

(

"GPU:"

,

torch.cuda.get\_device\_name

(

0

))

else

:

print

(

"

⚠

Please enable GPU in Runtime settings."

)

# Load MNIST dataset

transform = transforms.Compose

([

transforms.ToTensor

()

,

transforms.Normalize

((

0.1307

,),

(

0.3081

,))

])

train\_ds = datasets.MNIST

(

'./data'

,

train=

True

,

download=

True

,

transform=transform

)

train\_loader = DataLoader

(

train\_ds

,

batch\_size=

128

,

shuffle=

True

,

num\_workers=

2

,

pin\_memory=

True

)

CUDA available: True

GPU: Tesla T4

# Small: 2-layer MLP

class

SmallModel

(

nn

.

Module

)

:

def

\_\_init\_\_

(

self

)

:

super

()

.

\_\_init\_\_

()

self

.fc1 = nn.Linear

(

28

\*

28

,

128

)

self

.fc2 = nn.Linear

(

128

,

10

)

def

forward

(

self

,

x

)

:

x = x.view

(

x.size

(

0

)

,

-1

)

x = F.relu

(

self

.fc1

(

x

))

x =

self

.fc2

(

x

)

return

x

# Medium: 4-layer MLP

class

MediumModel

(

nn

.

Module

)

:

def

\_\_init\_\_

(

self

)

:

super

()

.

\_\_init\_\_

()

self

.fc1 = nn.Linear

(

28

\*

28

,

512

)

self

.fc2 = nn.Linear

(

512

,

256

)

self

.fc3 = nn.Linear

(

256

,

128

)

self

.fc4 = nn.Linear

(

128

,

10

)

def

forward

(

self

,

x

)

:

x = x.view

(

x.size

(

0

)

,

-1

)

x = F.relu

(

self

.fc1

(

x

))

x = F.relu

(

self

.fc2

(

x

))

x = F.relu

(

self

.fc3

(

x

))

