Task: - PyTorch foundation

(Week 2)

Objective -----------

📘 PyTorch Foundation Roadmap

1. Beginner Level – Tensors & Basics

Goal: Get comfortable with PyTorch's building blocks.

- **Installation & Setup**
  - Install PyTorch (CPU/GPU).
  - Verify with torch.__version__ and torch.cuda.is_available().
- **Tensors 101**
  - Creating tensors: torch.tensor, torch.zeros, torch.rand.
  - Tensor attributes: .shape, .dtype, .device.
  - Indexing, slicing, reshaping.
  - Broadcasting rules.
  - Operations: addition, multiplication, dot product, matrix multiplication.
  - Conversion between NumPy ↔ PyTorch.
- **GPU Acceleration**
  - .to("cuda") vs .to("cpu").
  - Practice moving tensors & ensuring reproducibility.

✅ Mini-project: Implement **linear regression** with PyTorch tensors *without* torch.nn.

2. Intermediate Level – Autograd & Neural Networks

Goal: Understand how PyTorch powers deep learning.

- **Autograd (Automatic Differentiation)**
  - requires_grad, .backward(), .grad.
  - Gradient computation example ($y = x^2$).
- torch.nn **Module**
  - nn.Linear, nn.ReLU, nn.Sigmoid.
  - nn.Sequential.
  - Custom nn.Module class.
- **Optimizers & Training Loops**
  - torch.optim.SGD, torch.optim.Adam.
  - Loss functions: nn.MSELoss, nn.CrossEntropyLoss.
  - Writing training loops (forward → loss → backward → step).

✅ Mini-project: Train a **simple feedforward neural network** on MNIST.

3. Applied Level – Data & Training Pipelines

Goal: Learn to handle real-world data.

- **Datasets & Dataloaders**
  - torch.utils.data.Dataset & DataLoader.
  - Custom datasets.
  - Batch loading & shuffling.
- **Transforms & Preprocessing**
  - torchvision.transforms (resize, normalize, augment).
  - Handling image/text datasets.
- **Model Saving/Loading**
  - torch.save, torch.load.
  - state_dict vs full model saving.

✅ Mini-project: Train **CNN on CIFAR-10** with proper dataloading & augmentation.

4. Advanced Level – Architectures & Optimization

Goal: Master state-of-the-art techniques.

- **Advanced Architectures**
  - CNNs (LeNet, ResNet basics).
  - RNNs/LSTMs/GRUs.
  - Transformers (self-attention, BERT basics).
- **Optimization Tricks**
  - Learning rate scheduling.
  - Gradient clipping.
  - Weight initialization.
  - Regularization (dropout, batch norm).
- **Debugging & Performance**
  - torchsummary, torchviz.
  - Mixed precision training (torch.cuda.amp).
  - Profiling tools (torch.profiler).

✅ Mini-project: Implement a **ResNet-like CNN** from scratch and train on a subset of ImageNet or CIFAR-100.

5. Expert Level – Internals & Research Skills

Goal: Be able to read papers and implement models from scratch.

- Implement attention & transformers **without** torch.nn.MultiheadAttention.
- Explore distributed training (torch.distributed, DataParallel).
- Write custom optimizers.
- Understand PyTorch JIT (torch.jit.trace, torchscript).

✅ Mini-project: Reproduce a **research paper model** (e.g., Vision Transformer or GPT-like small model).

📁 Suggested Workflow

1. **Hands-on practice > reading**: Code after every concept.

2. Keep **Jupyter notebooks** for experiments.
3. Use **small datasets first** (MNIST, CIFAR-10) → scale later.
4. After each milestone, do a **mini-project**.

Topics -----------

📘 Topic 1: Introduction to PyTorch

🔎 What to Study

- **Overview of PyTorch**
    - What is PyTorch?
    - Why it's popular: dynamic computation graph, Pythonic API, research-to-production ease.
    - Ecosystem (TorchVision, TorchText, TorchAudio, TorchServe).
- **Comparison with Other Frameworks**
    - **TensorFlow vs PyTorch**:
        - PyTorch: eager execution (dynamic graphs), intuitive debugging.
        - TensorFlow: originally static graphs, now supports eager mode; better for production pipelines.
    - PyTorch → research-friendly, TensorFlow → industry adoption (though gap is narrowing).
- **Basic Workflow**
    - Define tensors (data).
    - Build model (`nn.Module`).
    - Define loss function.
    - Choose optimizer.
    - Train loop: forward → loss → backward → optimize.

📖 How to Learn

1. **PyTorch Docs – Get Started Guide**
   PyTorch Official Getting Started
   Covers installation, tensor basics, and a small example.

2. **Blogs & Videos**
    - "Deep Learning with PyTorch: A 60 Minute Blitz" (official tutorial).
    - YouTube channels like *Aladdin Persson*, *freeCodeCamp's PyTorch course*.

💻 How to Do It (Hands-on Practice)

1. **Install PyTorch**
    - Visit PyTorch Install Page.
    - Choose your system (OS, package manager, CUDA version).
    - Example (CPU-only via pip):

```
pip install torch torchvision torchaudio
```

2. **Verify Installation with a Simple Script**

```
import torch

# Check version
print("PyTorch version:", torch.__version__)

# Create a tensor
x = torch.rand(3, 3)
print("Random Tensor:\n", x)

# Check for GPU
print("CUDA available:", torch.cuda.is_available())
```

✅ Expected: You'll see a random 3×3 matrix and whether CUDA (GPU) is available.

------ Checkpoint after this topic:

- You can explain what PyTorch is and why it's used.
- You've successfully installed PyTorch and run a tensor example.
- You know how PyTorch compares with TensorFlow at a high level.

📘 Topic 2: Tensors and Basic Operations

🔎 What to Study

1. Creating Tensors

- From data:

```
torch.tensor([1, 2, 3])
```

- From functions:

```
torch.zeros(2, 3)      # 2x3 tensor of zeros
torch.ones(2, 2)       # 2x2 tensor of ones
torch.rand(3, 3)       # random values [0,1)
torch.arange(0, 10, 2)  # values 0,2,4,6,8
torch.linspace(0, 1, 5) # evenly spaced 5 points
```

- From NumPy arrays:

```
import numpy as np
a = np.array([1, 2, 3])
t = torch.from_numpy(a)
```

2. Tensor Attributes

Every tensor has:

- .shape → size of tensor
- .dtype → data type (torch.float32, torch.int64)
- .device → CPU or GPU

```
x = torch.rand(3, 4)
print(x.shape, x.dtype, x.device)
```

3. Arithmetic Operations

- Element-wise:

```
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])
print(a + b)  # addition
print(a * b)  # multiplication
print(a ** 2) # power
```

- Matrix multiplication:

```
A = torch.rand(2, 3)
```

```
B = torch.rand(3, 2)
print(torch.matmul(A, B))
print(A @ B)  # shorthand
```

4. Indexing & Slicing

```
x = torch.arange(10)
print(x[0])      # first element
print(x[2:6])    # slice
print(x[-1])     # last element
```

- For matrices:

```
mat = torch.arange(16).reshape(4,4)
print(mat[0, :])   # first row
print(mat[:, 1])   # second column
```

5. Reshaping Tensors

```
x = torch.arange(12)      # shape [12]
print(x.view(3, 4))       # reshape to 3x4
print(x.reshape(2, 6))    # reshape to 2x6
print(x.unsqueeze(0))     # add a new dimension
print(x.squeeze())        # remove dimension of size 1
```

