MCDA 5511

Current Practices in Computing and Data Science

Building & Training Neural Networks

Part 2: nn.Module, Loss Functions & Training Loops

Somto Muotoe

January 2026

Part 2 Overview

What We'll Cover

1. **Neural Network Basics** - What they are and how they work
2. **nn.Module** - PyTorch's building block for networks
3. **Loss Functions** - Measuring prediction errors
4. **Optimizers** - Updating weights with gradients
5. **Training Loop** - Putting it all together
6. **Model Evaluation** - Testing and saving models

Prerequisites

You should be comfortable with:

- Creating and manipulating tensors
- Basic tensor operations
- Autograd and `backward()`
- `requires_grad` and gradient computation

Recap: Part 1 covered tensors and autograd - the foundation for everything today.

Neural Network Basics

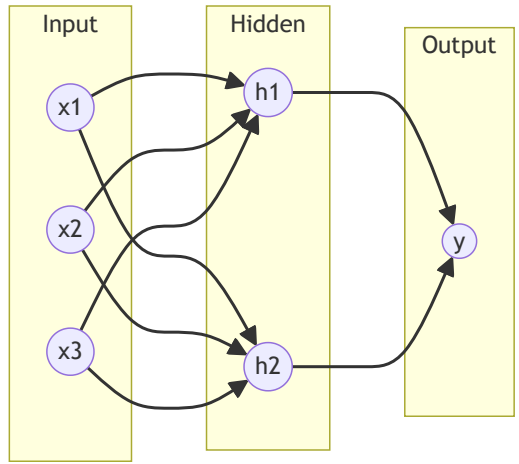
What they are and how they learn

What is a Neural Network?

The Concept

A neural network is a **function approximator** composed of:

- **Layers** of interconnected nodes (neurons)
- **Weights** that are learned from data
- **Activation functions** that add non-linearity



A simple feedforward network with one hidden layer

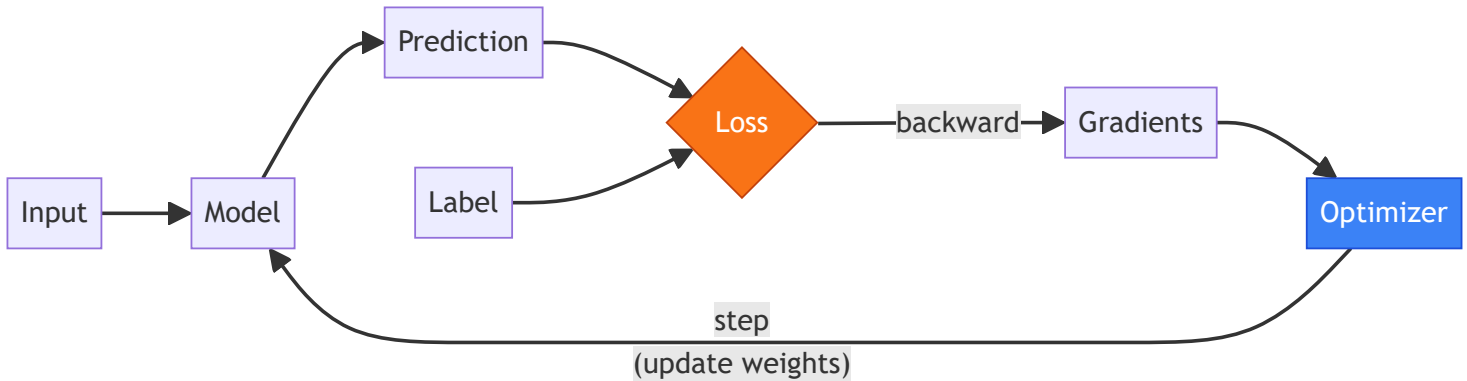
Why "Neural"?

Loosely inspired by biological neurons, but the math is what matters:

$$y = f(Wx + b)$$

Where W = weights, b = bias, f = activation

How Neural Networks Learn



1. Forward

Pass input through model to get prediction

2. Loss

Compare prediction to true label

3. Backward

Compute gradients via autograd

4. Update

Optimizer adjusts weights

This cycle repeats until the model learns the patterns in the data.

The nn.Module Class

PyTorch’s building block for neural networks

What is nn.Module?

