PyTorch Basics Cheat Sheet

---

1) Core ideas

- **Tensor**: PyTorch's n-dimensional array (like NumPy), can live on CPU/GPU.
- **Device**: where computation happens (`cpu` , `cuda` , `mps`).
- **Autograd**: automatic differentiation system that tracks operations to compute gradients.
- **`nn.Module`** : base class for neural network models/layers.
- **Parameters**: learnable tensors (usually weights/biases) stored in a module.
- **Logits**: raw model outputs before softmax/sigmoid (common for classification).
- **Training loop**: forward → loss → backward → optimizer step (repeat).
- **Evaluation**: run forward only (no gradients), compute metrics.

---

2) Most common imports

```
import torch
import torch.nn as nn
import torch.nn.functional as F

from torch.utils.data import Dataset, DataLoader, TensorDataset, random_split
import torchvision
from torchvision import datasets, transforms
```

---

3) Device setup (CPU/GPU/MPS)

Pick a device

```
device = (
    "cuda" if torch.cuda.is_available()
    else "mps" if torch.backends.mps.is_available()
    else "cpu"
)
print("device:", device)
```

Move tensors / models to device

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

Rule: model params and batch tensors must be on the same device.

---

4) Working with tensors

Create tensors

```
torch.tensor([1, 2, 3])                # from Python
torch.zeros(2, 3)                       # all zeros
torch.ones(2, 3)                        # all ones
torch.randn(2, 3)                       # N(0,1)
torch.rand(2, 3)                        # Uniform[0,1)
torch.arange(0, 10, 2)                  # 0,2,4,6,8
torch.linspace(0, 1, steps=5)           # 0..1 inclusive
```

Dtype and device

```
t = torch.randn(3, dtype=torch.float32, device=device)
t.dtype     # torch.float32
t.device    # cpu/cuda/mps
```

Convert:

```
t = t.float()
t = t.long()
t = t.to(torch.float16)
```

Shapes

```
t.shape          # torch.Size([...])
t.ndim           # number of dimensions
t.numel()        # number of elements
len(t)           # size of dimension 0
```

Reshape / view

```
t = torch.randn(2, 3, 4)

t.view(6, 4)         # shares memory (requires contiguous)
t.reshape(6, 4)      # safer, may copy
t.flatten()          # flatten all dims
t.unsqueeze(0)       # add dimension
t.squeeze()          # remove size-1 dimensions
t.permute(2, 0, 1)   # reorder dims
t.transpose(0, 1)    # swap 2 dims
```

Indexing and slicing

```
t[0]           # first row
t[:, 1]        # all rows, column 1
t[0, 1:3]      # slice
t[t > 0]       # boolean mask
```

Concatenate and stack

```
a = torch.randn(2, 3)
b = torch.randn(2, 3)

torch.cat([a, b], dim=0)     # (4, 3)
torch.stack([a, b], dim=0)   # (2, 2, 3)
```

Basic math

Elementwise:

```
a + b
a * b
torch.sin(a)
```

Matrix multiplication:

```
A = torch.randn(10, 5)
B = torch.randn(5, 3)
A @ B              # (10,3)
torch.matmul(A, B) # same
```

Reductions:

```
t.sum()
t.mean(dim=0)
t.max(dim=1).values
t.argmax(dim=1)
```

Convert to/from NumPy (CPU only)

```
np_arr = t.detach().cpu().numpy()
t2 = torch.from_numpy(np_arr)  # shares memory when possible
```

---

5) Autograd essentials (gradients)

Track gradients

```
x = torch.randn(3, requires_grad=True)
y = (x * x).sum()
y.backward()
x.grad
```

Stop tracking gradients

- Use when evaluating or doing non-learning operations.

```
with torch.no_grad():
    out = model(x)

# or (often faster than no_grad for inference)
with torch.inference_mode():
    out = model(x)
```

Detach a tensor from the graph

```
z = y.detach()      # no grad history
```

Zero gradients (important!)

Gradients accumulate by default.

```
optimizer.zero_grad(set_to_none=True)
```

---

6) Datasets & DataLoaders

Common pattern

- **Dataset**: defines how to get one sample
- **DataLoader**: batches, shuffles, parallel loading

Built-in dataset example (MNIST)

```
transform = transforms.ToTensor()

train_data = datasets.MNIST(
    root="data", train=True, download=True, transform=transform
)
test_data = datasets.MNIST(
    root="data", train=False, download=True, transform=transform
)
```

Train/val split

```
train_size = 55_000
val_size = len(train_data) - train_size
train_ds, val_ds = random_split(train_data, [train_size, val_size])
```

DataLoaders

```
train_loader = DataLoader(train_ds, batch_size=64, shuffle=True)
val_loader   = DataLoader(val_ds, batch_size=256, shuffle=False)
test_loader  = DataLoader(test_data, batch_size=256, shuffle=False)
```

Useful DataLoader args

- `batch_size`: samples per batch
- `shuffle=True`: shuffle each epoch (training)
- `num_workers`: parallel data loading (start with `0` for compatibility)
- `pin_memory=True`: can speed up CPU→GPU transfers (CUDA)

---

7) Preprocessing / transforms

Compose multiple transforms

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))  # MNIST mean/std
])
```

Common transforms:

- `Resize`, `CenterCrop`, `RandomCrop`
- `RandomHorizontalFlip`, `ColorJitter`
- `Normalize(mean, std)`
- `ToTensor()`

---

8) Building models

`nn.Module` template

