WHAT IS PYTORCH?

Definition

PyTorch is an open-source deep learning framework by Meta AI providing:
(1) N-dimensional tensor computation (GPU NumPy)
(2) Automatic differentiation (`autograd`)
(3) Dynamic computation graph (Define-by-Run)

Why Dynamic Graph?

Graph built *on the fly* during execution. Use normal Python `if/else`, `print()`, `pdb` directly on models. TF1.x used static graphs — harder to debug.

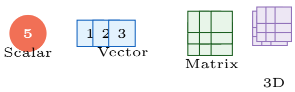
TENSOR — THE DNA

Definition

A **Tensor** is a generalized multi-dimensional array unifying scalars, vectors, and matrices under one structure.

Tensor Dimensionality

Name	Dim	Notation	Example
Scalar	0D	$x \in \mathbb{R}$	5
Vector	1D	$\mathbf{x} \in \mathbb{R}^n$	[1, 2, 3]
Matrix	2D	$\mathbf{X} \in \mathbb{R}^{m\times n}$	grid
3D Tensor	3D	$\mathcal{T} \in \mathbb{R}^{d\times m\times n}$	RGB image
4D Tensor	4D	$\mathcal{T} \in \mathbb{R}^{B\times C\times H\times W}$	batch



Batch Image Shape

$\mathcal{T} \in \mathbb{R}^{B\times C\times H\times W}$: B=32 (batch), C=3 (RGB), H, W=224

CREATING TENSORS

Syntax	Purpose
`torch.tensor([1,2,3])`	From Python list
`torch.zeros(3,4)`	All zeros
`torch.ones(3,4)`	All ones
`torch.rand(3,4)`	Uniform $\mathcal{U}[0,1)$
`torch.randn(3,4)`	Normal $\mathcal{N}(0,1)$
`torch.arange(0,10,2)`	Integer range
`torch.linspace(0,1,5)`	Evenly spaced
`torch.eye(3)`	Identity $\mathbf{I}_3$
`torch.zeros_like(t)`	Same shape, zeros
`torch.empty(2,3)`	Uninitialized

Key Distinction

`torch.tensor()` **copies** data.
`torch.as_tensor()` **shares** memory with NumPy arrays.

TENSOR ATTRIBUTES (BIG THREE)

Three Core Properties

`t.dtype` data type stored
`t.shape` dimensions (torch.Size)
`t.device` cpu / cuda / mps
Also: `t.ndim`, `t.numel()`

dtype	Use
`float32`	Default NN weights
`float16`	GPU half-precision
`int64`	Default integers
`bool`	Masks, conditions

DEVICE MANAGEMENT

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

```
t = t.to(device)     # move tensor
t = t.cuda()         # shorthand GPU
t = t.cpu()          # back to CPU
```

Common Error

Two tensors must be on **same device**. Mixing cpu/cuda throws `RuntimeError`. Move *both* model **and** data!

TENSOR OPERATIONS

Element-wise

$(a \odot b)_{ij} = a_{ij} \cdot b_{ij}$ — operates at each position independently

```
a*b; a+b; a-b; a/b
torch.sqrt(a); torch.exp(a); torch.log(a)
```

Matrix Multiplication — MOST IMPORTANT

For $\mathbf{A} \in \mathbb{R}^{m\times n}$, $\mathbf{B} \in \mathbb{R}^{n\times p}$:

$$\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{m\times p}, \quad C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}$$

```
A @ B              # preferred (batched
    too)
torch.matmul(A,B)  # same
torch.mm(A,B)      # 2D only
```

`a * b` = element-wise   `a @ b` = matmul

Aggregation

```
t.sum(); t.mean(); t.std()
t.max(); t.argmax(); t.argmin()
t.sum(dim=0)    # collapse rows
t.sum(dim=1)    # collapse columns
```

$\mu = \frac{1}{n}\sum_i x_i \qquad \sigma = \sqrt{\frac{1}{n}\sum_i (x_i-\mu)^2}$

SHAPE MANIPULATION

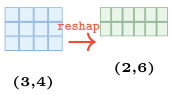
Operation	Effect
`reshape(r,c)`	New shape, may copy
`view(r,c)`	New shape, shares mem
`squeeze()`	Remove size-1 dims
`unsqueeze(0)`	Add dim at pos 0
`permute(2,0,1)`	Reorder all dims
`transpose(0,1)`	Swap two dims
`flatten()`	Collapse to 1D
`cat([a,b],dim=0)`	Concat existing dim
`stack([a,b],dim=0)`	Create new dim

The -1 Trick

`t.reshape(32,-1)`: PyTorch infers the second dim. If t has $32 \times N$ elements, result is $(32, N)$.