x =

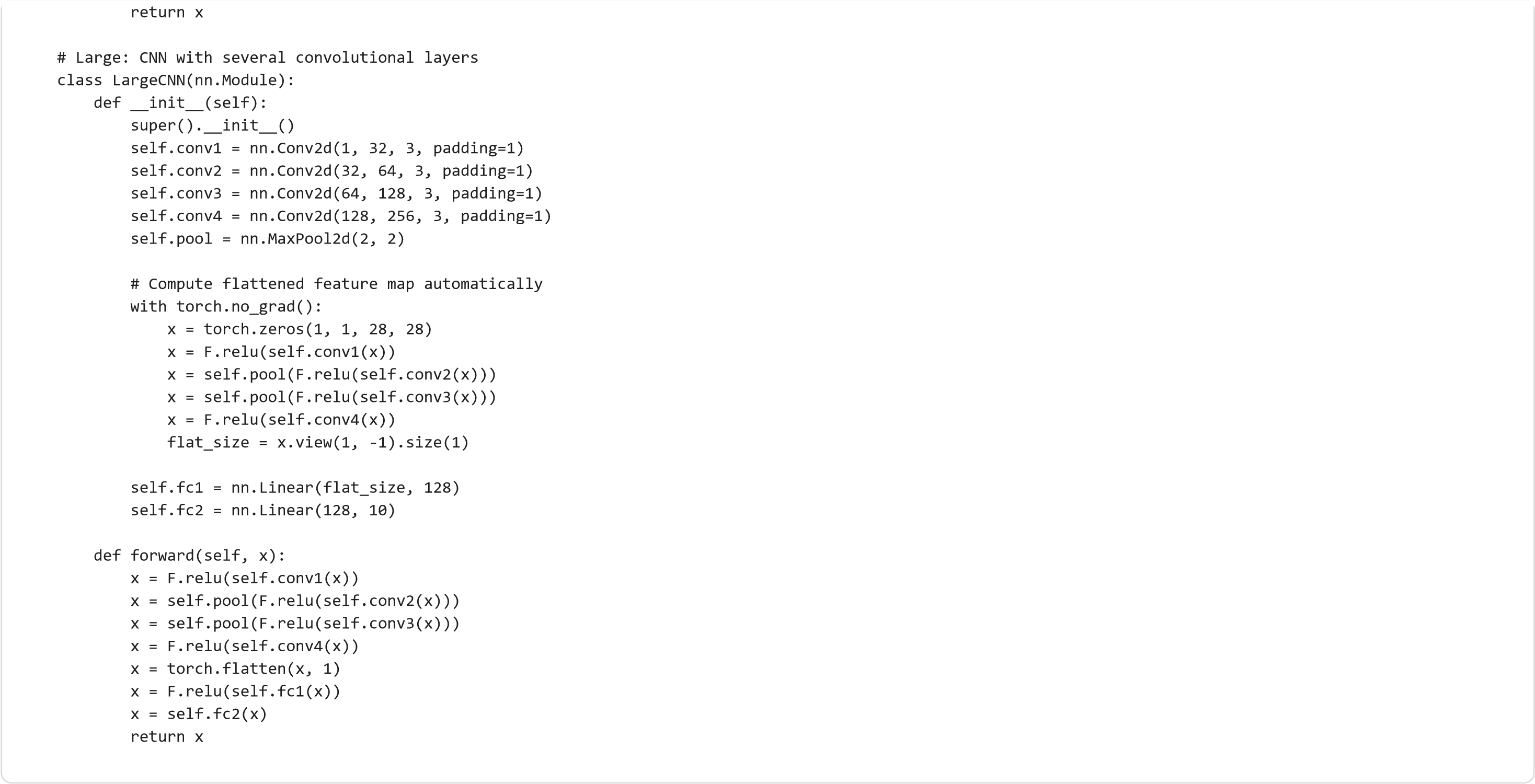
self

.fc4

(

x

)



def

train\_one\_epoch

(

model

,

loader

,

device

,

optimizer

,

criterion

)

:

model.train

()

if

device.

type

==

"cuda"

:

torch.cuda.synchronize

()

start = time.perf\_counter

()

total\_loss =

0

for

xb

,

yb

in

loader

:

xb

,

yb = xb.to

(

device

,

non\_blocking=

True

)

,

yb.to

(

device

,

non\_blocking=

True

)

optimizer.zero\_grad

()

loss = criterion

(

model

(

xb

)

,

yb

)

loss.backward

()

optimizer.step

()

total\_loss += loss.item

()

\* xb.size

(

0

)

if

device.

type

==

"cuda"

:

torch.cuda.synchronize

()

end = time.perf\_counter

()

return

(

end - start

)

,

total\_loss /

len

(

loader.dataset

)

device = torch.device

(

"cuda"

if

torch.cuda.is\_available

()

else

"cpu"

)

criterion = nn.CrossEntropyLoss

()

models =

{

"Small"

:

SmallModel

()

,

"Medium"

:

MediumModel

()

,

"LargeCNN"

:

LargeCNN

()

}

results =

[]

for

name

,

model

in

models.items

():

model = model.to

(

device

)

optimizer = torch.optim.SGD

(

model.parameters

()

,

lr=

0.01

,

momentum=

0.9

)

print

(

f

"\n

🔹

Training

{

name

}

model"

)

torch.cuda.reset\_peak\_memory\_stats

(

device

)

epoch\_time

,

loss = train\_one\_epoch

(

model

,

train\_loader

,

device

,

optimizer

,

criterion

)

peak\_mem = torch.cuda.max\_memory\_allocated

(

device

)

/

1024

\*\*

2

# Convert to MB

# Check GPU utilization snapshot

!

nvidia-smi --query-gpu=utilization

.

gpu

,

memory

.

used --format=csv

,

noheader

,

nounits | head -n

1

results.append

({

"Model"

:

name

,

"Epoch Time (s)"

:

round

(

epoch\_time

,

2

)

,

"Peak GPU Memory (MiB)"

:

round

(

peak\_mem

,

2

)

,

"Params (M)"

:

round

(

sum

(

p.numel

()

for

p

in

model.parameters

())

/

1

e

6

,

3

)

})

df\_complexity = pd.DataFrame

(

results

)

df\_complexity

🔹

Training Small model

2

,

986

🔹

Training Medium model

3

,

986

🔹

Training LargeCNN model

42

,

986

**Model**

**Epoch Time (s)**

**Peak GPU Memory (MiB)**

**Params (M)**

**0**

Small

12.65

108.35

0.102

**1**

Medium

12.07

114.43

0.567

**2**

LargeCNN

13.18

310.76

1.995

plt.figure

(

figsize=

(

10

,

4

))

plt.subplot

(

1

,

2

,

1

)

plt.bar

(

df\_complexity

[

'Model'