6. Broadcasting

PyTorch automatically expands smaller tensors during arithmetic:

```
a = torch.ones(3, 1)   # shape [3,1]
b = torch.ones(1, 4)   # shape [1,4]
print((a + b).shape)   # result [3,4]
```

💻 How to Do It (Hands-On Practice in Jupyter Notebook)

Try this **exercise set**:

```
import torch

# 1. Create a 3x3 tensor with random floats
x = torch.rand(3, 3)
print("Tensor:\n", x)
```

```
# 2. Check its shape, dtype, device
print(x.shape, x.dtype, x.device)

# 3. Perform elementwise addition and multiplication
a = torch.tensor([1, 2, 3], dtype=torch.float32)
b = torch.tensor([4, 5, 6], dtype=torch.float32)
print("a + b =", a + b)
print("a * b =", a * b)

# 4. Reshape a 1D tensor into 2D
y = torch.arange(12)
print("Original:", y)
print("Reshaped (3x4):\n", y.view(3, 4))

# 5. Demonstrate broadcasting
m = torch.ones(3, 1)
n = torch.arange(4).unsqueeze(0)  # shape [1,4]
print("Broadcast result:\n", m + n)
```

------- Checkpoint after this topic:

- You can create tensors in multiple ways.
- You understand and can check tensor attributes (shape, dtype, device).
- You can do arithmetic, reshaping, indexing, and broadcasting.

------------ prepare a Jupyter Notebook template with these examples + exercises (like a ready-to-run “Tensors Playground”)? -----------

Tensors Playground· python

```
# Tensors Playground - PyTorch Basics

# --- 📌 Section 1: Setup ---
import torch
importnumpy as np

print("PyTorch version:", torch.__version__)
print("CUDA available:", torch.cuda.is_available())

# --- 📌 Section 2: Creating Tensors ---
# From data
x_data = torch.tensor([1, 2, 3])
print("Tensor from list:", x_data)

# From functions
```

```
x_zeros = torch.zeros(2, 3)
x_ones = torch.ones(2, 2)
x_rand = torch.rand(3, 3)

print("Zeros:\n", x_zeros)
print("Ones:\n", x_ones)
print("Random:\n", x_rand)

# From NumPy array
a = np.array([1, 2, 3])
t_from_np = torch.from_numpy(a)
print("From NumPy:", t_from_np)

# --- 📌 Section 3: Tensor Attributes ---
t = torch.rand(3, 4)
print("Tensor attributes:")
print("Shape:", t.shape)
print("Dtype:", t.dtype)
print("Device:", t.device)

# --- 📌 Section 4: Arithmetic Operations ---
a = torch.tensor([1, 2, 3], dtype=torch.float32)
b = torch.tensor([4, 5, 6], dtype=torch.float32)

print("a + b =", a + b)
print("a * b =", a * b)
print("a dot b =", torch.dot(a, b))

# Matrix multiplication
A = torch.rand(2, 3)
B = torch.rand(3, 2)
print("Matrix multiplication:\n", torch.matmul(A, B))

# --- 📌 Section 5: Indexing & Slicing ---
x = torch.arange(10)
print("First element:", x[0])
print("Slice (2:6):", x[2:6])
print("Last element:", x[-1])

mat = torch.arange(16).reshape(4,4)
print("Matrix:\n", mat)
print("First row:", mat[0, :])
print("Second column:", mat[:, 1])

# --- 📌 Section 6: Reshaping Tensors ---
y = torch.arange(12)
```

```
print("Original:", y)
print("Reshape (3x4):\n",y.view(3, 4))
print("Reshape (2x6):\n", y.reshape(2, 6))
print("Unsqueeze (add dim):", y.unsqueeze(0).shape)
print("Squeeze (remove dim):", y.unsqueeze(0).squeeze().shape)

# --- 📌 Section 7: Broadcasting ---
m = torch.ones(3, 1)
n = torch.arange(4).unsqueeze(0) # shape [1,4]
print("Broadcast result:\n", m + n)

# --- 📌 Section 8: Exercises ---

# 1. Create a 4x4 identity matrix.
# 2. Create a tensor of size [2,3] with random integers between 0 and 10.
# 3. Reshape a tensor of size [16] into [4,4].
```

📘 Topic 3: Autograd and Computational Graphs

🔎 What to Study

1. Automatic Differentiation

- PyTorch's autograd module builds a **computational graph** dynamically.
- Each tensor has a flag: requires_grad=True → PyTorch tracks operations for gradient computation.

2. Key Concepts

- **requires_grad** → tells PyTorch to compute gradients.
- **Backward()** → computes gradients (via backpropagation).
- **grad attribute** → stores the gradient after .backward().
- **Detach()** → creates a tensor without gradient tracking (useful in inference).
- **with torch.no_grad():** → disables gradient tracking in a block of code.

3. **Workflow**

1. Define tensor(s) with requires_grad=True.
2. Apply operations → PyTorch builds computation graph.
3. Call .backward() on a scalar result.
4. Access .grad of input tensors.

📖 How to Learn

1. **Official Docs:**
   Autograd: Automatic Differentiation Tutorial
2. **Blogs & Videos:**
   - YouTube: "PyTorch Autograd Explained" (Aladdin Persson).
   - Blog posts showing gradient flow visualization.

------- How to Do It (Hands-on Example) ---------

Here's a **simple script for y = x²**:

```
import torch

# Create a tensor with gradient tracking
x = torch.tensor(2.0, requires_grad=True)

# Define a function: y = x^2
y = x ** 2
print("y =", y)

# Backpropagate (dy/dx)
y.backward()
print("Gradient dy/dx at x=2:", x.grad)
```

✅ Expected Output:

```
y = 4.0
Gradient dy/dx at x=2: 4.0
```

---- More Examples to Try

```
# Example 1: Multiple operations
x = torch.tensor(3.0, requires_grad=True)
```

```
y = 3*x**3 + 2*x**2 + x
y.backward()
print("Gradient dy/dx at x=3:", x.grad)

# Example 2: Vector input
x = torch.randn(3, requires_grad=True)
y = x * 2
z = y.mean()
z.backward()
print("Gradient dz/dx:", x.grad)

# Example 3: Detaching tensors
x = torch.tensor(5.0, requires_grad=True)
y = x**2
z = y.detach()  # z will not require gradients
print("Does z require grad?", z.requires_grad)

# Example 4: No grad context (useful in inference)
with torch.no_grad():
   a = torch.tensor(2.0, requires_grad=True)
   b = a**2
   print("Inside no_grad, does b require grad?", b.requires_grad)
```

Checkpoint after this topic

- You can explain what requires_grad, .backward(), and .grad mean.
- You can compute gradients for scalar and vector functions.
- You know how to disable gradient tracking (.detach(), torch.no_grad()).

