# Talent Bootcamp Series -1
# Day-2
# Computer Vision Lab

Lab slides prepared by: Hasan Shahid Kamal

CHAPTER 1

# Deep Learning Lab Workflow

## Load & Preprocess

Load images and labels, then transform raw data for neural networks

## Split Data

Divide into training, validation, and test sets

## Train & Optimize

Feed batches, compute loss, update parameters over epochs

## Evaluate

Measure performance and visualize results

CHAPTER 2

# MNIST Dataset Explained

## What is MNIST?

A collection of handwritten digits (0-9). Each image is 28×28 grayscale pixels.

## The Task

**Classification:** Given an image, predict which digit it represents across 10 possible classes.

## Why MNIST?

Simple, balanced, and clean—perfect for learning deep learning concepts without noisy data.

CHAPTER 3

# PyTorch Ecosystem

## torch

Tensor operations & automatic differentiation

## torch.nn

Neural network layers: Linear, ReLU, Conv2d

## torch.optim

Optimization algorithms for parameter updates

## torchvision

Vision datasets, transforms, and pretrained models

**torchvision** simplifies image data handling—automatically downloads, verifies, and loads datasets like MNIST so you can focus on model design.

# Torchvision

`torchvision` is the **computer vision companion library for PyTorch**. It provides everything you typically need to work with **images and videos**—from datasets and preprocessing to models and utilities.

## `torchvision.datasets` — Ready-to-use Vision Datasets

This module provides **pre-built dataset classes** so you don't have to manually download, unzip, and parse data.

| What it includes | Common datasets |
| --- | --- |
| • Popular **image classification datasets**<br>• Automatic **download & verification**<br>• Returns **(image, label)** pairs | • MNIST<br>• Fashion-MNIST<br>• CIFAR-10 / CIFAR-100<br>• ImageNet<br>• SVHN |

# Image Preprocessing & Augmentation-`torchvision.transforms`

Neural networks **cannot work with raw images**. Transforms convert and prepare images before training.

