Deep Learning in Scientific Computing

A Guided Introduction to PyTorch

Professor Vectorex Gradiens

November 25, 2025

Contents

Disclaimer

Important Notice: Apart from this paragraph, every single word in this document was written by a Large Language Model (Google Gemini). This document is intended as an aid for learning pyTorch, in particular for implementing the models presented in the course "AI in the sciences and Engineering" held at ETH Zürich in Fall Semester 2025.
The script is extremely likely to contain errors, so use it with care, and knowing that this isn't revisioned or approved by anybody who knows this subject. *Giovanni Guidarini*

A Note to My Student

My dear student,

Welcome. It is a distinct pleasure to guide you on this journey. You are already embedded in the world of scientific computing, a field of precision, rigor, and computational elegance. You will find that deep learning is not so different. At its heart, it is a powerful form of high-dimensional function approximation, built on the familiar foundations of linear algebra and calculus. The "magic" is simply that we have found a way to make these function approximators (*neural networks*) trainable on a massive scale.

Your lack of PyTorch experience is not a hindrance; it is an opportunity. You arrive with no bad habits. We will build your knowledge from the ground up, with the same care one takes in formulating a proof or designing a robust simulation. PyTorch is our "language" for this. It is expressive, powerful, and, once you are fluent, a genuine joy to use.

This document is the first in a series. We will walk, then run. We start with the fundamental atom of this universe: the `torch.Tensor`. We will then explore the "soul" of PyTorch, its automatic differentiation engine (`autograd`). From there, we will construct our first simple models, graduate to convolutional and recurrent networks, and, finally, assemble the magnificent architecture that is the Transformer.

Treat this not just as a technical exercise, but as an art. The code we write, like the LaTeX that renders this page, can and should be clean, precise, and beautiful.

Let us begin.

Part I

The Foundations of PyTorch

1 Module 1: The `torch.Tensor`

1.1 Introduction

Everything in PyTorch, from your input data (a simulation mesh, a time-series, an image) to the parameters of your neural network (the "weights" and "biases"), is represented as a **tensor**.

If you have used NumPy, you are already familiar with the core concept: the `ndarray`. A PyTorch tensor is, at its core, the same thing—a multi-dimensional array. So why not just use NumPy?

Two fundamental reasons:

1. **GPU Acceleration:** PyTorch tensors can be effortlessly moved to a Graphics Processing Unit (GPU). This allows for *massive* parallelization of computations, turning operations that would take hours on a CPU into minutes or seconds. For the scale of problems in scientific computing and deep learning, this is not optional; it is essential.

2. **Automatic Differentiation:** This is the engine of modern deep learning. PyTorch can automatically compute the gradient (the vector of partial derivatives) of any output with respect to any input. This mechanism, called `autograd`, allows us to perform *gradient descent* to "learn" our model's parameters. We will dedicate all of Module 2 to this concept.

1.2 Theory: A Rigorous Definition

Definition

Tensor – A **tensor** is a multi-dimensional array of numerical values. It is a generalization of scalars (0D tensor), vectors (1D tensor), and matrices (2D tensor) to an arbitrary number of dimensions.
In PyTorch, a `torch.Tensor` is an object that encapsulates this data and is characterized by three primary attributes:

- **`shape` (or `size`)**: A tuple of integers describing the size of each dimension. For example, a shape of `(3, 4, 5)` describes a 3D tensor.
- **`dtype` (data type)**: The type of data the tensor holds, e.g., `torch.float32` (32-bit floating point, the default for ML) or `torch.float64` (double precision, common in SciComp).
- **`device`**: The memory where the tensor is stored, e.g., `'cpu'` or `'cuda:0'` (for the first available NVIDIA GPU).

The number of dimensions is known as the **rank** of the tensor.

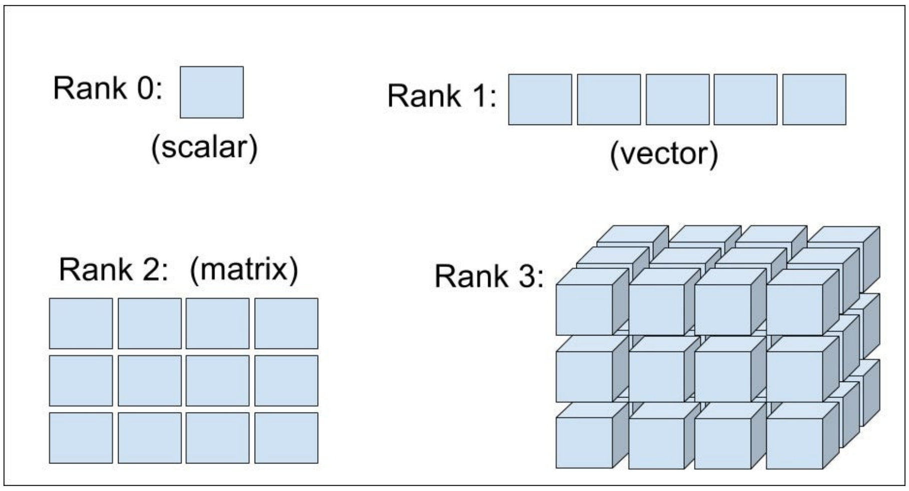

Figure 1: Visualizing Tensors: A 0D scalar, a 1D vector, a 2D matrix, and a 3D tensor.

1.3 Core PyTorch Commands for Module 1

Here are the essential commands you will need to master for tensor manipulation.

1.3.1 Tensor Creation

PyTorch Command

`torch.tensor(data)` – Creates a tensor from existing Python data (like a list or NumPy array). PyTorch infers the `dtype`.

```python
# From a Python list
a = torch.tensor([[1, 2], [3, 4]])

# Specifically setting dtype (good for SciComp)
b = torch.tensor([1.0, 2.0], dtype=torch.float64)
```

PyTorch Command

`torch.zeros(shape)`, `torch.ones(shape)` – Creates a tensor of a given `shape` filled entirely with 0s or 1s.

```python
# A 3x3 matrix of zeros
m_zeros = torch.zeros((3, 3))

# A 2x4x3 tensor of ones
t_ones = torch.ones((2, 4, 3))
```

PyTorch Command

`torch.rand(shape)`, `torch.randn(shape)` – Creates a tensor of a given `shape` with random numbers.

- `rand`: Uniform distribution on $[0, 1)$.
- `randn`: Standard normal (Gaussian) distribution with mean 0 and variance 1. This is *extremely* common for initializing neural network weights.

```python
# 5x2 tensor, standard normal distribution
weights = torch.randn((5, 2))
```

Professor's Note

You will also see `torch.zeros_like(input)` and `torch.rand_like(input)`. These are excellent helper functions that create a new tensor with the *same shape, dtype, and device* as an existing `input` tensor. This avoids many common bugs.

1.3.2 Tensor Attributes and Device Management

PyTorch Command

`T.shape`, `T.dtype`, `T.device` – These are not functions, but *attributes* you access to inspect your tensor.

```python
T = torch.rand((4, 2))
print(T.shape)     # Output: torch.Size([4, 2])
print(T.dtype)     # Output: torch.float32
print(T.device)    # Output: cpu
```

PyTorch Command

`T.to(device)` – This is how you move a tensor between devices (e.g., from CPU to GPU). **This is a critical operation.** Computations can only happen between tensors that live on the *same device*.

```python
# Check if a GPU is available
if torch.cuda.is_available():
    device = torch.device('cuda')
    print('GPU is available!')
else:
    device = torch.device('cpu')
    print('GPU not found, using CPU.')

# Create a tensor on the CPU
T_cpu = torch.randn((10, 10))

# Move it to the GPU
T_gpu = T_cpu.to(device)

# T_cpu is NOT modified. .to() returns a NEW tensor.
# This is a common pattern.
print(T_cpu.device)   # 'cpu'
print(T_gpu.device)   # 'cuda:0' (if available)
```

1.3.3 Tensor Operations

Professor's Note

Element-wise vs. Matrix Operations – This is the single most common point of confusion for new students.