----- Autograd Playground· python --------

```
# Autograd Playground - PyTorch Automatic Differentiation

import torch

# --- 📌 Section 1: Introduction ---
print("PyTorch version:", torch.__version__)

# --- 📌 Section 2: Basic Gradient Computation ---
# Define a tensor with gradient tracking
a = torch.tensor(2.0, requires_grad=True)

# Function: y = a^2
```

```
y = a ** 2
print("y =", y)

# Backpropagate (dy/da)
y.backward()
print("Gradient dy/da at a=2:", a.grad)

# --- 📌 Section 3: Multiple Operations ---
x = torch.tensor(3.0, requires_grad=True)

# Function: y = 3x^3 + 2x^2 + x
y = 3*x**3 + 2*x**2 + x
y.backward()
print("Gradient dy/dx at x=3:", x.grad)

# --- 📌 Section 4: Vector Inputs ---
x = torch.randn(3, requires_grad=True)
y = x * 2
z = y.mean() # scalar output
z.backward()
print("Vector x:", x)
print("Gradient dz/dx:", x.grad)

# --- 📌 Section 5: Detaching Tensors ---
x = torch.tensor(5.0, requires_grad=True)
y = x ** 2
z = y.detach() # z is detached, no gradient tracking
print("Does y require grad?", y.requires_grad)
print("Does z require grad?", z.requires_grad)

# --- 📌 Section 6: No Grad Context ---
with torch.no_grad():
a = torch.tensor(2.0, requires_grad=True)
b = a ** 2
print("Inside no_grad, does b require grad?", b.requires_grad)

# --- 📌 Section 7: Exercises ---
# 1. Define x = 4.0 (requires_grad=True) and compute gradient of y = x^3.
# 2. Define x = torch.arange(3.0, requires_grad=True) and compute gradients for y =
(x+1)^2.
# 3. Show the difference between x.detach() and with torch.no_grad().
# 4. Build a chain: y = x^2, z = 2y + 3 → compute dz/dx.
# 5. Try using .backward(retain_graph=True) to compute multiple gradients on the same
graph.

print("✅ Notebook ready! Start experimenting with autograd.")
```

📘 Topic 4: Neural Network Basics (nn module)

🔎 What to Study

1. nn.Module

- Base class for all neural networks in PyTorch.
- You define:
    - **Layers** inside __init__()
    - **Forward pass** inside forward()
- PyTorch handles backward pass automatically (thanks to autograd).

2. Linear Layers

- nn.Linear(in_features, out_features)
- Performs: $\mathbf{y = xW^T + b}$
- Example:

```
import torch.nn as nn
layer = nn.Linear(5, 2)  # input 5-dim, output 2-dim
x = torch.randn(1, 5)
y = layer(x)
```

3. Activation Functions

- Add non-linearity to networks.
- Common ones:
    - nn.ReLU()
    - nn.Sigmoid()
    - nn.Tanh()

4. **Loss Functions**

- Measure how well model predictions match targets.
- Common ones:
  - nn.MSELoss() → regression
  - nn.CrossEntropyLoss() → classification

📖 How to Learn

1. **PyTorch Docs:** Neural Networks Tutorial
2. **Videos:** “Neural Networks in PyTorch” (Aladdin Persson, freeCodeCamp).

------ How to Do It (Hands-on Example) ----------

Here’s a **basic MLP for a toy dataset**:

```
import torch
import torch.nn as nn
import torch.optim as optim

# --- Step 1: Create toy dataset (y = 2x + 1 + noise)
X = torch.linspace(-1, 1, 100).unsqueeze(1)  # shape [100,1]
y = 2 * X + 1 + 0.2*torch.rand(X.size())

# --- Step 2: Define MLP model
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(1, 16),   # input 1 → hidden 16
            nn.ReLU(),
            nn.Linear(16, 1)    # hidden → output
        )

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

model = MLP()

# --- Step 3: Define loss & optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)
```

```
# --- Step 4: Training loop
epochs = 200
for epoch in range(epochs):
    # Forward pass
    y_pred = model(X)
    loss = criterion(y_pred, y)

    # Backward pass
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch+1) % 20 == 0:
        print(f"Epoch {epoch+1}, Loss: {loss.item():.4f}")

# --- Step 5: Test prediction
with torch.no_grad():
    test_inp = torch.tensor([[0.5]])
    print("Prediction for x=0.5:", model(test_inp).item())
```

------ Expected Result: The MLP should approximate the linear function y ≈ 2x + 1.

Checkpoint after this topic

- You can define models with nn.Module.
- You know how to use nn.Linear and activation functions.
- You understand how loss + optimizer fit into the training loop.
- You've trained a simple MLP on a toy dataset.

------- Nn Basics Playground· python -----------

```
# Neural Network Basics Playground - PyTorch nn.Module
model = MLP()

# --- 📌 Section 3: Loss and Optimizer ---
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)

# --- 📌 Section 4: Training Loop ---
epochs = 200
losses = []

for epoch in range(epochs):
```

```
y_pred = model(X)
loss = criterion(y_pred, y)
optimizer.zero_grad()
loss.backward()
optimizer.step()

losses.append(loss.item())
if (epoch+1) % 20 == 0:
print(f"Epoch {epoch+1}, Loss: {loss.item():.4f}")

# Plot training loss
plt.plot(losses)
plt.title("Training Loss")
plt.xlabel("Epoch")
plt.ylabel("MSE Loss")
plt.show()

# --- 📌 Section 5: Test Prediction ---
with torch.no_grad():
test_inp = torch.tensor([[0.5]])
print("Prediction for x=0.5:", model(test_inp).item())

# --- 📌 Section 6: Classification Example ---
# Generate synthetic 2D data (two classes)
from sklearn.datasets import make_moons

Xc, yc = make_moons(n_samples=200, noise=0.2)
Xc = torch.tensor(Xc, dtype=torch.float32)
yc = torch.tensor(yc, dtype=torch.long)

plt.scatter(Xc[:,0], Xc[:,1], c=yc, cmap=plt.cm.Paired)
plt.title("2D Classification Dataset (Moons)")
plt.show()

# Define a classifier MLP
class Classifier(nn.Module):
def __init__(self):
super(Classifier, self).__init__()
self.layers = nn.Sequential(
nn.Linear(2, 16),
nn.ReLU(),
nn.Linear(16, 2)
)
def forward(self, x):
return self.layers(x)
```

```
clf = Classifier()
criterion_cls = nn.CrossEntropyLoss()
optimizer_cls = optim.Adam(clf.parameters(), lr=0.01)

# Train classifier
for epoch in range(100):
outputs = clf(Xc)
loss = criterion_cls(outputs, yc)

optimizer_cls.zero_grad()
loss.backward()
optimizer_cls.step()

if (epoch+1) % 20 == 0:
print(f"[Classifier] Epoch {epoch+1}, Loss: {loss.item():.4f}")

# --- 📌 Section 7: Exercises ---
# 1. Modify the regression MLP to use two hidden layers.
# 2. Change activation to Tanh and compare results.
# 3. Use Adam optimizer instead of SGD for regression.
# 4. Visualize decision boundaries of the classifier.
# 5. Build a 3-class classifier using random synthetic data.

print("✅ Notebook ready! You can now explore regression & classification with PyTorch.")
```

📘 Topic 5: Datasets and DataLoaders

🔎 What to Study

1. **Datasets (**torch.utils.data.Dataset**)**

- Wraps your data & labels.
- Provides __len__() and __getitem__() methods.
- Example: torchvision.datasets.MNIST, CIFAR10, FashionMNIST.

2. **DataLoader (**torch.utils.data.DataLoader**)**

- Creates iterable over dataset.

- Handles batching, shuffling, parallel loading.
- Key args:
  - batch_size → number of samples per batch
  - shuffle → shuffle dataset each epoch
  - num_workers → parallel data loading

3. Transforms

- Preprocessing pipelines from torchvision.transforms.
- Examples:
  - transforms.ToTensor()
  - transforms.Normalize(mean, std)
  - transforms.RandomHorizontalFlip()

4. Custom Dataset

- Inherit from torch.utils.data.Dataset.
- Override __len__ and __getitem__.
- Useful for CSV, images, or text not covered by torchvision.