## What it includes

- Image → Tensor conversion
- Normalization
- Data augmentation

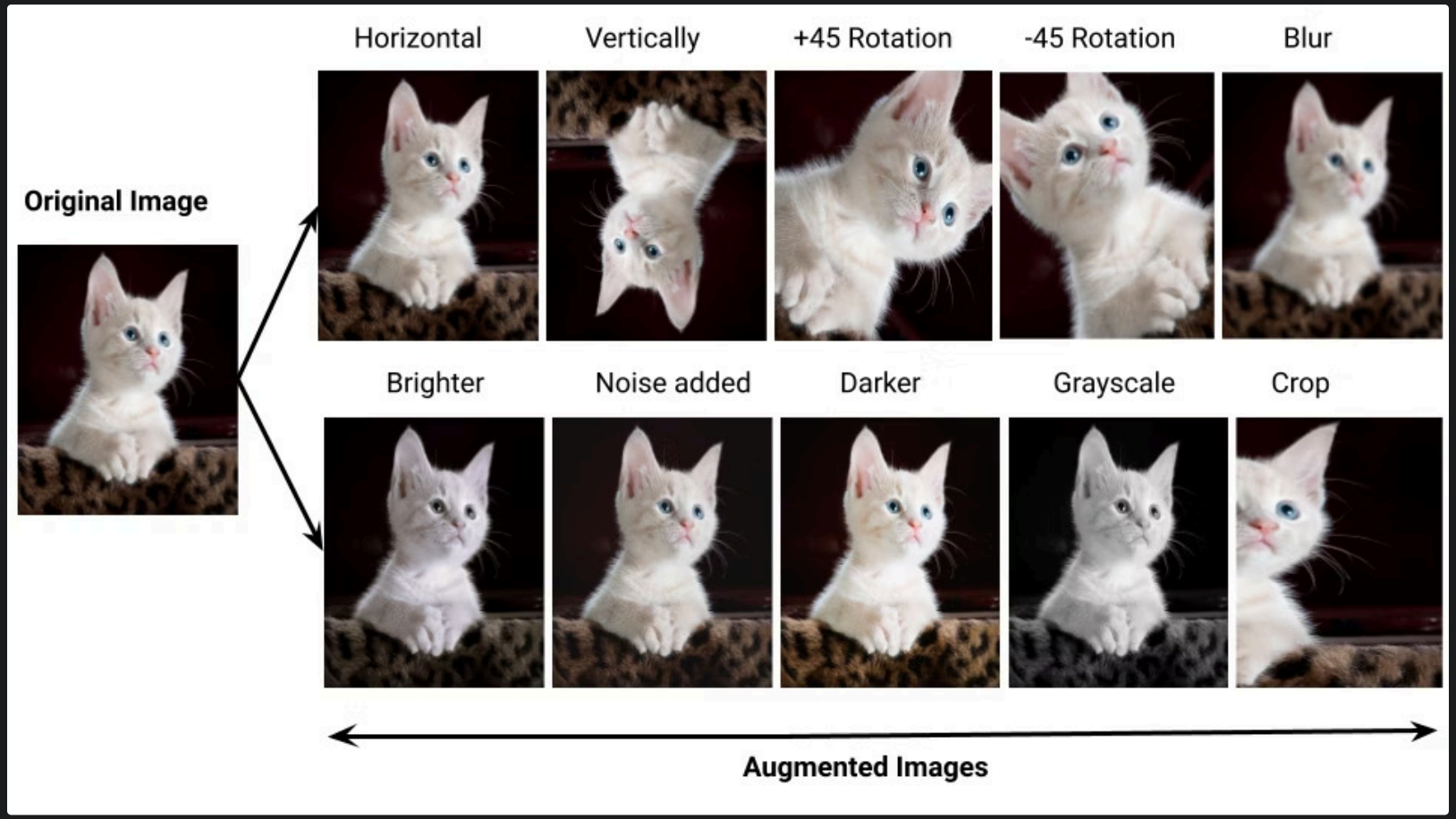
## Common transforms

- `ToTensor()` – converts image to PyTorch tensor
- `Normalize()` – scales pixel values
- `Resize()`, `CenterCrop()`
- `RandomHorizontalFlip()`, `RandomRotation()`

## Why this matters

- Improves **training stability**
- Augmentation helps **prevent overfitting**
- Standard preprocessing ensures fair comparisons

# Example for different transforms



# `torchvision.models` — Pretrained Vision Models

This module provides **state-of-the-art neural network architectures** already implemented and optionally pretrained.

## What it includes

- CNN architectures
- Pretrained weights (trained on ImageNet)

## Popular models

- ResNet
- VGG
- AlexNet
- MobileNet
- EfficientNet

## Why this matters

- No need to implement complex models from scratch
- Enables **transfer learning**
- Faster convergence and better accuracy

# Torch.nn module

> `torch.nn` **is the neural-network module of PyTorch.**

It provides **everything needed to define, organize, and run neural networks**, but **not** to train them fully (training also needs optimizers, data, etc.).

> `torch.nn` = **building blocks + structure for neural networks**

# Why do we need `torch.nn`?

> Neural networks consist of:

- Layers (Linear, Conv, ReLU, etc.)
- Parameters (weights & biases)
- Loss functions
- Model structure

> `torch.nn` provides **ready-made, optimized implementations** of all these.

Without `torch.nn`, you would have to:

- Define weights manually
- Write forward computations yourself
- Track parameters by hand

# What does `torch.nn` contain?

We can divide `torch.nn` into **4 major categories**:

| | |
|---|---|
| Layers (Core Building Blocks) | Loss Functions (Error Measurement) |
| Model Base Class: `nn.Module` | Container Modules (Organizing Layers) |