- **Element-wise** (Hadamard product): `A * B` or `torch.mul(A, B)`. This multiplies corresponding elements. `A` and `B` must have the same shape (or be "broadcastable").
- **Matrix Multiplication**: `A @ B` or `torch.matmul(A, B)`. This performs standard matrix multiplication. The inner dimensions must align (e.g., $(m, k)@(k, n) \to (m, n)$).

In deep learning, we use **both** all the time. Be precise!

PyTorch Command

`A @ B` or `torch.matmul(A, B)` – Performs matrix multiplication (or more generally, tensor dot products).

```python
A = torch.randn((5, 3))
B = torch.randn((3, 4))
C = A @ B
print(C.shape) # Output: torch.Size([5, 4])
```

PyTorch Command

Broadcasting – Broadcasting is a powerful mechanism that allows PyTorch to perform operations on tensors of different shapes. The "smaller" tensor is "broadcast" (expanded) to match the shape of the "larger" one.
Rule: Two tensors are "broadcastable" if for each dimension (starting from the trailing dimension), the dimension sizes are either equal, one of them is 1, or one of them does not exist.

```python
# Example: Adding a bias vector to a matrix
M = torch.rand((4, 3))  # Shape (4, 3)
v = torch.rand((3,))    # Shape (3,) or (1, 3)
# v is broadcast to shape (4, 3)
# by "copying" its content 4 times vertically.
R = M + v
print(R.shape) # Output: torch.Size([4, 3])
```

1.3.4 Indexing and Reshaping

PyTorch Command

Indexing – This works exactly like NumPy. You use standard `[]` notation and `:` for slices.

```python
T = torch.rand((4, 3, 2)) # 4 "matrices" of 3x2

# Get the first matrix
m0 = T[0]   # Shape: (3, 2)

# Get the second row from all matrices
rows = T[:, 1, :] # Shape: (4, 2)

# Get the last column from the last matrix
col = T[-1, :, -1] # Shape: (3,)
```

PyTorch Command

`T.reshape(shape)` or `T.view(shape)` – This is the "plumbing" of deep learning. You are constantly changing the shape of your tensors to fit the next layer of your network. `reshape` changes the shape without changing the data. The new shape must have the same total number of elements.

```python
A = torch.arange(1, 13) # Vector [1, 2, ..., 12]
print(A.shape) # torch.Size([12])

# Reshape it
B = A.reshape((3, 4))
# B is now:
# [[ 1,  2,  3,  4],
#  [ 5,  6,  7,  8],
#  [ 9, 10, 11, 12]]

# -1 tells PyTorch to infer the dimension
C = A.reshape((2, -1)) # Shape: (2, 6)
```

Professor's Note

`view` vs. `reshape` – You will see both. `T.view()` creates a tensor that *shares the same underlying memory* as `T`. `T.reshape()` *may* do this, but may also return a copy if the memory layout isn't compatible. My advice: Start with `.reshape()`. It's more flexible and safer.

PyTorch Command

`T.unsqueeze(dim)` and `T.squeeze(dim)` – This is an indispensable tool for adding or removing dimensions of size 1. This is most often used to add a "batch" dimension to a single data sample.

```python
# A 1D vector (e.g., a single data sample)
v = torch.rand(10) # Shape (10,)

# NN models expect a "batch" of data.
# We need to add a batch dimension of size 1.
# We want shape (1, 10)
v_batch = v.unsqueeze(0) # Add a new dim at index 0
print(v_batch.shape) # Output: torch.Size([1, 10])

# To remove it, we "squeeze"
v_original = v_batch.squeeze(0)
print(v_original.shape) # Output: torch.Size([10])
```

2 Module 1: Exercises

Now, it is time to practice. Open your Python environment (a Jupyter notebook is excellent for this). After `import torch`, complete the following challenges. Do not look at the answers until you have tried.

Exercise

Creation and Attributes

1. Create a 2D tensor (a matrix) from the following Python list: `[[1, 5, 9], [2, 6, 10]]`.
2. Print its `shape`, `dtype`, and `device`.
3. Create a 3D tensor of shape `(4, 2, 5)` filled with random numbers from a standard normal distribution.
4. Create another tensor with the *exact same shape* as the one from step 3, but fill it with ones, and ensure its data type is `torch.float64` (double precision).

Exercise

Operations and Broadcasting

1. Create a random matrix `A` of shape `(5, 3)`.
2. Create a random matrix `B` of shape `(3, 7)`.
3. Compute the matrix product of `A` and `B`, storing the result in `C`. What is the shape of `C`?
4. Create a random matrix `D` of shape `(5, 3)`.
5. Compute the *element-wise* product of `A` and `D`, storing the result in `E`. What is the shape of `E`?
6. Create a 1D tensor (vector) `v` of shape `(7,)`.
7. Add `v` to the matrix `C` from step 3. What broadcasting rule makes this possible?

Exercise

Indexing and Slicing

1. Create a 3D tensor `T` of shape `(5, 4, 3)` filled with integers from 0 up to (but not including) 60. (Hint: use `torch.arange()` and `.reshape()`).
2. Select and print the 2D matrix at index 2 (the third matrix). Its shape should be `(4, 3)`.
3. Select and print the vector at `T[0, 1, :]` (from the first matrix, the second row, all columns). Its shape should be `(3,)`.
4. Select and print all elements from all matrices, but only from the *last column*. The resulting tensor should have a shape of `(5, 4)`.

Exercise

Reshaping and Device Transfer (Challenge)

1. Create a 1D tensor `x` with 100 elements.
2. Reshape `x` into a 2D tensor `x_2d` of shape `(20, 5)`.
3. Now, "flatten" `x_2d` back into a 1D tensor `x_flat` using `x.reshape(-1)`. This `-1` trick is a vital shortcut.
4. Imagine `x_2d` is a single image from a "batch". Add a "batch" dimension (size 1) at the beginning, so its shape becomes `(1, 20, 5)`. (Hint: `unsqueeze`).
5. Write a code block (using the `if torch.cuda.is_available()` template) that defines your `device`.
6. Move your 3D tensor from step 4 to this `device`.
7. Print the `.device` attribute of your new tensor to confirm it worked.

Conclusion of Module 1

Well done. You have just mastered the single most important object in the PyTorch ecosystem. Every complex model, every simulation you run, every piece of data you process will be a `torch.Tensor`.

Review these operations until they feel second nature. In our next session, we will uncover the "magic" that makes PyTorch a *learning* library: **Autograd and the Computational Graph**. This is where we learn how to compute gradients automatically, the engine that powers all of modern deep learning.

Part II

The Engine of Learning

3 Module 2: Autograd & The Computational Graph

3.1 Introduction

In Module 1, we built our "bricks": the `torch.Tensor`. Now, we learn how to make them "smart." We will explore the "soul" of PyTorch: its automatic differentiation engine, `autograd`.

This is what separates a mere tensor library (like NumPy) from a deep learning framework. In scientific computing, you often must manually derive and implement your gradients to solve optimization problems. In deep learning, the problems are of such high dimensionality that this is utterly intractable.

`autograd` solves this. It "listens" to your operations and, when you are done, it can automatically compute the gradient of your final output (e.g., an error value) with respect to *any* tensor that was involved in the computation. This process is known as **backpropagation**.