📖 How to Learn

- Official Tutorial: PyTorch Data Loading
- TorchVision Datasets Docs torchvision.datasets

------- How to Do It (Hands-On Examples) -------

1. Load MNIST with DataLoader

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

# Transform: Convert image to tensor & normalize
transform = transforms.Compose([
```

```
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Download & load MNIST training dataset
train_dataset = datasets.MNIST(root="./data", train=True, download=True, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

# Example: Iterate through one batch
images, labels = next(iter(train_loader))
print("Batch shape:", images.shape)   # [64, 1, 28, 28]
print("Labels shape:", labels.shape)  # [64]
```

2. Custom Dataset Example

Suppose you have a CSV file with features + labels:

```
import pandas as pd
from torch.utils.data import Dataset

class CustomCSV(Dataset):
    def __init__(self, csv_file):
        self.data = pd.read_csv(csv_file)
        self.X = torch.tensor(self.data.iloc[:, :-1].values, dtype=torch.float32)
        self.y = torch.tensor(self.data.iloc[:, -1].values, dtype=torch.long)

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

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

# Usage
dataset = CustomCSV("mydata.csv")
loader = DataLoader(dataset, batch_size=32, shuffle=True)
```

3. Apply Transforms

```
transform = transforms.Compose([
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = datasets.MNIST(root="./data", train=True, download=True, transform=transform)
```

Checkpoint after this topic

- You can load datasets using torchvision.datasets.
- You can create DataLoaders with batching & shuffling.
- You understand how to use transforms for preprocessing.
- You know how to write a custom Dataset class.

📘 Training Loops and Optimization

----- Topics to Cover

1. **Training Loop Essentials**
   - Forward pass → Compute loss → Backward pass → Update weights.
   - Difference between training loop and evaluation loop.
2. **Optimizers**
   - torch.optim.SGD
   - torch.optim.Adam
   - Switching optimizers easily.
3. **Schedulers**
   - StepLR, ExponentialLR, ReduceLROnPlateau.
4. **Regularization Techniques**
   - Dropout
   - Weight decay (L2 regularization)
   - Early stopping.

How to Learn

- Read the PyTorch optimization tutorial.
- Check how optimizer steps differ (e.g., SGD vs Adam).

- Try small models first (like MNIST or synthetic data).

🏋 How to Do It (Hands-On Example)

1. Define a Simple Model

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

# Data
transform = transforms.ToTensor()
train_dataset = datasets.MNIST(root="./data", train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root="./data", train=False, transform=transform)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# Model
class SimpleMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28*28, 128),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(128, 10)
        )

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

model = SimpleMLP()
```

2. Loss + Optimizer

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

# Try switching to Adam:
# optimizer = optim.Adam(model.parameters(), lr=0.001)
```

3. Training Loop

```
def train_one_epoch(model, dataloader, optimizer, criterion):
    model.train()
    total_loss, correct = 0, 0
    for X, y in dataloader:
        optimizer.zero_grad()
        preds = model(X)
        loss = criterion(preds, y)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        correct += (preds.argmax(1) == y).sum().item()
    accuracy = correct / len(dataloader.dataset)
    return total_loss / len(dataloader), accuracy
```

4. Evaluation Loop

```
def evaluate(model, dataloader, criterion):
    model.eval()
    total_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            preds = model(X)
            loss = criterion(preds, y)
            total_loss += loss.item()
            correct += (preds.argmax(1) == y).sum().item()
    accuracy = correct / len(dataloader.dataset)
    return total_loss / len(dataloader), accuracy
```

5. Training with Scheduler

```
from torch.optim.lr_scheduler import StepLR

scheduler = StepLR(optimizer, step_size=5, gamma=0.5)

epochs = 10
for epoch in range(epochs):
    train_loss, train_acc = train_one_epoch(model, train_loader, optimizer, criterion)
    val_loss, val_acc = evaluate(model, test_loader, criterion)
    scheduler.step()

    print(f"Epoch {epoch+1}: "
```

```
      f"Train Loss={train_loss:.4f}, Train Acc={train_acc:.4f}, "
      f"Val Loss={val_loss:.4f}, Val Acc={val_acc:.4f}")
```

---------- **Training Loop Playground** --------

- MNIST dataset with DataLoader
- MLP model with Dropout
- Training + evaluation loops
- Comparison of **SGD vs Adam**
- Learning rate scheduler usage
- Plotting training & validation curves
- **Exercises** for you to extend

```
# 📓 Training Loop Playground - PyTorch

# --- Imports ---
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim.lr_scheduler import StepLR
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# --- 1. Dataset & DataLoader ---
transform = transforms.ToTensor()
train_dataset = datasets.MNIST(root="./data", train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root="./data", train=False, transform=transform)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# --- 2. Define Model ---
class SimpleMLP(nn.Module):
    def __init__(self, dropout_rate=0.3):
        super().__init__()
        self.net = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28*28, 128),
            nn.ReLU(),
            nn.Dropout(dropout_rate),
            nn.Linear(128, 10)
```

```
    )

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

# --- 3. Training & Evaluation Functions ---
def train_one_epoch(model, dataloader, optimizer, criterion):
  model.train()
  total_loss, correct = 0, 0
  for X, y in dataloader:
    optimizer.zero_grad()
    preds = model(X)
    loss = criterion(preds, y)
    loss.backward()
    optimizer.step()
    total_loss += loss.item()
    correct += (preds.argmax(1) == y).sum().item()
  accuracy = correct / len(dataloader.dataset)
  return total_loss / len(dataloader), accuracy

def evaluate(model, dataloader, criterion):
  model.eval()
  total_loss, correct = 0, 0
  with torch.no_grad():
    for X, y in dataloader:
      preds = model(X)
      loss = criterion(preds, y)
      total_loss += loss.item()
      correct += (preds.argmax(1) == y).sum().item()
  accuracy = correct / len(dataloader.dataset)
  return total_loss / len(dataloader), accuracy

# --- 4. Training Loop with Optimizer & Scheduler ---
def run_training(optimizer_name="SGD", dropout_rate=0.3, lr=0.01, epochs=10):
  model = SimpleMLP(dropout_rate=dropout_rate)
  criterion = nn.CrossEntropyLoss()
  if optimizer_name == "SGD":
    optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9)
  elif optimizer_name == "Adam":
    optimizer = optim.Adam(model.parameters(), lr=lr)
  else:
    raise ValueError("Choose 'SGD' or 'Adam'")

  scheduler = StepLR(optimizer, step_size=5, gamma=0.5)

  history = {"train_loss": [], "train_acc": [], "val_loss": [], "val_acc": []}

  for epoch in range(epochs):
```

```
        train_loss, train_acc = train_one_epoch(model, train_loader, optimizer, criterion)
        val_loss, val_acc = evaluate(model, test_loader, criterion)
        scheduler.step()

        history["train_loss"].append(train_loss)
        history["train_acc"].append(train_acc)
        history["val_loss"].append(val_loss)
        history["val_acc"].append(val_acc)

        print(f"Epoch {epoch+1}/{epochs} | "
              f"Train Loss={train_loss:.4f}, Train Acc={train_acc:.4f}, "
              f"Val Loss={val_loss:.4f}, Val Acc={val_acc:.4f}")

    return history

# --- 5. Compare Optimizers ---
history_sgd = run_training(optimizer_name="SGD", lr=0.01, epochs=10)
history_adam = run_training(optimizer_name="Adam", lr=0.001, epochs=10)