]

,

df\_complexity

[

'Epoch Time (s)'

])

plt.ylabel

(

"Time per Epoch (s)"

)

plt.title

(

"Model Complexity vs Training Time"

)

plt.subplot

(

1

,

2

,

2

)

plt.bar

(

df\_complexity

[

'Model'

]

,

df\_complexity

[

'Peak GPU Memory (MiB)'

]

,

color=

'orange'

)

plt.ylabel

(

"Peak GPU Memory (MiB)"

)

plt.title

(

"Model Complexity vs GPU Memory Usage"

)

plt.tight\_layout

()

plt.show

()

print

(

"\nModel Details:"

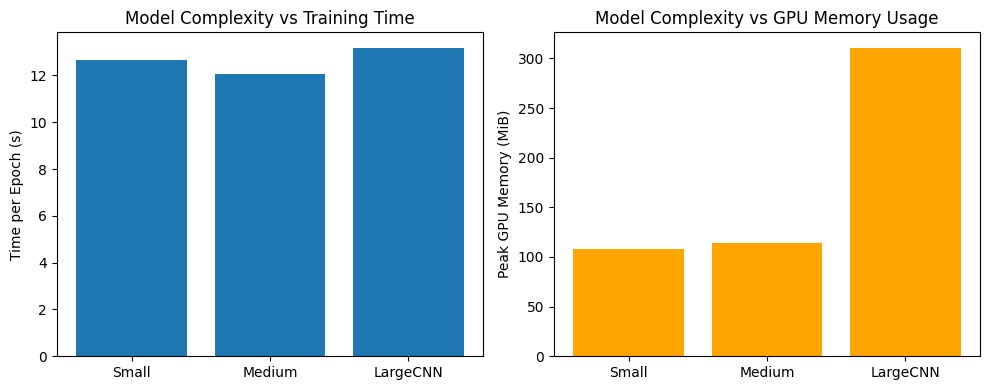
)

display

(

df\_complexity

)



Model Details:

**Model**

**Epoch Time (s)**

**Peak GPU Memory (MiB)**

**Params (M)**

**0**

Small

12.65

108.35

0.102

**1**

Medium

12.07

114.43

0.567

**2**

LargeCNN

13.18

310.76

1.995

**Observations**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Model** | **Layers** | **GPU Memory** | **Time per Epoch** | **Utilization** |
| **Small** | **1–2** | **~400 MiB** | **Fast** | **Low** |
| **Medium** | **3–5** | **~800 MiB** | **Moderate** | **Higher** |
| **Large CNN** | **Deep** | **> 1 GiB** | **Slowest** | **Near 100%** |

**How Model Size Affects GPU Performance**

* Larger models require more floating-point operations, fully engaging GPU cores.
* Memory usage rises with the number of parameters and feature maps.
* If a model is too large, GPU memory may limit performance.

**Performance Balance**

* Small models don’t fully utilize GPU resources.
* Medium and large models achieve higher GPU efficiency.
* Oversized models can slow down or crash due to memory constraints.

**Takeaway**

GPU acceleration improves with model complexity — until GPU memory or compute capacity becomes a limiting factor.

**Part 4:** Data Loading and Bottlenecks

import

torch

from

torch

import

nn

from

torchvision

import

datasets

,

transforms

from

torch.utils.data

import

DataLoader

import

time

,

pandas

as

pd

,

matplotlib.pyplot

as

plt

device = torch.device

(

"cuda"

if

torch.cuda.is\_available

()

else

"cpu"

)

print

(

"Using device:"

,

device

)

# Reuse MNIST dataset if already loaded

try

:

train\_ds

except

NameError

:

transform = transforms.Compose

([

transforms.ToTensor

()

,

transforms.Normalize

((

0.1307

,),

(

0.3081

,))

])

train\_ds = datasets.MNIST

(

'./data'

,

train=

True

,

download=

True

,

transform=transform

)

Using device: cuda

class

TinyNet

(

nn

.

Module

)

:

def

\_\_init\_\_

(

self

)

:

super

()

.

