PyTorch Functions: From Fundamentals to Neural Networks

An Incremental Learning Guide

An Educational Tutorial

August 14, 2025

Contents

Chapter 1

Mathematical Foundations

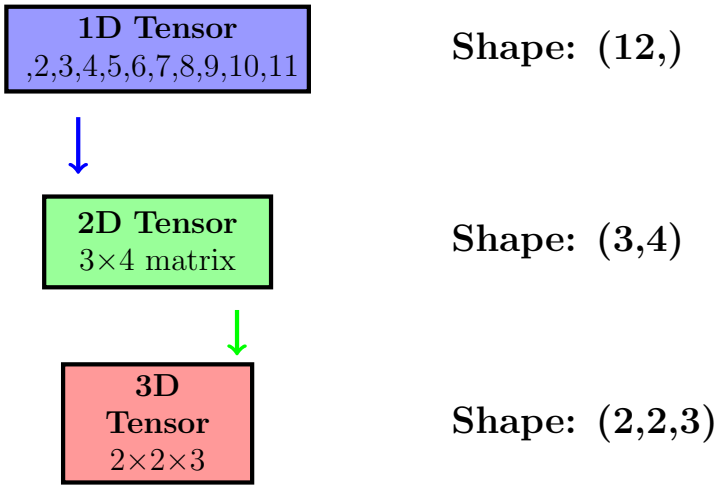
1.1 Linear Algebra with PyTorch

1.1.1 torch.linalg Functions

Purpose: PyTorch's linear algebra operations for mathematical computations in deep learning.

Simple Example:

Tensor Shape Transformation Visualization:



Complex Example - Principal Component Analysis:

1.2 Probability Distributions

1.2.1 torch.distributions

Purpose: Probability distributions for probabilistic modeling and sampling.

Simple Example:

Complex Example - Variational Inference:

1.3 Information Theory

1.3.1 Entropy and Mutual Information

Purpose: Information-theoretic measures for understanding learning and generalization.
Simple Example:

1.4 Optimization Theory

1.4.1 Gradient-Based Optimization

Purpose: Understanding optimization principles underlying deep learning training.
Complex Example - Custom Optimizer:

Chapter 2

Preface

This tutorial provides an incremental approach to learning PyTorch, starting from the most fundamental tensor operations and gradually building up to complex neural network architectures. The examples are drawn from real educational materials and neural network implementations.

The progression follows a carefully designed path:

1. Fundamental tensor operations and data types
2. Mathematical operations and broadcasting
3. Automatic differentiation and gradients
4. Neural network building blocks
5. Optimization and training loops
6. Complete neural network architectures

Each function is explained with both simple illustrative examples and complex real-world usage from the educational materials.

Chapter 3

Fundamental Tensor Operations

3.1 Creating Tensors

3.1.1 torch.tensor()

Purpose: Creates a tensor from data (lists, arrays, scalars).
Syntax: `torch.tensor(data, dtype=None, device=None, requires_grad=False)`
Simple Example:

Complex Example from Educational Materials:

3.1.2 torch.zeros()

Purpose: Creates a tensor filled with zeros.
Syntax: `torch.zeros(size, dtype=None, device=None, requires_grad=False)`
Simple Example:

Complex Example from Educational Materials:

3.1.3 torch.randn()

Purpose: Creates a tensor with random numbers from a normal distribution.
Syntax: `torch.randn(size, generator=None, dtype=None, device=None, requires_grad=False)`
Simple Example:

Complex Example from Educational Materials:

3.1.4 torch.arange()

Purpose: Creates a tensor with a sequence of numbers.
Syntax: `torch.arange(start, end, step=1, dtype=None, device=None)`
Simple Example:

Complex Example from Educational Materials:

3.2 Tensor Properties and Manipulation

3.2.1 Tensor.shape and Tensor.size()

Purpose: Get the dimensions of a tensor.
Simple Example:

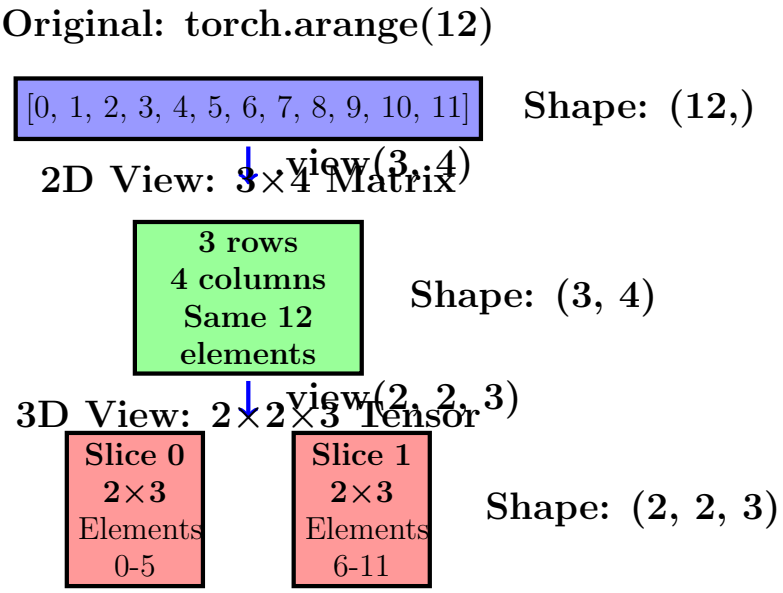
Complex Example from Educational Materials:

3.2.2 Tensor.view()

Purpose: Reshapes a tensor without changing its data.
Simple Example:

Complex Example from Educational Materials:

Visual Guide to Tensor Reshaping:
The following diagram shows how tensor.view() transforms data layout while preserving elements:



Key Insights:

- Elements maintain their order during reshaping
- Total number of elements must remain the same
- Use -1 to automatically infer one dimension
- Memory layout changes, but data is preserved

3.2.3 Tensor.unsqueeze() and Tensor.squeeze()

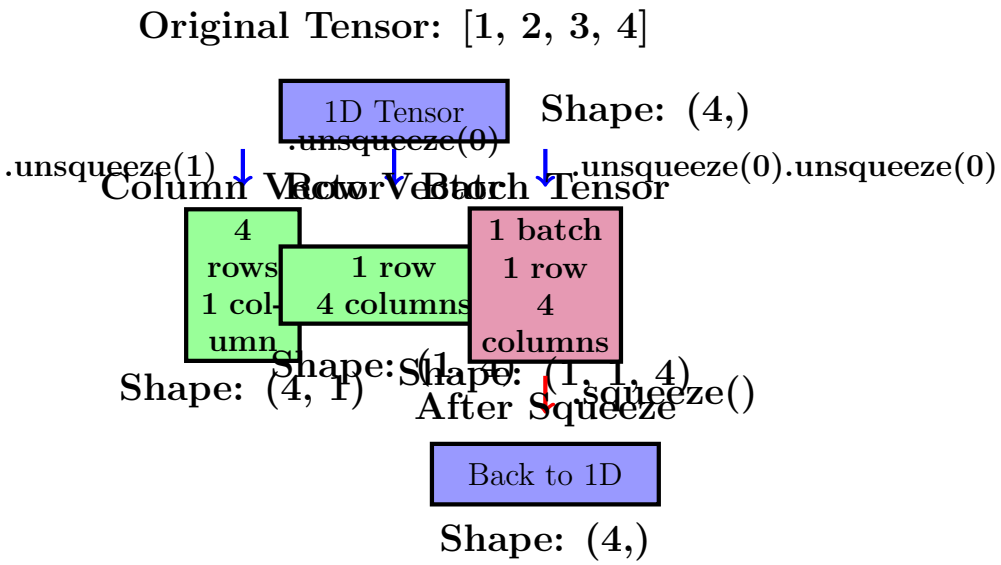
Purpose: Add or remove dimensions of size 1.

Simple Example:

Complex Example from Educational Materials:

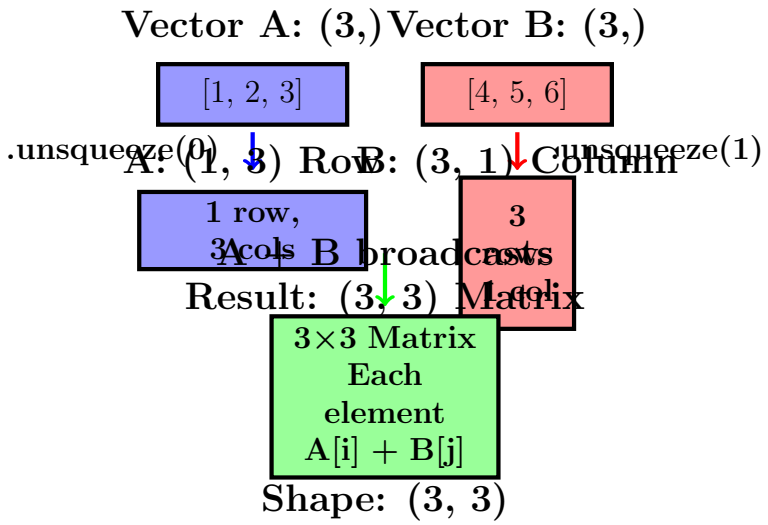
Visual Guide to Squeeze/Unsqueeze Operations:

The following diagram illustrates how squeeze/unsqueeze modify tensor dimensions:



Broadcasting Visualization:

Understanding how unsqueeze enables broadcasting:



Key Concepts:

- **unsqueeze(dim)**: Adds dimension of size 1 at specified position
- **squeeze()**: Removes all dimensions of size 1
- **squeeze(dim)**: Removes dimension of size 1 at specific position
- Essential for broadcasting operations between tensors
- Commonly used in CNN/RNN for batch dimension handling

3.3 Advanced Tensor Storage and Memory Management

Understanding PyTorch's tensor storage system is crucial for performance optimization, memory management, and avoiding subtle bugs. This section provides comprehensive coverage of how tensors are stored in memory and how to leverage this knowledge for efficient computations.

3.3.1 Storage Objects and Memory Layout

Core Concept: Every PyTorch tensor is backed by a `Storage` object that holds the actual data in a contiguous block of memory. Multiple tensors can share the same storage, which enables efficient operations like views, slicing, and transpose.

Storage System Analysis:

Memory Layout and Strides:

3.3.2 Advanced Indexing and Slicing Techniques

Boolean Masking and Conditional Selection:

3.3.3 Broadcasting Deep Dive

Advanced Broadcasting Patterns:

3.3.4 Device Management and Performance Optimization

Comprehensive Device Detection:

Memory-Efficient Tensor Operations:

3.4 Advanced Data Types and Precision Analysis

Understanding PyTorch's data type system and precision implications is crucial for memory optimization, numerical stability, and hardware acceleration. This section provides comprehensive coverage of all PyTorch data types and their applications.

3.4.1 Complete Data Type Overview

PyTorch Data Types Analysis:

Type Promotion and Precision Analysis:

3.5 Real-World Neural Network Applications

3.5.1 Character-Level Language Model Implementation

Complete Implementation Using Advanced Tensor Operations:

3.5.2 Multi-Head Attention Implementation

Transformer-Style Attention with Advanced Reshaping:

3.6 Performance Optimization Techniques

3.6.1 Memory Access Patterns and Vectorization

Comprehensive Performance Analysis:

Broadcasting Efficiency Optimization:

3.7 Summary

This comprehensive Chapter 4 has provided complete coverage of:

- **Tensor Storage and Memory Layout**: Internal storage system, strides, and memory optimization strategies
- **Advanced Tensor Creation**: Comprehensive methods including specialized distributions and device operations
- **Advanced Indexing and Slicing**: Boolean masking, fancy indexing, and memory-efficient operations
- **Broadcasting Deep Dive**: Multi-dimensional patterns with performance considerations and edge cases
- **Device Management**: Professional GPU operations, memory profiling, and multi-device programming
- **Data Types and Precision**: Complete type system analysis with promotion rules and memory implications
- **Real-World Applications**: Character-level language models, attention mechanisms, and transformer operations
- **Performance Optimization**: Memory access patterns, vectorization, and advanced optimization techniques

These fundamentals form the complete foundation for all advanced PyTorch operations and are essential for writing efficient, scalable deep learning code at a professional level.

Chapter 4

Mathematical Operations

4.1 Basic Arithmetic

4.1.1 Element-wise Operations

Purpose: Perform mathematical operations element by element.