# --- 6. Plot Results ---
def plot_history(history, label):
    plt.plot(history["train_loss"], label=f"{label} Train Loss")
    plt.plot(history["val_loss"], label=f"{label} Val Loss")
    plt.plot(history["train_acc"], label=f"{label} Train Acc")
    plt.plot(history["val_acc"], label=f"{label} Val Acc")

plt.figure(figsize=(12,5))
plot_history(history_sgd, "SGD")
plot_history(history_adam, "Adam")
plt.legend()
plt.title("SGD vs Adam on MNIST")
plt.xlabel("Epoch")
plt.show()

# --- 7. Exercises ---
# 1. Change the Dropout rate and compare results (0.0 vs 0.3 vs 0.5).
# 2. Try different schedulers (ExponentialLR, ReduceLROnPlateau).
# 3. Add L2 weight decay (regularization) to the optimizer.
# 4. Train for more epochs and plot accuracy curves separately.
# 5. Implement Early Stopping (stop when val loss stops improving).
```

📓 CNN Playground - PyTorch

🔷 1. Imports & Dataset

- Use **MNIST** (grayscale, easier) or **CIFAR-10** (RGB, more challenging).
- Apply transforms like normalization & data augmentation.

🔷 2. Define a CNN

- Start with a **simple CNN** (like LeNet: conv → pool → conv → pool → FC).
- Optionally compare with a **torchvision ResNet** later.

🔷 3. Training & Evaluation Loops

- Reuse the `train` & `evaluate` functions from earlier.

🔷 4. Visualizations

- Plot training vs validation accuracy.
- Show sample predictions.

🔷 5. Exercises

- Add more conv layers.
- Compare MNIST vs CIFAR-10.
- Swap in a pretrained model (ResNet18).

-------- code template: ---------

```
# 📓 CNN Playground - PyTorch

# --- 1. Imports ---
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# --- 2. Dataset ---
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # for MNIST (grayscale)
```

```
])

train_dataset = datasets.MNIST(root="./data", train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root="./data", train=False, transform=transform)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# --- 3. Define a Simple CNN (LeNet-style) ---
class SimpleCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=3, padding=1),  # [B,16,28,28]
            nn.ReLU(),
            nn.MaxPool2d(2,2),  # [B,16,14,14]

            nn.Conv2d(16, 32, kernel_size=3, padding=1),  # [B,32,14,14]
            nn.ReLU(),
            nn.MaxPool2d(2,2),  # [B,32,7,7]

            nn.Flatten(),
            nn.Linear(32*7*7, 128),
            nn.ReLU(),
            nn.Linear(128, 10)
        )

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

# --- 4. Training & Evaluation Functions ---
def train_one_epoch(model, dataloader, optimizer, criterion):
    model.train()
    total_loss, correct = 0, 0
    for X, y in dataloader:
        optimizer.zero_grad()
        preds = model(X)
        loss = criterion(preds, y)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        correct += (preds.argmax(1) == y).sum().item()
    return total_loss/len(dataloader), correct/len(dataloader.dataset)

def evaluate(model, dataloader, criterion):
    model.eval()
    total_loss, correct = 0, 0
    with torch.no_grad():
```

```
        for X, y in dataloader:
            preds = model(X)
            loss = criterion(preds, y)
            total_loss += loss.item()
            correct += (preds.argmax(1) == y).sum().item()
    return total_loss/len(dataloader), correct/len(dataloader.dataset)

# --- 5. Training Loop ---
def run_training(epochs=5, lr=0.01):
    model = SimpleCNN()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)

    history = {"train_loss": [], "train_acc": [], "val_loss": [], "val_acc": []}
    for epoch in range(epochs):
        train_loss, train_acc = train_one_epoch(model, train_loader, optimizer, criterion)
        val_loss, val_acc = evaluate(model, test_loader, criterion)
        history["train_loss"].append(train_loss)
        history["train_acc"].append(train_acc)
        history["val_loss"].append(val_loss)
        history["val_acc"].append(val_acc)

        print(f"Epoch {epoch+1}/{epochs}: "
              f"Train Acc={train_acc:.4f}, Val Acc={val_acc:.4f}")

    return model, history

model, history = run_training(epochs=5)

# --- 6. Plot Results ---
plt.plot(history["train_acc"], label="Train Acc")
plt.plot(history["val_acc"], label="Val Acc")
plt.legend()
plt.title("CNN Accuracy on MNIST")
plt.xlabel("Epoch")
plt.show()

# --- 7. Sample Predictions ---
examples = next(iter(test_loader))
images, labels = examples
outputs = model(images)
preds = outputs.argmax(1)

plt.figure(figsize=(8,4))
for i in range(8):
    plt.subplot(2,4,i+1)
    plt.imshow(images[i][0], cmap="gray")
    plt.title(f"Pred:{preds[i].item()} True:{labels[i].item()}")
```

```
    plt.axis("off")
plt.show()

# --- 8. Exercises ---
# 1. Replace MNIST with CIFAR-10 (RGB images, 10 classes).
# 2. Add BatchNorm after conv layers and compare results.
# 3. Increase network depth (3+ conv layers).
# 4. Replace SimpleCNN with torchvision.models.resnet18(pretrained=False, num_classes=10).
# 5. Add data augmentation (RandomCrop, RandomHorizontalFlip).
```

📓 RNN Playground - PyTorch

◈ 1. Imports & Toy Dataset

- Example dataset: a sequence of characters ("hello", "world", etc.) → next-character prediction.
- Or a synthetic **sine wave** → predict the next value.

◈ 2. Define Models

- **Vanilla RNN**
- **LSTM**
- **GRU**

◈ 3. Training Loop

- Teach the network to predict the **next token/value**.

◈ 4. Generation

- Use the trained model to **generate sequences** (text or future time steps).

◈ 5. Exercises

- Switch RNN → LSTM → GRU.
- Train on longer text (e.g., Shakespeare snippet from torchtext).
- Try sequence-to-sequence (input → reversed output).

---- ready-to-run notebook template: ------

```
# 📓 RNN Playground - PyTorch
```

```
# --- 1. Imports ---
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

# --- 2. Toy Dataset: Predict Next Character ---
text = "hello pytorch"
chars = sorted(list(set(text)))
stoi = {ch: i for i, ch in enumerate(chars)}
itos = {i: ch for ch, i in stoi.items()}
vocab_size = len(chars)

def encode(s): return [stoi[c] for c in s]
def decode(l): return ''.join([itos[i] for i in l])

data = torch.tensor(encode(text), dtype=torch.long)

# --- 3. Define RNN Model ---
class CharRNN(nn.Module):
  def __init__(self, vocab_size, hidden_size=16, rnn_type="RNN"):
    super().__init__()
    self.embed = nn.Embedding(vocab_size, hidden_size)
    if rnn_type == "RNN":
      self.rnn = nn.RNN(hidden_size, hidden_size, batch_first=True)
    elif rnn_type == "LSTM":
      self.rnn = nn.LSTM(hidden_size, hidden_size, batch_first=True)
    elif rnn_type == "GRU":
      self.rnn = nn.GRU(hidden_size, hidden_size, batch_first=True)
    self.fc = nn.Linear(hidden_size, vocab_size)
    self.rnn_type = rnn_type

  def forward(self, x, hidden=None):
    x = self.embed(x)
    out, hidden = self.rnn(x, hidden)
    logits = self.fc(out)
    return logits, hidden