PIL vs PyTorch Format

PIL/NumPy: (H, W, C)
PyTorch: (C, H, W)
Convert: `tensor.permute(2,0,1)`



AUTOGRAD — AUTO DIFFERENTIATION

Definition

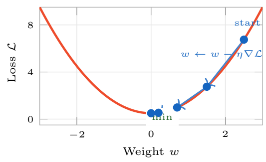
Autograd records operations on tensors with `requires_grad=True`, builds a DAG, and computes gradients via backpropagation when `.backward()` is called.

Gradient Descent — The Core Math

Update Rule

$$\boxed{\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \cdot \nabla_{\mathbf{w}} \mathcal{L}}$$

η=learning rate, $\quad \nabla_{\mathbf{w}} \mathcal{L} = \dfrac{\partial \mathcal{L}}{\partial \mathbf{w}}$



Chain Rule — Foundation of Backprop

If $\mathcal{L} = f(g(h(\mathbf{x})))$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \underbrace{\frac{\partial \mathcal{L}}{\partial f}}_{\text{loss}} \cdot \underbrace{\frac{\partial f}{\partial g}}_{\text{L3}} \cdot \underbrace{\frac{\partial g}{\partial h}}_{\text{L2}} \cdot \underbrace{\frac{\partial h}{\partial \mathbf{x}}}_{\text{L1}}$$

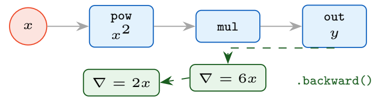
Worked Example

$y = 3x^3 + 2x + 1$, find dy/dx at $x = 4$:

$\frac{dy}{dx} = 9x^2 + 2 \qquad$ At x=4: $9(16) + 2 = \mathbf{146}$

```
x = torch.tensor(4.0, requires_grad=True)
y = 3*x**3 + 2*x + 1
y.backward()
print(x.grad)     # tensor(146.)
```

Computation Graph




```
x = torch.tensor(3.0, requires_grad=True)
y = x ** 2
y.backward()
print(x.grad)      # 6.0  (= 2*3)
```

```
# Stop tracking (inference)
with torch.no_grad():
    pred = model(x)
t.detach()   # new tensor, no grad history
```

Sacred Training Loop Order

1.	`pred=model(X)`	forward
2.	`loss=criterion(pred,y)`	loss
3.	`optimizer.zero_grad()`	CLEAR
4.	`loss.backward()`	backprop
5.	`optimizer.step()`	update

Why zero_grad Matters

Gradients **accumulate** (add to `.grad`). Without clearing, batch 2's gradients pile on batch 1's — updates are wrong from iteration 2 onward.

nn.MODULE — THE BACKBONE

Definition

`nn.Module` is the base class for all models. Every custom model **must** inherit from it. Implement: `__init__` (layers) and `forward` (data flow).

```python
import torch.nn as nn

class MyNet(nn.Module):
    def __init__(self):
        super().__init__()   # ALWAYS
            first!
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        return self.fc2(x)

model = MyNet()
```

KEY LAYERS — MATH

nn.Linear — Fully Connected

$$\mathbf{y} = \mathbf{x}\mathbf{W}^\top + \mathbf{b}$$

$\mathbf{W} \in \mathbb{R}^{\text{out}\times\text{in}}$, $\mathbf{b} \in \mathbb{R}^{\text{out}}$

Params $= (\text{in} \times \text{out}) + \text{out}$

`Linear(512,256)`: $512\times256 + 256 =$ **131,328**

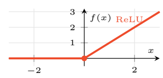
Why Activations Are Needed

Key Insight

Without non-linearity: $\mathbf{W}_2(\mathbf{W}_1\mathbf{x}) = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x}$ — stacking linear layers $\equiv$ one linear layer. Activations enable complex patterns.

ReLU

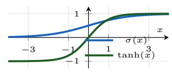
$$f(x) = \max(0, x), \quad f'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \le 0 \end{cases}$$



Sigmoid and Tanh

$$\sigma(x) = \frac{1}{1+e^{-x}} \in (0,1), \quad \sigma' = \sigma(1-\sigma)$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \in (-1,1)$$



Softmax

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad \sum_i p_i = 1$$

Activation Summary

Name	Range	Use	Vanish?
ReLU	$[0,\infty)$	Hidden	No
Sigmoid	$(0,1)$	Binary out	Yes
Tanh	$(-1,1)$	Hidden,RNN	Mild
Softmax	$(0,1)^K$	Multi-class	—

LOSS FUNCTIONS

MSE — Regression

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

`nn.MSELoss()` — linear output.