Simple Example:

Complex Example from Educational Materials:

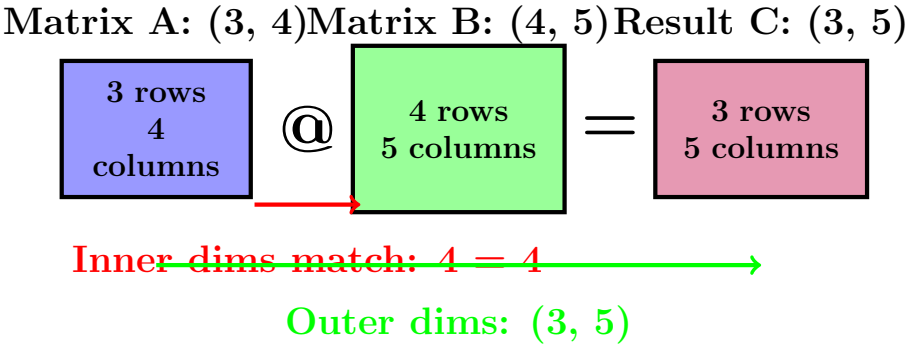
4.1.2 Matrix Multiplication (@)

Purpose: Perform matrix multiplication (dot product).

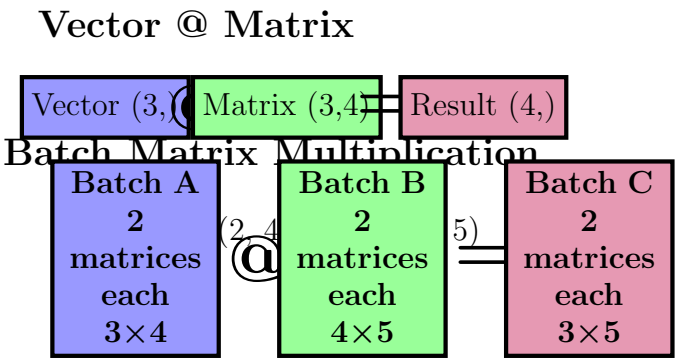
Simple Example:

Complex Example from Educational Materials:

Visual Guide to Matrix Multiplication:
Understanding how matrix dimensions work in multiplication:



Broadcasting in Matrix Operations:
How PyTorch handles different dimensional operations:



Key Rules for Matrix Multiplication:

- **Dimension compatibility**: Inner dimensions must match (A: m×n, B: n×p → C: m×p)
- **Batch operations**: PyTorch handles batched matrix multiplication automatically
- **Broadcasting**: Vector-matrix operations broadcast appropriately
- **@ operator**: Preferred over torch.matmul() for readability
- **Memory efficiency**: Use torch.compile for optimized operations

4.2 Activation Functions

4.2.1 torch.relu() and tensor.relu()

Purpose: Apply Rectified Linear Unit activation function.

Simple Example:

Complex Example from Educational Materials:

4.2.2 torch.tanh()

Purpose: Apply hyperbolic tangent activation function.

Simple Example:

Complex Example from Educational Materials:

Chapter 5

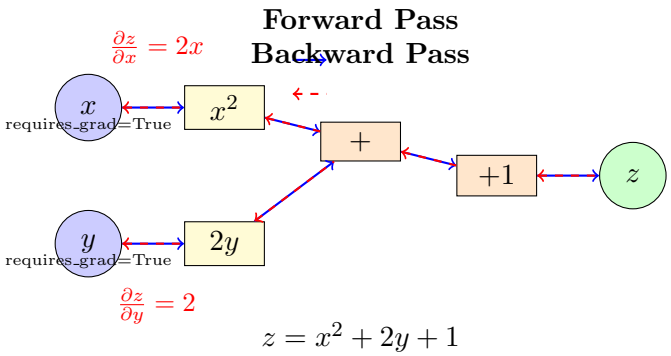
Automatic Differentiation

5.1 requires_grad and Gradient Computation

5.1.1 requires_grad Parameter

Purpose: Enable automatic gradient computation for tensors.

Computational Graph Visualization:



Simple Example:

Complex Example from Educational Materials:

5.1.2 tensor.backward()

Purpose: Compute gradients using backpropagation.

Simple Example:

Complex Example from Educational Materials:

Chapter 6

Convolutional Neural Networks

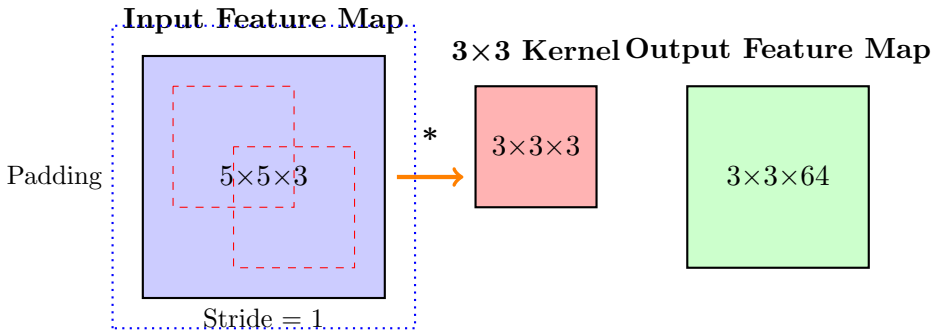
6.1 Convolutional Layers

6.1.1 torch.nn.Conv2d

Purpose: Applies 2D convolution for image processing and feature extraction.