3.2 Theory: A Rigorous Definition

Definition

The Computational Graph – PyTorch performs its magic by building a **Computational Graph** in real-time. This is a Directed Acyclic Graph (DAG) where:

- **Leaf Nodes** are the tensors you create (e.g., your input data, or your model's weights).
- **Intermediate Nodes** (or `grad_fn` nodes) represent the operations you perform (e.g., `AddBackward0`, `MulBackward0`, `MatMulBackward0`).
- **The Root Node** is your final output tensor (typically, a scalar "loss" value).

When you create a tensor, you can "flag" it with `requires_grad=True`. This tells PyTorch: "I will eventually want to know the gradient of some future result with respect to *this* tensor."

When you call `loss.backward()`, PyTorch traverses this graph *backwards* from the root (the loss). At each node, it applies the multivariate **chain rule** to compute the gradient and passes it down to the connected leaf nodes.

Mathematically, if we have a loss L computed from a weight w via an intermediate variable y (i.e., $y = f(w)$ and $L = g(y)$), the chain rule states:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w}$$

`autograd` simply applies this principle recursively, layer by layer, all the way back to the leaves.

3.3 Core PyTorch Commands for Module 2

3.3.1 Tracking Gradients

PyTorch Command

`T.requires_grad_()` or `requires_grad=True` – This is the "on/off" switch for a tensor.

```python
# 1. At creation time
# By default, model parameters (nn.Linear) have this
    True.
# By default, raw data (torch.tensor) has this False.
w = torch.randn((5, 2), requires_grad=True)

# 2. In-place modification (note the trailing underscore
    )
x = torch.randn((10, 5))
x.requires_grad_() # Now x is tracked
```

PyTorch Command

`L.backward()` – This is the function that *triggers* the gradient computation. It must be called on a **scalar** (0D tensor), which is why we almost always call it on the `loss`.

```python
# w, x are defined above and have requires_grad=True
y = x @ w  # 'y' now has a grad_fn (MatMulBackward0)
z = y.mean() # 'z' now has a grad_fn (MeanBackward0)

# Triggers backpropagation from 'z'
z.backward()
```

PyTorch Command

`T.grad` – After `.backward()` is called, the computed gradients are *accumulated* in the `.grad` attribute of the leaf nodes.

```python
# Following the code above
print(w.grad) # Prints a (5, 2) tensor: d(z)/d(w)
print(x.grad) # Prints a (10, 5) tensor: d(z)/d(x)
```

Professor's Note

CRITICAL: Gradients *accumulate* by default. When you run your next training step, you must **zero the gradients** first, otherwise you will be adding new gradients to old ones. This is a classic "gotcha".

3.3.2 Clearing Gradients and Disabling Tracking

PyTorch Command

`T.grad.zero_()` – An in-place operation to reset the gradient of a tensor to zero.

```python
w.grad.zero_() # Resets the gradient for 'w'
```

Professor's Note

In practice, we will use an `optimizer` object that conveniently calls `.zero_grad()` on *all* model parameters at once. We will see this in Module 3.

PyTorch Command

`with torch.no_grad():` – This is a **context manager** that disables the `autograd` engine within its block. This is *essential*. When you are evaluating your model (i.e., running "inference"), you are not training, so you do not need gradients. This context manager makes your code:

- **Faster:** It avoids building the computational graph.
- **Memory-efficient:** It doesn't store intermediate `grad_fn` objects.

```python
print(w.requires_grad) # True

with torch.no_grad():
    y_pred = x @ w
    # y_pred has no grad_fn
    # Calling y_pred.backward() here would fail!

print(y_pred.requires_grad) # False
```

PyTorch Command

`T.detach()` – Creates a new tensor that shares the same data as `T` but is "detached" from the computational graph. It will never have a gradient computed for it.

```python
a = torch.randn(5, requires_grad=True)
b = a.detach() # b shares data, but requires_grad=False
```

4 Module 2: Exercises

Your turn. These exercises are designed to build your intuition for how the graph works.

Exercise

Scalar Backpropagation

1. Define three *scalar* (0D) tensors, `a`, `w`, and `b`, with `requires_grad=True`. Initialize them to some values (e.g., `a=2.0`, `w=3.0`, `b=4.0`).
2. Define the computation $y = w \cdot a + b$. (This is `y = w * a + b`).
3. Print the `.grad_fn` attribute of `y`. What operation does it represent?
4. Now, define a final scalar "loss" $L = y^2$.
5. **On paper:** Manually compute the partial derivatives: $\frac{\partial L}{\partial w}$, $\frac{\partial L}{\partial a}$, and $\frac{\partial L}{\partial b}$. (Hint: $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w}$).
6. In code, call `L.backward()`.
7. Print `w.grad`, `a.grad`, and `b.grad`. Do they match your manual calculations?

Exercise

Gradients of Matrix Operations

1. Define a random 2D tensor `X` of shape `(10, 3)` (our data).
2. Define a random 2D tensor `W` of shape `(3, 2)` (our weights), with `requires_grad=True`.
3. Define a random 1D tensor `b` of shape `(2,)` (our bias), with `requires_grad=True`.
4. Compute the model's prediction: `Y_pred = X @ W + b`. (Broadcasting will handle `b`).
5. Create a "target" tensor `Y_target = torch.ones_like(Y_pred)`.
6. Compute the Mean Squared Error (MSE) loss: $L = \frac{1}{N} \sum (Y_{pred} - Y_{target})^2$. (Hint: `loss = ((Y_pred - Y_target)**2).mean()`).
7. Call `.backward()` on the loss.
8. Print the `.shape` of `W.grad` and `b.grad`. Do they match the shapes of `W` and `b`? Why is this crucial?

Exercise

The `no_grad` Context

1. Create a tensor `a` with `requires_grad=True`.
2. Create a `for` loop that runs 10 times.
3. **Inside** the loop, wrap the following computation in a `with torch.no_grad():` block:
4. Compute `b = a * 2`.
5. Check the `.requires_grad` attribute of `b`. Is it `True` or `False`?
6. **Outside** the `with` block (but still in the loop), compute `c = a * 2`.
7. Check the `.requires_grad` attribute of `c`. Is it `True` or `False`?
8. This exercise should demonstrate the power of the `no_grad` context for "turning off" the graph.

Exercise

Gradient Accumulation – This is a critical concept.

1. Create a weight tensor `w = torch.tensor([5.0], requires_grad=True)`.
2. Compute $L_1 = (w * 2)^2$.
3. Call `L1.backward()`.
4. Print `w.grad`. (You should get 40.0, as $\frac{dL_1}{dw} = 2 \cdot (2w) \cdot 2 = 8w = 40$).
5. **Do not zero the gradient.**
6. Now, compute $L_2 = (w * 3)^2$.
7. Call `L2.backward()`.
8. Print `w.grad` again. What is the value? Is it 90.0? Or 130.0?
9. Now, explicitly call `w.grad.zero_()`.
10. Re-run `L2.backward()` (just this call) and print `w.grad` a final time. What is it now?
11. You have just proven to yourself why `optimizer.zero_grad()` is the first step in every training loop.

Part III

Building and Training Models

5 Module 3: Your First Neural Network & Training Loop

5.1 Introduction

With Tensors (bricks) and Autograd (the physics) understood, we can finally build our "machine." In PyTorch, these machines are encapsulated in a beautiful abstraction: the `nn.Module`.

We will also formalize the "training" process by introducing **loss functions** (to score our model) and **optimizers** (to update our model's parameters using the gradients).

This module culminates in the **canonical PyTorch training loop**—a 5-step-process that is the foundation for training everything from a simple line-fitter to a massive Transformer.