```
class MyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer = nn.Linear(10, 5)

    def forward(self, x):
        return self.layer(x)
```

A typical MLP (for MNIST-like data)

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28*28, 256),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(128, 10)
        )

    def forward(self, x):
        return self.net(x)  # returns logits
```

Inspect parameters

```
sum(p.numel() for p in model.parameters())            # total params
sum(p.numel() for p in model.parameters() if p.requires_grad)
```

Switch modes

```
model.train()  # enables dropout, etc.
model.eval()   # disables dropout, uses running stats for batchnorm
```

---

9) Loss functions

Regression

```
loss_fn = nn.MSELoss()   # mean squared error
loss_fn = nn.L1Loss()    # mean absolute error
```

Binary classification

- Prefer `BCEWithLogitsLoss` (expects logits, more stable than sigmoid + BCE):

```
loss_fn = nn.BCEWithLogitsLoss()
# model outputs shape: (B,) or (B,1)
# targets are float: 0.0/1.0
```

Multi-class classification (e.g., MNIST)

- Prefer `CrossEntropyLoss` (expects logits):

```
loss_fn = nn.CrossEntropyLoss()
# logits: (B, C)
# targets: (B,) long with class indices 0..C-1
```

Important:
Do **not** apply `softmax` before `CrossEntropyLoss`.

---

10) Optimizers

Common optimizers

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-2)
```

Typical update step

```
optimizer.zero_grad(set_to_none=True)
loss.backward()
optimizer.step()
```

(Optional) Learning rate scheduler

```
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)

# call once per epoch
scheduler.step()
```

---

11) Training loop

Single-epoch training loop

```
def train_one_epoch(model, loader, loss_fn, optimizer, device):
    model.train()
    total_loss = 0.0
    total_correct = 0
    total_examples = 0

    for X, y in loader:
        X, y = X.to(device), y.to(device)

        logits = model(X)
        loss = loss_fn(logits, y)

        optimizer.zero_grad(set_to_none=True)
        loss.backward()
        optimizer.step()

        batch_size = y.size(0)
        total_loss += loss.item() * batch_size
        total_correct += (logits.argmax(dim=1) == y).sum().item()
        total_examples += batch_size

    return total_loss / total_examples, total_correct / total_examples
```

---

12) The evaluation loop (no gradients)

```python
@torch.inference_mode()
def evaluate(model, loader, loss_fn, device):
    model.eval()
    total_loss = 0.0
    total_correct = 0
    total_examples = 0

    for X, y in loader:
        X, y = X.to(device), y.to(device)

        logits = model(X)
        loss = loss_fn(logits, y)

        batch_size = y.size(0)
        total_loss += loss.item() * batch_size
        total_correct += (logits.argmax(dim=1) == y).sum().item()
        total_examples += batch_size

    return total_loss / total_examples, total_correct / total_examples
```

---

13) Full training skeleton

```python
epochs = 10

for epoch in range(1, epochs + 1):
    train_loss, train_acc = train_one_epoch(model, train_loader, loss_fn, optimizer, devi
    val_loss, val_acc = evaluate(model, val_loader, loss_fn, device)

    print(
        f"Epoch {epoch:02d} | "
        f"train loss {train_loss:.4f}, train acc {train_acc*100:.2f}% | "
        f"val loss {val_loss:.4f}, val acc {val_acc*100:.2f}%"
    )
```

14) Saving & loading models

Best practice: save `state_dict`

```
torch.save(model.state_dict(), "model.pt")
```

Load:

```
model = MLP().to(device)
model.load_state_dict(torch.load("model.pt", map_location=device))
model.eval()
```

Save optimizer

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

Load checkpoint:

```
ckpt = torch.load("checkpoint.pt", map_location=device)
model.load_state_dict(ckpt["model_state"])
optimizer.load_state_dict(ckpt["optim_state"])
start_epoch = ckpt["epoch"] + 1
```

15) Transfer learning & fine-tuning

Load a pretrained model (example)

```
import torchvision.models as models
model = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)
```

Freeze backbone

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

Replace classifier head (ResNet example)

```
num_features = model.fc.in_features
model.fc = nn.Linear(num_features, 10)  # new task with 10 classes
model = model.to(device)
```

Optimizer should only see trainable params

```
optimizer = torch.optim.AdamW(
    (p for p in model.parameters() if p.requires_grad),
    lr=1e-3
)
```

---

16) Debugging

Check a single batch

```
X, y = next(iter(train_loader))
print(X.shape, X.dtype, X.min().item(), X.max().item())
print(y.shape, y.dtype, y[:10])
```

Check model output shapes

```
X = X.to(device)
logits = model(X)
print(logits.shape)  # should be (batch_size, num_classes)
```

Common shape expectations

- `CrossEntropyLoss`:
    - logits: `(B, C)`
    - target labels: `(B,)` **long**
- `BCEWithLogitsLoss`:
    - logits: `(B,)` or `(B,1)`
    - targets: same shape, **float** in {0,1}

If loss becomes NaN

- Lower `lr`
- Ensure stable loss usage (use logits-based losses)
- Check inputs are normalized
- Look for exploding gradients (try gradient clipping):

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

---

17) Reproducibility

```
import random
import numpy as np
import torch

seed = 42
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)
```

---

18) Quick glossary of “modes” and “contexts”

- `model.train()`
  Enables training behaviors (Dropout active, BatchNorm updates running stats)
- `model.eval()`
  Disables training-only behaviors (Dropout off, BatchNorm uses stored stats)
- `torch.no_grad()`
  Don’t track gradients (evaluation, saves memory)
- `torch.inference_mode()`
  Like `no_grad` but even more optimized for inference