Part 1 of the Assignment:

Creating and Exploring Tensors - Create one-dimensional tensors in both TensorFlow and PyTorch, each containing at least 5 values of your choice. Print the tensors and their shapes. Leveraged the LLM to improve the display.

Answer

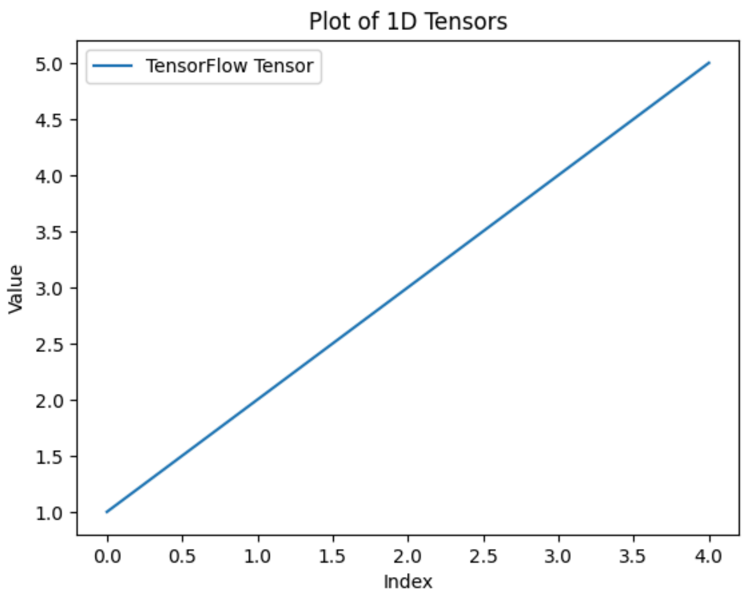
```
# Creating 1D Tensors for TensorFlow and PyTorch
import torch
import tensorflow as tf
import numpy as np
import pandas as pd


# Values 2,4,6,8,10 used for Pytorch tensor
torch_1d = torch.tensor([2.0, 4.0, 6.0, 8.0, 10.0])
torch_1d_shape = torch_1d.shape

print("PyTorch 1D Tensor:----------> ")
print(torch_1d)
print("Shape:------------>", torch_1d_shape)

# Values 1,2,3,4,5 used for Tensorflow tensor
tf_1d = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0])
tf_1d_shape = tf_1d.shape
print("\nTensorFlow 1D Tensor:--------->")
print(tf_1d)
print("Shape:---------->", tf_1d_shape)
```

```
PyTorch 1D Tensor:---------->
tensor([ 2.,  4.,  6.,  8., 10.])
Shape:------------> torch.Size([5])

TensorFlow 1D Tensor:--------->
tf.Tensor([1. 2. 3. 4. 5.], shape=(5,), dtype=float32)
Shape:----------> (5,)
```

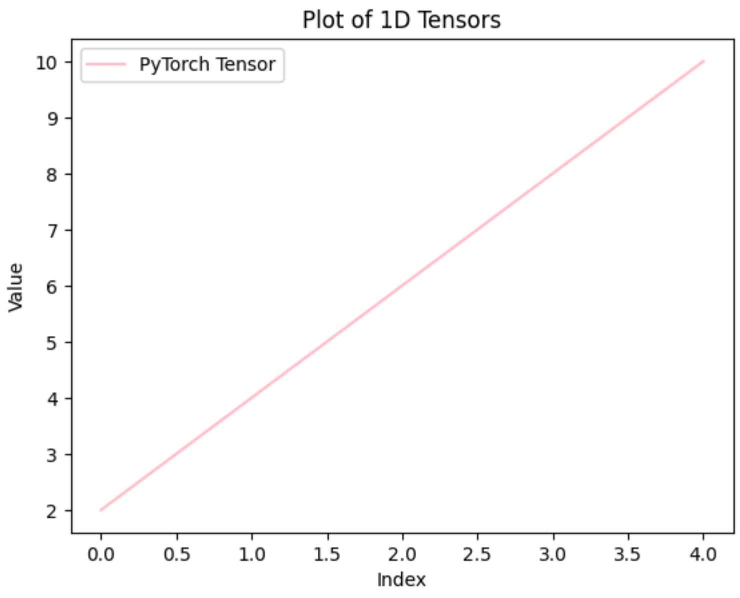
```
#Print the Dataset


pd.DataFrame(torch_1d)
```

	0
0	2.0
1	4.0
2	6.0
3	8.0
4	10.0

```
# plot the graph of the tensors - tensorflow

import matplotlib.pyplot as plt

# Convert tensors to NumPy arrays for plotting
torch_1d_np = torch_1d.numpy()
tf_1d_np = tf_1d.numpy()

# Create a figure and axes for the plot
fig, ax = plt.subplots()

# Plot the data

ax.plot(tf_1d_np, label='TensorFlow Tensor')

# Add labels and title
ax.set_xlabel("Index")
ax.set_ylabel("Value")
ax.set_title("Plot of 1D Tensors")

# Add a legend
ax.legend()

# Show the plot
plt.show()
```



```
# pytorch tensor plot

import matplotlib.pyplot as plt
# Convert tensors to NumPy arrays for plotting
torch_1d_np = torch_1d.numpy()

# Create a figure and axes for the plot
fig, ax = plt.subplots()

# Plot the data

ax.plot(torch_1d_np, color='pink', label='PyTorch Tensor')


# Add labels and title
ax.set_xlabel("Index")
ax.set_ylabel("Value")
ax.set_title("Plot of 1D Tensors")

# Add a legend
ax.legend()

# Show the plot
plt.show()
```



2.Create two-dimensional tensors (matrices) in both frameworks with dimensions of at least 3×4. Print the tensors and their shapes.