5.2 Theory: A Rigorous Definition

Definition

`torch.nn.Module` – In PyTorch, a model is defined as a Python `class` that **inherits** from `torch.nn.Module`. This base class provides an enormous amount of functionality. Your only true responsibilities are:

- **In `__init__(self)`:** Define your network's "layers" (which are themselves `nn.Modules`, like `nn.Linear`). When you assign an `nn.Module` as an attribute (e.g., `self.layer1 = ...`), it is *automatically registered* as part of your model. Its parameters are tracked.
- **In `forward(self, x)`:** Define the *computation* of your model. You take the input `x` and pass it through your defined layers, returning the final output. The `autograd` engine automatically builds the computational graph based on this `forward` pass.

Definition

`nn.Linear(in_features, out_features)` – This is our first "layer" module. It applies a standard affine transformation to the input data. Given an input x with shape $(N, \ldots, \text{in\_features})$, it produces an output y with shape $(N, \ldots, \text{out\_features})$ by computing:

$$y = xW^T + b$$

The module automatically creates and manages its own parameters:
- `weight` (W): A tensor of shape $(\text{out\_features}, \text{in\_features})$.
- `bias` (b): A tensor of shape (out_features).

Both `weight` and `bias` have `requires_grad=True` by default.

Definition

Loss Function (`nn.MSELoss`) – A loss function (or "criterion") is a function that computes a scalar value measuring the "error" or "distance" between your model's `output` and the ground `target`. We call `.backward()` on this scalar. For example, the **Mean Squared Error (MSE)** loss is:

$$L(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

In PyTorch, `criterion = nn.MSELoss()` creates an object that computes this for us.

Definition

Optimizer (`torch.optim`) – An optimizer is an object that implements a specific algorithm to update your model's parameters using their computed gradients (`T.grad`). The simplest is **Stochastic Gradient Descent (SGD)**. When we call `optimizer.step()`, it performs the following update for every parameter w it manages:

$$w_{new} \leftarrow w_{old} - \eta \cdot w_{old}.grad$$

where η (eta) is the **learning rate** (a hyperparameter you must set).

5.3 Core PyTorch Commands for Module 3

We will now use two new libraries. The imports are canonical:

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

5.3.1 Defining the Model

PyTorch Command

Defining an `nn.Module` – This is the standard boilerplate for a simple model.

```python
class MyLinearModel(nn.Module):
    # 1. Initialization
    def __init__(self, input_size, output_size):
        # Call the parent class's init function first!
        super(MyLinearModel, self).__init__()

        # Define our layer(s)
        self.linear_layer = nn.Linear(input_size,
    output_size)

    # 2. The forward pass
    def forward(self, x):
        # Define the computation
        # Here, we just pass the input through our one
    layer
        return self.linear_layer(x)

# --- Usage ---
# Instantiate the model: 1 input feature, 1 output
    feature
model = MyLinearModel(input_size=1, output_size=1)
```

PyTorch Command

`model.parameters()` – This is a helper method from `nn.Module` that returns an iterator over *all* parameters (tensors with `requires_grad=True`) that were registered in `__init__`.

```python
# You pass this to the optimizer
list(model.parameters())
# Will show two tensors: the 'weight' and 'bias'
# from self.linear_layer
```

5.3.2 The Training Loop

This is the most important pattern in all of PyTorch.

PyTorch Command

The Canonical 5-Step Training Loop – We combine the model, loss, and optimizer.

```
# --- 1. Setup ---
# (Assume 'model' is defined as above)
# (Assume 'X_train' and 'y_train' are our data tensors)
learning_rate = 0.01

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

# Define our optimizer, telling it
# WHAT to optimize (model.parameters()) and
# HOW to optimize (lr)
optimizer = optim.SGD(model.parameters(), lr=
   learning_rate)

# --- 2. Training Epochs ---
num_epochs = 50
for epoch in range(num_epochs):
    # THE 5-STEP LOOP

    # 1. Forward Pass: Compute predictions
    y_pred = model(X_train)

    # 2. Compute Loss
    loss = criterion(y_pred, y_train)

    # 3. Zero Gradients
    # (VERY IMPORTANT: reset from previous loop)
    optimizer.zero_grad()

    # 4. Backward Pass: Compute gradients
    loss.backward()

    # 5. Optimizer Step: Update weights
    optimizer.step()

    if (epoch + 1) % 5 == 0:
        # Print progress (formatting omitted for LaTeX
   compatibility)
        print("Epoch", epoch+1, "/", num_epochs, "Loss:"
   , loss.item())

# After the loop, the model.parameters() have been "
   learned"
```

Professor's Note

Note: `loss.item()` is used to extract the scalar value of the loss (which is a 0D tensor) as a standard Python number. You use `.item()` for any 1-element tensor.

6 Module 3: Exercises

Time to build and train. This is your first complete deep learning workflow.

Exercise

Define a Multi-Layer Model – Your first task is to define the model class.

1. `import torch.nn as nn`.
2. Define a new class called `SimpleNet` that inherits from `nn.Module`.
3. In the `__init__` function, define two `nn.Linear` layers:
   - `self.layer1` should take 10 input features and produce 5 output features.
   - `self.layer2` should take 5 input features and produce 1 output feature.
4. In the `forward(self, x)` function, define the computation:
   - Pass the input `x` through `self.layer1`.
   - (We will add activation functions in the next module. For now, just pass the output of layer 1 directly into layer 2).
   - Pass the result of that through `self.layer2`.
   - Return the final result.
5. Instantiate your model: `model = SimpleNet()`.
6. Use `print(model)` to see a summary of its structure.

Exercise

The Full Training Loop – This is the main event. We will perform linear regression to find the parameters of a known line.

1. **Generate Data:**
   - Create a "ground truth" weight $W_{true} = 5.0$ and bias $b_{true} = -2.0$.
   - Create a 1D tensor `X` of 100 points, e.g., `torch.randn(100, 1)`.
   - Create the "ground truth" `y` using the line equation, adding some Gaussian noise: `y = X * W_true + b_true + torch.randn(100, 1) * 0.1`.
2. **Setup Model:**
   - Instantiate a linear model. You can use the `MyLinearModel(1, 1)` class from the example, or just use `model = nn.Linear(1, 1)` directly (since it's also an `nn.Module`!).
3. **Setup Optimizer and Loss:**
   - Define a `learning_rate = 0.01`.
   - Instantiate `criterion = nn.MSELoss()`.
   - Instantiate `optimizer = optim.SGD(model.parameters(), lr=learning_rate)`.
4. **Write the Training Loop:**
   - Write a `for` loop that runs for 100 `epochs`.
   - Inside the loop, implement the **5-step training process** exactly as shown in the `pytorchcmd` block.
   - Use `X` as your input and `y` as your target.
   - Add a print statement that prints the `loss.item()` every 10 epochs.
5. **Check Results:**
   - After the loop finishes, inspect your learned parameters.
   - `model.weight` should be close to 5.0.
   - `model.bias` should be close to -2.0.
   - (Hint: The parameters are themselves tensors, access their values with `.item()` or `.data`).

Conclusion of Part 2

If you have completed these exercises, you have achieved something significant. You have gone from zero to building and training a complete neural network. You now understand the *process* of deep learning in PyTorch.

This 5-step loop is the "heartbeat" of everything we will do.

In our next part, we will make our models more powerful by introducing **non-linear activation functions** (the "spark" that gives neural networks their power) and explore how to handle **batches** of data, which is essential for training on large-scale scientific datasets.

Take your time. Master this. I shall be here when you are ready for Module 4.