The Base Class

`nn.Module` is the **base class** for all neural network components in PyTorch.

Key Responsibilities

- **Holds parameters** (weights & biases)
- **Defines forward pass** computation
- **Tracks submodules** (nested layers)
- **Provides utilities** for training/eval modes

Rule: Every neural network you build will inherit from `nn.Module`.

```python
import torch.nn as nn

class MyNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        # Define layers here
        self.layer1 = nn.Linear(10, 5)
        self.layer2 = nn.Linear(5, 2)

    def forward(self, x):
        # Define computation here
        x = self.layer1(x)
        x = torch.relu(x)
        x = self.layer2(x)
        return x

# Create an instance
model = MyNetwork()
```

Two required methods: `__init__` and `forward`

Anatomy of nn.Module

`__init__` : Define Your Layers

```
def __init__(self):
    super().__init__()  # Always call this!

    # Define layers as attributes
    self.fc1 = nn.Linear(784, 256)
    self.fc2 = nn.Linear(256, 128)
    self.fc3 = nn.Linear(128, 10)

    # Activation (stateless, can reuse)
    self.relu = nn.ReLU()
    self.dropout = nn.Dropout(0.2)
```

- Call `super().__init__()` first
- Assign layers as `self.` attributes
- PyTorch auto-registers parameters

`forward` : Define the Computation

```
def forward(self, x):
    # Input: (batch_size, 784)

    x = self.fc1(x)      # (batch, 256)
    x = self.relu(x)
    x = self.dropout(x)

    x = self.fc2(x)      # (batch, 128)
    x = self.relu(x)
    x = self.dropout(x)

    x = self.fc3(x)      # (batch, 10)
    return x

# Call model like a function
output = model(input_tensor)
```

- Defines data flow through layers
- Called automatically when you do `model(x)`
- Never call `model.forward(x)` directly

Introducing MNIST

The "Hello World" of Machine Learning

MNIST is a dataset of 70,000 handwritten digit images (0-9).

Property	Value
Images	70,000 (60k train, 10k test)
Size	28 x 28 pixels, grayscale
Classes	10 (digits 0-9)

Why MNIST? Small, simple, and perfect for learning neural networks.

Your First Neural Network

The Task

Build a classifier that takes a 28x28 image and predicts which digit (0-9) it represents.

Input: Image as tensor `(1, 28, 28)`

Output: 10 scores (one per digit)

Prediction: Digit with highest score

```
# Image tensor shape: (channels, height, width)
# For MNIST: (1, 28, 28) - 1 channel, 28×28 pixels

# Flatten to vector: 1 * 28 * 28 = 784 features
# Output: 10 scores (one per digit)
```

Building the Classifier

```
import torch.nn as nn

class MNISTClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.network = nn.Sequential(
            nn.Flatten(),        # (1,28,28) → 784
            nn.Linear(784, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 10)    # 10 digits
        )

    def forward(self, x):
        return self.network(x)

model = MNISTClassifier()
# Total parameters: 109,386
```

Common Layer Types

Core Layers

Layer	Purpose
`nn.Linear(in, out)`	Fully connected layer
`nn.Conv2d(in, out, k)`	2D convolution (images)
`nn.LSTM(in, hidden)`	Recurrent (sequences)
`nn.Embedding(vocab, dim)`	Word embeddings (NLP)

```
# Fully connected: every input connects to every output
fc = nn.Linear(784, 256)  # 784 in, 256 out

# Convolution: slides filter over image
conv = nn.Conv2d(1, 32, kernel_size=3)
```

How nn.Linear Works

A linear layer computes: y = xW + b

```
# Input: 784 features, Output: 256 features
layer = nn.Linear(784, 256)

# Parameters created automatically:
# - weight: (256, 784) = 200,704 values
# - bias: (256,) = 256 values

x = torch.randn(1, 784)   # 1 sample
y = layer(x)              # shape: (1, 256)
```

Note: Linear layers are the building blocks of most networks. Each neuron learns its own weights.

What are Activation Functions?

A Function Applied After Each Layer

Activation functions transform the output of a layer before passing it to the next:

```
# Without activation (just linear)
x = linear1(x)
x = linear2(x)

# With activation
x = linear1(x)
x = relu(x)      # ⟵ Activation function
x = linear2(x)
```

In PyTorch:

```
self.network = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),          # Activation
    nn.Linear(128, 10)
)
```

Why Do We Need Them?

