

PyTorch Quick Reference

Tensors, autograd, neural networks, and training

Tensors

Creating Tensors

```
import torch
a = torch.tensor([1, 2, 3])
b = torch.zeros(2, 3)
c = torch.ones(3, 3)
d = torch.randn(2, 4) # normal dist
```

Tensor Constructors

torch.zeros(m, n)	All zeros, shape (m, n)
torch.ones(m, n)	All ones, shape (m, n)
torch.randn(m, n)	Standard normal random
torch.arange(start, end, step)	Evenly spaced values
torch.linspace(start, end, steps)	Fixed number of points
torch.eye(n)	Identity matrix
torch.empty(m, n)	Uninitialized memory

NumPy Interop

```
t = torch.from_numpy(np_array)
arr = tensor.numpy() # shares memory
t = torch.as_tensor(np_array)
```

Autograd

Tracking Gradients

```
x = torch.tensor([2.0, 3.0],
                 requires_grad=True)
y = (x ** 2).sum()
y.backward()
print(x.grad) # tensor([4., 6.]
```

Disabling Gradient Tracking

```
with torch.no_grad():
    pred = model(x) # inference only
x_det = x.detach() # detach from graph
```

Gradient Control

x.requires_grad_(True)	Enable grad tracking in-place
x.grad.zero_()	Reset accumulated gradients
x.detach()	New tensor without grad history
x.grad	Access stored gradients

Neural Networks

Define a Model

```
import torch.nn as nn
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 10)
    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return self.fc2(x)
```

Sequential Model

```
model = nn.Sequential(
    nn.Linear(784, 256),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(256, 10))
```

Common Layers

nn.Linear(in, out)	Fully connected layer
nn.Conv2d(c_in, c_out, k)	2D convolution, kernel size k
nn.BatchNorm2d(n)	Batch normalization
nn.LSTM(in, hidden)	LSTM recurrent layer
nn.Dropout(p)	Dropout with probability p
nn.Embedding(vocab, dim)	Embedding lookup table

Data Loading

Custom Dataset

```
from torch.utils.data import Dataset, DataLoader
class MyData(Dataset):
    def __init__(self, X, y):
        self.X, self.y = X, y
    def __len__(self): return len(self.X)
    def __getitem__(self, i):
        return self.X[i], self.y[i]
```

DataLoader

```
loader = DataLoader(dataset, batch_size=32,
                   shuffle=True, num_workers=2)
for batch_x, batch_y in loader:
    output = model(batch_x)
```

Built-in Datasets

```
from torchvision import datasets, transforms
t = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, ), (0.5,))])
data = datasets.MNIST("data", train=True,
                    download=True, transform=t)
```

Training Loop

Standard Training Loop

```
model.train()
for epoch in range(num_epochs):
    for X, y in train_loader:
        optimizer.zero_grad()
        loss = criterion(model(X), y)
        loss.backward()
        optimizer.step()
```

Evaluation

```
model.eval()
with torch.no_grad():
    correct = 0
    for X, y in test_loader:
        pred = model(X).argmax(dim=1)
        correct += (pred == y).sum().item()
```

Training Checklist

model.train()	Enable dropout / batch norm training
model.eval()	Switch to inference mode
optimizer.zero_grad()	Clear gradients before backward
loss.backward()	Compute gradients
optimizer.step()	Update parameters

Optimizers

Common Optimizers

```
import torch.optim as optim
opt = optim.SGD(model.parameters(), lr=0.01,
               momentum=0.9)
opt = optim.Adam(model.parameters(), lr=1e-3)
opt = optim.AdamW(model.parameters(), lr=1e-3,
                  weight_decay=0.01)
```

Learning Rate Scheduler

```
sched = optim.lr_scheduler.StepLR(
    opt, step_size=10, gamma=0.1)
# in loop: sched.step() after each epoch
```

Optimizer Comparison

SGD	Simple, needs tuning, good with momentum
Adam	Adaptive LR, fast convergence, default
AdamW	Adam with decoupled weight decay
RMSprop	Adaptive, good for RNNs

Loss Functions

Common Loss Functions

nn.CrossEntropyLoss()	Classification (logits, no softmax)
nn.BCEWithLogitsLoss()	Binary classification (logits)
nn.MSELoss()	Regression (mean squared error)
nn.L1Loss()	Regression (mean absolute error)
nn.NLLLoss()	Negative log-likelihood (after log_softmax)
nn.HuberLoss()	Robust regression (less outlier-sensitive)

Usage

```
criterion = nn.CrossEntropyLoss()
loss = criterion(logits, targets)
# logits: (batch, classes), targets: (batch,)
```

Custom Loss

```
def focal_loss(pred, target, gamma=2.0):
    ce = nn.functional.cross_entropy(
        pred, target, reduction="none")
    pt = torch.exp(-ce)
    return ((1 - pt) ** gamma * ce).mean()
```

Saving & Loading

Save / Load State Dict (Recommended)

```
torch.save(model.state_dict(), "model.pt")
model = Net()
model.load_state_dict(
    torch.load("model.pt", weights_only=True))
```

Save Full Checkpoint

```
torch.save({
    "epoch": epoch,
    "model": model.state_dict(),
    "optimizer": opt.state_dict(),
    "loss": loss}, "checkpoint.pt")
```

Load Checkpoint

```
ckpt = torch.load("checkpoint.pt",
                 weights_only=False)
model.load_state_dict(ckpt["model"])
opt.load_state_dict(ckpt["optimizer"])
```

GPU

Device Management

```
device = torch.device(
    "cuda" if torch.cuda.is_available()
    else "cpu")
model = model.to(device)
x = x.to(device)
```

PyTorch Quick Reference

GPU Utilities

<code>torch.cuda.is_available()</code>	Check if CUDA is available
<code>torch.cuda.device_count()</code>	Number of GPUs
<code>torch.cuda.memory_allocated()</code>	Current GPU memory usage (bytes)
<code>torch.cuda.empty_cache()</code>	Free unused cached memory

Multi-GPU

```
if torch.cuda.device_count() > 1:
    model = nn.DataParallel(model)
model = model.to(device)
```

Common Patterns

Weight Initialization

```
def init_weights(m):
    if isinstance(m, nn.Linear):
        nn.init.xavier_uniform_(m.weight)
        m.bias.data.fill_(0.01)
model.apply(init_weights)
```

Gradient Clipping

```
torch.nn.utils.clip_grad_norm_(
    model.parameters(), max_norm=1.0)
```

Freeze Layers

```
for param in model.fc1.parameters():
    param.requires_grad = False
```

Model Summary

```
total = sum(p.numel()
            for p in model.parameters())
trainable = sum(p.numel()
                for p in model.parameters()
                if p.requires_grad)
```