Part IV

Deepening the Architecture

7 Module 4: Non-Linearity & The Data Pipeline

7.1 Introduction

My dear student, reflect on the linear models of Module 3. If you stack two linear layers, say $y = (xW_1)W_2$, this is mathematically equivalent to $y = x(W_1W_2) = xW_{new}$. No matter how deep you stack them, a sequence of linear layers remains a linear function.

To approximate complex, non-linear phenomena (turbulence, wavefunctions, stock markets), we must inject **non-linearity**. We do this via **Activation Functions**.

Simultaneously, we must address engineering. We cannot load terabytes of simulation data into RAM at once. We need a mechanism to load data "lazily" and in batches. PyTorch provides the `Dataset` and `DataLoader` for this exact purpose.

7.2 Theory: Activation Functions

Definition

The Activation Function – An activation function $\sigma(\cdot)$ is a non-linear function applied element-wise to the output of a layer. Our layer definition becomes:

$$h = \sigma(xW^T + b)$$

Common choices in scientific computing include:
1. ReLU (Rectified Linear Unit):

$$\text{ReLU}(x) = \max(0, x)$$

The industry standard. It is computationally free and solves the "vanishing gradient" problem for positive values.
2. Tanh (Hyperbolic Tangent):

$$\text{Tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Outputs values in $(-1, 1)$. This is often preferred in scientific computing (physics-informed neural networks) because it is smooth (differentiable everywhere) and centered around zero, unlike ReLU.

7.3 Theory: The Data Pipeline

Definition

`Dataset` and `DataLoader` – PyTorch separates the "what" from the "how" of data loading.

- **`Dataset`:** A class that defines *where* your data is and *how to get a single item*. You must implement two magic methods:
  - `__len__`: Returns the total size of the dataset.
  - `__getitem__`: Given an index i, returns the i-th sample (input and target).
- **`DataLoader`:** An iterator that handles the logistics. It takes a `Dataset` and automatically handles:
  - **Batching:** Grouping B samples into a single tensor.
  - **Shuffling:** Randomizing order (crucial for SGD).
  - **Parallelism:** Loading data using multiple CPU workers.

7.4 Core PyTorch Commands for Module 4

7.4.1 Activations

PyTorch Command

`nn.ReLU()` and `nn.Tanh()` – These are `nn.Module`s, just like `nn.Linear`. You typically define them in `__init__` and call them in `forward`.

```python
class NeuralNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Linear(10, 20)
        self.act = nn.ReLU() # or nn.Tanh()
        self.layer2 = nn.Linear(20, 1)

    def forward(self, x):
        x = self.layer1(x)
        x = self.act(x) # Apply non-linearity
        x = self.layer2(x)
        return x
```

7.4.2 Data Handling

PyTorch Command

Custom Dataset

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

class ScientificDataset(Dataset):
    def __init__(self, x_tensor, y_tensor):
        # Store the data (usually as tensors)
        self.x = x_tensor
        self.y = y_tensor

    def __len__(self):
        # Return total number of samples
        return len(self.x)

    def __getitem__(self, idx):
        # Return tuple (input, target) for index idx
        return self.x[idx], self.y[idx]
```

PyTorch Command

Using DataLoader – This modifies our training loop. Instead of passing the whole dataset at once, we iterate over the loader.

```python
# Create dataset and loader
dataset = ScientificDataset(x_train, y_train)
dataloader = DataLoader(dataset, batch_size=32, shuffle=
    True)

# New Training Loop Structure
for epoch in range(num_epochs):
    # Iterate over batches
    for batch_x, batch_y in dataloader:
        # 1. Forward
        pred = model(batch_x)
        # 2. Loss
        loss = criterion(pred, batch_y)
        # 3, 4, 5. Zero, Backward, Step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

8 Module 4: Exercises

Exercise

The Multi-Layer Perceptron (MLP) – We will upgrade our model from Module 3.

1. Define a class `NonLinearNet`.
2. In `__init__`, define a structure with two hidden layers:
   - Layer 1: Linear (input 1 → 20 features).
   - Activation: Tanh.
   - Layer 2: Linear (20 → 20 features).
   - Activation: Tanh.
   - Layer 3: Linear (20 → 1 output).
3. Implement the `forward` pass connecting these components.
4. **Question:** Why do we typically *not* put an activation function after the very last layer for a regression problem?

Exercise

Implementing a Dataset

1. Create synthetic data: `X = torch.linspace(-10, 10, 1000).reshape(-1, 1)` `y = torch.sin(X) + 0.1 * torch.randn_like(X)`
2. Define a class `SineWaveDataset` inheriting from `torch.utils.data.Dataset`.
3. Implement `__init__`, `__len__`, and `__getitem__` to handle this data.
4. Instantiate your dataset and verify it works by printing `len(dataset)` and `dataset[0]`.

Exercise

Mini-Batch Training Loop – Combine the previous two exercises.

1. Instantiate your `SineWaveDataset`.
2. Create a `DataLoader` with `batch_size=64` and `shuffle=True`.
3. Instantiate your `NonLinearNet`, an optimizer (SGD or Adam), and MSELoss.
4. Write the training loop. **Crucial Change:** You now need a nested loop. The outer loop counts epochs; the inner loop iterates through the `dataloader`.
5. Train for 50 epochs.
6. (Optional) Use `matplotlib` to plot your model's predictions against the noisy data. The model should capture the sine wave curve, which a linear model could never do.

Part V

The Topology of Data

9 Module 5: Convolutional Neural Networks (CNNs)

9.1 Introduction: From Vectors to Manifolds

In scientific computing, structure is information. A fluid velocity field, a stress tensor on a plate, or a microscope image all possess **spatial topology**. Point (i, j) is physically correlated with point $(i + 1, j)$.

Up to this point, we have used Dense Layers (`nn.Linear`). To feed a 28×28 grid into a Dense Layer, we had to flatten it into a vector of size 784. This is destructive. It throws away the 2D topology. The network has to "re-learn" that pixel 1 is next to pixel 28.

The Convolutional Neural Network (CNN) preserves this topology. It processes data locally using a **Kernel** (or filter). For the scientific computing student, the analogy is exact: **A Convolutional Layer is a learnable Stencil operation**.

9.2 Theory: The Building Blocks

The Tensor Image

In PyTorch CNNs, data is always a 4-dimensional tensor:

$$(N, C, H, W)$$

- **N (Batch Size):** Number of samples.
- **C (Channels):** The depth of the data.
  - Input Layer: RGB image = 3 channels. Scalar field (temp) = 1 channel.
  - Hidden Layers: The number of "features" extracted at that spatial location.
- **H, W (Height, Width):** The spatial dimensions.

9.2.1 1. The Convolution Operation (`Conv2d`)

The core operation is the sliding dot product. We define a small matrix of weights called a **Kernel** (e.g., 3×3). We slide this kernel over every pixel of the input. At each position, we perform an element-wise multiplication and sum the result.

Key Hyperparameters:

1. **Kernel Size (k):** The size of the window. 3×3 is the standard. Larger kernels ($5 \times 5, 7 \times 7$) look at a wider context but are more expensive.

2. **Stride (s):** The step size.
   - $s = 1$: We slide pixel by pixel. Preserves resolution (mostly).
   - $s = 2$: We skip every other pixel. This **halves** the spatial dimension (Downsampling).
3. **Padding (p):** How we handle boundaries.
   - *Valid Padding ($p = 0$):* We only compute where the kernel fits inside the image. The image shrinks.
   - *Same Padding:* We pad the border with zeros so the Output Size = Input Size (when stride=1).