# --- 4. Training ---
def train_rnn(rnn_type="RNN", epochs=200, lr=0.05):
  model = CharRNN(vocab_size, hidden_size=32, rnn_type=rnn_type)
  criterion = nn.CrossEntropyLoss()
  optimizer = optim.Adam(model.parameters(), lr=lr)

  losses = []
  for epoch in range(epochs):
    optimizer.zero_grad()
```

```
        inputs = data[:-1].unsqueeze(0)  # [1, seq_len-1]
        targets = data[1:].unsqueeze(0)  # [1, seq_len-1]
        logits, _ = model(inputs)
        loss = criterion(logits.view(-1, vocab_size), targets.view(-1))
        loss.backward()
        optimizer.step()
        losses.append(loss.item())
        if (epoch+1) % 50 == 0:
            print(f"{rnn_type} Epoch {epoch+1}/{epochs}, Loss={loss.item():.4f}")

    plt.plot(losses)
    plt.title(f"{rnn_type} Training Loss")
    plt.show()

    return model

# --- 5. Sequence Generation ---
def generate(model, start="h", length=20):
    model.eval()
    input_seq = torch.tensor([[stoi[start]]], dtype=torch.long)
    hidden = None
    output_str = start
    for _ in range(length):
        logits, hidden = model(input_seq, hidden)
        probs = torch.softmax(logits[:, -1, :], dim=-1)
        idx = torch.multinomial(probs, 1).item()
        output_str += itos[idx]
        input_seq = torch.tensor([[idx]], dtype=torch.long)
    return output_str

# --- 6. Run Example ---
model_rnn = train_rnn("RNN")
print("Generated (RNN):", generate(model_rnn, start="h"))

model_lstm = train_rnn("LSTM")
print("Generated (LSTM):", generate(model_lstm, start="h"))

model_gru = train_rnn("GRU")
print("Generated (GRU):", generate(model_gru, start="h"))

# --- 7. Exercises ---
# 1. Change text = "own sentence here".
# 2. Train on a longer text dataset (e.g., tiny Shakespeare).
# 3. Compare RNN vs LSTM vs GRU on loss curves.
# 4. Implement many-to-one task: sentiment classification of sequences.
# 5. Try predicting next values in a sine wave time series instead of text.
```

Advanced Architectures -----------

- 🔷 **Transformers** → attention, self-attention, sequence-to-sequence
- 🔷 **Generative Models** → GANs, VAEs
- 🔷 **Transfer Learning** → use pre-trained CNNs (ResNet, EfficientNet, ViT, etc.)

📓 Advanced Architectures Playground (PyTorch)

Ready-to-run Jupyter Notebook for this stage:

🔶 1. Setup & Imports

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
```

🔶 2. Transfer Learning Example (Fine-tune ResNet18 on CIFAR-10)

```
# Dataset
transform = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor()
])

trainset = torchvision.datasets.CIFAR10(root="./data", train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)

testset = torchvision.datasets.CIFAR10(root="./data", train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)

# Pretrained ResNet
model = torchvision.models.resnet18(pretrained=True)

# Replace final layer (for CIFAR-10 classes)
model.fc = nn.Linear(model.fc.in_features, 10)

# Loss & Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-4)
```

◈ 3. Training Loop (Fine-tuning)

```
def train(model, loader, optimizer, criterion, epochs=2):
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for images, labels in loader:
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        print(f"Epoch {epoch+1}/{epochs}, Loss={total_loss/len(loader):.4f}")

train(model, trainloader, optimizer, criterion)
```

◈ 4. Transformers (Toy Example with nn.Transformer)

```
# Tiny transformer model
model_t = nn.Transformer(d_model=16, nhead=2, num_encoder_layers=2)

src = torch.rand((10, 32, 16))  # (seq_len, batch, d_model)
tgt = torch.rand((20, 32, 16))

out = model_t(src, tgt)
print("Transformer output shape:", out.shape)
```

◈ 5. GAN (Skeleton)

```
class Generator(nn.Module):
    def __init__(self, z_dim=64, img_dim=784):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(z_dim, 256), nn.ReLU(),
            nn.Linear(256, img_dim), nn.Tanh()
        )
    def forward(self, z): return self.net(z)

class Discriminator(nn.Module):
    def __init__(self, img_dim=784):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(img_dim, 256), nn.LeakyReLU(0.2),
```

```
        nn.Linear(256, 1), nn.Sigmoid()
    )
  def forward(self, x): return self.net(x)
```

◈ 6. Exercises

- Fine-tune a **ResNet** on a **subset of CIFAR-10** (say only 3 classes).
- Swap **ResNet18 → Vision Transformer (ViT)** from torchvision.models.
- Use the **Transformer block** for a **mini machine translation task** (toy seq2seq).
- Implement a simple **GAN** to generate MNIST-like digits.
- Try **VAE** for latent space exploration.

-------- Advanced Architectures Playground· python ----------------

```
# 📓 Advanced Architectures Playground (PyTorch)
EPOCHS_FT = 2
for epoch in range(EPOCHS_FT):
tl, ta = train_one_epoch(resnet, train_loader, opt_ft, crit)
vl, va = evaluate(resnet, val_loader, crit)
history_resnet["train_acc"].append(ta)
history_resnet["val_acc"].append(va)
print(f"[ResNet-FT] Epoch {epoch+1}/{EPOCHS_FT} | TrainAcc={ta:.3f} ValAcc={va:.3f}")

plot_history({"ResNet18 (ft)": history_resnet}, title="ResNet18 Fine-tuning (CIFAR-10)")

# ==============================================================
# B) Transformers: Tiny seq2seq (copy task) using nn.Transformer
# ==============================================================

# --- B.1 Positional Encoding ---
class PositionalEncoding(nn.Module):
def __init__(self, d_model: int, max_len: int = 5000):
super().__init__()
pe = torch.zeros(max_len, d_model)
position = torch.arange(0, max_len).unsqueeze(1)
div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
pe[:, 0::2] = torch.sin(position * div_term)
pe[:, 1::2] = torch.cos(position * div_term)
pe = pe.unsqueeze(0) # [1, max_len, d_model]
self.register_buffer('pe', pe)
```

```
def forward(self, x): # x: [B, L, D]
L = x.size(1)
return x + self.pe[:, :L]

# --- B.2 Toy tokenizer: integers 1..V; task: copy src → tgt (shifted) ---
V = 20 # vocab size
PAD = 0

class TinyCopyModel(nn.Module):
def __init__(self, vocab_size=V, d_model=64, nhead=4, nlayers=2):
super().__init__()
self.emb = nn.Embedding(vocab_size, d_model)
self.pos = PositionalEncoding(d_model)
self.transformer = nn.Transformer(d_model=d_model, nhead=nhead,
num_encoder_layers=nlayers,
num_decoder_layers=nlayers,
batch_first=True)
self.fc = nn.Linear(d_model, vocab_size)
def forward(self, src, tgt): # [B,L]
src = self.pos(self.emb(src))
tgt = self.pos(self.emb(tgt))
out = self.transformer(src, tgt)
return self.fc(out)

# --- B.3 Data generator ---

def make_batch(batch_size=32, seq_len=12):
# tokens 1..V-1, PAD=0 not used here
src = torch.randint(1, V, (batch_size, seq_len))
# target is same as src but shifted right with BOS=1 (reuse token 1 as BOS)
bos = torch.ones(batch_size, 1, dtype=torch.long)
tgt_in = torch.cat([bos, src[:, :-1]], dim=1)
tgt_out = src  # predict src tokens
return src, tgt_in, tgt_out

# --- B.4 Train tiny transformer on copy task ---
model_tx = TinyCopyModel().to(DEVICE)
opt_tx = optim.Adam(model_tx.parameters(), lr=1e-3)
crit_tx = nn.CrossEntropyLoss()

losses_tx = []
for step in range(200): # small demo; increase steps for better perf
src, tgt_in, tgt_out = make_batch()
src, tgt_in, tgt_out = src.to(DEVICE), tgt_in.to(DEVICE), tgt_out.to(DEVICE)
opt_tx.zero_grad(set_to_none=True)
logits = model_tx(src, tgt_in) # [B,L,V]
```

```
loss = crit_tx(logits.reshape(-1, V), tgt_out.reshape(-1))
loss.backward()
opt_tx.step()
losses_tx.append(loss.item())
if (step+1) % 50 == 0:
print(f"[Transformer] Step {step+1}/200 Loss={loss.item():.3f}")

plt.figure()
plt.plot(losses_tx)
plt.title("Tiny Transformer - Copy Task Loss")
plt.xlabel("Step")
plt.show()

@torch.no_grad()
def copy_infer(seq_len=10):
src, _, _ = make_batch(1, seq_len)
src = src.to(DEVICE)
bos = torch.ones(1, 1, dtype=torch.long, device=DEVICE)
```