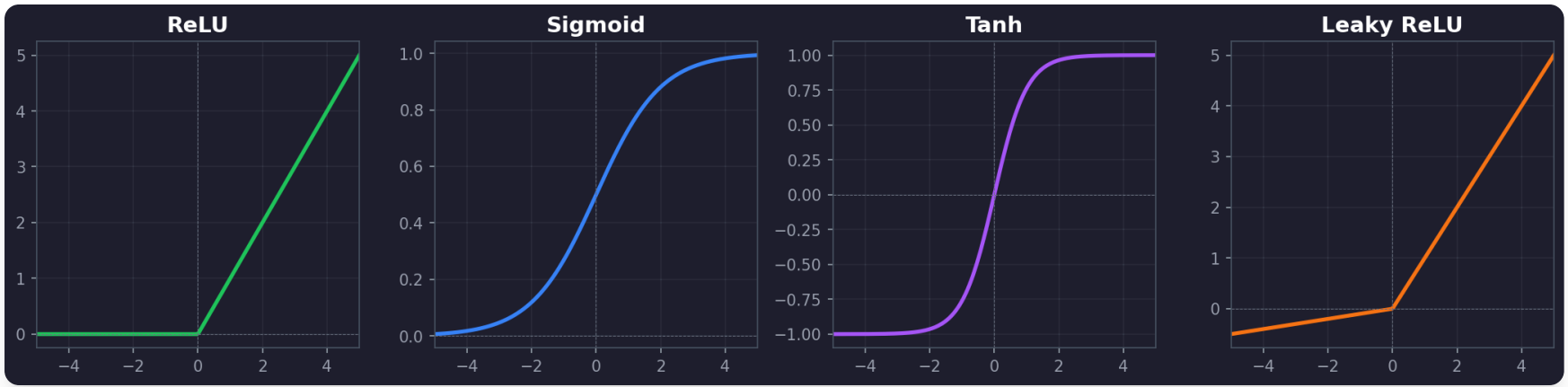
Without activation functions, stacking layers just gives another linear function:

```
# Two linear layers without activation
y = W2 @ (W1 @ x)
# Simplifies to:
y = W_combined @ x  # Still linear!
```

Key insight: Linear functions can only learn linear patterns. Activation functions add **non-linearity**, allowing the network to learn complex patterns like curves and edges.

Think of it this way: linear = straight lines only. Activations = curves and complex shapes.

Common Activation Functions



ReLU - `max(0, x)`

Most common choice. Simple and fast to compute.

Sigmoid - Output: (0, 1)

Good for output layer when you need probabilities.

Tanh - Output: (-1, 1)

Zero-centered output. Sometimes used in RNNs.

Leaky ReLU

Like ReLU but allows small negative values through.

Rule of thumb: Start with ReLU for hidden layers. It works well in most cases.

Choosing an Activation Function

When to Use What

Location	Recommended	Why
Hidden layers	**ReLU**	Fast, works well
Binary output	Sigmoid	Outputs 0-1 probability
Multi-class output	None*	Use CrossEntropyLoss

*CrossEntropyLoss applies softmax internally, so don't add it yourself.

Simple rule: Use ReLU between hidden layers, and let your loss function handle the output layer.

Why Leaky ReLU Exists

ReLU has a problem: if a neuron outputs negative values, it becomes "dead" (always outputs 0).

```
# ReLU: negative inputs → 0
relu(-5)  # Returns 0
relu(-100)  # Returns 0 (neuron is "dead")

# Leaky ReLU: small slope for negatives
leaky_relu(-5)   # Returns -0.05
leaky_relu(-100) # Returns -1.0 (still learning!)
```

When to try Leaky ReLU:

- If your model isn't learning well with ReLU
- Deep networks where neurons might "die"
- Usually not needed for simple networks

Regularization Layers

What is a Neuron?

A **neuron** is a single unit in a layer. It receives inputs, multiplies each by a learned weight, adds them up, and applies an activation.

```
output = activation(w1*x1 + w2*x2 + ... + bias)
```

In `nn.Linear(784, 128)`, there are **128 neurons**, each learning 784 weights.

