PyTorch Tutorial

CS 185/285 Deep RL

PyTorch Tutorial

CS 185/285 Deep RL

What you will leave with:

- Tensor basics (shapes, indexing, broadcasting, dtype/device)

PyTorch Tutorial

CS 185/285 Deep RL

What you will leave with:

- Tensor basics (shapes, indexing, broadcasting, dtype/device)
- Autograd mental model (when gradients are tracked & why)

PyTorch Tutorial

CS 185/285 Deep RL

What you will leave with:

- Tensor basics (shapes, indexing, broadcasting, dtype/device)
- Autograd mental model (when gradients are tracked & why)
- How to write a correct training loop (zero_grad → backward → step)

PyTorch Tutorial

CS 185/285 Deep RL

What you will leave with:

- Tensor basics (shapes, indexing, broadcasting, dtype/device)
- Autograd mental model (when gradients are tracked & why)
- How to write a correct training loop (zero_grad → backward → step)
- A full modern training recipe: AdamW, schedulers, grad clipping, eval mode

PyTorch Tutorial

CS 185/285 Deep RL

What you will leave with:

- Tensor basics (shapes, indexing, broadcasting, dtype/device)
- Autograd mental model (when gradients are tracked & why)
- How to write a correct training loop (zero_grad → backward → step)
- A full modern training recipe: AdamW, schedulers, grad clipping, eval mode
- Common gotchas + debugging tools

PyTorch Tutorial

CS 185/285 Deep RL

What you will leave with:

- Tensor basics (shapes, indexing, broadcasting, dtype/device)
- Autograd mental model (when gradients are tracked & why)
- How to write a correct training loop (zero_grad → backward → step)
- A full modern training recipe: AdamW, schedulers, grad clipping, eval mode
- Common gotchas + debugging tools

Scope

Single GPU, single process. We skip multi-GPU, sharding, and distributed details.

1) Tensors	Shapes, indexing, broadcasting, dtype & device
2) Autograd	Computation graphs, requires_grad, no_grad, detach
3) Modules	nn.Module, parameters, buffers, state_dict
4) Training loop	loss, backward, optimizer, schedulers, clipping
5) Data + eval	DataLoader, model.train/eval, inference_mode
6) Practical recipes	Checkpointing, debugging & gotchas
7) What's new in modern PyTorch	torch.compile

Part 1 — Tensors

The core data structure (and where it lives)

Tensors: typed n-D arrays on a device

A tensor is like a NumPy ndarray, but with: GPU support + autograd metadata.

Key attributes

shape
- dtype
- device
- requires_grad
- layout (rare)
- strides (important for views)

Tensors: typed n-D arrays on a device

A tensor is like a NumPy ndarray, but with: GPU support + autograd metadata.

Key attributes

shape
- dtype
- device
- requires_grad
- layout (rare)
- strides (important for views)

shape = (4, 6)

Indexing examples:

x[0] → first row (shape (6,))
x[:, 2:5] → slice columns 2..4 (shape (4,3))
x.reshape(2, 2, 6) → same data, new view if contiguous

Tensors: typed n-D arrays on a device

A tensor is like a NumPy ndarray, but with: GPU support + autograd metadata.

Key attributes

shape
- dtype
- device
- requires_grad
- layout (rare)
- strides (important for views)

shape = (4, 6)

Indexing examples:

x[0] → first row (shape (6,))
x[:, 2:5] → slice columns 2..4 (shape (4,3))
x.reshape(2, 2, 6) → same data, new view if contiguous

Frequent tensors in RL code

Rollouts are just tensors: obs (B,T,<ob_dim>), actions (B,T,<a_dim>), rewards (B,T). Getting shapes right is most of the battle.

Creating tensors (and controlling dtype/device)

Common constructors

```
import torch

# Constructors
x = torch.zeros(3, 4)              # float32 on CPU
y = torch.randn(10, device='cuda') # on GPU

# From Python / NumPy
a = torch.tensor([1, 2, 3])        # copies data
b = torch.as_tensor(a)             # avoids copy when possible

# dtype
w = torch.ones(5, dtype=torch.float64)

# Randomness control (more later)
torch.manual_seed(0)
```

Creating tensors (and controlling dtype/device)

Common constructors

```
import torch

# Constructors
x = torch.zeros(3, 4)              # float32 on CPU
y = torch.randn(10, device='cuda') # on GPU

# From Python / NumPy
a = torch.tensor([1, 2, 3])        # copies data
b = torch.as_tensor(a)             # avoids copy when possible

# dtype
w = torch.ones(5, dtype=torch.float64)

# Randomness control (more later)
torch.manual_seed(0)
```

Defaults

- Default device: CPU
- Default floating dtype: float32
- Integers default to int64

Tip: for large tensors, create on the target device to avoid extra copies.

Creating tensors (and controlling dtype/device)

Common constructors

```
1 import torch
2
3 # Constructors
4 x = torch.zeros(3, 4)              # float32 on CPU
5 y = torch.randn(10, device='cuda') # on GPU
6
7 # From Python / NumPy
8 a = torch.tensor([1, 2, 3])        # copies data
9 b = torch.as_tensor(a)             # avoids copy when possible
10
11 # dtype
12 w = torch.ones(5, dtype=torch.float64)
13
14 # Randomness control (more later)
15 torch.manual_seed(0)
```

Defaults

- Default device: CPU
- Default floating dtype: float32
- Integers default to int64

Tip: for large tensors, create on the target device to avoid extra copies.

Gotchas about copying (don't really need to worry about this)

torch.tensor(np_array) copies.
Use torch.from_numpy / as_tensor for zero-copy views (CPU only).

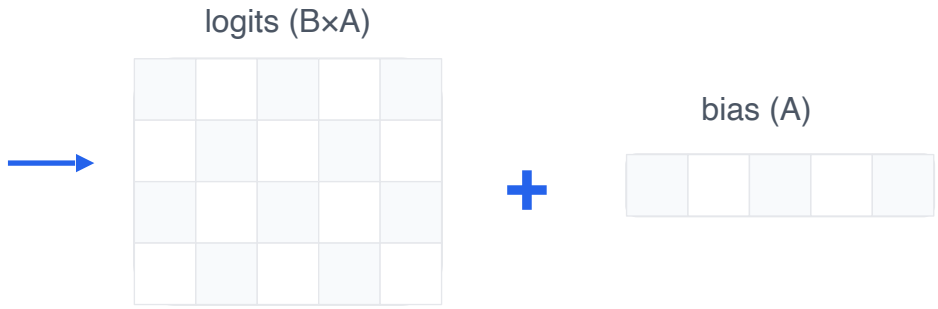
Once data is on GPU, keep the whole pipeline in torch.