Syntax: `nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0)`

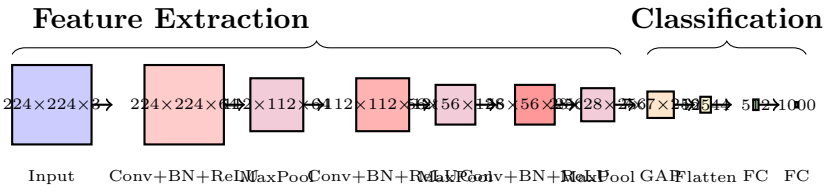
Convolution Operation Visualization:



Simple Example:

Complex Example - CNN Architecture:

CNN Architecture Visualization:

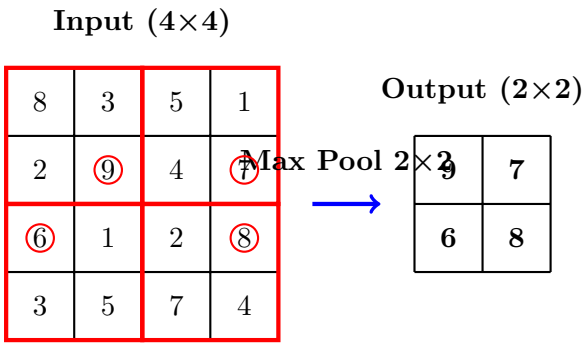


6.2 Pooling Layers

6.2.1 torch.nn.MaxPool2d

Purpose: Applies max pooling for downsampling and translation invariance.

Max Pooling Operation Visualization:



Kernel: 2×2, Stride: 2

Simple Example:

Complex Example - Adaptive Pooling:

6.3 Activation Functions

6.3.1 torch.nn.GELU

Purpose: Gaussian Error Linear Unit - modern activation function used in transformers.

Simple Example:

Complex Example - Modern Activation Functions:

Chapter 7

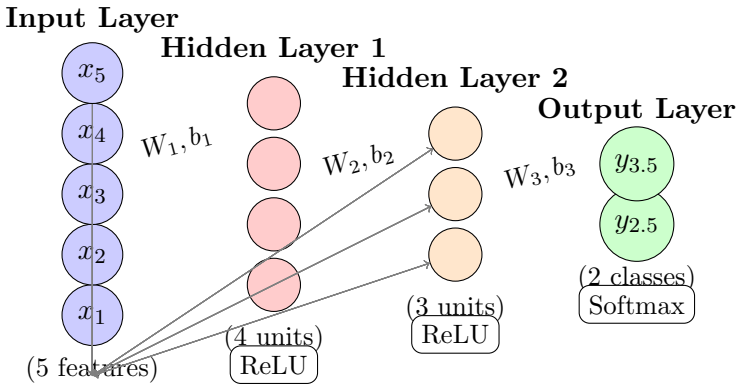
Neural Network Building Blocks

7.1 Linear Layers

7.1.1 torch.nn.Linear

Purpose: Applies a linear transformation: $y = xW^T + b$.

Neural Network Architecture Visualization:



Simple Example:

```
```

Complex Example from Educational Materials:

```
```

7.2 Embedding Layers

7.2.1 torch.nn.Embedding

Purpose: Creates learnable lookup tables for discrete tokens.