The "Ghost Cells" of Deep Learning

In Finite Volume Methods, you add ghost cells to boundaries to apply stencils. **Padding** is exactly the same concept. In CNNs, we typically use "Zero Padding" (Dirichlet BCs with value 0), but PyTorch also supports "Reflection Padding" (Neumann BCs), which is better for generative science tasks to avoid edge artifacts.

9.2.2 2. Pooling Layers

Pooling reduces the spatial resolution (H, W) while keeping the depth (C) constant.

- **Max Pooling:** Selects the maximum value in a window (e.g., 2×2).
- **Average Pooling:** Calculates the mean.

Why? It provides *invariance*. If a feature moves slightly, the max value in the window remains the same. It also reduces computational cost by shrinking the grid.

9.2.3 3. Batch Normalization (BN)

Training deep CNNs is notoriously unstable due to "Internal Covariate Shift"—the distribution of inputs to layer L changes as the parameters of layer $L - 1$ change. Batch Normalization fixes this by normalizing the activations *during training*:

$$\hat{x} = \frac{x - \mu_{batch}}{\sigma_{batch}} \cdot \gamma + \beta$$

Where γ and β are learnable parameters. This forces the data to stay centered and scaled.

9.3 Professor's Design Patterns: How to Architect

Knowing the tools is not enough; you must know how to arrange them. Here are the heuristics of the trade:

- **The VGG Block:** Do not use large kernels like 5×5 or 7×7. Instead, stack two 3×3 layers.

  $$\text{Conv}(3 \times 3) \to \text{ReLU} \to \text{Conv}(3 \times 3) \to \text{ReLU}$$

  This covers the same receptive field as a 5×5 but with fewer parameters and more non-linearity.

- **The Pyramid Principle:** As you go deeper into the network, the spatial resolution (H, W) should *decrease*, and the number of channels (C) should *increase*.

  $$\text{Resolution: } 256 \to 128 \to 64 \to 32$$

  $$\text{Channels: } 3 \to 64 \to 128 \to 256$$

  Rationale: We trade spatial precision for semantic complexity.

- **Order of Operations:** The canonical order for a single layer block is:

  $$\text{Conv2d} \longrightarrow \text{BatchNorm} \longrightarrow \text{ReLU}$$

- **Downsampling:** Modern architectures often avoid MaxPool in favor of using `Conv2d` with `stride=2`. This lets the network *learn* how to downsample rather than just taking the max.

9.4 The PyTorch Syntax

nn.Conv2d

```python
# nn.Conv2d(in_channels, out_channels, kernel_size,
    stride, padding)

# Example: Input is RGB (3), we want 64 feature maps.
# We use 3x3 kernel, stride 1, and padding 1 to keep
    size constant.
layer = nn.Conv2d(3, 64, kernel_size=3, stride=1,
    padding=1)
```

nn.MaxPool2d

```python
# Halves the dimension (H/2, W/2)
pool = nn.MaxPool2d(kernel_size=2, stride=2)
```

nn.BatchNorm2d

```python
# Argument must match the OUTPUT channels of the
    previous conv
bn = nn.BatchNorm2d(64)
```

Calculating Output Dimensions

You must memorize this formula. Failure to do so results in shape mismatch errors.

$$H_{out} = \left\lfloor \frac{H_{in} + 2p - k}{s} + 1 \right\rfloor$$

`nn.Sequential`

A cleaner way to stack layers, especially for repeated blocks (like Conv-BN-ReLU).

```
self.block = nn.Sequential(
    nn.Conv2d(1, 32, 3, padding=1),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.MaxPool2d(2)
)
```

10 Module 5: Exercises

Exercise 5.1: The Arithmetic of Geometry

Do this with pen and paper. Do not skip. Assume an input tensor of shape `(Batch=1, Channels=1, Height=32, Width=32)`. Calculate the output shape (C, H, W) after each layer sequentially:

1. `Conv2d(1, 16, kernel=5, stride=1, padding=0)`
2. `MaxPool2d(kernel=2, stride=2)`
3. `Conv2d(16, 32, kernel=3, stride=1, padding=1)`
4. `Conv2d(32, 32, kernel=3, stride=2, padding=1)`
5. `Flatten()` (What is the size of the resulting vector?)

Exercise 5.2: The "Mechanic" - Manual Weights

We will inspect the "guts" of a CNN. We will manually set the weights to create an Edge Detector (Sobel Filter).

1. Create a dummy "image" tensor: A 10×10 matrix of zeros, with a vertical strip of ones (value 5.0) in the middle (columns 4 and 5). Reshape to $(1, 1, 10, 10)$.
2. Define a `Conv2d(1, 1, kernel_size=3, stride=1, padding=0, bias=False)`.
3. Access the weights via `conv.weight.data`. This will be shape $(1, 1, 3, 3)$.
4. Overwrite the weights with the Vertical Edge Kernel:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

5. Pass the image through the layer.
6. Print the output. You should see high positive values on the left of the strip (light to dark transition) and high negative values on the right (dark to light).
7. **Reflection:** This is all a CNN does. It learns thousands of these little filters to detect edges, then shapes, then objects.

Exercise 5.3: The "Architect" - VGG Construction

Create a class `MyVGG` that strictly follows this architecture. Do not worry about training yet; just ensure the `forward` pass works without crashing.

Architecture:

- Input: $1 \times 28 \times 28$ (MNIST size)
- **Block 1:** Conv(1→32, k=3, p=1) → BN → ReLU → MaxPool(2)
- **Block 2:** Conv(32→64, k=3, p=1) → BN → ReLU → MaxPool(2)
- **Block 3:** Conv(64→128, k=3, p=1) → BN → ReLU → MaxPool(2)
- **Classifier:** Flatten → Linear($\cdots \to 10$).

Task: Calculate the input size for the Linear layer. (Hint: Trace the spatial dimension $28 \to 14 \to 7 \to 3$). Pass a random tensor through it to verify.

Exercise 5.4: The "Experiment" - Real World Training

Now we put it all together. We will train on real data: MNIST (handwritten digits).

1. **Data Loading:** Use `torchvision` to load data (standard procedure in PyTorch).

```
  from torchvision import datasets, transforms
  transform = transforms.ToTensor()
  train_data = datasets.MNIST(root='./data', train
=True, download=True, transform=transform)
  train_loader = torch.utils.data.DataLoader(
train_data, batch_size=64, shuffle=True)

```

2. Instantiate your `MyVGG` model from Exercise 5.3.
3. Define Optimizer (`Adam`, lr=0.001) and Loss (`CrossEntropyLoss`).
4. **The Loop:** Write the standard training loop (Forward $\rightarrow$ Loss $\rightarrow$ ZeroGrad $\rightarrow$ Backward $\rightarrow$ Step).
5. Train for 5 epochs.
6. Calculate accuracy on the training set. It should easily exceed 98%.
7. **Challenge:** Add `model.train()` at the start of the loop and `model.eval()` during inference. This is crucial because Batch Norm behaves differently in training vs. testing!

Conclusion of Expanded Module 5

You have now moved beyond the "black box" view of CNNs. You understand them as hierarchical feature extractors and learnable stencil operators. You can calculate their dimensions precisely, and you can manipulate them for both classification (reduction) and generation (reconstruction).

Part VI

The Architecture of Time and Context

11 Module 6: Recurrent Neural Networks (RNNs)

11.1 Theory: The Problem of Sequence

Standard Feed-Forward networks (and CNNs) function on the assumption of **independence**. They assume that input X_i is independent of input X_{i+1}. In scientific computing, this is rarely true. In a time-series simulation of a chaotic system, X_t is strictly determined by X_{t-1}. In a DNA sequence, a base pair at position i interacts with one at position j.

To model this, we need **Memory**. We need a network that maintains an internal **Hidden State** (h_t) that acts as a summary of everything it has seen so far.