Broadcasting & shape sanity checks

Broadcasting is powerful — and the #1 source of silent bugs.

Example: add a bias to a batch

logits: (B, A)
bias: (A,)
→ logits + bias: (B, A)

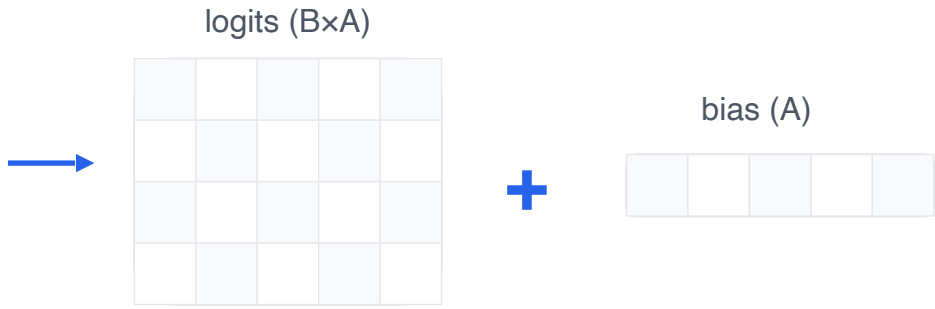


Broadcasting & shape sanity checks

Broadcasting is powerful — and the #1 source of silent bugs.

Example: add a bias to a batch

logits: (B, A)
bias: (A,)
→ logits + bias: (B, A)



Best practices

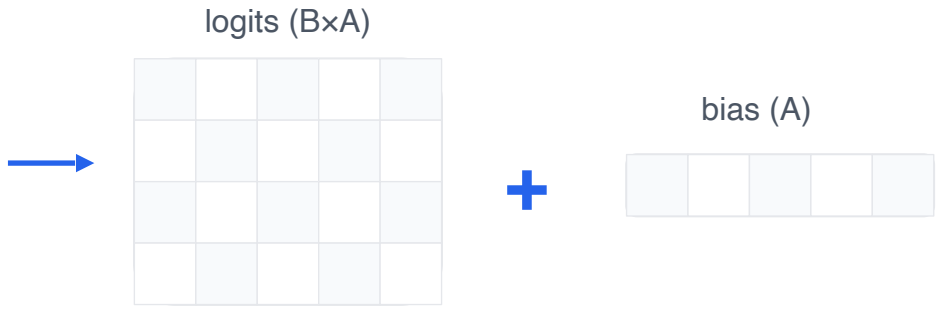
- Assert shapes early
- Use .shape prints liberally
- Prefer explicit reshape/unsqueeze over “magic” broadcasting
- Watch out for (B,) vs (B,1)

Broadcasting & shape sanity checks

Broadcasting is powerful — and the #1 source of silent bugs.

Example: add a bias to a batch

logits: (B, A)
bias: (A,)
→ logits + bias: (B, A)



Best practices

- Assert shapes early
- Use .shape prints liberally
- Prefer explicit reshape/unsqueeze over “magic” broadcasting
- Watch out for (B,) vs (B,1)

Recommendation: first reshape bias to (1, A), then broadcast (better clarity)

Device placement: CPU vs GPU

Tensors live on a device. Operations generally require all inputs to be on the same device.

Canonical pattern

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = MyNet().to(device)

for batch in loader:
    obs, target = batch
    obs = obs.to(device, non_blocking=True)
    target = target.to(device, non_blocking=True)
```

Device placement: CPU vs GPU

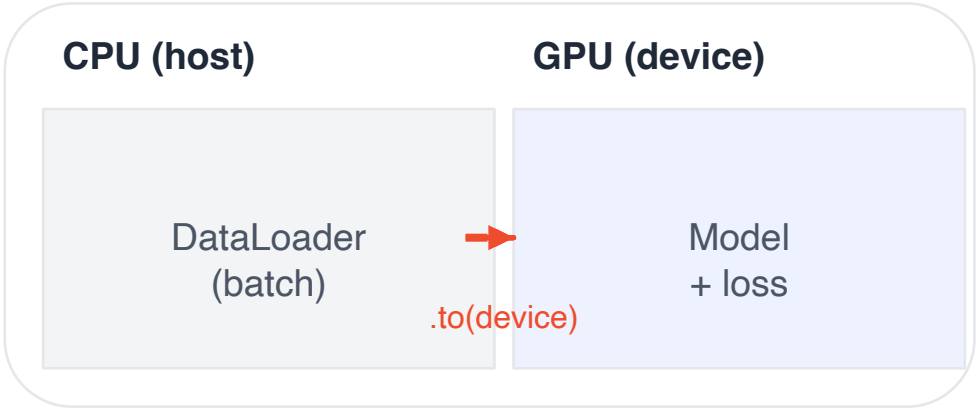
Tensors live on a device. Operations generally require all inputs to be on the same device.

Canonical pattern

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = MyNet().to(device)

for batch in loader:
    obs, target = batch
    obs = obs.to(device, non_blocking=True)
    target = target.to(device, non_blocking=True)
```



Device placement: CPU vs GPU

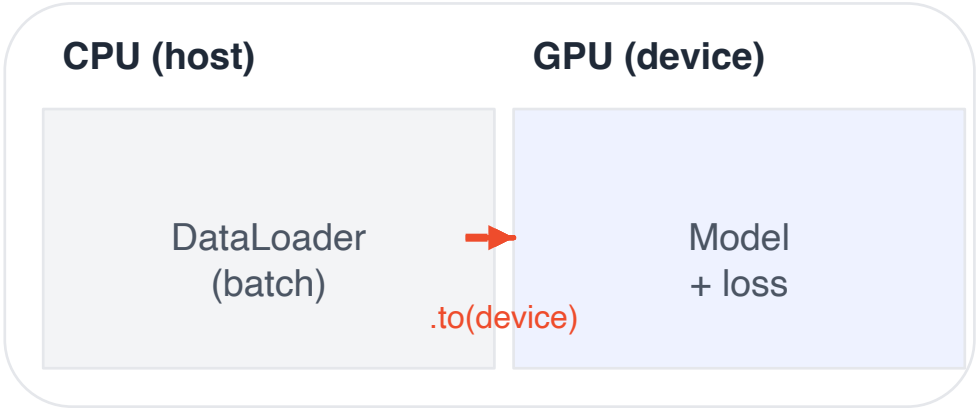
Tensors live on a device. Operations generally require all inputs to be on the same device.