Binary Cross Entropy

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{n}\sum_i [y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)]$$

Use BCEWithLogitsLoss

Applies sigmoid internally. More numerically stable. Never use `BCELoss` + manual sigmoid.

Cross Entropy — Multiclass

$$\mathcal{L}_{\text{CE}} = -\frac{1}{n}\sum_i \sum_j y_{ij} \log \hat{y}_{ij}$$

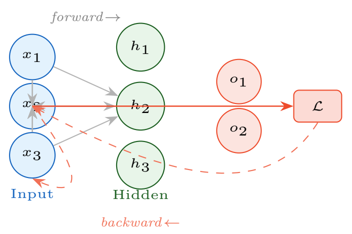
CRITICAL — No Softmax Before CE

`CrossEntropyLoss`=LogSoftmax+NLLLoss. Passing softmax outputs applies softmax twice — **silent** wrong results.

Loss Selection — Memorise

Task	Loss	Output
Regression	`MSELoss`	linear
Binary cls	`BCEWithLogitsLoss`	linear
Multi-class	`CrossEntropyLoss`	linear

BACKPROPAGATION — VISUAL



Forward: data flows input → output, graph built.
Backward: gradients flow output → input via chain rule.
Each weight: $w \leftarrow w - \eta \dfrac{\partial \mathcal{L}}{\partial w}$

OPTIMIZERS

SGD

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta g_t$$

With momentum: $v_t = \beta v_{t-1} + (1-\beta) g_t$

```
torch.optim.SGD(model.parameters(), lr
    =0.01)
torch.optim.SGD(model.parameters(), lr
    =0.01, momentum=0.9)
```

Adam — Adaptive Moment Estimation

Adam Equations

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \quad \text{(1st moment)}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \quad \text{(2nd moment)}$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \quad \text{(bias corr)}$$
$$\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

Defaults: β_1=0.9, β_2=0.999, ε=10^{-8}

```
torch.optim.Adam(model.parameters(), lr=1e
    -3)
```

Optimizer	Best For	lr
SGD	Vision tasks	0.01
SGD+momentum	Faster conv.	0.01
Adam	Default choice	0.001
AdamW	Transformers	0.001

BATCH NORMALISATION

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \varepsilon}}, \qquad y_i = \gamma \hat{x}_i + \beta$$

γ, β are *learned*. During `eval()`: uses running stats.

```
nn.BatchNorm1d(num_features)   # after
    Linear
nn.BatchNorm2d(num_channels)   # after
    Conv2d
```

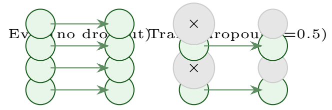
Benefits: faster training, higher LR tolerance, mild regularisation.

DROPOUT

$$\tilde{h}_i = \frac{h_i \cdot m_i}{1-p}, \quad m_i \sim \text{Bernoulli}(1-p)$$

Randomly zeros neurons with prob p during training.

```
nn.Dropout(p=0.5)     # 50% zeroed (train)
# eval(): automatically disabled
```



nn.Sequential

```
model = nn.Sequential(
    nn.Linear(784, 256),
    nn.BatchNorm1d(256),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(256, 128),
    nn.ReLU(),
    nn.Linear(128, 10)
)
```

Sequential vs Module

`Sequential`: simple linear stacks.
`nn.Module`: skip connections, multiple inputs/outputs, custom logic.

DATASETS & DATALOADERS

Definition

Dataset: Abstract class. Must implement `__len__` and `__getitem__`.
DataLoader: Wraps Dataset; batching, shuffling, parallel loading.

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

class MyDataset(Dataset):
    def __init__(self, X, y):
        self.X, self.y = X, y

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

ds = MyDataset(X_tensor, y_tensor)
dl = DataLoader(ds, batch_size=32, shuffle=True)
```

Param	Meaning
`batch_size`	Samples per iteration
`shuffle=True`	Randomise each epoch
`num_workers`	Parallel loading threads
`drop_last`	Drop final small batch

MODEL MODES

Always Switch Modes

model.train(): Dropout active, BN uses batch stats.
model.eval(): Dropout off, BN uses running stats.
Forgetting this **silently** degrades performance!