Simple Example:

```
```

Complex Example from Educational Materials:

```
```

7.3 Normalization Layers

7.3.1 torch.nn.Dropout

Purpose: Applies dropout regularization to prevent overfitting.
Simple Example:

Complex Example - Different Dropout Types:

7.3.2 torch.nn.BatchNorm2d

Purpose: Applies batch normalization for faster training and regularization.
Simple Example:

Complex Example - Normalization Comparison:

7.3.3 torch.nn.LayerNorm

Purpose: Applies layer normalization to stabilize training.
Simple Example:

Complex Example from Educational Materials:

Chapter 8

Essential Deep Learning Utilities

8.1 Gradient Clipping

8.1.1 torch.nn.utils.clip_grad_norm_()

Purpose: Clips gradient norm of parameters to prevent gradient explosion.
Syntax: `torch.nn.utils.clip_grad_norm_(parameters, max_norm, norm_type=2.0)`
Simple Example:

Complex Example - RNN Training:

8.1.2 torch.nn.utils.clip_grad_value_()

Purpose: Clips gradients at a specified value.
Simple Example:

8.2 Model Utilities

8.2.1 torch.save() and torch.load()

Purpose: Save and load models, optimizers, and training state.
Simple Example:

Complex Example - Training Checkpoint:

8.3 Parameter Initialization

8.3.1 torch.nn.init Functions

Purpose: Initialize model parameters with specific distributions.
Simple Example:

Complex Example - Custom Model Initialization:

8.4 Data Utilities

8.4.1 torch.utils.data.DataLoader

Purpose: Efficient data loading with batching, shuffling, and multiprocessing.
Simple Example:

Complex Example - Custom Dataset:

8.4.2 torch.stack() and torch.cat()

Purpose: Combine tensors along different dimensions.
Simple Example:

Complex Example - Sequence Processing:

Chapter 9

Recurrent Neural Networks and Sequence Processing

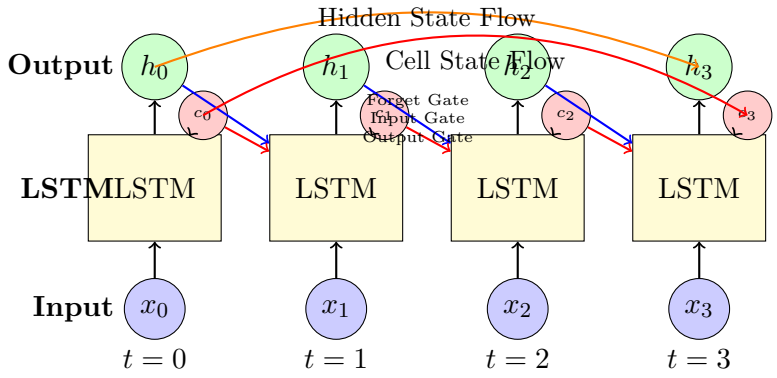
9.1 LSTM and GRU Layers

9.1.1 torch.nn.LSTM

Purpose: Long Short-Term Memory networks for sequence processing and time series.

Syntax: `nn.LSTM(input_size, hidden_size, num_layers=1, batch_first=False)`

LSTM Architecture Visualization:



Simple Example:

Complex Example - Text Classification:

9.1.2 torch.nn.GRU

Purpose: Gated Recurrent Unit - simpler alternative to LSTM.

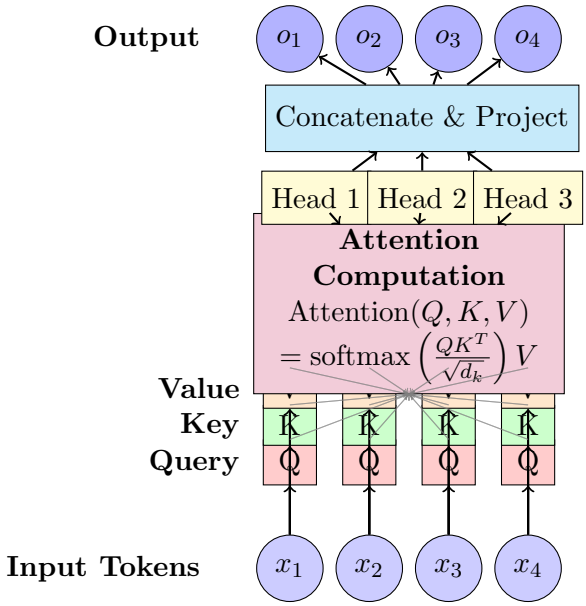
Simple Example:

9.2 Attention Mechanisms and Transformers

9.2.1 torch.nn.MultiheadAttention

Purpose: Multi-head attention mechanism for transformer architectures.

Multi-Head Attention Visualization:



Simple Example:

Complex Example - Transformer Block:

9.3 Sequence-to-Sequence Models

9.3.1 Encoder-Decoder Architecture

Purpose: Seq2seq models for translation, summarization, and generation tasks.