Canonical pattern

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = MyNet().to(device)

for batch in loader:
    obs, target = batch
    obs = obs.to(device, non_blocking=True)
    target = target.to(device, non_blocking=True)
```



Common device errors

“Expected all tensors to be on the same device” → some tensor stayed on CPU (often targets / masks).

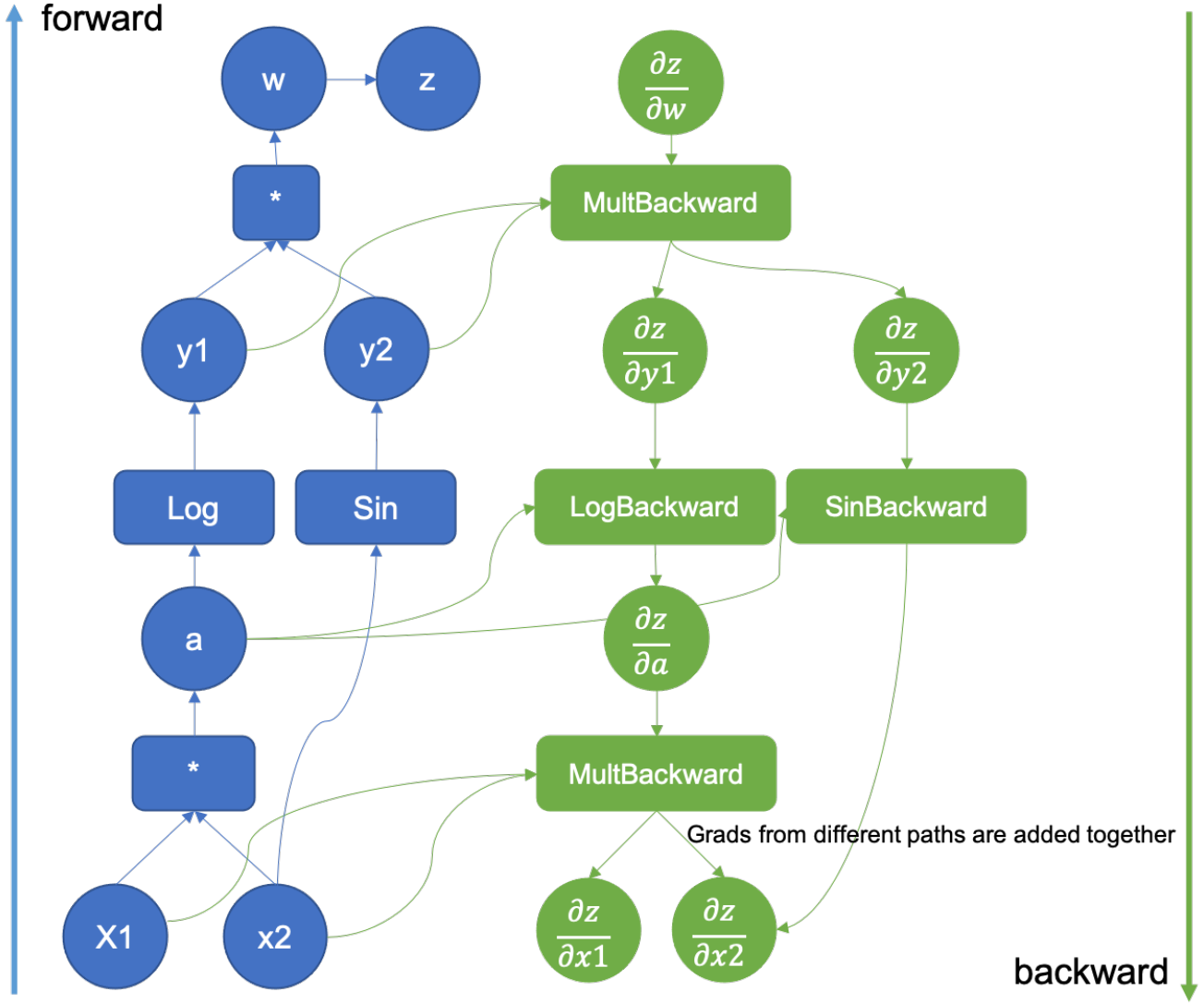
Fix: move *every* tensor used in the loss.

Part 2 — Autograd

How PyTorch tracks gradients

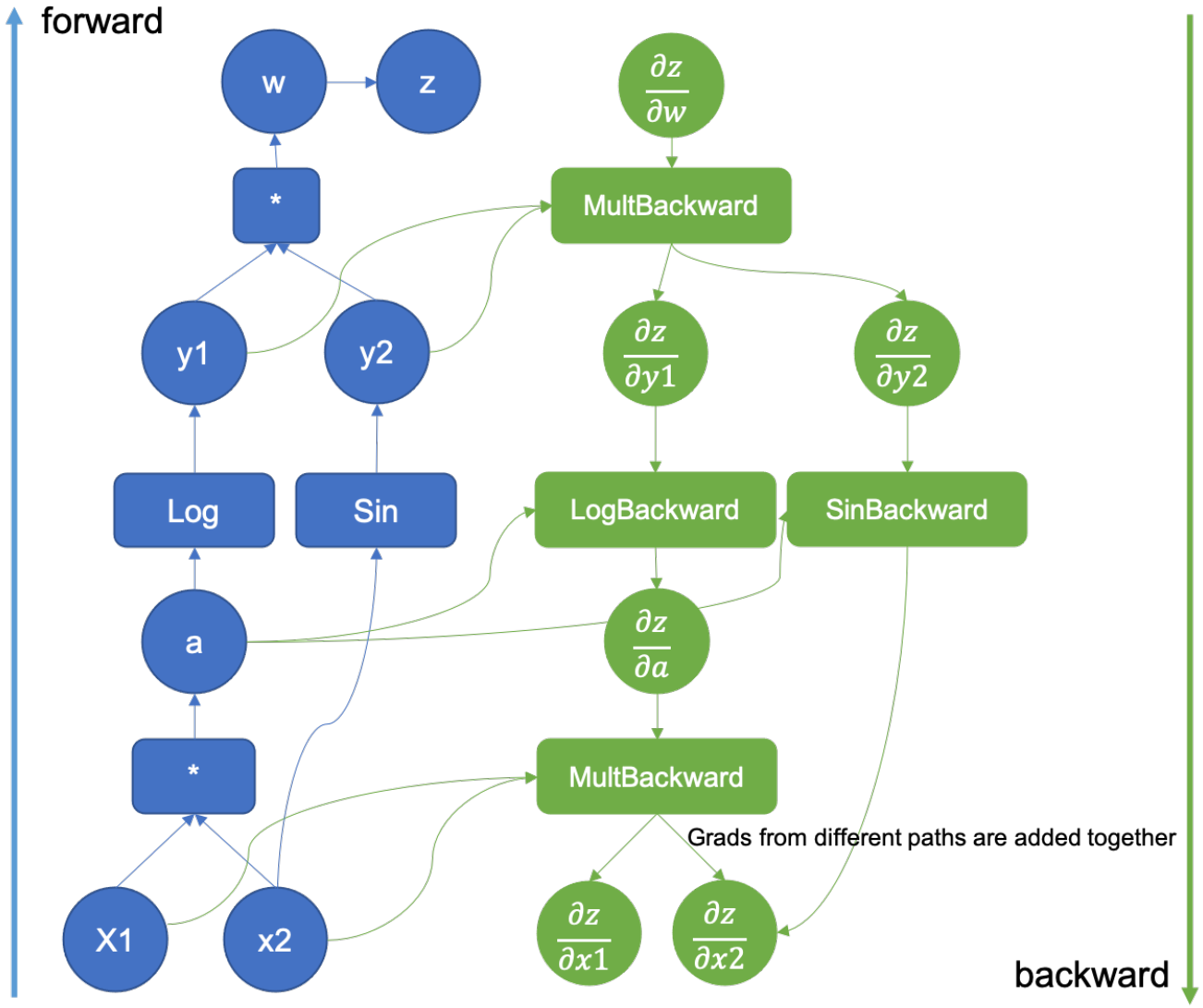
Autograd: dynamic computation graphs

During the forward pass, PyTorch records operations so it can run backprop later.



Autograd: dynamic computation graphs