11.1.1 1. The Vanilla RNN: A Dynamical System

Mathematically, an RNN is a discrete-time dynamical system. Let $x_t \in \mathbb{R}^d$ be the input at time t. Let $h_t \in \mathbb{R}^h$ be the hidden state at time t. The update rule is:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b)$$

$$y_t = W_{hy}h_t + b_y$$

Here, W_{hh} is the **Recurrent Weight Matrix**. It is applied over and over again. This is where the power and the danger lie.

11.1.2 2. The Vanishing Gradient Problem

Why do we rarely use vanilla RNNs? Consider the gradient of the loss at time T with respect to the input at time 1. By the Chain Rule, this involves multiplying the weight matrix W_{hh} by itself $T-1$ times.

$$\frac{\partial L_T}{\partial h_1} \propto \prod_{k=2}^{T} W_{hh}^T \cdot \mathrm{diag}(\tanh')$$

If the largest eigenvalue of W_{hh} is < 1, the gradient decays exponentially to zero ("Vanishes"). The network "forgets" inputs from long ago. If the largest eigenvalue is > 1, the gradient explodes.

11.1.3 3. The Solution: Gating (LSTM & GRU)

To solve this, we introduce **Gating**. Gates are sigmoid-activated neurons that output values in $[0, 1]$, acting as "valves" for information flow.

The LSTM (Long Short-Term Memory) Cell: The LSTM maintains *two* states: the hidden state h_t (short-term) and the **Cell State** C_t (long-term). The update involves four distinct gates/operations:

1. **Forget Gate (f_t):** "What percentage of the old cell state should we keep?"
2. **Input Gate (i_t):** "What percentage of the new candidate information should we add?"
3. **Candidate Cell ($\tilde{C}_t$):** The new information to be added.
4. **Output Gate (o_t):** "What part of the cell state should we reveal as the new hidden state?"

The core magic is the Cell Update:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

Note the **addition** (+). In calculus, the derivative of a sum is 1. This allows gradients to flow backwards through time without decaying, creating an "information superhighway."

11.2 Module 6 Exercises: The Mechanics of Memory

6.1: The "Watchmaker" - RNN from Scratch

You will implement a Vanilla RNN Cell using **only** `nn.Linear`.

1. Define a class `MyRNNCell(nn.Module)`.
2. In `__init__`: define `self.i2h` (Input to Hidden Linear layer) and `self.h2h` (Hidden to Hidden Linear layer).
3. In `forward(x, h_prev)`:
   - Compute $h_{new} = \tanh(\text{self.i2h}(x) + \text{self.h2h}(h\_prev))$.
   - Return h_{new}.
4. **Test:** Create a random input sequence of shape $(10, 1, 5)$ (SeqLen, Batch, Feats). Iterate through the sequence using a Python `for` loop, updating the hidden state at each step.

6.2: The "Engineer" - LSTM from Scratch

This is a rite of passage. You must implement the LSTM equations manually.
Formulas:

$$\begin{pmatrix} i \\ f \\ g \\ o \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \tanh \\ \sigma \end{pmatrix} (W \cdot [h_{t-1}, x_t] + b)$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Task:

1. Create `MyLSTMCell`.
2. Use a single `nn.Linear` that maps $(input\_dim + hidden\_dim) \to (4 \times hidden\_dim)$. This is a common optimization (calculating all gates at once).
3. Slice the output of this linear layer into 4 chunks (i, f, g, o).
4. Apply the correct activations (Sigmoid for gates, Tanh for candidate).
5. Implement the cell state update and hidden state update.
6. Verify your outputs against `nn.LSTMCell` with the same weights (you will need to manually copy weights to verify).

6.3: The "Analyst" - Time Series Prediction

Now use the native `nn.LSTM`. **Data:** Generate a synthetic signal: $y = \sin(t) + \sin(0.5t) + \text{noise}$. Length 1000. **Task:** Many-to-One prediction.

1. Create a dataset where Input is a window of 50 steps, and Target is step 51.
2. **Model:** `nn.LSTM(input_size=1, hidden_size=64, num_layers=2, batch_first=True)`. Followed by `nn.Linear(64, 1)`.
3. **Forward:** Pass the sequence. Take the **last** hidden state (output of the last time step). Feed it to the Linear layer.
4. Train with MSE Loss.
5. Plot the True vs Predicted future.

12 Module 7: The Transformer and Attention

12.1 Theory: Attention is All You Need

RNNs suffer from a sequential bottleneck. To process token 100, you must wait for token 99. This precludes parallelization. Furthermore, "memory" is compressed into a single vector.

The Transformer proposes a radical idea: **Self-Attention**. Instead of remembering a summary, let every element in the sequence look at every other element and decide what is important.

12.1.1 1. The Query-Key-Value Mechanism

This is a database retrieval concept made differentiable. For every vector x_i in our sequence, we apply three linear transformations to get three new vectors:

- **Query (q_i):** What I am looking for?
- **Key (k_i):** What do I contain? (My label).
- **Value (v_i):** What is my actual information content?

The Attention Score: We compute the similarity between Query i and Key j using a dot product.

$$\text{score}_{ij} = q_i \cdot k_j$$

We scale this by $\frac{1}{\sqrt{d_k}}$ (to stabilize gradients) and apply Softmax. This gives us a probability distribution: "How much attention should i pay to j?"

$$\alpha_{ij} = \text{softmax}_j \left(\frac{q_i k_j^T}{\sqrt{d_k}} \right)$$

Finally, the output for position i is the weighted sum of all Values:

$$z_i = \sum_j \alpha_{ij} v_j$$

12.1.2 2. Multi-Head Attention

One attention head can only focus on one type of relationship (e.g., "previous word"). We want to capture multiple relationships simultaneously (e.g., "previous word", "subject of sentence", "prepositional object"). We run H attention mechanisms in parallel, each with its own learnable W^Q, W^K, W^V matrices. We concatenate their outputs.

12.1.3 3. Positional Encoding

The attention mechanism is set-invariant. It has no notion of order. $A \to B$ looks the same as $B \to A$. To fix this, we **add** a positional vector to the input embeddings. In scientific computing, this is often a vector of sin and cos frequencies, allowing the network to learn periodicity.

12.2 Module 7 Exercises: Mastering the Transformer

7.1: The "Mathematician" - Attention by Hand

Goal: Implement Scaled Dot-Product Attention using only matrix multiplication (`torch.matmul`).

1. Inputs: Q, K, V. All shape (B, H, L, D) (Batch, Heads, SeqLen, HeadDim).
2. Compute $S = QK^T$. Warning: You need to transpose the last two dimensions of K. Use `K.transpose(-2, -1)`.
3. Result shape should be (B, H, L, L). This is the **Attention Matrix**.
4. Scale by $1/\sqrt{D}$.
5. Apply Mask (optional): Set elements where you want "no attention" to $-\infty$.
6. Apply `F.softmax(dim=-1)`.
7. Multiply by V: $Out = \text{Attention} \cdot V$.
8. Return *Out* and the Attention Matrix (for visualization).

7.2: The "Architect" - Transformer Block from Scratch

You will build a single Encoder Layer. **Structure:**

$$x \rightarrow \text{LayerNorm}(x + \text{Attention}(x)) \rightarrow \text{LayerNorm}(z + \text{FeedForward}(z))$$

Task:

1. Class `MyTransformerBlock(nn.Module)`.
2. Init: `nn.MultiheadAttention`, two `nn.LayerNorm`s, and a small FeedForward network (Linear $\rightarrow$ ReLU $\rightarrow$ Linear).
3. Forward:
   - Save input as `residual`.
   - Run Attention. Add `residual`. Run LayerNorm.
   - Save result as `residual`.
   - Run FeedForward. Add `residual`. Run LayerNorm.
4. **Why Add & Norm?** This is the "Residual Connection". It is crucial for deep networks. It allows gradients to flow through the network without bottlenecking at the layers.