Dropout

Randomly "turns off" neurons during training (sets output to 0).

```
dropout = nn.Dropout(p=0.5)  # 50% drop rate
model.train()  # Dropout ON
model.eval()   # Dropout OFF
```

Why Drop Neurons?

Without dropout, neurons become **co-dependent** - they rely on specific other neurons. This causes **overfitting** (memorizing data instead of learning).

Impact of dropout:

- Forces each neuron to learn independently
- Creates redundancy (multiple neurons learn similar things)
- Like training many networks and averaging them

Note: During eval, all neurons are active but scaled. Always call `model.eval()` before inference!

nn.Sequential: Quick Model Building

When to Use Sequential

`nn.Sequential` chains layers in order - data flows straight through A → B → C.

```
# Simple way to build a network
model = nn.Sequential(
    nn.Linear(784, 256),
    nn.ReLU(),
    nn.Linear(256, 128),
    nn.ReLU(),
    nn.Linear(128, 10)
)

output = model(input_tensor)
```

Use `nn.Sequential` when:

- Layers flow linearly (no branching)
- Quick prototyping

When You Need Custom nn.Module

Sometimes data doesn't just flow straight through. A **skip connection** lets data "skip over" some layers and get added back later:

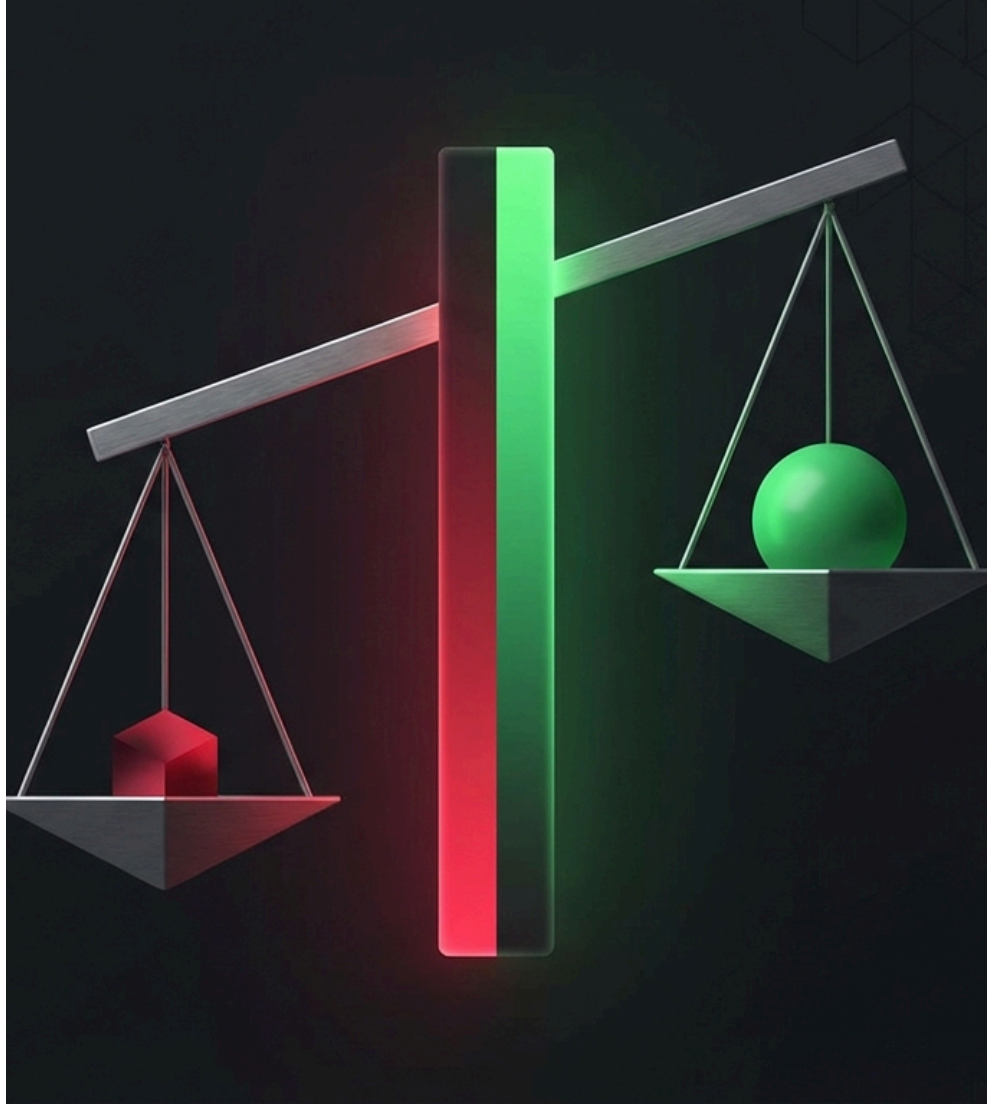
```
class BlockWithSkip(nn.Module):
    def forward(self, x):
        original = x          # Save the input
        out = self.layers(x)  # Process through layers
        return out + original # Add original back!
```