During the forward pass, PyTorch records operations so it can run backprop later.



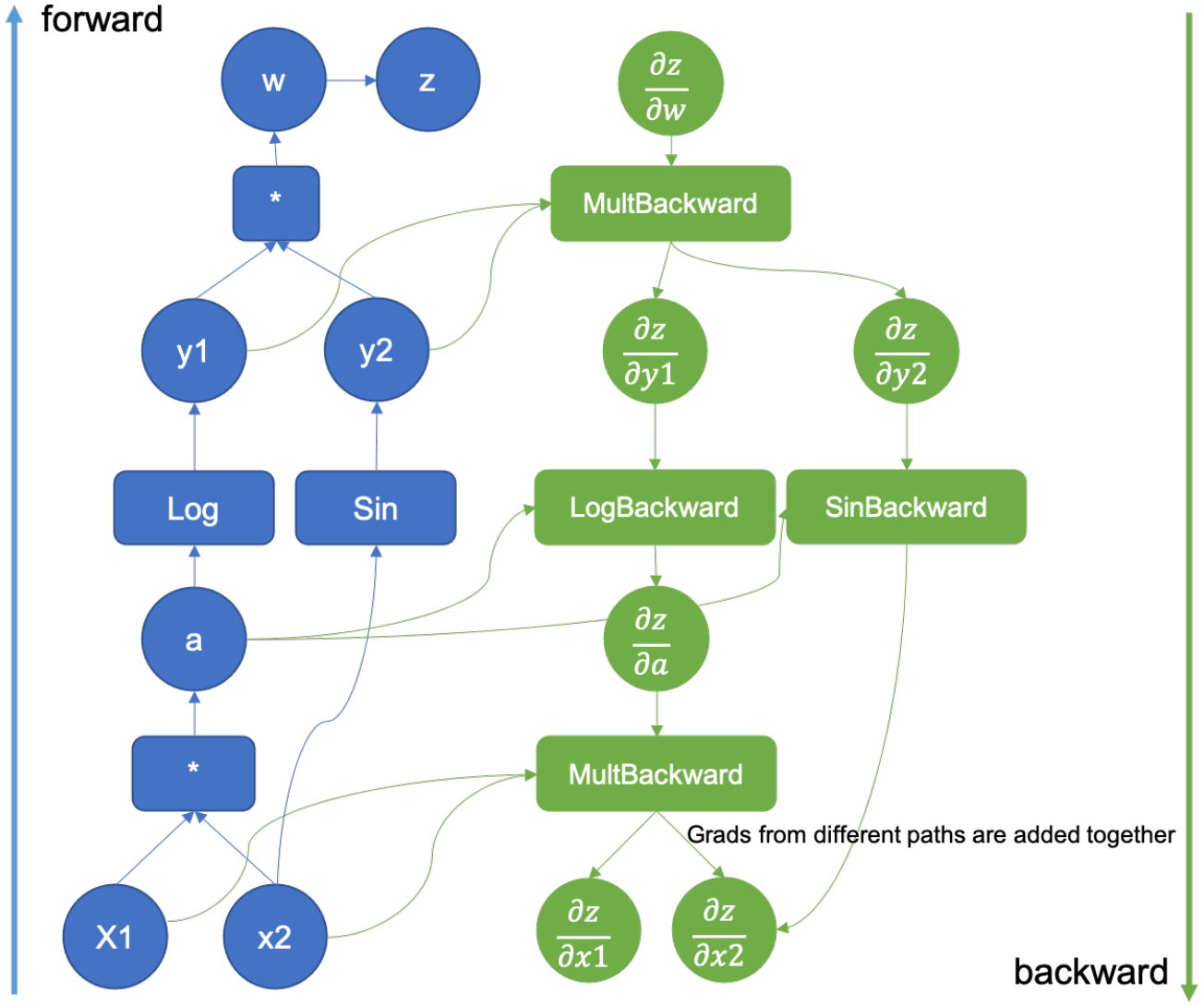
Key idea

Graph is built *as you run Python*.

If you do control flow (if/for), the graph matches what actually ran.

Autograd: dynamic computation graphs

During the forward pass, PyTorch records operations so it can run backprop later.



Key idea

Graph is built *as you run Python*.

If you do control flow (if/for), the graph matches what actually ran.

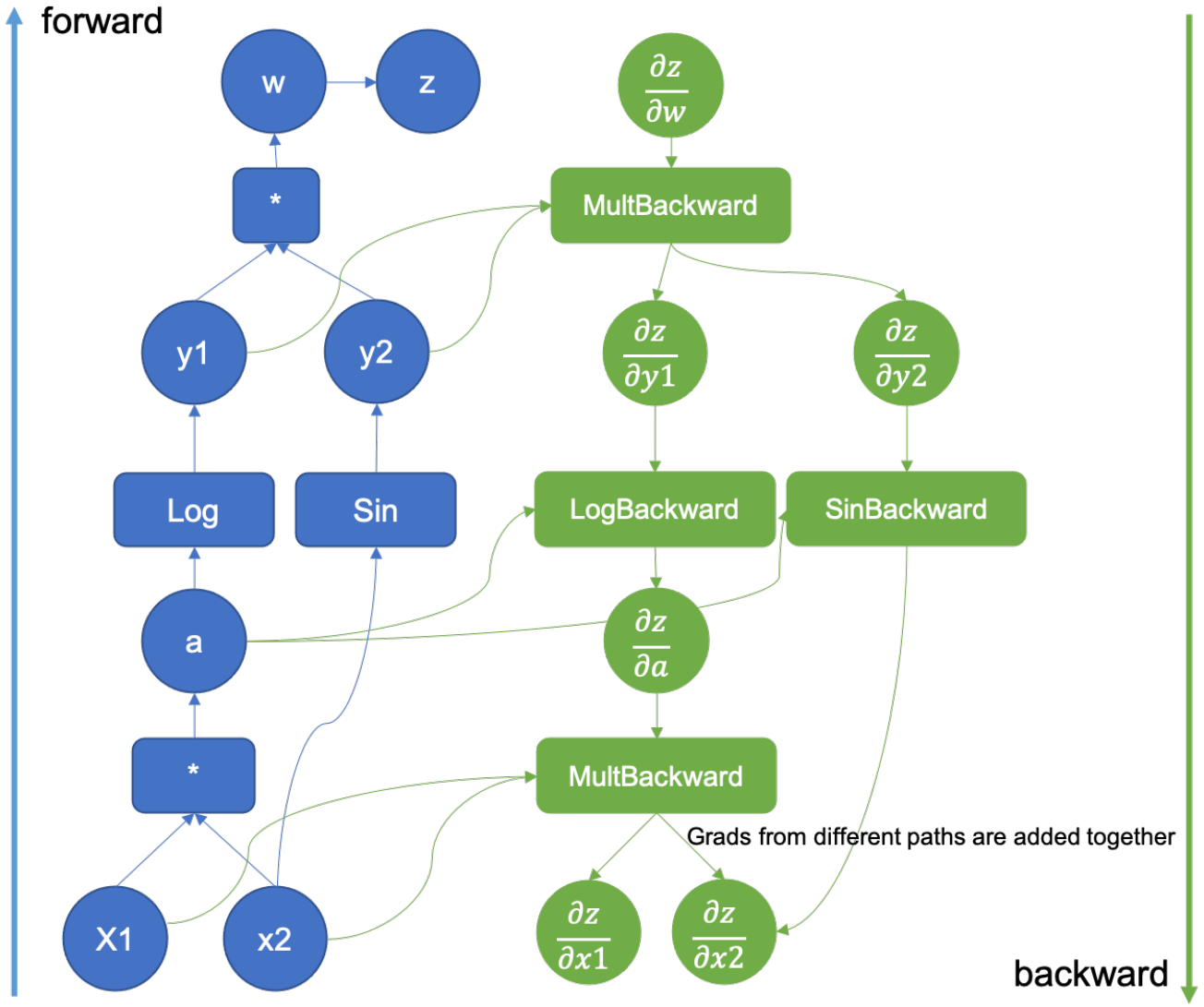
Backprop requires a scalar

Typically you compute a scalar loss (e.g., MSE, cross-entropy).

loss.backward() computes gradients for all parameters that contributed.

Autograd: dynamic computation graphs

During the forward pass, PyTorch records operations so it can run backprop later.



Key idea

Graph is built *as you run Python*.

If you do control flow (if/for), the graph matches what actually ran.

Backprop requires a scalar

Typically you compute a scalar loss (e.g., MSE, cross-entropy).

loss.backward() computes gradients for all parameters that contributed.

RL Losses

Your loss can be a policy gradient objective, TD error, or PPO clip loss — but it still ends as a scalar you backprop.

More on gradients

Where gradient information is stored

- Parameters in nn.Module have requires_grad=True by default
- Other tensors will not have requires_grad=True by default
- Gradients accumulate into .grad
- .grad is only populated for leaf tensors

Tiny example

