Tensor-based Computing

Introduction

Tensor-based computation is at the core of modern deep learning frameworks, enabling efficient manipulation of multi-dimensional data structures. A tensor generalizes scalars, vectors, and matrices to higher dimensions, serving as the fundamental unit for representing input data, model parameters, and intermediate activations in neural networks.
Deep learning frameworks optimize tensor operations usingg automatic differentiation, which simplifies gradient computation for backpropagation, and hardware acceleration through GPUs and TPUs. These frameworks utilize specialized linear algebra libraries such as cuBLAS, cuDNN, and MKL to ensure efficient execution of matrix multiplications, convolutions, and other tensor-based computations.

Common Tensor Operations

PyTorch provides a comprehensive set of tensor operations, enabling efficient numerical computations on CPUs and GPUs. Tensors in PyTorch are multi-dimensional arrays similar to NumPy arrays but with the added ability to utilize hardware acceleration (GPUs). PyTorch tensors provide a flexible and powerful way to perform mathematical operations efficiently.

Creating Tensors

PyTorch provides several ways to create tensors, including direct initialization from Python lists, random initialization, and predefined structures like identity matrices or sequences.
Some common ways to create tensors include:

- **From Lists/NumPy Arrays**: Using `torch.tensor()`, tensors can be created from Python lists or NumPy arrays.
- **Random Initialization**: `torch.rand()`, `torch.randn()`, and `torch.randint()` generate tensors with random values.
- **Predefined Structures**: `torch.zeros()`, `torch.ones()`, and `torch.eye()` create common tensor patterns.
- **Arange and Linspace**: `torch.arange()` and `torch.linspace()` generate evenly spaced values.

- **Data Type and Device Control**: Tensors can have specific data types (e.g., `torch.float32`) and be assigned to a CPU or GPU.

```
# Creating a tensor from a list
tensor_a = torch.tensor([1, 2, 3, 4])

# Creating a tensor of zeros
tensor_b = torch.zeros(3, 4)

# Creating a tensor of ones
tensor_c = torch.ones(2, 2)

# Creating a tensor with random values
tensor_d = torch.rand(4, 4)

# Creating an identity matrix
tensor_e = torch.eye(3)

# Creating a tensor with a specific data type
tensor_f = torch.tensor([1.5, 2.3, 3.1], dtype=torch.float64)
```

Tensor Attributes

PyTorch tensors have several important attributes that provide information about their structure, data type, and computational properties. Understanding these attributes helps optimize operations and manage memory efficiently.

- **`shape` (or `size()`)**: Returns the dimensions of the tensor.
- **`dtype`**: Specifies the data type (e.g., `torch.float32`, `torch.int64`).
- **`device`**: Indicates whether the tensor is stored on the CPU or GPU.
- **`requires_grad`**: Shows whether the tensor tracks gradients for automatic differentiation.
- **`numel()`**: Returns the total number of elements in the tensor.

```
tensor = torch.rand(3, 4)
print(tensor.shape)     # Shape of the tensor
print(tensor.dtype)     # Data type
print(tensor.device)    # Device (CPU or GPU)
```

```
print(tensor.requires_grad)  # Whether gradients are required
```

Tensor Indexing and Slicing

Tensor slicing and indexing in PyTorch allow for efficient access and manipulation of specific elements, rows, columns, or subarrays within a tensor. Similar to NumPy, PyTorch supports various indexing techniques, including basic indexing, slicing, boolean indexing, and advanced indexing.

- **Basic Indexing**: Access individual elements using square brackets (`tensor[i]`).
- **Slicing**: Extract sub-tensors using the `start:stop:step` notation.
- **Ellipsis (`...`)**: Used for selecting all elements along unspecified dimensions.
- **Boolean Masking**: Select elements based on conditions.
- **Advanced Indexing**: Use index lists or tensors for selecting arbitrary elements.

```
tensor = torch.arange(10)  # Creates a tensor [0, 1, ..., 9]
# Indexing
print(tensor[2])   # Get element at index 2
print(tensor[-1])  # Get last element

# Slicing
print(tensor[2:6])  # Get elements from index 2 to 5
print(tensor[:5])   # Get first five elements
print(tensor[::2])  # Get every second element

# Boolean indexing
tensor = torch.tensor([1, 2, 3, 4, 5])
mask = tensor > 3
print(tensor[mask])  # Returns tensor([4, 5])
```

Arithmetic Operations

PyTorch provides a wide range of arithmetic operations for element-wise computations on tensors. These operations include addition, subtraction, multiplication, division, exponentiation, and modulus, all of which can be performed using operators (`+`, `-`, `*`, `/`, `**`, `%`) or equivalent PyTorch functions (`torch.add()`, `torch.mul()`, etc.).