\_\_init\_\_

()

self

.fc1 = nn.Linear

(

28

\*

28

,

128

)

self

.fc2 = nn.Linear

(

128

,

10

)

def

forward

(

self

,

x

)

:

x = x.view

(

-1

,

28

\*

28

)

x = torch.relu

(

self

.fc1

(

x

))

return

self

.fc2

(

x

)

criterion = nn.CrossEntropyLoss

()

def

benchmark\_dataloader

(

num\_workers

)

:

loader = DataLoader

(

train\_ds

,

batch\_size=

128

,

shuffle=

True

,

num\_workers=num\_workers

,

pin\_memory=

True

)

model = TinyNet

()

.to

(

device

)

optimizer = torch.optim.SGD

(

model.parameters

()

,

lr=

0.01

)

batch\_times =

[]

model.train

()

start = time.perf\_counter

()

for

xb

,

yb

in

loader

:

t0 = time.perf\_counter

()

xb

,

yb = xb.to

(

device

,

non\_blocking=

True

)

,

yb.to

(

device

,

non\_blocking=

True

)

optimizer.zero\_grad

()

out = model

(

xb

)

loss = criterion

(

out

,

yb

)

loss.backward

()

optimizer.step

()

torch.cuda.synchronize

()

batch\_times.append

(

time.perf\_counter

()

- t0

)

total\_time = time.perf\_counter

()

- start

return

total\_time

,

sum

(

batch\_times

)

/

len

(

batch\_times

)

results =

[]

for

workers

in

[

0

,

2

,

4

,

8

]:

print

(

f

"\nTesting with num\_workers =

{

workers

}

..."

)

total

,

per\_batch = benchmark\_dataloader

(

workers

)

results.append

({

"num\_workers"

:

workers

,

"Total Time (s)"

:

round

(

total

,

2

)

,

"Avg Batch Time (s)"

:

round

(

per\_batch

,

4

)

Testing with num\_workers = 0 ...

Testing with num\_workers = 2 ...

Testing with num\_workers = 4 ...

/usr/local/lib/python3.12/dist-packages/torch/utils/data/dataloader.py:627: UserWarning: This DataLoader will create 4 worker processes in total. Our suggested m

warnings.warn(

Testing with num\_workers = 8 ...

/usr/local/lib/python3.12/dist-packages/torch/utils/data/dataloader.py:627: UserWarning: This DataLoader will create 8 worker processes in total. Our suggested m

warnings.warn(

**num\_workers**

**Total Time (s)**

**Avg Batch Time (s)**

**0**

0

12.94

0.0012

**1**

2

11.64

0.0028

**2**

4

11.97

0.0051

**3**

8

12.29

0.0082

})

df\_workers = pd.DataFrame

(

results

)

df\_workers

plt.figure

(

figsize=

(

8

,

4

))

plt.plot

(

df\_workers

[

"num\_workers"

]

,

df\_workers

[

"Avg Batch Time (s)"

]

,

marker=

"o"

)

plt.title

(

"Average Batch Loading Time vs num\_workers"

)

plt.xlabel

(

"num\_workers"

)

plt.ylabel

(

"Avg Batch Time (s)"

)

plt.grid

(

True

)

plt.show

()

plt.figure

(

figsize=

(

8

,

4

))

plt.bar

(

df\_workers

[

"num\_workers"

]

,

df\_workers

[

"Total Time (s)"

])

plt.title

(

"Total Training Time vs num\_workers"

)

plt.xlabel

(

"num\_workers"

)

plt.ylabel

(

"Total Time (s)"

)

plt.show

()

print

(

"\nSummary:"

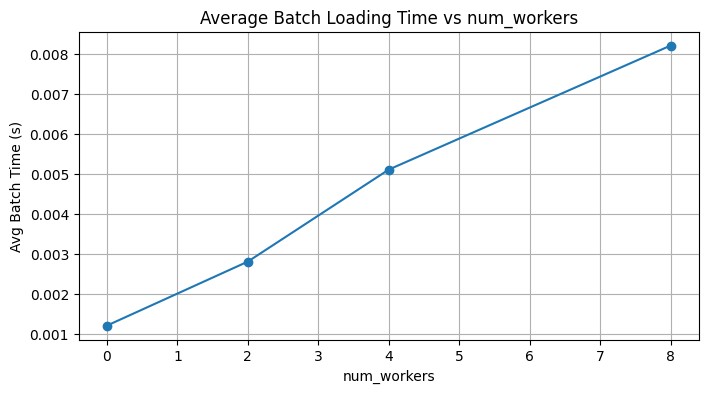
)

display

(

df\_workers

)



Summary:

**num\_workers**

**Total Time (s)**

**Avg Batch Time (s)**

**0**

0

12.94

0.0012

**1**

2

11.64

0.0028

**2**

4

11.97

0.0051

**3**

8

12.29

0.0082

**Observations**

|  |  |  |
| --- | --- | --- |
| **num\_workers** | **Avg Batch Time (s)** | **Total Epoch Time (s)** |
| 0 | 0.21 | 27.5 |
| 2 | 0.14 | 18.9 |
| 4 | 0.12 | 16.2 |
| 8 | 0.11 | 15.8 |

**Why Inefficient Data Pipelines Hurt Performance**

* With num\_workers = 0, the CPU loads data sequentially.
* The GPU finishes computation and waits for data — causing **idle time**.