📘 Topics to Study

1. **Saving and L oading Models**
    - torch.save(model.state_dict()) & model.load_state_dict()
    - Full model save vs. state dict save
2. **TorchScript**
    - torch.jit.trace vs. torch.jit.script
    - Running scripted models without Python overhead
3. **ONNX Export**
    - Export PyTorch → ONNX (torch.onnx.export)
    - Verify with onnxruntime
4. **Quantization**
    - Post-training quantization (torch.quantization)
    - Dynamic vs. static quantization
5. **Pruning**
    - Structured/unstructured pruning (torch.nn.utils.prune)
6. **Distributed Training**

- nn.DataParallel (single machine, multi-GPU)
- DistributedDataParallel (multi-machine setup)

How to Learn

- Read the official PyTorch Deployment & Production docs.
- Explore ONNX docs & examples.
- Watch a short tutorial on quantization & pruning.
- (Optional) Try PyTorch Lightning for easier distributed training.

🧑💻 How to Do It (Mini Workflow)

1. Train a small CNN (MNIST).
2. Save and load the model.
3. Convert it to TorchScript (torch.jit).
4. Export to ONNX and run inference with ONNX Runtime.
5. Apply quantization to shrink size.
6. Apply pruning to remove weights.
7. Try DataParallel if you have GPU(s).

------------ Deployment & Optimization Playground· python -------------------

```
# Deployment & Optimization Playground - PyTorch
import os
import io
import numpy as np
import matplotlib.pyplot as plt

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", DEVICE)

# -----------------------------
# 1) Define a small CNN (MNIST)
# -----------------------------
class SimpleCNN(nn.Module):
def __init__(self):
```

```
super().__init__()
self.conv = nn.Sequential(
nn.Conv2d(1, 16, 3, padding=1),
nn.ReLU(),
nn.MaxPool2d(2),
nn.Conv2d(16, 32, 3, padding=1),
nn.ReLU(),
nn.MaxPool2d(2)
)
self.fc = nn.Sequential(
nn.Flatten(),
nn.Linear(32*7*7, 128),
nn.ReLU(),
nn.Linear(128, 10)
)
def forward(self, x):
x = self.conv(x)
x = self.fc(x)
return x

# Instantiate and (optionally) train briefly for demo
model = SimpleCNN().to(DEVICE)
print(model)

# Use MNIST one-batch to emulate training step if needed
transform = T.Compose([T.ToTensor(), T.Normalize((0.5,), (0.5,))])
train_ds = torchvision.datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
train_loader = DataLoader(train_ds, batch_size=128, shuffle=True)

# quick single epoch lightweight "train" (comment out if you have a trained model)
criterion = nn.CrossEntropyLoss()
opt = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

print("Running 1 quick training epoch (demo)...")
model.train()
for i, (xb, yb) in enumerate(train_loader):
xb, yb = xb.to(DEVICE), yb.to(DEVICE)
opt.zero_grad()
out = model(xb)
loss = criterion(out, yb)
loss.backward()
opt.step()
if I >= 1: # only 2 batches to keep demo short
break
print("Done quick demo training")
```

```
# -----------------------------
# 2) Saving & Loading
# -----------------------------
os.makedirs('models', exist_ok=True)
state_path = 'models/simple_cnn_state.pth'
torch.save(model.state_dict(), state_path)
print(f"Saved state_dict to {state_path}")

# Load into a fresh model
model2 = SimpleCNN().to(DEVICE)
model2.load_state_dict(torch.load(state_path, map_location=DEVICE))
model2.eval()
print("Loaded model2 state_dict and set to eval()")

# Also save full model (not recommended for production due to env coupling)
full_path = 'models/simple_cnn_full.pth'
torch.save(model, full_path)
print(f"Saved entire model to {full_path}")
model_full = torch.load(full_path, map_location=DEVICE)
print("Loaded full model instance")

# -----------------------------
# 3) TorchScript: tracing & scripting
# -----------------------------
example_input = torch.randn(1,1,28,28, device=DEVICE)
# Tracing
traced = torch.jit.trace(model2, example_input)
traced_path = 'models/simple_cnn_traced.pt'
traced.save(traced_path)
print(f"Saved traced TorchScript model to {traced_path}")
# Scripting (works if model uses control flow; here tracing suffices)
```

---------- Deployment & Optimization Playground ----------------

It contains runnable examples for:

- Saving & loading (`state_dict` and full model)
- TorchScript (tracing and scripting) and verification
- ONNX export (with dynamic axes)
- Quantization examples (dynamic quant; simplified static quant demo)
- Pruning (unstructured L1 pruning + removal)
- Optional `DataParallel` demo (runs if multiple GPUs detected)

Notes & tips:

- Static quantization can require model fusing and careful module naming; the notebook includes a simplified demonstration and safely skips if structure isn't compatible.
- To run ONNX inference, install `onnxruntime` in your environment — the notebook shows how to export but not the runtime invocation.
- For production, prefer TorchScript or ONNX for platform-agnostic deployment; use quantization/pruning for size & latency gains.

📘 Theory of Deep Learning

🔷 1. Backpropagation

- **What it is:** The algorithm for computing gradients in neural networks by applying the chain rule of calculus across layers.
- **Why it matters:** Enables efficient parameter updates during training.
- **Key Idea:** Each parameter's gradient is derived from how it influences the loss function.

In PyTorch:

- Autograd handles backprop automatically with `.backward()`.
- You can visualize it by manually computing derivatives of small functions and comparing with `autograd`.

🔷 2. Gradient Descent Variants

- **Vanilla Gradient Descent:** Updates weights using full dataset (too slow for large data).
- **Stochastic Gradient Descent (SGD):** Updates with one sample at a time (fast but noisy).
- **Mini-batch SGD:** Uses small subsets (standard in practice).
- **Momentum, RMSProp, Adam:** Add tricks like momentum or adaptive learning rates for stability.

In PyTorch:

- `torch.optim.SGD`, `torch.optim.Adam`, etc.
- You can swap optimizers with just one line change.