Why skip? In very deep networks, gradients can become tiny. Skip connections give gradients a "shortcut" path, making training easier.

Note: Skip connections are used in advanced architectures like ResNet. For now, `nn.Sequential` works great for simple networks.

Loss Functions

Measuring how wrong your predictions are



What is a Loss Function?

The Concept

A loss function (or criterion) **quantifies the error** between predictions and true values.

Properties of Good Loss Functions

- **Differentiable** - so we can compute gradients
- **Lower is better** - 0 = perfect predictions
- **Task-appropriate** - regression vs classification

$$\text{Loss} = f(\text{prediction}, \text{target})$$

The goal of training: **minimize the loss**.

```python
import torch.nn as nn

# Define a loss function
criterion = nn.MSELoss()

# Compute loss
predictions = model(inputs)
loss = criterion(predictions, targets)

print(f"Loss: {loss.item():.4f}")

# Backpropagate
loss.backward()
```

Key insight: The loss value tells the optimizer *how much* to adjust weights, and gradients tell it *which direction*.

Common Loss Functions

Loss Function	Use Case	Output Range
`nn.MSELoss()`	Regression	Any real number
`nn.L1Loss()`	Regression (robust to outliers)	Any real number
`nn.CrossEntropyLoss()`	Multi-class classification	Raw logits (no softmax)
`nn.BCEWithLogitsLoss()`	Binary classification	Raw logits (no sigmoid)

Regression: MSE Loss

```
criterion = nn.MSELoss()
pred = torch.tensor([2.5, 0.0, 2.1])
target = torch.tensor([3.0, -0.5, 2.0])
loss = criterion(pred, target)  # = 0.135
```

Averages squared differences: $\text{MSE} = \frac{1}{n}\sum(y_i - \hat{y}_i)^2$

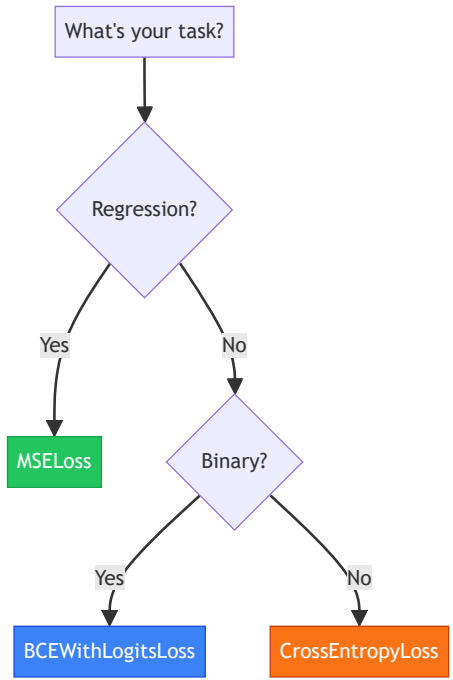
Classification: Cross-Entropy

```
criterion = nn.CrossEntropyLoss()
# Raw logits (NOT probabilities!)
logits = torch.tensor([[2.0, 1.0, 0.1]])
target = torch.tensor([0])  # Class index
loss = criterion(logits, target)  # = 0.417
```

Penalizes wrong class predictions: $\text{CE} = -\sum y_c \log(\hat{y}_c)$

Choosing the Right Loss Function

Decision Guide



Common Mistakes

Don't apply softmax before `CrossEntropyLoss` - it's built in!

Don't apply sigmoid before `BCEWithLogitsLoss` - it's built in!

Do use `BCEWithLogitsLoss` over `BCELoss` - more stable.

Do check your target shape matches what the loss expects.