```
1 w = torch.randn(3, requires_grad=True)
2 x = torch.randn(3)
3
4 y = (w * x).sum()
5 y.backward()
6
7 print(w.grad)   # tensor([...])
8 print(x.grad)   # None (requires_grad=False)
```

When gradients are tracked (and how to turn them off)

Useful tools:

Tool	What it does	Typical use
with torch.no_grad():	Disables gradient tracking (but tensors can later be used with autograd).	Evaluation loops, target networks, data preprocessing
with torch.inference_mode():	Like no_grad, but more restrictive and can be faster (disables extra autograd bookkeeping).	Pure inference / evaluation when you never need grads
x = x.detach()	Cuts the graph: x becomes a tensor that shares storage but does not track gradients.	Stop-grad in RL (e.g., advantage estimates, target values)

Part 3 — nn.Module

How you define models

nn.Module essentials

Modules are Python classes that hold parameters and define a forward pass.

A simple MLP

```
import torch
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self, in_dim, hidden=256, out_dim=10):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(in_dim, hidden),
            nn.ReLU(),
             nn.Linear(hidden, hidden),
             nn.ReLU(),
             nn.Linear(hidden, out_dim),
        )

    def forward(self, x):
        return self.net(x)

model = MLP(128)
```

nn.Module essentials

Modules are Python classes that hold parameters and define a forward pass.

A simple MLP

```python
import torch
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self, in_dim, hidden=256, out_dim=10):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(in_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, hidden),
            nn.ReLU(),
            nn.Linear(hidden, out_dim),
        )

    def forward(self, x):
        return self.net(x)

model = MLP(128)
```

Parameters & buffers

- Parameters: nn.Parameter → optimized
- Buffers: persistent state not optimized (e.g., running_mean in BatchNorm) → don't worry about this (we won't use BatchNorm)

Both appear in state_dict().

nn.Module essentials

Modules are Python classes that hold parameters and define a forward pass.

A simple MLP

```python
import torch
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self, in_dim, hidden=256, out_dim=10):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(in_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, hidden),
            nn.ReLU(),
            nn.Linear(hidden, out_dim),
        )

    def forward(self, x):
        return self.net(x)

model = MLP(128)
```

Parameters & buffers

- Parameters: nn.Parameter → optimized
- Buffers: persistent state not optimized (e.g., running_mean in BatchNorm) → don't worry about this (we won't use BatchNorm)

Both appear in state_dict().

Typical Modules in RL

Actor and critic can be separate Modules.

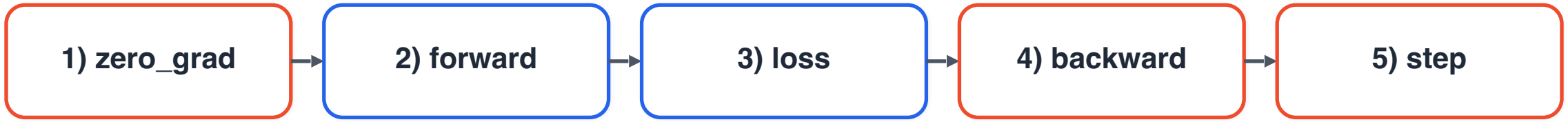
A common thing to do is share a trunk (e.g., a vision encoder) and have two heads (two Linear layers) for the actor and critic.

Part 4 — Training Loop

The “three-liner” + modern recipes

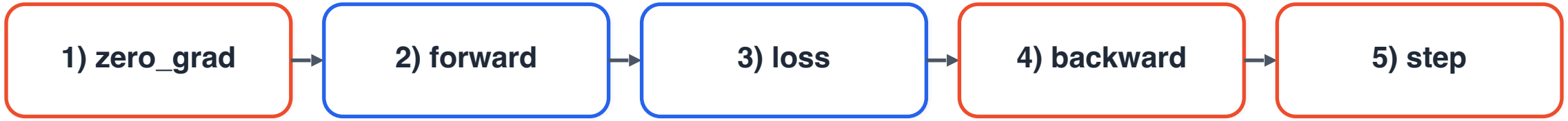
The canonical training step

Most training code is just this pattern, plus a few “recipe” extras:



The canonical training step

Most training code is just this pattern, plus a few “recipe” extras:

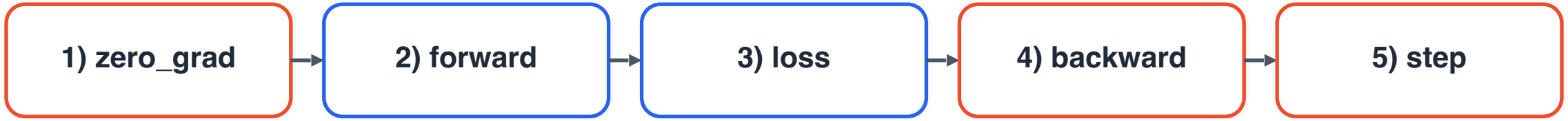


Minimal code

```
optimizer.zero_grad(set_to_none=True)
loss = compute_loss(model, batch)
loss.backward()
optimizer.step()
```

The canonical training step

Most training code is just this pattern, plus a few “recipe” extras:



Minimal code

```
optimizer.zero_grad(set_to_none=True)
loss = compute_loss(model, batch)
loss.backward()
optimizer.step()
```

Recipe extras

- gradient clipping
- weight decay (AdamW)
- lr schedulers
- mixed precision
- logging (loss, grad norms)

optimizer.zero_grad(): why and how

Why zero grads?

Because grads accumulate into param.grad.

If you forget, training usually “blows up” or behaves like you increased batch size unpredictably.

Two modes

```
# Recommended (often slightly faster + saves memory)
optimizer.zero_grad(set_to_none=True)

# Equivalent to setting grads to 0 (more work)
optimizer.zero_grad(set_to_none=False)
```

optimizer.zero_grad(): why and how

Why zero grads?

Because grads accumulate into param.grad.

If you forget, training usually “blows up” or behaves like you increased batch size unpredictably.

Gradient accumulation

If you *want* accumulation: do NOT zero every microbatch.

Example: accumulate N microbatches then step once.

Two modes

```
# Recommended (often slightly faster + saves memory)
optimizer.zero_grad(set_to_none=True)

# Equivalent to setting grads to 0 (more work)
optimizer.zero_grad(set_to_none=False)
```

Accumulation pattern

```
for i, batch in enumerate(loader):
    loss = compute_loss(batch) / accum_steps
    loss.backward()

    if (i + 1) % accum_steps == 0:
        clip_grad_norm_(model.parameters(), 1.0)
        optimizer.step()
        optimizer.zero_grad(set_to_none=True)
```

Optimizers: AdamW, SGD, and parameter groups

AdamW + param groups

```
import torch.optim as optim

# Common default for modern deep nets
optimizer = optim.AdamW(
    model.parameters(),
    lr=3e-4,
    weight_decay=1e-2,
)