- **Addition**: `tensor1 + tensor2` or `torch.add(tensor1, tensor2)`

- **Subtraction**: `tensor1 - tensor2`
- **Multiplication**: `tensor1 * tensor2` (element-wise)
- **Matrix Multiplication**: `tensor1 @ tensor2` or `torch.mm(tensor1, tensor2)`
- **Division**: `tensor1 / tensor2`
- **Exponentiation**: `tensor1 ** 2`
- **Modulus**: `tensor1 % 2`

```
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])

# Addition
print(a + b)      # Element-wise addition
print(torch.add(a, b))

# Subtraction
print(a - b)      # Element-wise subtraction

# Multiplication
print(a * b)      # Element-wise multiplication
print(torch.mul(a, b))

# Division
print(a / b)      # Element-wise division

# Power
print(a ** 2)     # Element-wise exponentiation

# Modulus
print(a % 2)      # Element-wise modulus
```

Reduction Operations

Reduction operations in PyTorch compute summary statistics from tensors by aggregating values across one or more dimensions. These operations are essential for numerical analysis and optimization.

- **Summation**: `torch.sum(tensor, dim=None)` – Computes the sum of all elements or along a specific dimension.

- **Mean & Standard Deviation**: `torch.mean(tensor)`, `torch.std(tensor)` – Compute the average and standard deviation.
- **Min & Max**: `torch.min(tensor)`, `torch.max(tensor)` – Find the smallest or largest value.
- **Product**: `torch.prod(tensor)` – Computes the product of all elements.
- **Argmin & Argmax**: `torch.argmin(tensor)`, `torch.argmax(tensor)` – Return the index of the min/max value.

```
tensor = torch.tensor([[1, 2], [3, 4]])

print(torch.sum(tensor))   # Sum of all elements
print(torch.mean(tensor.float()))  # Mean value
print(torch.std(tensor.float()))   # Standard deviation
print(torch.min(tensor))    # Minimum value
print(torch.max(tensor))    # Maximum value
print(torch.prod(tensor))   # Product of elements

# Reduction Along a Dimension

tensor = torch.tensor([[1, 2], [3, 4]])

print(torch.sum(tensor, dim=0))  # Sum along rows
print(torch.sum(tensor, dim=1))  # Sum along columns
```

Linear Algebra Operations

PyTorch provides a comprehensive set of linear algebra operations that allow for efficient manipulation of matrices and vectors. These operations are essential for solving systems of equations, performing transformations, and handling geometric computations.
Key linear algebra operations include:

- **Matrix Multiplication**: `torch.mm(A, B)` or `A @ B` – Performs standard matrix multiplication.
- **Dot Product**: `torch.dot(a, b)` – Computes the dot product of two 1D tensors.
- **Transpose**: `A.T` or `torch.transpose(A, 0, 1)` – Swaps rows and columns of a matrix.
- **Determinant**: `torch.det(A)` – Computes the determinant of a square matrix.

- **Inverse**: `torch.inverse(A)` – Finds the inverse of a square matrix.
- **Eigenvalues & Eigenvectors**: `torch.eig(A, eigenvectors=True)` – Computes the eigenvalues and eigenvectors.
- **Singular Value Decomposition (SVD)**: `torch.svd(A)` – Factorizes a matrix into singular vectors and values.

```
A = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
B = torch.tensor([[5, 6], [7, 8]], dtype=torch.float32)

# Matrix multiplication
print(torch.mm(A, B))
print(A @ B)  # Equivalent to mm

# Transpose
print(A.T)

# Determinant
print(torch.det(A))

# Inverse
print(torch.inverse(A))

# Eigenvalues and Eigenvectors
eigenvalues, eigenvectors = torch.eig(A, eigenvectors=True)
print(eigenvalues)
print(eigenvectors)

# Singular Value Decomposition (SVD)
U, S, V = torch.svd(A)
print(U, S, V)
```

Tensor Reshaping and Manipulation

Tensor reshaping and manipulation in PyTorch allow for efficient transformation of tensor structures without altering their underlying data. These operations are used for data preprocessing, feature engineering, and optimizing computations.