Optimizers

Updating weights to minimize loss

What is an Optimizer?

The Role of an Optimizer

An optimizer **updates model parameters** using gradients to minimize loss.

The Update Rule

$$\theta_{\text{new}} = \theta_{\text{old}} - \alpha \cdot \nabla L$$

- θ = model parameters (weights)
- α = learning rate (step size)
- ∇L = gradient (direction of steepest increase)

Intuition: Imagine you're blindfolded on a hilly landscape, trying to find the lowest point. The gradient tells you which way is uphill. To go downhill, you step in the **opposite direction** (the minus sign). The learning rate is how big each step is.

Key insight: Every weight in your network gets its own gradient. Some weights need big adjustments, others tiny tweaks. The optimizer handles this automatically.

Using an Optimizer

The Three-Step Dance

Every training iteration follows this pattern:

1. **zero_grad()** - Clear old gradients (they accumulate by default!)
2. **backward()** - Compute gradients via backpropagation
3. **step()** - Update weights using the gradients

Code Example

```
import torch.optim as optim

# Create optimizer (pass model parameters)
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training step
optimizer.zero_grad()   # 1. Clear old gradients
loss = criterion(model(x), y)
loss.backward()         # 2. Compute gradients
optimizer.step()        # 3. Update weights
```

Remember: Always call `zero_grad()` before `backward()`, or gradients will accumulate across batches!

Common Optimizers

SGD

Stochastic Gradient Descent

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

- Simple and well-understood
- Often needs momentum
- Good for fine-tuning
- Can generalize better

Adam

Adaptive Moment Estimation

```
optim.Adam(
    model.parameters(),
    lr=0.001,
    weight_decay=1e-5
)
```

- **Most popular choice**
- Adapts learning rate per-param
- Works well out of the box
- Good default for beginners

AdamW

Adam with Weight Decay

```
optim.AdamW(
    model.parameters(),
    lr=0.001,
    weight_decay=0.01
)
```

- Better regularization
- Preferred for transformers
- Decoupled weight decay
- Current best practice

Rule of thumb: Start with Adam (lr=0.001). Switch to AdamW for larger models or if overfitting.

Learning Rate: The Most Important Hyperparameter

What is Learning Rate?

The learning rate (α) controls **step size** during optimization.

Learning Rate	Effect
Too high	Overshoots, loss explodes
Too low	Very slow training
Just right	Smooth convergence

Typical Starting Points

- Adam: `lr=0.001` (1e-3)
- SGD: `lr=0.01` to `0.1`
- Fine-tuning: `lr=1e-5` to `1e-4`

Learning Rate Schedulers

For advanced training, you can **adjust the learning rate** during training using schedulers.

Scheduler	What it Does
`StepLR`	Reduce LR every N epochs
`ReduceLROnPlateau`	Reduce when loss stops improving
`CosineAnnealingLR`	Smooth decay following cosine curve

```
from torch.optim.lr_scheduler import StepLR
scheduler = StepLR(optimizer, step_size=10, gamma=0.1)

# Call after each epoch
scheduler.step()
```

The Training Loop

Putting it all together

The Training Loop Overview

```
for epoch in range(num_epochs):
    for batch_x, batch_y in dataloader:
        optimizer.zero_grad()       # 1. Zero gradients
        predictions = model(batch_x)       # 2. Forward pass
        loss = criterion(predictions, batch_y)  # 3. Compute loss
        loss.backward()             # 4. Backward pass
        optimizer.step()            # 5. Update weights

    print(f"Epoch {epoch}: Loss = {loss.item():.4f}")
```

zero_grad	forward	loss	backward	step
Clear accumulated gradients	Model makes predictions	Measure prediction error	Compute gradients	Update weights

Datasets and DataLoaders

Built-in Datasets (torchvision)

PyTorch provides common datasets ready to use.

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

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

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

DataLoader: Batching & Shuffling

```
train_loader = DataLoader(
    train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(
    test_data, batch_size=64, shuffle=False)

# Iterate over batches
for images, labels in train_loader:
    print(images.shape)  # (64, 1, 28, 28)
    print(labels.shape)  # (64,)
    break
```

DataLoader handles: batching, shuffling, parallel loading with `num_workers`.

Other datasets: CIFAR-10, ImageNet, FashionMNIST in `torchvision.datasets`.