# Parameter groups (e.g., different LR for head)
optimizer = optim.AdamW([
    {'params': model.trunk.parameters(), 'lr': 3e-4},
    {'params': model.head.parameters(),  'lr': 1e-3},
], weight_decay=1e-2)
```

Optimizers: AdamW, SGD, and parameter groups

AdamW + param groups

```python
import torch.optim as optim

# Common default for modern deep nets
optimizer = optim.AdamW(
    model.parameters(),
    lr=3e-4,
    weight_decay=1e-2,
)

# Parameter groups (e.g., different LR for head)
optimizer = optim.AdamW([
    {'params': model.trunk.parameters(), 'lr': 3e-4},
    {'params': model.head.parameters(),  'lr': 1e-3},
], weight_decay=1e-2)
```

Weight decay vs L2

AdamW uses decoupled weight decay (preferred over L2 penalty in Adam).

Tip: typically do NOT weight-decay biases or LayerNorm params (use groups).

Optimizers: AdamW, SGD, and parameter groups

AdamW + param groups

```python
import torch.optim as optim

# Common default for modern deep nets
optimizer = optim.AdamW(
    model.parameters(),
    lr=3e-4,
    weight_decay=1e-2,
)

# Parameter groups (e.g., different LR for head)
optimizer = optim.AdamW([
    {'params': model.trunk.parameters(), 'lr': 3e-4},
    {'params': model.head.parameters(),  'lr': 1e-3},
], weight_decay=1e-2)
```

Weight decay vs L2

AdamW uses decoupled weight decay (preferred over L2 penalty in Adam).

Tip: typically do NOT weight-decay biases or LayerNorm params (use groups).

RL note

Actor/critic often benefit from separate learning rates. You might experiment with tuning learning rates separately for these two (e.g., higher learning rate for the critic).

Gradient clipping (a common stability tool)

Why clip?

Prevents rare but huge gradients from destabilizing training.

Common in just about all modern NN training runs

Clip before optimizer.step()

```
1 from torch.nn.utils import clip_grad_norm_
2
3 loss.backward()
4
5 # Clip by global norm
6 max_norm = 1.0
7 grad_norm = clip_grad_norm_(model.parameters(),
max_norm)
8
9 optimizer.step()
```

Gradient clipping (a common stability tool)

Why clip?

Prevents rare but huge gradients from destabilizing training.

Common in just about all modern NN training runs

Clip before optimizer.step()

```
1 from torch.nn.utils import clip_grad_norm_
2
3 loss.backward()
4
5 # Clip by global norm
6 max_norm = 1.0
7 grad_norm = clip_grad_norm_(model.parameters(),
max_norm)
8
9 optimizer.step()
```

Mental model:

If $\|g\| > \text{max\_norm}$, rescale: $g \leftarrow g \cdot (\text{max\_norm} / \|g\|)$

Learning rate schedules

Schedulers don’t fix bad hyperparameters, but they often improve final performance.

Cosine schedule

```
from torch.optim.lr_scheduler import CosineAnnealingLR

optimizer = torch.optim.AdamW(model.parameters(), lr=3e-4)
scheduler = CosineAnnealingLR(optimizer, T_max=num_steps)

for step, batch in enumerate(loader):
    ...
    optimizer.step()
    scheduler.step()  # usually after optimizer.step()
```

Learning rate schedules

Schedulers don’t fix bad hyperparameters, but they often improve final performance.

Cosine schedule

```
1 from torch.optim.lr_scheduler import CosineAnnealingLR
2
3 optimizer = torch.optim.AdamW(model.parameters(), lr=3e-4)
4 scheduler = CosineAnnealingLR(optimizer, T_max=num_steps)
5
6 for step, batch in enumerate(loader):
7     ...
8     optimizer.step()
9     scheduler.step()  # usually after optimizer.step()
```

Where to call scheduler.step()

Most step-per-batch schedulers: after optimizer.step().

Plateau schedulers: call after validation metric (e.g., after validation loss plateaus)

Learning rate schedules

Schedulers don’t fix bad hyperparameters, but they often improve final performance.

Cosine schedule

```python
from torch.optim.lr_scheduler import CosineAnnealingLR

optimizer = torch.optim.AdamW(model.parameters(), lr=3e-4)
scheduler = CosineAnnealingLR(optimizer, T_max=num_steps)

for step, batch in enumerate(loader):
    ...
    optimizer.step()
    scheduler.step()  # usually after optimizer.step()
```

Where to call scheduler.step()

Most step-per-batch schedulers: after optimizer.step().

Plateau schedulers: call after validation metric (e.g., after validation loss plateaus)

Warmup (common recipe)

For large models: ramp LR from 0 → target over first few % of steps.
Implement with LambdaLR or a custom schedule.

Learning rate schedules

Schedulers don’t fix bad hyperparameters, but they often improve final performance.

Cosine schedule

```python
from torch.optim.lr_scheduler import CosineAnnealingLR

optimizer = torch.optim.AdamW(model.parameters(), lr=3e-4)
scheduler = CosineAnnealingLR(optimizer, T_max=num_steps)

for step, batch in enumerate(loader):
    ...
    optimizer.step()
    scheduler.step()  # usually after optimizer.step()
```

Where to call scheduler.step()

Most step-per-batch schedulers: after optimizer.step().

Plateau schedulers: call after validation metric (e.g., after validation loss plateaus)

Warmup (common recipe)

For large models: ramp LR from 0 → target over first few % of steps.
Implement with LambdaLR or a custom schedule.

Schematic: warmup + cosine decay

Part 5 — Data & Evaluation

Feeding in training data, and also evaluating models

Data pipeline: Dataset → DataLoader

Typical setup

```python
from torch.utils.data import Dataset, DataLoader

class MyDataset(Dataset):
    def __len__(self):
        return N
    def __getitem__(self, idx):
        return obs[idx], target[idx]

loader = DataLoader(
    MyDataset(),
    batch_size=256,
    shuffle=True,
    num_workers=4,
    pin_memory=True,
)
```

Key knobs

shuffle: True for SGD
num_workers: parallel CPU loading
pin_memory: faster CPU→GPU copies

In RL, your training dataset will typically be a replay buffer

Data pipeline: Dataset → DataLoader

Typical setup