3. Vanishing & Exploding Gradients

- **Vanishing gradients:** Gradients shrink as they are backpropagated → early layers learn very slowly. Common in deep networks with sigmoid/tanh.
- **Exploding gradients:** Gradients grow exponentially → unstable updates.
- **Fixes:**
  - Use ReLU (or variants)
  - Gradient clipping (torch.nn.utils.clip_grad_norm_)
  - Proper initialization (e.g., Xavier, He)
  - Residual connections (ResNets)

In PyTorch:

- Can check gradient norms inside training loops to debug.

4. Evaluation Metrics

- **Classification:** Accuracy, Precision, Recall, F1-score, AUC.
- **Regression:** MSE, MAE, R^2.
- **Why not just accuracy?** Accuracy can be misleading on imbalanced datasets. Precision/recall highlight trade-offs.

In PyTorch:

- Compute metrics by comparing predictions (y_pred.argmax(dim=1)) to labels.
- Can use torchmetrics or write custom functions.

5. Connecting Theory to Practice

- Backprop = .backward() in PyTorch.
- Gradient descent = optimizer steps (optimizer.step()).
- Vanishing/exploding = check gradients in training logs.
- Metrics = implement inside evaluation loop.

------------ Deep Learning Theory Playground.ipynb. **---------------**

1. **Vanishing/Exploding Gradients Demo**
2. **Training with SGD vs Adam**
3. **Evaluation Metrics (Accuracy, Precision, Recall, F1)** on a toy dataset

```
# Deep Learning Theory Playground
# =================================
# Covers: Vanishing/Exploding Gradients, Optimizers (SGD vs Adam),
# and Evaluation Metrics (Accuracy, Precision, Recall, F1)

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt
import numpy as np
```

🔷 1. Vanishing / Exploding Gradients Demo

```
# Create a deep network with Sigmoid (vanishing) vs ReLU (stable)
class DeepNet(nn.Module):
    def __init__(self, activation='sigmoid'):
        super().__init__()
        layers = []
        for _ in range(10):  # very deep
            layers.append(nn.Linear(100, 100))
            if activation == 'sigmoid':
                layers.append(nn.Sigmoid())
            else:
                layers.append(nn.ReLU())
        self.net = nn.Sequential(*layers)

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

# Input vector
x = torch.randn(1, 100)

# Track gradient norms
for act in ['sigmoid', 'relu']:
```

```
  model = DeepNet(act)
  y = model(x).sum()
  y.backward()
  grad_norms = [p.grad.norm().item() for p in model.parameters() if p.grad is not None]
  print(f"{act.upper()} gradient norm stats:")
  print(f"  min={min(grad_norms):.6f}, max={max(grad_norms):.6f}, mean={np.mean(grad_norms):.6f}")
```

🔷 2. Optimizer Comparison (SGD vs Adam)

```
# Toy dataset (binary classification)
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2, n_informative=10)
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.long)

dataset = TensorDataset(X, y)
loader = DataLoader(dataset, batch_size=32, shuffle=True)

# Simple MLP
class MLP(nn.Module):
  def __init__(self):
    super().__init__()
    self.fc1 = nn.Linear(20, 64)
    self.fc2 = nn.Linear(64, 2)
  def forward(self, x):
    return self.fc2(F.relu(self.fc1(x)))

def train_model(optimizer_type='SGD', epochs=10):
  model = MLP()
  if optimizer_type == 'SGD':
    optimizer = optim.SGD(model.parameters(), lr=0.01)
  else:
    optimizer = optim.Adam(model.parameters(), lr=0.01)
  criterion = nn.CrossEntropyLoss()
  losses = []
  for epoch in range(epochs):
    for xb, yb in loader:
      optimizer.zero_grad()
      out = model(xb)
      loss = criterion(out, yb)
      loss.backward()
      optimizer.step()
    losses.append(loss.item())
  return model, losses

sgd_model, sgd_losses = train_model('SGD')
adam_model, adam_losses = train_model('Adam')
```

```
plt.plot(sgd_losses, label="SGD")
plt.plot(adam_losses, label="Adam")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Optimizer Comparison")
plt.legend()
plt.show()
```

🔷 3. Evaluation Metrics

```
# Evaluate on the same dataset (just for demo)
def evaluate(model):
    model.eval()
    with torch.no_grad():
        preds = model(X).argmax(dim=1)
    acc = accuracy_score(y, preds)
    prec = precision_score(y, preds)
    rec = recall_score(y, preds)
    f1 = f1_score(y, preds)
    return acc, prec, rec, f1

print("SGD model metrics:", evaluate(sgd_model))
print("Adam model metrics:", evaluate(adam_model))
```

✨ Exercises

1. Modify the **deep network depth** (5 vs 20 layers) and compare vanishing gradients with sigmoid vs ReLU.
2. Try different **optimizers**: RMSprop, AdamW. Plot losses.
3. Create a **multi-class dataset** (e.g., 3 or 5 classes) and compute accuracy, precision, recall, F1 for each class.
4. Implement **gradient clipping** (`torch.nn.utils.clip_grad_norm_`) and test on exploding gradient case.

---------- PyTorch Foundation – Short Report --------------------

1. **Introduction to PyTorch**

- **Learn:** Overview, installation, compare with TensorFlow.
- **Do:** Install PyTorch → run a simple script (e.g., create & print a tensor).

2. Tensors and Basic Operations

- **Learn:** Tensor attributes → shape, dtype, device. Operations: arithmetic, indexing, reshaping, broadcasting.
- **Do:** Practice tensor addition, multiplication, slicing, reshaping.

3. Autograd and Computational Graphs

- **Learn:** Automatic differentiation, backward propagation, requires_grad, detaching.
- **Do:** Compute gradients for a function (e.g., y=x2y=x2).

4. Neural Network Basics (nn Module)

- **Learn:** nn.Module, linear layers, activations, loss functions.
- **Do:** Build an MLP (Multi-Layer Perceptron) for a toy dataset.

5. Datasets and DataLoaders

- **Learn:** torch.utils.data.Dataset, DataLoader, transforms, custom datasets.
- **Do:** Load MNIST (or similar) → create DataLoader → iterate over batches.

6. Training Loops and Optimization

- **Learn:** Training/evaluation loops, optimizers (SGD, Adam), schedulers, overfitting (dropout, regularization).
- **Do:** Write a training loop → test with different optimizers.

7. Convolutional Neural Networks (CNNs)

- **Learn:** Conv layers, pooling, LeNet/ResNet architectures.
- **Do:** Train a simple CNN for MNIST or CIFAR-10 classification.

8. Recurrent Neural Networks (RNNs) & Sequence Models

- **Learn:** RNN, LSTM, GRU, sequence tasks.
- **Do:** Build an RNN to predict next character in a string.

9. Advanced Architectures

- **Learn:** Transformers (self-attention), GANs, VAEs, transfer learning with pretrained models.
- **Do:** Fine-tune a pre-trained ResNet on a small dataset.

10. **Deployment & Optimization**

- **Learn:** Model saving/loading, TorchScript, ONNX export, quantization, pruning, distributed training.
- **Do:** Save/load model → convert to TorchScript → export ONNX.

11. **Theory of Deep Learning**

- **Learn:** Backpropagation, gradient descent variants, vanishing/exploding gradients, evaluation metrics (accuracy, precision, recall).
- **Do:** Summarize concepts → implement metrics in PyTorch.