Training vs Evaluation Mode

Why Modes Matter

Some layers behave differently during training vs inference:

Layer	Training	Evaluation
`Dropout`	Randomly zeros neurons	Disabled
`BatchNorm`	Uses batch statistics	Uses running statistics

Critical: Forgetting `model.eval()` during testing will give wrong results!

Setting the Mode

```python
# Training mode (default)
model.train()
for batch_x, batch_y in train_loader:
    optimizer.zero_grad()
    loss = criterion(model(batch_x), batch_y)
    loss.backward()
    optimizer.step()

# Evaluation mode
model.eval()
with torch.no_grad():  # Disable gradient computation
    for batch_x, batch_y in test_loader:
        predictions = model(batch_x)
```

`torch.no_grad()` : Disables gradient tracking for faster inference and less memory usage. Always use during evaluation!

Complete MNIST Training Example

```
import torch, torch.nn as nn
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# 1. Load MNIST
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])
train_data = datasets.MNIST('./data', train=True, download=True, transform=transform)
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)

# 2. Create model, loss, optimizer
model = nn.Sequential(nn.Flatten(), nn.Linear(784, 128), nn.ReLU(), nn.Linear(128, 10))
criterion, optimizer = nn.CrossEntropyLoss(), torch.optim.Adam(model.parameters(), lr=0.001)

# 3. Training loop
for epoch in range(5):
    model.train(); total_loss = 0
    for images, labels in train_loader:
        optimizer.zero_grad(); loss = criterion(model(images), labels)
        loss.backward(); optimizer.step(); total_loss += loss.item()
    print(f"Epoch {epoch+1}: Avg Loss = {total_loss / len(train_loader):.4f}")
```

This trains a digit classifier to ~97% accuracy in just a few epochs!

Tracking Training Progress

Recording Metrics

```
train_losses, val_losses, val_accs = [], [], []

for epoch in range(num_epochs):
    # Training
    model.train()
    epoch_loss = 0
    for x, y in train_loader:
        # ... training step ...
        epoch_loss += loss.item()
    train_losses.append(epoch_loss / len(train_loader))

    # Validation
    model.eval()
    val_loss, correct = 0, 0
    with torch.no_grad():
        for x, y in val_loader:
            out = model(x)
            val_loss += criterion(out, y).item()
            correct += (out.argmax(1) == y).sum()
    val_losses.append(val_loss / len(val_loader))
    val_accs.append(correct / len(val_dataset))
```

Visualizing Progress

```
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

ax1.plot(train_losses, label='Train')
ax1.plot(val_losses, label='Validation')
ax1.set_xlabel('Epoch'); ax1.set_ylabel('Loss')
ax1.legend(); ax1.set_title('Loss Over Time')

ax2.plot(val_accs)
ax2.set_xlabel('Epoch'); ax2.set_ylabel('Accuracy')
ax2.set_title('Validation Accuracy')

plt.tight_layout()
plt.show()
```

Watch for: loss decreasing, train/val gap (overfitting), accuracy improving.

GPU Training

Moving to GPU

```
# Check GPU availability
device = torch.device('cuda' if torch.cuda.is_available()
print(f"Using: {device}")

# Move model to GPU
model = model.to(device)

# Training loop with GPU
for batch_x, batch_y in train_loader:
    batch_x, batch_y = batch_x.to(device), batch_y.to(devi
    optimizer.zero_grad()
    loss = criterion(model(batch_x), batch_y)
    loss.backward()
    optimizer.step()
```

Best Practices

Do define device once at the start and reuse it.

Do use `pin_memory=True` in DataLoader for faster GPU transfer.

Don't call `.to(device)` inside the loop more than needed.

Device-Agnostic Code

```
device = torch.device('cuda' if torch.cuda.is_available()
model = MyModel().to(device)
output = model(torch.randn(32, 10).to(device))
```

Model Evaluation

Testing, metrics, and saving models

Evaluating Your Model

Train / Validation / Test Split

Set	Purpose	When to Use
Train	Learn patterns	During training
Validation	Tune hyperparams	After each epoch
Test	Final evaluation	Once, at the end

Golden rule: Never use test data during development!