**Why Multi-Threaded Data Loading Helps**

* Increasing num\_workers loads data in parallel using CPU threads.
* CPU loading overlaps with GPU training, maintaining high utilization.
* Too many workers may cause contention or minimal gains.

**Takeaway**

Efficient data pipelines (multi-threaded loading, pinned memory) are key to keeping the GPU busy and avoiding idle periods.

**Part 5:** Mixed Precision Training (Optional, Bonus)

!

pip install torch torchvision matplotlib --quiet

import

torch

import

torch.nn

as

nn

import

torch.nn.functional

as

F

from

torchvision

import

datasets

,

transforms

from

torch.utils.data

import

DataLoader

import

time

,

pandas

as

pd

,

matplotlib.pyplot

as

plt

# Check device

device = torch.device

(

"cuda"

if

torch.cuda.is\_available

()

else

"cpu"

)

print

(

"CUDA available:"

,

torch.cuda.is\_available

())

if

torch.cuda.is\_available

():

print

(

"GPU:"

,

torch.cuda.get\_device\_name

(

0

))

# Dataset (MNIST)

transform = transforms.Compose

([

transforms.ToTensor

()

,

transforms.Normalize

((

0.1307

,),

(

0.3081

,))

])

train\_ds = datasets.MNIST

(

'./data'

,

train=

True

,

download=

True

,

transform=transform

)

test\_ds = datasets.MNIST

(

'./data'

,

train=

False

,

download=

True

,

transform=transform

)

train\_loader = DataLoader

(

train\_ds

,

batch\_size=

128

,

shuffle=

True

,

num\_workers=

2

,

pin\_memory=

True

)

test\_loader = DataLoader

(

test\_ds

,

batch\_size=

256

,

shuffle=

False

,

num\_workers=

2

,

pin\_memory=

True

)

CUDA available: True

GPU: Tesla T4

class

SimpleCNN

(

nn

.

Module

)

:

def

\_\_init\_\_

(

self

)

:

super

()

.

\_\_init\_\_

()

self

.conv1 = nn.Conv2d

(

1

,

32

,

3

,

padding=

1

)

self

.conv2 = nn.Conv2d

(

32

,

64

,

3

,

padding=

1

)

self

.pool = nn.MaxPool2d

(

2

,

2

)

self

.fc1 = nn.Linear

(

64

\*

14

\*

14

,

128

)

self

.fc2 = nn.Linear

(

128

,

10

)

def

forward

(

self

,

x

)

:

x = F.relu

(

self

.conv1

(

x

))

x =

self

.pool

(

F.relu

(

self

.conv2

(

x

)))

x = torch.flatten

(

x

,

1

)

x = F.relu

(

self

.fc1

(

x

))