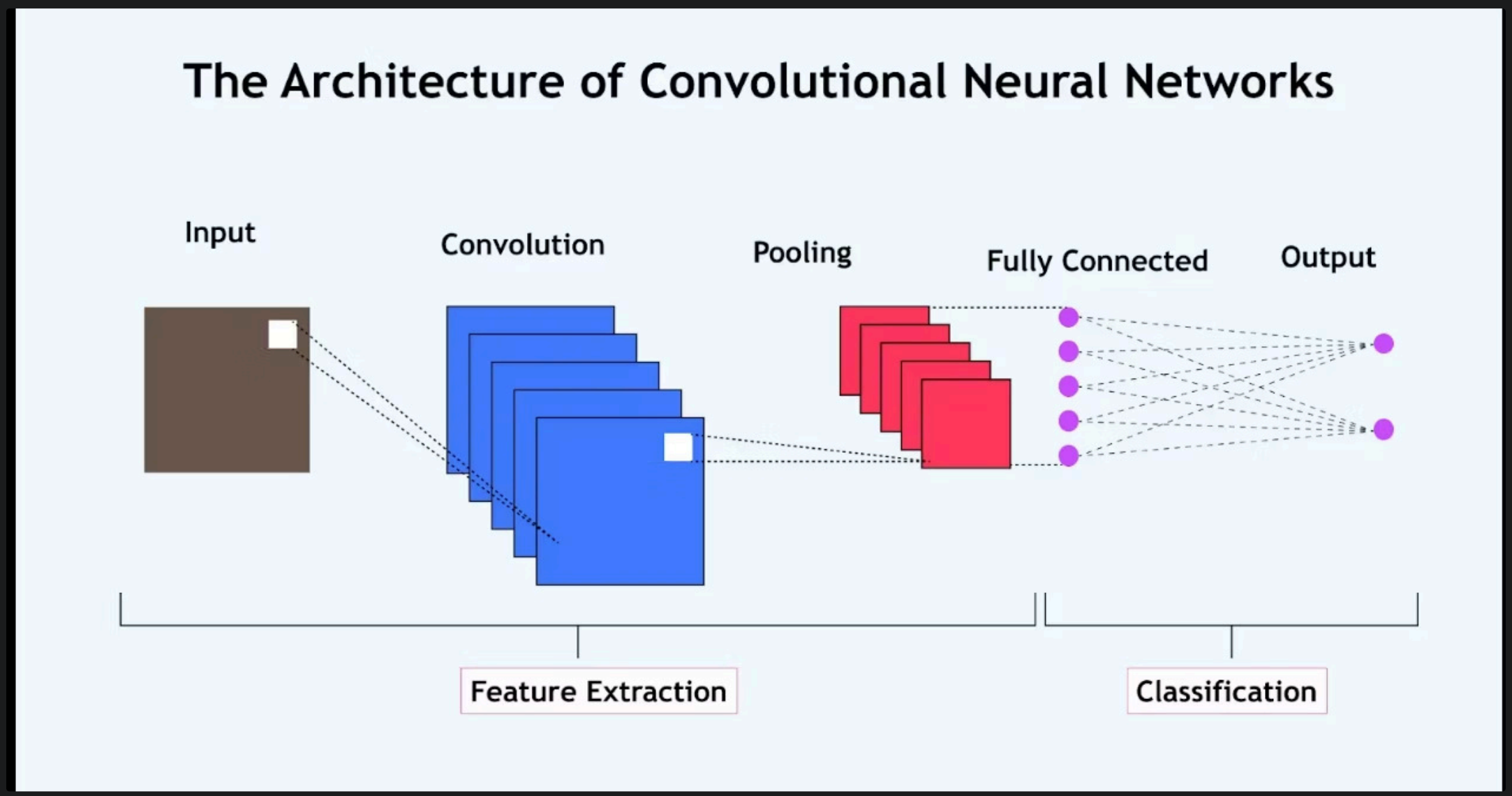
Let's go through each one in detail.

# Layers (Core Building Blocks)

These are **learnable transformations**.

## Common layers

- `nn.Linear` – fully connected layer
- `nn.Conv2d` – convolutional layer
- `nn.ReLU` – activation function
- `nn.MaxPool2d` – pooling
- `nn.Dropout` – regularization



# Loss Functions (Error Measurement)

> Loss functions **quantify how wrong the model's prediction is**.

# Model Base Class: nn.Module

> This is the **most important concept** in torch.nn.

## What is nn.Module?

- Base class for **all models and layers**
- Handles parameter registration
- Enables training & evaluation modes

## Why models inherit from it

- Automatically tracks parameters
- Works with optimizers
- Supports .train() and .eval()

Example:

python Copy code

```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Linear(784, 128)

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

# Container Modules (Organizing Layers)

> These help **organize layers cleanly**.

## Common containers

- `nn.Sequential` – layers in order
- `nn.ModuleList`
- `nn.ModuleDict`

## Problem with `nn.Sequential`

- Does **not naturally handle reshaping**
- Assumes every step is a module

**Example:**

python Copy code