Detecting Overfitting

- Train loss down, val loss up = overfitting
- Large train/val accuracy gap

Evaluation Loop

```python
def evaluate(model, loader, criterion, device):
    model.eval()
    total_loss, correct, total = 0, 0, 0
    with torch.no_grad():
        for x, y in loader:
            x, y = x.to(device), y.to(device)
            out = model(x)
            total_loss += criterion(out, y).item()
            correct += (out.argmax(1) == y).sum().item()
            total += y.size(0)
    return total_loss / len(loader), correct / total
```

Common Evaluation Metrics

Classification Metrics

```
from sklearn.metrics import (
    accuracy_score, precision_score,
    recall_score, f1_score, confusion_matrix)

# Get predictions
model.eval()
all_preds, all_labels = [], []
with torch.no_grad():
    for x, y in test_loader:
        preds = model(x.to(device)).argmax(1).cpu()
        all_preds.extend(preds)
        all_labels.extend(y)

# Calculate metrics
acc = accuracy_score(all_labels, all_preds)
prec = precision_score(all_labels, all_preds, average='wei
rec = recall_score(all_labels, all_preds, average='weighte
f1 = f1_score(all_labels, all_preds, average='weighted')
```

What Each Metric Tells You

Metric	Question Answered
Accuracy	What % of predictions are correct?
Precision	Of predicted positives, how many correct?
Recall	Of actual positives, how many found?
F1 Score	Harmonic mean of precision & recall

Confusion Matrix

```
import seaborn as sns
cm = confusion_matrix(all_labels, all_preds)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted'); plt.ylabel('Actual')
```

Saving and Loading Models

Saving Models

```
# Save state dict (recommended)
torch.save(model.state_dict(), 'model_weights.pth')

# Save checkpoint for resuming training
torch.save({
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
}, 'checkpoint.pth')
```

Best practice: Save `state_dict()` - portable and doesn't depend on code structure.

Loading Models

```
# Load state dict
model = MyModel()  # Create model first
model.load_state_dict(torch.load('model_weights.pth'))
model.eval()  # Set to evaluation mode

# Load checkpoint for resuming
checkpoint = torch.load('checkpoint.pth')
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict

# Load with device mapping (GPU → CPU)
model.load_state_dict(
    torch.load('weights.pth', map_location=device))
```

Summary & Exercises

Wrapping up Part 2

Part 2 Recap

What We Learned

1. **nn.Module** - Base class for neural networks. Define layers in `__init__`, computation in `forward`.
2. **Loss Functions** - MSELoss for regression, CrossEntropyLoss for classification.
3. **Optimizers** - Adam is a great default. Learning rate is the key hyperparameter.
4. **Training Loop** - zero_grad, forward, loss, backward, step. Repeat!
5. **Evaluation** - Use model.eval() and torch.no_grad() for inference.

The Complete Picture

```
# Define
model = MyModel().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)

# Train
model.train()
for x, y in train_loader:
    x, y = x.to(device), y.to(device)
    optimizer.zero_grad()
    loss = criterion(model(x), y)
    loss.backward()
    optimizer.step()

# Evaluate
model.eval()
with torch.no_grad():
    predictions = model(test_x.to(device))

# Save
torch.save(model.state_dict(), 'model.pth')
```

Part 2 Exercises

Build a banknote forgery detector using the Banknote Authentication dataset (1372 samples, 4 features, binary classification).

Exercise 1: Build a Binary Classifier

```
# TODO: Create a model with:
# - Input: 4 features (variance, skewness, etc.)
# - Hidden: 32 neurons with ReLU
# - Output: 1 neuron (binary classification)
```

Exercise 2: Training Loop

- Use `BCEWithLogitsLoss` for binary classification
- Train for 100 epochs with Adam optimizer
- Track and plot train/validation loss

Exercise 3: Hyperparameter Tuning

Compare different configurations:

- Learning rates: 0.01, 0.001, 0.0001
- Hidden sizes: 16, 32, 64

Exercise 4: Evaluation

- Calculate accuracy, precision, recall, F1
- Plot confusion matrix
- Save your best model

Exercises: `exercises/pytorch-2-exercises.ipynb`

Bonus: Train an MNIST classifier with >95% accuracy!

You Made It!

You now have the foundation for deep learning!

- **Build** neural networks with nn.Module
- **Train** with loss functions and optimizers
- **Evaluate** and save your models

Questions? Bring them to the session!