```python
from torch.utils.data import Dataset, DataLoader

class MyDataset(Dataset):
    def __len__(self):
        return N
    def __getitem__(self, idx):
        return obs[idx], target[idx]

loader = DataLoader(
    MyDataset(),
    batch_size=256,
    shuffle=True,
    num_workers=4,
    pin_memory=True,
)
```

Key knobs

shuffle: True for SGD
num_workers: parallel CPU loading
pin_memory: faster CPU→GPU copies

In RL, your training dataset will typically be a replay buffer

Gotcha

If your Dataset returns NumPy arrays, convert to torch tensors in __getitem__.
Avoid mixing numpy ops in the training step.

Training vs evaluation mode

Two separate switches:

1) model.train() / model.eval()

Changes behavior of certain layers:
- Dropout: on/off
- BatchNorm: batch stats vs running stats

Always set the right mode.

2) grad mode (no_grad / inference_mode)

Controls whether autograd tracks ops.

During evaluation:

```
model.eval()
with torch.inference_mode():
    ...
```

Training vs evaluation mode

Two separate switches:

1) model.train() / model.eval()

Changes behavior of certain layers:
- Dropout: on/off
- BatchNorm: batch stats vs running stats

Always set the right mode.

2) grad mode (no_grad / inference_mode)

Controls whether autograd tracks ops.

During evaluation:
model.eval()
with torch.inference_mode():
    ...

Recommended pattern

```
model.train()
for batch in train_loader:
    loss = ...
    ...

model.eval()
with torch.inference_mode():
    for batch in val_loader:
        metrics = ...
```

Part 6 — Other training loop things

Checkpointing, debugging, best practices

Saving & loading: state_dict and checkpoints

Save weights (and optimizer state) so training can resume exactly.

Checkpoint pattern

```
# Save
ckpt = {
  'model': model.state_dict(),
  'optim': optimizer.state_dict(),
  'step': step,
  'rng': torch.get_rng_state(),
}
torch.save(ckpt, 'ckpt.pt')

# Load
ckpt = torch.load('ckpt.pt', map_location='cpu', weights_only=False)
model.load_state_dict(ckpt['model'])
optimizer.load_state_dict(ckpt['optim'])
```

Best practice

For *weights only* files: save and load just model.state_dict().

When loading from untrusted sources, prefer weights_only=True.

Newer PyTorch (>=2.6)

torch.load changed the default for weights_only to improve security from False to True. If you rely on loading arbitrary pickled objects, you may need to manually specify weights_only=False.

Debugging toolbox

First steps

- Print tensor shapes + dtypes
- Check devices (CPU vs CUDA)
- Look for NaNs/Infs: torch.isnan / torch.isfinite
- Overfit a tiny batch (sanity test)

Debugging toolbox

First steps

- Print tensor shapes + dtypes
- Check devices (CPU vs CUDA)
- Look for NaNs/Infs: torch.isnan / torch.isfinite
- Overfit a tiny batch (sanity test)

Common culprits

- Forgetting model.train()/eval()
- Wrong target dtype (CE needs int64)
- Learning rate too high
- Silent broadcasting bug
- Accidental in-place ops on tensors needed for backward

Debugging toolbox

First steps

- Print tensor shapes + dtypes
- Check devices (CPU vs CUDA)
- Look for NaNs/Infs: torch.isnan / torch.isfinite
- Overfit a tiny batch (sanity test)

Common culprits

- Forgetting model.train()/eval()
- Wrong target dtype (CE needs int64)
- Learning rate too high
- Silent broadcasting bug
- Accidental in-place ops on tensors needed for backward

RL-specific gotcha

Stop-grad mistakes: accidentally backprop through targets/returns/advantages can cause instability. Use detach() intentionally.

Debugging toolbox

First steps

- Print tensor shapes + dtypes
- Check devices (CPU vs CUDA)
- Look for NaNs/Infs: torch.isnan / torch.isfinite
- Overfit a tiny batch (sanity test)

Useful snippets

```
torch.autograd.set_detect_anomaly(True)

# Track grad norms
with torch.no_grad():
    total = 0.0
    for p in model.parameters():
        if p.grad is not None:
            total += p.grad.norm().item()
    print('grad_norm', total)

# Helpful for NaNs
torch.cuda.synchronize()
```

Common culprits

- Forgetting model.train()/eval()
- Wrong target dtype (CE needs int64)
- Learning rate too high
- Silent broadcasting bug
- Accidental in-place ops on tensors needed for backward

RL-specific gotcha

Stop-grad mistakes: accidentally backprop through targets/returns/advantages can cause instability. Use detach() intentionally.

Gotchas & best practices

Gotchas

- Mixing NumPy and torch in the training step (breaks device/grad flow)
- Forgetting .to(device) on targets/masks
- In-place ops (x += y) on values needed for backward
- view() on non-contiguous tensors (use reshape or .contiguous())
- Calling softmax before CrossEntropyLoss

Best practices

- Keep everything in torch; convert NumPy at the boundary
- Use optimizer.zero_grad(set_to_none=True)
- Log: loss, learning rate, grad_norm, parameter_norm
- Save checkpoints with model/optimizer state_dict
- Start simple: overfit a tiny batch → scale up
- Use inference_mode for evaluation loops

Part 7 — Modern PyTorch (2.x)

Not needed for this class, but nonetheless might be useful

torch.compile: speed up training with minimal changes

PyTorch 2.x introduced a compiler stack you can often use with one line.

One-liner

```
model = MyNet().to(device)

# Drop-in compilation (PyTorch 2.0+)
model = torch.compile(model)

for batch in loader:
    ...
```

When it helps

- Larger models
- Many repeated ops
- GPU-bound training

With “graph breaks”, you may get smaller gains (but the code will still be correct).

When to skip (for class)

First implementation: start in eager mode.
Turn on compile only after your code seems to work.

Checklist: a solid PyTorch training script

- Set device + move model and all batch tensors
- model.train() in training; model.eval() + inference_mode() in evaluation
- Training step: zero_grad(set_to_none=True) → forward → loss → backward → clip → step
- Use AdamW (and parameter groups if needed) + an LR schedule
- Log loss, LR, grad norms; checkpoint state_dict regularly
- Don’t mix NumPy ops into the training step; keep everything in torch