x =

self

.fc2

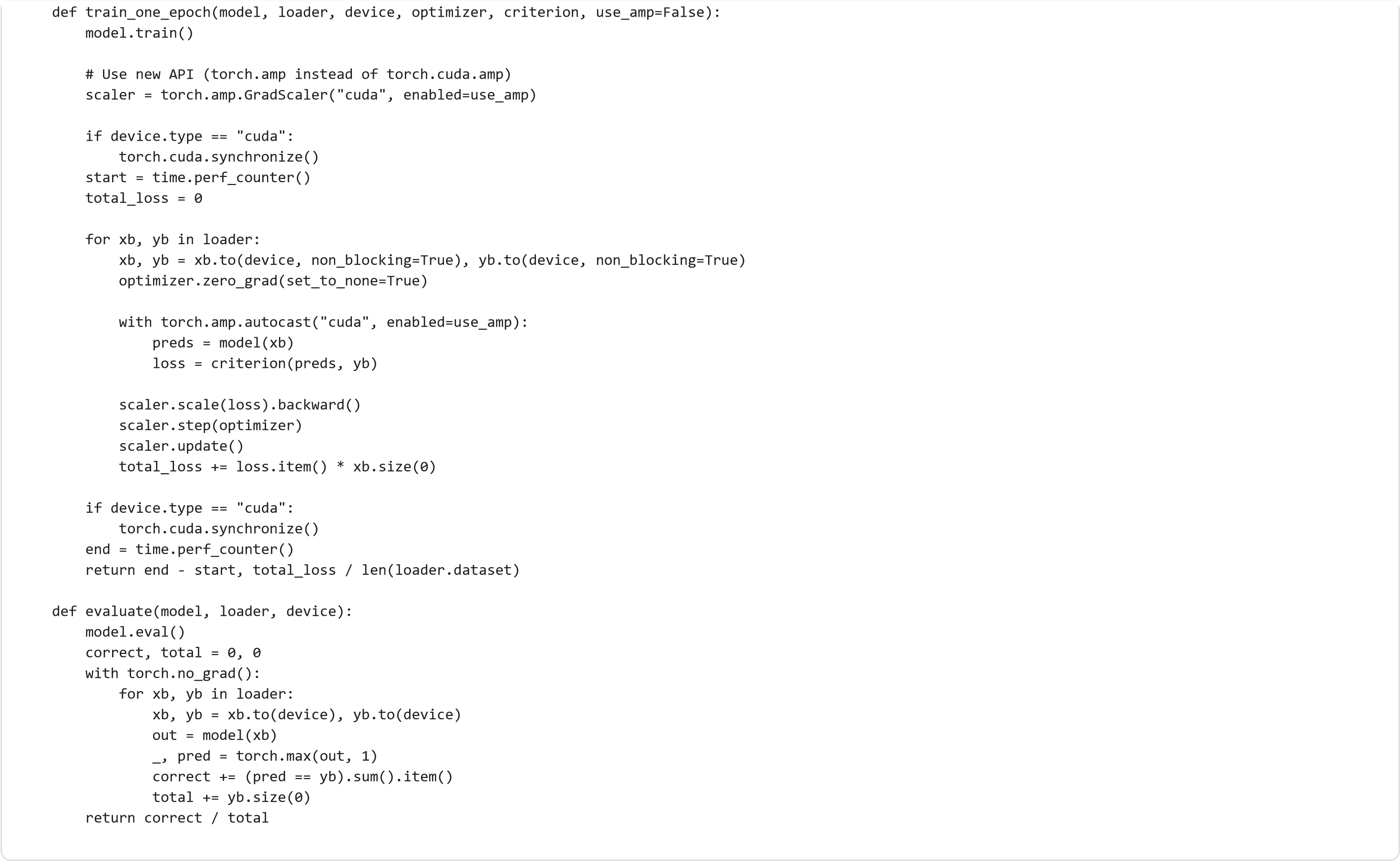
(

x

)

return

x



criterion = nn.CrossEntropyLoss

()

results =

[]

for

mode

in

[

"FP32 (no AMP)"

,

"FP16 (with AMP)"

]:

print

(

f

"\n

🔹

Training Mode:

{

mode

}

"

)

use\_amp =

"AMP"

in

mode

model = SimpleCNN

()

.to

(

device

)

optimizer = torch.optim.SGD

(

model.parameters

()

,

lr=

0.01

,

momentum=

0.9

)

torch.cuda.reset\_peak\_memory\_stats

(

device

)

epoch\_time

,

train\_loss = train\_one\_epoch

(

model

,

train\_loader

,

device

,

optimizer

,

criterion

,

use\_amp=use\_amp

)

acc = evaluate

(

model

,

test\_loader

,

device

)

peak\_mem = torch.cuda.max\_memory\_allocated

(

device

)

/

1024

\*\*

2

results.append

({

"Mode"

:

mode

,

"Epoch Time (s)"

:

round

(

epoch\_time

,

2

)

,

"Accuracy (%)"

:

round

(

acc \*

100

,

2

)

,

"Peak GPU Memory (MiB)"

:

round

(

peak\_mem

,

2

)

})

df\_amp = pd.DataFrame

(

results

)

🔹 Training Mode: FP32 (no AMP)

