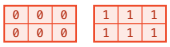
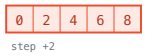
PyTorch

The everyday PyTorch loop, from tensors and autograd to building models, training them, and running on a GPU.

Create Tensors

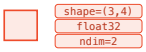
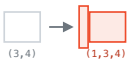
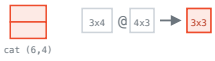
Spin up tensors from data, ranges, and random noise.

PURPOSE	COMMAND	RESULT
From a Python list.	`torch.tensor([[1, 2], [3, 4]])`	
Filled with zeros / ones.	`torch.zeros(2,3) torch.ones(2,3)`	
Evenly spaced range.	`torch.arange(0, 10, 2)`	
Random normal noise.	`torch.randn(2, 2)`	
Bridge from NumPy.	`torch.from_numpy(arr)`	
Match shape of another.	`torch.zeros_like(x)`	

a tensor is PyTorch's n-dimensional array; dtype and shape are inferred at creation

Shapes, Indexing & Ops

Reshape, slice, and combine tensors without copying when you can.

PURPOSE	COMMAND	RESULT
Inspect shape / dtype.	`x.shape x.dtype x.ndim`	
Reshape / flatten.	`x.reshape(2, 6) x.view(-1)`	
Add / drop a dim.	`x.unsqueeze(0) x.squeeze()`	
Transpose / permute axes.	`x.permute(1, 0)`	
Index, slice, mask.	`x[:, 1] x[x > 5]`	
Concatenate / matrix multiply.	`torch.cat([a, b]) a @ b.T`	

reshape/view reinterpret the same data; @ does the matrix multiply at the heart of every layer

Autograd (gradients)

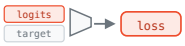
Tensors that track history; .backward() fills in the gradients.

PURPOSE	COMMAND	RESULT
Track gradients.	`torch.tensor([2.], requires_grad=True)`	
Build a computation.	`y = w * x + b`	
Compute gradients.	`y.backward()`	
Read a gradient.	`w.grad`	
Stop tracking (eval / infer).	`withtorch.no_grad(): ...`	
Detach a branch.	`z.detach()`	

requires_grad=True records every op; .backward() walks the graph in reverse, filling each leaf's .grad

Build a Model (nn.Module)

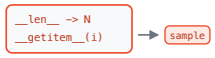
Layers are parameters with a forward; stack them into a module.

PURPOSE	COMMAND	RESULT
A linear (dense) layer.	`nn.Linear(10, 20)`	
Stack layers in order.	`nn.Sequential(nn.Linear(10,20),` `nn.ReLU(), nn.Linear(20,2))`	
Subclass for custom forward.	`classNet(nn.Module):` `defforward(self, x): ...`	
Apply an activation.	`F.relu(x)`	
Pick a loss.	`nn.CrossEntropyLoss()`	
Count / list parameters.	`sum(p.numel() for p in m.parameters())`	

layers go in __init__, the data path in forward; the module auto-registers weights as parameters

Data Loading

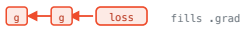
A Dataset indexes samples; a DataLoader batches and shuffles them.

PURPOSE	COMMAND	RESULT
Wrap tensors as a dataset.	`TensorDataset(X, y)`	
Custom dataset (3 methods).	`class DS(Dataset): __len__ __getitem__`	
Batch + shuffle.	`DataLoader(ds, batch_size=16, shuffle=True)`	
Iterate batches.	`for xb, yb in loader:`	
Grab one batch (debug).	`xb, yb = next(iter(loader))`	
Split train / val.	`random_split(ds, [80, 20])`	

a Dataset answers __len__ and __getitem__; next(iter(loader)) sanity-checks one batch's shapes

The Training Loop

Zero grads, forward, loss, backward, step. Repeat each batch.

PURPOSE	COMMAND	RESULT
Pick an optimizer.	`optim.AdamW(model.parameters(), lr=1e-3)`	
Clear old gradients.	`optimizer.zero_grad()`	
Forward + loss.	`loss = loss_fn(model(xb), yb)`	
Backpropagate.	`loss.backward()`	
Update the weights.	`optimizer.step()`	
Anneal the rate.	`scheduler.step()`	

five beats: zero_grad, forward, loss, backward, step; gradients accumulate, so zeroing first is mandatory

Save and Load

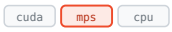
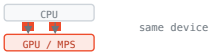
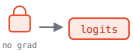
Save the state_dict, not the object; reload with weights_only=True.

PURPOSE	COMMAND	RESULT
Save model weights.	`torch.save(model.state_dict(), "model.pth")`	
Load weights back.	`model.load_state_dict(torch.load("model.pth",weights_only=True))`	
Save a full checkpoint.	`torch.save({"model":.., "opt":.., "epoch":e})`	
Set the right mode.	`model.eval()` `model.train()`	
Save a plain tensor.	`torch.save(t, "t.pt")`	
Export a portable graph.	`torch.export.export(model, (ex,))`	

save the state_dict (a plain dict of tensors); always pass weights_only=True to avoid running pickled code

Devices and Inference

Move model and data to the accelerator; run with no gradients.

PURPOSE	COMMAND	RESULT
Pick the accelerator.	`device =torch.accelerator.current_accelerator() ... else "cpu"`	
Move model + data over.	`model.to(device) xb.to(device)`	 same device
Run inference safely.	`withtorch.inference_mode(): out = model(x)`	
Logits to a prediction.	`out.argmax(dim=1)`	
Logits to probabilities.	`torch.softmax(out, dim=1)`	
Make a run reproducible.	`torch.manual_seed(0)`	 same every run

torch.accelerator picks CUDA / MPS / CPU; run predictions under inference_mode, then argmax or softmax