```python
model = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 10)
)
```

# What `torch.nn` does NOT do

`torch.nn` does **not**:

- Update weights
- Perform optimization
- Handle datasets or batching

Those are handled by:

- `torch.optim`
- `torch.utils.data`

CHAPTER 4

# DataLoaders, Epochs & Data Splitting

1

## DataLoader

Groups data into batches, shuffles to prevent order bias, enables parallel loading

2

## Epochs & Batches

Epoch = full dataset pass. 60K samples ÷ 100 batch size = 600 batches per epoch

3

## Avoid Data Leakage

Always evaluate on unseen test data —never on training data

CHAPTER 5

# Dataset Class

> Used to create a custom Dataset class when the data isn't directly available to use.

Defines how data is accessed:

- **__len__():** Returns total samples
- **__getitem__():** Retrieves single data point + label

CHAPTER 6

# Neural Network Architecture

01

## nn.Module

Foundation of PyTorch models—tracks parameters and manages devices

02

## __init__()

Defines layers and initializes parameters

03

## forward()

Defines how input flows through layers

The model learns patterns like curves and edges that define each digit.

# The enumerate() function

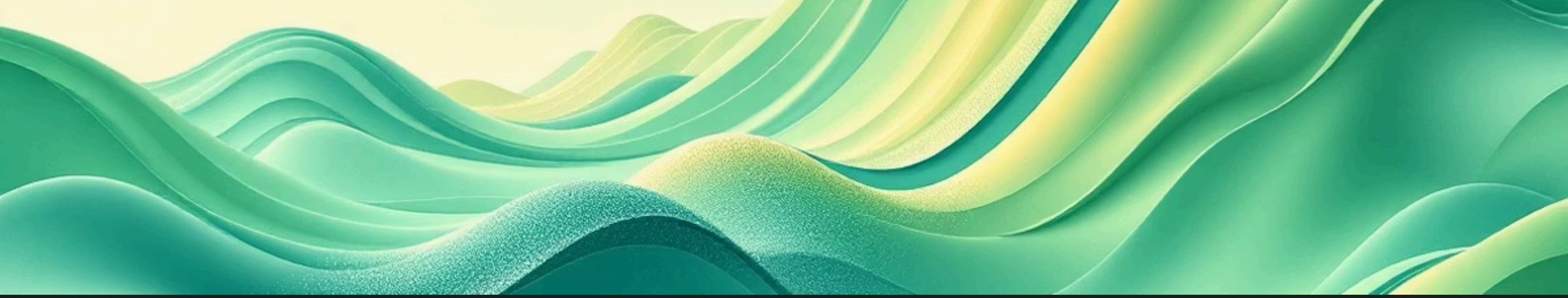
> `enumerate()` **returns both an index and a value when looping**

for i, (images, labels) in enumerate(train_loader):

- `i` → **batch index**
- `(images, labels)` → **data from the DataLoader**

## Why is `enumerate()` useful in training loops?

1️⃣ Progress tracking
2️⃣ Debugging
3️⃣ Logging & visualization



CHAPTER 7

# Loss Functions & Optimizers

## Loss Functions

Quantify prediction error. Lower loss = better performance.

- **CrossEntropyLoss:** Classification
- **MSELoss:** Regression

## Optimizers

Update parameters based on gradients from backpropagation.

- **SGD:** Fixed learning rate
- **Adam:** Adaptive per-parameter rates

# When to use what Loss function?

| Loss Function | Problem Type | Model Output | Target Format | When to Use | Example Use Case |
|---|---|---|---|---|---|
| `nn.CrossEntropyLoss` | Multi-class classification | Raw scores (logits), shape `(N, C)` | Class labels `(N,)` | When each input belongs to **exactly one class** | MNIST (digits 0–9), CIFAR-10 |
| `nn.NLLLoss` | Multi-class classification | Log-probabilities | Class labels | When model already applies `LogSoftmax` | Advanced/custom architectures |
| `nn.BCEWithLogitsLoss` | Binary / multi-label classification | Raw scores (logits) | 0/1 labels | Binary or **multiple independent labels** | Spam detection, multi-tag images |
| `nn.BCELoss` | Binary classification | Probabilities (after sigmoid) | 0/1 labels | When sigmoid is applied manually | Rarely used in practice |
| `nn.MSELoss` | Regression | Continuous values | Continuous values | Predicting real-valued outputs | House price prediction |
| `nn.L1Loss` | Regression | Continuous values | Continuous values | When robustness to outliers is needed | Noisy sensor data |
| `nn.SmoothL1Loss` **(Huber)** | Regression | Continuous values | Continuous values | Balance between MSE and L1 | Bounding box regression |
| `nn.KLDivLoss` | Distribution matching | Log-probabilities | Probabilities | Comparing probability distributions | Knowledge distillation |
| `nn.CosineEmbeddingLoss` | Similarity learning | Embeddings | Similar / dissimilar labels | Measuring similarity | Face verification |
| `nn.TripletMarginLoss` | Metric learning | Embeddings | Anchor/positive/negative | Learning distances | Face recognition |
| `nn.CTCLoss` | Sequence alignment | Probabilities | Sequences | Unaligned sequence tasks | Speech recognition, OCR |

# What are **logits**?

## Logits are the raw, unnormalized outputs of a neural network

They are the **numbers produced by the last layer of a model *before* applying any activation like Softmax or Sigmoid**.

> **Logits = raw class scores**

# Logits vs Probabilities

| Property | Logits | Probabilities |
|---|---|---|
| Range | (−∞, +∞) | [0, 1] |
| Sum to 1? | ❌ No | ✅ Yes |
| Direct model output | ✅ Yes | ❌ No |
| Used by loss function | ✅ Yes (CrossEntropy) | ❌ No |
| Human-interpretable | ❌ No | ✅ Yes |

# QUICK EXAMPLE FOR LOGITS

Example:

```text
Logits:      [2.3, 0.4, 5.6]
Probabilities: [0.03, 0.01, 0.96]
```

Copy code

Now:

- Values are between 0 and 1
- Sum to 1
- Can be interpreted as probabilities

# Optimizers

## Optimizers in PyTorch (with Functions)

| Optimizer | PyTorch Function | Learning Rate Type | Speed | Stability | When to Use |
|---|---|---|---|---|---|
| **SGD** | `torch.optim.SGD()` | Fixed | Slow | Medium | When you want full control; theory-focused training |
| **SGD + Momentum** | `torch.optim.SGD(momentum=...)` | Fixed | Medium | Good | CNNs and vision tasks; smoother convergence |
| **Adam** | `torch.optim.Adam()` | Adaptive | Fast | Very good | **Default choice** for most deep learning tasks |
| **RMSprop** | `torch.optim.RMSprop()` | Adaptive | Fast | Good | RNNs, noisy or non-stationary problems |
| **AdamW** | `torch.optim.AdamW()` | Adaptive | Fast | Very good | Modern best practice; better weight decay |
| **Adagrad** | `torch.optim.Adagrad()` | Adaptive | Slow | Medium | Sparse data (e.g., NLP embeddings) |

# What is `model.parameters()`?

> `model.parameters()` **returns all the trainable parameters of the model**

These parameters are:

- **Weights**
- **Biases**

stored inside layers such as:

- `nn.Linear`
- `nn.Conv2d`
- etc.

> These are the values the optimizer updates during training.

CHAPTER 8

# Training & Evaluation Pipeline

Forward Pass

Compute Loss

Update Params

Backpropagation

### model.train()

Enables dropout & training behaviors

### model.eval()

Disables for consistent evaluation

### Visualize

Loss & accuracy curves show progress

# Model.train() vs Model.eval()

## model.train() vs model.eval()

| Aspect | model.train() | model.eval() |
|---|---|---|
| **Purpose** | Puts the model in **training mode** | Puts the model in **evaluation (inference) mode** |
| **When used** | During **training phase** | During **validation / testing / inference** |
| **Affects weights?** | ❌ No (weights updated by optimizer, not this) | ❌ No |
| **Dropout layers** | ✅ **Enabled** (random neurons dropped) | ❌ **Disabled** (all neurons active) |
| **Batch Normalization** | ✅ Uses **batch statistics** | ✅ Uses **running statistics** |
| **Parameter updates** | ❌ Does not update parameters by itself | ❌ Does not update parameters |
| **Gradient computation** | ✅ Gradients are usually computed | ❌ Gradients usually disabled (torch.no_grad()) |
| **Random behavior** | ✅ Yes (dropout, batch stats) | ❌ No (deterministic) |
| **Loss.backward()** | ✅ Yes | ❌ No |
| **Optimizer.step()** | ✅ Yes | ❌ No |
| **Common mistake** | Forgetting to switch back to train mode | Forgetting to switch to eval mode |

FINAL CHAPTER

# Key Takeaways

## Device Management

GPUs accelerate matrix operations. Model and data must be on the same device.

## Reproducibility

Set random seeds to ensure consistent results and fair comparisons.

## Understanding > Memorizing

MNIST provides a foundation for advanced computer vision tasks.