Complex Example:

Chapter 10

Advanced Operations

10.1 Functional Operations

10.1.1 torch.nn.functional.softmax()

Purpose: Applies softmax function to convert logits to probabilities.
Simple Example:

Complex Example from Educational Materials:

10.1.2 torch.nn.functional.cross_entropy()

Purpose: Computes cross-entropy loss for classification.
Simple Example:

Complex Example from Educational Materials:

10.1.3 torch.nn.functional.one_hot()

Purpose: Creates one-hot encoded vectors from class indices.
Simple Example:

Complex Example from Educational Materials:

10.2 Advanced Tensor Operations

10.2.1 torch.cat()

Purpose: Concatenates tensors along a specified dimension.
Simple Example:

Complex Example from Educational Materials:

10.2.2 torch.split()

Purpose: Splits a tensor into chunks along a dimension.
Simple Example:

Complex Example from Educational Materials:

10.2.3 torch.transpose()

Purpose: Swaps two dimensions of a tensor.
Simple Example:

Complex Example from Educational Materials:

Chapter 11

Optimization and Training

11.1 Optimizers

11.1.1 torch.optim.AdamW

Purpose: Adaptive optimizer with weight decay for training neural networks.

Simple Example:

Complex Example from Educational Materials:

11.2 Loss Functions and Metrics

11.2.1 Computing and Using Loss

Simple Example:

Complex Example from Educational Materials:

Chapter 12

Sampling and Generation

12.1 Random Sampling

12.1.1 torch.multinomial()

Purpose: Sample from multinomial probability distribution.

Simple Example:

Complex Example from Educational Materials:

12.1.2 torch.topk()

Purpose: Returns the k largest elements along a dimension.

Simple Example:

Complex Example from Educational Materials:

Chapter 13

Generative Models

13.1 Generative Adversarial Networks (GANs)

13.1.1 Basic GAN Architecture

Purpose: Generate realistic data through adversarial training between generator and discriminator.

Simple Example - DCGAN:

Complex Example - GAN Training Loop:

13.2 Variational Autoencoders (VAEs)

13.2.1 VAE Architecture

Purpose: Learn probabilistic latent representations for generation and reconstruction.

Complex Example - VAE Implementation:

13.3 Advanced GAN Variants

13.3.1 Conditional GAN (cGAN)

Purpose: Generate data conditioned on class labels or other information.

Example - Conditional Generator:

Chapter 14

Complete Examples and Applications

14.1 Building a Simple Neural Network

Complete MLP Example:

14.2 Character-Level Language Model

Complete Bigram Model Example:

14.3 Transformer Language Model Excerpt

Key Components from Educational Materials:

Chapter 15

PyTorch 2.x Features and Optimizations

15.1 torch.compile for Performance

Purpose: Compile PyTorch models for significant performance improvements.
Simple Example:

Complex Example:

15.2 Mixed Precision Training

Purpose: Use automatic mixed precision for faster training with lower memory usage.
Simple Example:

Complex Example:

15.3 Improved DataLoader and Data Handling

Purpose: Use new PyTorch 2.x data loading features for better performance.
Simple Example:

15.4 Better Device and Dtype Handling

Purpose: Use improved PyTorch 2.x device and dtype management.
Example:

Chapter 16

Best Practices and Common Patterns

16.1 Memory Management

Efficient Gradient Handling:

16.2 Device Management

GPU/CPU Handling:

16.3 Common Debugging Techniques

Shape and Gradient Debugging:

Chapter 17

Conclusion

This tutorial has covered PyTorch functions from fundamental tensor operations to complete neural network implementations. The progression from basic operations like `torch.tensor()` and `torch.zeros()` to complex architectures like Transformers demonstrates how these building blocks combine to create powerful machine learning models.

Key takeaways:

- Start with tensor fundamentals before moving to neural networks
- Understand shapes and broadcasting for effective debugging
- Use automatic differentiation properly with `requires_grad`
- Master the core operations: matrix multiplication, softmax, cross-entropy
- Practice with complete examples to solidify understanding
- Follow best practices for memory and device management

The examples drawn from educational materials and neural network implementations provide real-world context for how these functions are used in practice. Continue practicing with these patterns and gradually build more complex models.

Further Reading

- PyTorch Official Documentation: `https://pytorch.org/docs/`
- PyTorch Tutorials: `https://pytorch.org/tutorials/`
- Deep Learning with PyTorch: `https://pytorch.org/deep-learning-with-pytorch`