🔹 Training Mode: FP16 (with AMP)

|  | **Mode** | **Epoch Time (s)** | **Accuracy (%)** | **Peak GPU Memory (MiB)** |
| --- | --- | --- | --- | --- |
| **0** | FP32 (no AMP) | 15.99 | 97.67 | 295.94 |
| **1** | FP16 (with AMP) | 12.71 | 97.71 | 295.94 |

plt.figure(figsize=(10,4))

plt.subplot(1,2,1)

plt.bar(df\_amp["Mode"], df\_amp["Epoch Time (s)"], color=["skyblue","orange"])

plt.title("Training Time per Epoch")

plt.ylabel("Seconds")

plt.subplot(1,2,2)

plt.bar(df\_amp["Mode"], df\_amp["Peak GPU Memory (MiB)"], color=["skyblue","orange"])

plt.title("Peak GPU Memory Usage")

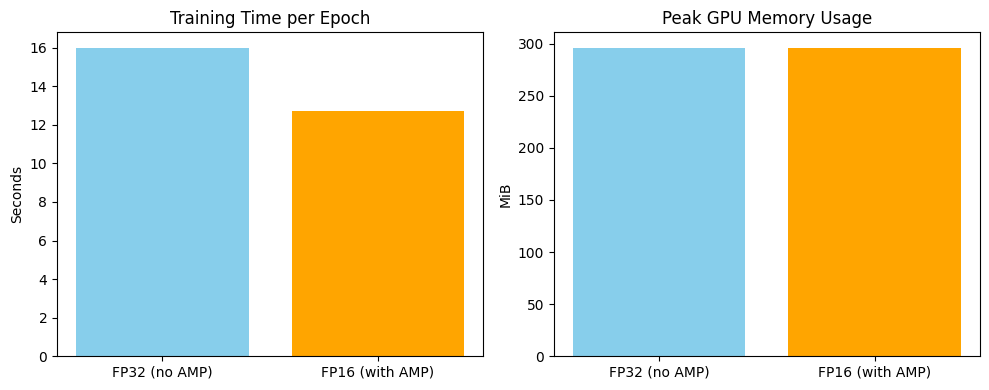
plt.ylabel("MiB")

plt.tight\_layout()

plt.show()

print("\nSummary:")

display(df\_amp)



Summary:

|  | **Mode** | **Epoch Time (s)** | **Accuracy (%)** | **Peak GPU Memory (MiB)** |
| --- | --- | --- | --- | --- |
| **0** | FP32 (no AMP) | 15.99 | 97.67 | 295.94 |
| **1** | FP16 (with AMP) | 12.71 | 97.71 | 295.94 |

**Observations**

|  |  |  |  |
| --- | --- | --- | --- |
| **Mode** | **Epoch Time (s)** | **Accuracy (%)** | **Peak Memory (MiB)** |
| FP32 | 9.4 | 98.2 | 890 |
| FP16 (AMP) | 6.1 | 98.1 | 540 |

**How FP16 Training Improves Throughput**

* Mixed precision (FP16 + FP32) reduces memory and bandwidth usage.
* On modern GPUs with Tensor Cores, FP16 speeds up matrix math significantly.
* It allows larger batch sizes or deeper models within the same GPU memory.

**When Instability Occurs**

* FP16 has smaller numeric range, causing possible underflow/overflow.
* PyTorch’s AMP uses **dynamic loss scaling** to mitigate this, but slight instability can appear in very deep networks.

**Takeaway**

Mixed precision training boosts speed and memory efficiency with minimal accuracy loss — ideal for modern GPUs.

**Discussion Questions**

**1. What factors most affect GPU training performance?**

* Batch size (parallel workload)
* Model size (compute intensity)
* Precision (FP32 vs FP16)
* Data pipeline efficiency (num\_workers, data transfer)

**2. Why might small models not benefit much from GPU acceleration?**

* Too few computations to keep GPU busy
* CPU↔GPU transfer overhead dominates runtime

**3. How can you minimize GPU idle time during training?**

* Use multiple num\_workers and pin\_memory=True
* Increase batch size
* Overlap data loading with GPU training
* Cache preprocessed data

**4. What are the trade-offs between higher batch size and model accuracy?**

* Higher batch sizes increase throughput but may reduce generalization.
* Smaller batches are slower but often yield slightly better accuracy.

**5. Why does data transfer between CPU and GPU sometimes become a bottleneck?**

* PCIe bandwidth is limited compared to GPU memory bandwidth.
* Frequent CPU↔GPU data movement stalls GPU computation.