```python
model.train()
for X, y in train_dl:
    ...   # training loop

model.eval()
with torch.no_grad():
    for X, y in val_dl:
        pred = model(X)
```

CONVOLUTIONAL LAYERS (CNN)

Output Spatial Size

$$H_{out} = \left\lfloor \frac{H_{in} + 2P - K}{S} \right\rfloor + 1$$

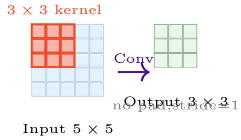
P=padding, K=kernel, S=stride. Same for W_{out}.

Parameter Count

$$\text{params} = C_{out} \times (C_{in} \times K^2 + 1)$$

`Conv2d(3,64,3)`: $64 \times (3 \times 9 + 1) = \mathbf{1{,}792}$

```python
nn.Conv2d(3, 64, kernel_size=3, padding=1)
# padding=1 with kernel=3 -> same spatial size
nn.MaxPool2d(kernel_size=2, stride=2)
# halves both H and W
```



SAVING & LOADING

```python
torch.save(model.state_dict(), "model.pth")

model = MyNet()
model.load_state_dict(torch.load("model.pth"))
model.eval()
```

Why state_dict?

Saves only parameter tensors. Portable across codebases. Saving whole model pickles the class — breaks on refactor or rename.

COMPLETE TRAINING PIPELINE

```python
import torch, torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset

# Data
X = torch.randn(1000, 20)
y = torch.randint(0, 3, (1000,))
ds = TensorDataset(X, y)
train_ds, val_ds = torch.utils.data.random_split(
    ds, [800, 200])
train_dl = DataLoader(train_ds, batch_size=32, shuffle=True)
val_dl   = DataLoader(val_ds, batch_size=32)

# Model
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(20,64), nn.ReLU(),
            nn.Linear(64,32), nn.ReLU(),
            nn.Linear(32,3))
    def forward(self, x): return self.net(x)

device = "cuda" if torch.cuda.is_available() else "cpu"
model  = Net().to(device)
crit   = nn.CrossEntropyLoss()
optim  = torch.optim.Adam(model.parameters(), lr=1e-3)

# Loop
for epoch in range(10):
    model.train(); tloss = 0
    for Xb,yb in train_dl:
        Xb,yb = Xb.to(device), yb.to(device)
        loss = crit(model(Xb), yb)
        optim.zero_grad(); loss.backward()
        optim.step(); tloss += loss.item()
    model.eval(); correct = 0
    with torch.no_grad():
        for Xb,yb in val_dl:
            Xb,yb = Xb.to(device), yb.to(device)
            correct += (model(Xb).argmax(1)==yb).sum().item()
    print(f"Ep{epoch+1} Loss:{tloss/len(train_dl):.3f} Acc:{correct/200:.3f}")

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

PRACTICE QUESTIONS

Section A — Conceptual

Q1. Explain dynamic vs static computation graphs. Why does PyTorch's approach aid debugging?
Q2. What are the **three** core tensor attributes? Describe each.
Q3. Why must `optimizer.zero_grad()` be called before `loss.backward()`?
Q4. Why should you *never* apply softmax before `nn.CrossEntropyLoss()`?
Q5. Difference between `model.train()` and `model.eval()`? Name 2 affected layers.
Q6. Write chain rule for $\mathcal{L} = f(g(h(\mathbf{x})))$ w.r.t. $\mathbf{x}$.
Q7. What does `requires_grad=True` do? When is the graph built?

Section B — Shape Tracing

Q8. `t=randn(4,3,28,28)`. Shapes after: `view(4,-1)`, `permute(0,2,3,1)`, `[:,0,:,:]`?
Q9. `a=randn(32,64)`, `b=randn(64,128)`. Shape of `a @ b`?
Q10. `t=randn(8,16)`, shape of `t.unsqueeze(0).unsqueeze(2)`?
Q11. `t=randn(3,1,5)`, shape of `t.squeeze()`?
Q12. Shape $(100,)\to(100,1)$. What single call?

Section C — Debug the Code

Q13. User wants matmul but writes `result=x*y` for $(10,5)$ matrices.
Q14. `model` on CPU, `x=randn(32,784).to("cuda")`, then `model(x)`.
Q15. Training loop missing one line — gradients accumulate across batches.
Q16. `probs=Softmax(logits)`, then `loss=CrossEntropyLoss(probs,y)`.
Q17. Eval loop has no `torch.no_grad()` wrapper.