Key tensor reshaping and manipulation operations:

- **Reshaping**: `tensor.view(new_shape)` or `tensor.reshape(new_shape)` – Changes the shape of a tensor.

- **Flattening**: `tensor.view(-1)` – Converts a multi-dimensional tensor into a 1D tensor.
- **Transposing**: `tensor.T` or `tensor.permute(dim_order)` – Swaps tensor dimensions.
- **Squeezing & Unsqueezing**: `tensor.squeeze()` removes dimensions of size 1, while `tensor.unsqueeze(dim)` adds a new dimension.
- **Expanding**: `tensor.expand(new_shape)` – Duplicates tensor elements along specified dimensions.

```
tensor = torch.arange(12)

# Reshaping
reshaped = tensor.view(3, 4)
print(reshaped)

# Flattening
flattened = reshaped.view(-1)
print(flattened)

# Permute dimensions
tensor = torch.rand(2, 3, 4)
permuted = tensor.permute(2, 0, 1)
print(permuted.shape)

# Expanding
tensor = torch.tensor([[1], [2], [3]])
expanded = tensor.expand(3, 4)
print(expanded)

# Squeezing (removing dimensions of size 1)
tensor = torch.tensor([[[5]]])
squeezed = tensor.squeeze()
print(squeezed)

# Unsqueezing (adding dimensions)
unsqueezed = squeezed.unsqueeze(0)
print(unsqueezed.shape)
```

Tensor Concatenation and Splitting

Tensor concatenation and splitting in PyTorch allow for combining multiple tensors or dividing a tensor into smaller parts, which is useful for data processing, batching, and parallel computation.

Key operations:

- **Concatenation**: `torch.cat((tensor1, tensor2), dim=axis)` – Joins tensors along a specified dimension.
- **Stacking**: `torch.stack((tensor1, tensor2), dim=axis)` – Creates a new dimension and stacks tensors along it.
- **Chunking**: `torch.chunk(tensor, chunks, dim=axis)` – Splits a tensor into a specified number of chunks.
- **Splitting**: `torch.split(tensor, split_size, dim=axis)` – Splits a tensor into parts of a given size.

```
a = torch.rand(2, 3)
b = torch.rand(2, 3)

# Concatenation
print(torch.cat((a, b), dim=0))  # Along rows
print(torch.cat((a, b), dim=1))  # Along columns

# Splitting
splitted = torch.chunk(a, chunks=2, dim=1)
print(splitted)

# Stack tensors
stacked = torch.stack((a, b), dim=0)
print(stacked.shape)
```

Copying and Cloning Tensors

Copying and cloning tensors in PyTorch ensures efficient memory management and prevents unintentional modifications to data when working with tensor operations.

Key methods for copying tensors:

- **Shallow Copy (Assignment)**: `tensor_b = tensor_a` – Does not create a new tensor; `tensor_b` shares memory with `tensor_a`.
- **Cloning**: `tensor.clone()` – Creates a new tensor with the same data but detached from the original memory.

- **Detaching**: `tensor.detach()` – Creates a new tensor that shares the same memory but is not tracked for gradient computation.
- **Converting to NumPy**: `tensor.numpy()` – Returns a NumPy array sharing the same memory (if on CPU).
- **Copying to a Different Device**: `tensor.to(device)` – Moves a tensor between CPU and GPU.

```
tensor = torch.tensor([1, 2, 3], dtype=torch.float32)

# Cloning (creates a new tensor with the same data)
clone_tensor = tensor.clone()

# Detaching (useful when working with computation graphs)
detached_tensor = tensor.detach()

# Convert tensor to NumPy
numpy_array = tensor.numpy()

# Convert NumPy array back to tensor
tensor_from_numpy = torch.from_numpy(numpy_array)
```

Tensor Broadcasting

Tensor broadcasting in PyTorch allows operations between tensors of different shapes by automatically expanding the smaller tensor along missing dimensions. This eliminates the need for explicit reshaping, making computations more efficient and intuitive.
When performing element-wise operations (e.g., addition, multiplication) between tensors of different shapes, PyTorch follows these broadcasting rules:

1. **Align Right**: The dimensions of the tensors are aligned from the right.
2. **Expand Dimensions**: If a dimension mismatch occurs, the smaller tensor is **expanded** to match the larger tensor's shape.
3. **Size Compatibility**: A dimension can either be the same for both tensors or one of them must be `1` (which will be expanded).

```
# Broadcasting beween scalar and tensor:
tensor = torch.tensor([[1, 2, 3], [4, 5, 6]])
scalar = 2
result = tensor * scalar  # The scalar is broadcasted across all elements
```

```
# Broadcasting between vector and matrix:
matrix = torch.tensor([[1, 2, 3], [4, 5, 6]])
vector = torch.tensor([1, 2, 3])  # Shape: (3,)
result = matrix + vector  # The vector is broadcasted along rows

# Higher-dimensional Broadcasting
A = torch.rand(2, 1, 3)  # Shape: (2,1,3)
B = torch.rand(1, 4, 3)  # Shape: (1,4,3)
result = A + B  # A is expanded to (2,4,3)
```

Broadcasting Errors

Broadcasting only works if the dimensions are either the same or one of them is 1. Otherwise, PyTorch will raise a shape mismatch error.

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

# This will fail because (2,3) and (3,2) are incompatible
result = A + B  # RuntimeError
```

Moving Tensors to GPU

PyTorch allows seamless acceleration of computations by moving tensors to a **GPU** (Graphics Processing Unit). This may result in significantly faster numerical operations compared to CPU execution.

Key operations for moving tensors between devices:

- **Check if GPU is available**: `torch.cuda.is_available()`
- **Move tensor to GPU**: `tensor.to("cuda")` or `tensor.cuda()`
- **Move tensor back to CPU**: `tensor.to("cpu")`
- **Specify device during tensor creation**: `torch.tensor([1, 2, 3], device="cuda")`

```
# Check if CUDA is available
device = "cuda" if torch.cuda.is_available() else "cpu"

# Move tensor to GPU
```

```
tensor = torch.rand(3, 3)
tensor = tensor.to(device)

# Move tensor back to CPU
tensor = tensor.cpu()
```

Tasks

Implement the following using PyTorch tensors. Use tensor operations (no for/while loops):

1.1. Create a tensor of shape (3, 4) filled with random values between 0 and 5

1.2. Create a 5x5 identity matrix

1.2. Create a 1D tensor of evenly-spaced values between 0 and 10 with a step of 0.5

1.3. Given the following tensor:

```
t = torch.arange(1, 17).reshape(4, 4)
```

- extract the first row
- extract the first column
- extract the submatrix containing rows 1 and 2 and columns 2 and 3
- modify the diagonal elements to be 0
- reverse the order of the rows

1.4. Given the following tensor:

```
t = torch.arange(24).reshape(2, 3, 4)
```

- reshape it into a tensor of shape (4, 6)
- flatten it into a 1D tensor
- transpose the first and last dimensions

1.5. Create a 5x5 tensor of consecutive values

- compute the sum of all elements
- compute the mean of each row
- compute the maximum value of each column
- compute the indices of the minimum values of each row

1.6. Create two tensors of shapes (4, 1) and (1, 5) and computye their sum using broadcasting

1.7. Generate a tensor of shape (5, 5) where the values are drawn from a normal distribution of mean 0 and standard deviation 1

2. Create a random 5x5 matrix A

- Extract the second and fourth rows as a new matrix B
- Create a vector V of shape (5, 1) with elements 1, -1, 2, -2, 3
- Compute C = A x B^T (matrix multiplication)
- Normalize C so that its mean is 0 and standard deviation is 1
- Add V to each column of C using broadcasting
- Make A symmetrical by multiplying it with its tranpose
- Add 5 to its main diagonal
- Solve the system of equations Ax = V

3. Generate a random matrix A of size NxN and a random vector b of size N

3.1. Perform a row normalized exponential decay transformation. Transform each row a_i of A as follows:

$$\mathbf{a}_i^{'} = \frac{e^{-a_i}}{\sum e^{-a_i}}$$

Store the transformed matrix as $A^{'}$.

3.2. Constuct the Weighted Graph Laplacian Matrix

- Define the weight matrix W as:

$$W = A'A'^T$$

- Compute the diagonal degree matrix D, where

$$D_{ii} = \sum_j W_{ij}$$

- Compute the unnormalized Laplacian matrix L:

$$L = D - W$$

3.3. Solve the following system of linear equations:

$$(L + \lambda I)x = b$$

Where $\lambda > 0$ is a regularization parameter and I is the identity matrix of size N

3.4. Compute a nonlinear aggregation of x: define a transformed output y where:

$$y_i = \tanh\left(\sum_j L_{ij} x_j\right)$$