7.3: The "Scientist" - Solving a PDE with Transformers

This is an advanced research problem. **Problem:** Learning the solution operator for the Burgers' Equation (1D fluid flow). **Data:** We treat the discretization of the domain $x \in [0, 1]$ as a sequence of points.

1. **Input:** Initial condition $u_0(x)$ sampled at 64 points. Sequence length = 64.
2. **Target:** Solution $u_T(x)$ at time $T = 1.0$.
3. **Model:**
   - Input Embedding: Map scalar u to vector dim 64.
   - Positional Encoding: Add sine/cos embeddings so the model knows $x = 0.1$ vs $x = 0.9$.
   - Transformer Encoder: Stack 4 layers of your `MyTransformerBlock`.
   - Output Projection: Map vector dim 64 back to scalar u.
4. Train with MSE Loss.
5. **Analysis:** Extract the Attention Matrix from the first layer. Visualize it as a heatmap. You should see "bands" representing the propagation of the wave front. The network "attends" to the upstream location that determines the current location's value!

Final Words on Sequence Modeling

You have now implemented the engines that drive ChatGPT, AlphaFold, and modern weather forecasting models. Remember:

- **RNNs** iterate. They are O(N) in time and O(1) in memory. They struggle with long context.
- **Transformers** parallelize. They are O(1) in time but O(N^2) in memory (due to the attention matrix). They capture global context perfectly.

Choose your weapon based on your scientific constraints.

Part VII

The Grand Examination: Integrative Challenges

A Note on Difficulty

Professor's Note

Warning – The following exercises are not tutorials. They are problems. They are designed to be frustrating. You will likely encounter dimension mismatches, exploding gradients, and silent failures. This is normal. This is research.

13 Challenge Set 1: The "Raw Metal" Operations

Exercise

Grand Challenge I: Logistic Regression from Scratch – We often rely too heavily on `nn.Linear` and `nn.CrossEntropyLoss`. To understand them, you must recreate them.

Task: Train a binary classifier for 2D data (two blobs of points) without using `nn.Module` or `torch.optim`.

1. Generate synthetic data: 100 points for Class 0 (centered at -2, -2) and 100 points for Class 1 (centered at 2, 2).
2. Initialize weights W (shape 2x1) and bias b (shape 1) as tensors with `requires_grad=True`.
3. **The Loop:**
   - Compute logits: $z = XW + b$.
   - Compute probabilities manually using the Sigmoid formula: $\sigma(z) = \frac{1}{1+e^{-z}}$.
   - Compute Binary Cross Entropy Loss manually: $L = -\frac{1}{N}\sum[y\log(\hat{y}) + (1-y)\log(1-\hat{y})]$.
   - Call `L.backward()`.
   - Update parameters: $W = W - \eta \cdot W.grad$. **Hint:** You must wrap this update in `with torch.no_grad():`, otherwise PyTorch will try to track the update step itself in the graph!
   - Zero the gradients manually: `W.grad = None`.
4. Achieve $> 95\%$ accuracy.

Exercise

Grand Challenge II: Custom Autograd Function – Sometimes, you need an operation that PyTorch doesn't support, or you want to override the gradient calculation (e.g., for numerical stability).
Task: Implement a custom activation function $f(x) = x \cdot \sin(x)$.

1. Inherit from `torch.autograd.Function`.
2. Implement the static method `forward(ctx, input)`. Save the input for the backward pass using `ctx.save_for_backward`.
3. Implement the static method `backward(ctx, grad_output)`.
   - Retrieve input.
   - Analytically calculate $f'(x) = \sin(x) + x\cos(x)$.
   - Return `grad_output * f'(x)` (Chain rule).
4. Wrap this function in a standard `nn.Module` and use it to train a small network.
5. Verify your gradient is correct using `torch.autograd.gradcheck`.

14 Challenge Set 2: Scientific Architectures

Exercise

Grand Challenge III: The Convolutional Autoencoder – In scientific computing, we often need to compress complex fields (like fluid velocity) into a small "latent space" or remove noise from sensor data.
Task: Build a Denoising Autoencoder for MNIST.

1. **Data:** Load MNIST. Create a version of the dataset where you add Gaussian noise to every image.
2. **Encoder:** A series of `Conv2d` layers with `stride=2` (to downsample). Compress the 28×28 image into a $3 \times 3 \times 64$ latent vector.
3. **Decoder:** A series of `ConvTranspose2d` layers (sometimes called Deconvolutions) to upsample back to 28×28.
4. **Training:** Input is the *Noisy* image. Target is the *Clean* image. Loss is MSE.
5. **Goal:** The network effectively "learns" to remove noise. Visualise the Input, the Output, and the Original Clean image side-by-side.

Exercise

Grand Challenge IV: Physics-Informed Neural Network (PINN) – This is the holy grail of Deep Learning in Science. We will not train on data; we will train on a *differential equation*.

Problem: Solve the Harmonic Oscillator: $\frac{d^2u}{dt^2} + u = 0$ for $t \in [0, 2\pi]$, with $u(0) = 1, u'(0) = 0$. (True solution: $u(t) = \cos(t)$).

1. **Network:** A simple MLP: Input t (1 neuron) $\rightarrow$ Hidden layers $\rightarrow$ Output u (1 neuron). **Important:** Use `nn.Tanh` activation (it has smooth non-zero 2nd derivatives).
2. **Physics Loss:**
   - Generate random points t in $[0, 2\pi]$. Set `requires_grad=True` on inputs.
   - Compute prediction $u_{pred} = \text{model}(t)$.
   - Compute first derivative $u_t = \text{torch.autograd.grad}(u_{pred}, t, \dots)[0]$.
   - Compute second derivative $u_{tt} = \text{torch.autograd.grad}(u_t, t, \dots)[0]$.
   - Define residual: $R = u_{tt} + u_{pred}$.
   - Loss PDE = Mean(R^2).
3. **Boundary Loss:**
   - Compute $u(0)$ and $u'(0)$ using the network.
   - Loss BC = $(u(0) - 1)^2 + (u'(0) - 0)^2$.
4. **Total Loss:** Loss = Loss PDE + Loss BC.
5. Train the network. Plot the network's prediction against the analytical $\cos(t)$. It should match perfectly without ever seeing the cosine function!

Exercise

Grand Challenge V: The "Vectorex" Capstone – Combine everything.

Task: Predict the future state of a chaotic system (The Lorenz Attractor) using a Transformer.

1. **Data Generation:** Use `scipy.integrate.odeint` to generate a trajectory of the Lorenz system (x, y, z) for 10,000 time steps.
2. **Preprocessing:** Create a dataset of windows. Input: sequence of length 50. Target: sequence of length 50 (shifted by 1 step into the future).
3. **Model:** Implement a Transformer Decoder-only model (like GPT).
   - You will need to implement *Causal Masking* in the attention mechanism (so position t cannot see $t + 1$).
4. **Inference:** Feed the model an initial seed of 50 points. Have it generate the next 500 points auto-regressively (predict one, append to input, predict next).
5. **Viz:** Plot the generated 3D trajectory against the true trajectory. How long does the model stay on track before chaos diverges?

Final Certification

If you complete Grand Challenge IV (The PINN) and Grand Challenge V (Lorenz Transformer), you have my permission to consider yourself proficient in PyTorch for Scientific Computing. You have moved beyond "using" the library to "wielding" it as an instrument of science.