Section D — Math Questions

Q18. $y = 3x^3 + 2x + 1$: find dy/dx at x=4. What does `x.grad` print?
Q19. `Linear(512,256)`: how many trainable parameters?
Q20. `Conv2d(3,64,3)`: how many parameters?
Q21. PIL image (H=256, W=256, C=3): what converts to PyTorch format?
Q22. Output $(32, 10)$: write 2 lines for predictions and accuracy vs `y` shape $(32,)$.

SOLUTIONS

Section A

A1. Static graphs compile before execution (fixed). Dynamic build during execution — normal Python control flow, `print()`, `pdb` all work directly.

A2. `dtype` (data type), `shape` (dimensions), `device` (cpu/cuda/mps).

A3. Gradients *accumulate* (add to `.grad`). Without clearing, batches 2+ have wrong accumulated gradients baked into updates.

A4. `CrossEntropyLoss` applies log-softmax internally. Passing softmax outputs doubles it — wrong loss, no error thrown. Silent bug.

A5. `train()`: Dropout active, BN uses batch stats. `eval()`: Dropout off, BN uses running stats accumulated during training.

A6. $\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \frac{\partial \mathcal{L}}{\partial f} \cdot \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial \mathbf{x}}$

A7. Tells PyTorch to track all ops on this tensor in a DAG. Graph built *dynamically* during forward pass. `.backward()` traverses in reverse.

Section B

B8. `view(4,-1)`: $(4, 2352)$ since 3×28^2=2352; `permute(0,2,3,1)`: $(4, 28, 28, 3)$; `[:,0,:,:]`: $(4, 28, 28)$

B9. $(32, 128)$ — inner dims match; outer dims form result.

B10. $(8, 16) \to (1, 8, 16) \to (\mathbf{1}, \mathbf{8}, \mathbf{1}, \mathbf{16})$

B11. $(3, 5)$ — size-1 middle dim removed.

B12. `t.unsqueeze(1)` or `t.reshape(100,1)`

Section C

C13. `x*y`=element-wise. For matmul: `x @ y` or `torch.matmul(x,y)`.

C14. Model is on CPU, data on CUDA. Fix: `model.to("cuda")` before `model(x)`.

C15. Missing `optimizer.zero_grad()` before `loss.backward()`.

C16. Softmax before CrossEntropyLoss — double applies. Pass raw logits.

C17. Wrap with `with torch.no_grad():` — saves memory, faster.

Section D

D18. $dy/dx = 9x^2 + 2$. At x=4: **146**. `x.grad = tensor(146.)`

D19. $(512 \times 256) + 256 = \mathbf{131{,}328}$

D20. $64 \times (3 \times 9 + 1) = 64 \times 28 = \mathbf{1{,}792}$

D21. `tensor.permute(2,0,1)`: $(H, W, C) \to (C, H, W)$

D22.
```
preds=output.argmax(dim=1)
acc=(preds==y).float().mean().item()
```

TIPS & TRICKS — LOCK IN

1. **Shape debug first.** Print `t.shape` at every step. 90% of errors are shape mismatches.
2. **Always** `loss.item()` to log — avoids accumulating GPU tensors.
3. **Normalise inputs** to $[-1, 1]$: use $(x - \mu)/\sigma$.
4. **Default: Adam, lr=1e-3.** Too high: loss explodes. Too low: stuck.
5. **Set seed:** `torch.manual_seed(42)` for reproducibility.
6. **No softmax before CrossEntropyLoss.** Silent error, not a crash.
7. **model.parameters()** for optimizer; **state_dict()** for saving.
8. **no_grad() always** during inference — faster, less memory.
9. **Classification predictions:** `output.argmax(dim=1)`.
10. **Start small.** Debug on 10 samples, then scale up.

QUICK REFERENCE

Concept	Syntax
Matrix multiply	`A @ B`
Enable gradient	`requires_grad=True`
Backward pass	`loss.backward()`
Zero gradients	`optimizer.zero_grad()`
Weight update	`optimizer.step()`
Stop gradients	`torch.no_grad()`
Move to GPU	`tensor.to(device)`
Predictions	`out.argmax(dim=1)`
Save model	`torch.save(m.state_dict(),...)`
Train mode	`model.train()`
Eval mode	`model.eval()`

The Sacred 5-Step Training Loop

1. `pred=model(X)` forward
2. `loss=crit(pred,y)` loss
3. `optimizer.zero_grad()` clear grads
4. `loss.backward()` backprop
5. `optimizer.step()` update weights