Answer

```
# 2D Tensors created for pytorch and tensorflow
torch_2d = torch.tensor([[1, 2, 3, 4],
                         [5, 6, 7, 8],
                         [9, 10, 11, 12]])
torch_2d_shape = torch_2d.shape



print("PyTorch 2D Tensor:")
print(torch_2d)
print("Shape:", torch_2d_shape)
```

```
PyTorch 2D Tensor:
tensor([[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12]])
Shape: torch.Size([3, 4])
```

```
tf_2d = tf.constant([[1, 2, 3, 4],
                     [5, 6, 7, 8],
                     [9, 10, 11, 12]])

tf_2d_shape = tf_2d.shape
print("\nTensorFlow 2D Tensor:")
print(tf_2d)
print("Shape:", tf_2d_shape)
```

```

TensorFlow 2D Tensor:
tf.Tensor(
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]], shape=(3, 4), dtype=int32)
Shape: (3, 4)
```

Print the Dataframe

```
pd.DataFrame([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

3.Create three-dimensional tensors in both frameworks with dimensions of your choice. Print the tensors and their shapes, and demonstrate indexing by accessing specific elements.

Answer

```
# 3D Tensors
torch_3d = torch.tensor([
    [[1, 2, 3], [4, 5, 6]],
    [[7, 8, 9], [10, 11, 12]]
])
torch_3d_shape = torch_3d.shape


tf_3d = tf.constant([
    [[10, 20, 30], [40, 50, 60]],
    [[70, 80, 90], [100, 110, 120]]
])
tf_3d_shape = tf_3d.shape

torch_3d_index = torch_3d[1][0][2].item()
tf_3d_index = tf_3d[1, 0, 2].numpy().item()



print("Torch 3D Tensor:")
print(torch_3d)
print("Shape:", torch_3d_shape)
```

```
Torch 3D Tensor:
tensor([[[ 1,  2,  3],
         [ 4,  5,  6]],

        [[ 7,  8,  9],
         [10, 11, 12]]])
Shape: torch.Size([2, 2, 3])
```

```
print("PyTorch 3D Tensor:")
print(tf_3d)
print("Shape:", tf_3d_shape)
```

```
PyTorch 3D Tensor:
tf.Tensor(
[[[ 10  20  30]
  [ 40  50  60]]

 [[ 70  80  90]
  [100 110 120]]], shape=(2, 2, 3), dtype=int32)
Shape: (2, 2, 3)
```

```
pd.DataFrame([
    [[1, 2, 3], [4, 5, 6]],
    [[7, 8, 9], [10, 11, 12]]
])
```

	0	1
0	[1, 2, 3]	[4, 5, 6]
1	[7, 8, 9]	[10, 11, 12]

4. Create a function in both TensorFlow and PyTorch that takes two tensors as input and returns the sum of their elements. Test the function with 1D, 2D, and 3D tensors. This function will demonstrate how to perform element-wise addition between two tensors and then sum all elements of the resulting tensor.

Answer

```
# Function to add tensors and return sum
def fn_torch_add_and_sum(a, b):
    return torch.sum(a + b).item()

# sums
torch_sum_1d = fn_torch_add_and_sum(torch_1d, torch_1d)
torch_sum_2d = fn_torch_add_and_sum(torch_2d, torch_2d)
torch_sum_3d = fn_torch_add_and_sum(torch_3d, torch_3d)

def fn_tf_add_and_sum(a, b):
    return tf.reduce_sum(tf.add(a, b)).numpy()

tf_sum_1d = fn_tf_add_and_sum(tf_1d, tf_1d)
tf_sum_2d = fn_tf_add_and_sum(tf_2d, tf_2d)
tf_sum_3d = fn_tf_add_and_sum(tf_3d, tf_3d)



# Prepare DataFrame for display
df = pd.DataFrame({
    "Tensor Type": ["1D Torch", "1D TensorFlow", "2D Torch", "2D TensorFlow", "3D Torch", "3D TensorFlow"],
    "Shape": [str(torch_1d_shape), str(tf_1d_shape), str(torch_2d_shape), str(tf_2d_shape), str(torch_3d_shape), str(tf_3d_s
    "Example Index [1][0][2]": ["-", "-", "-", "-", torch_3d_index, tf_3d_index],
    "Sum Result": [torch_sum_1d, tf_sum_1d, torch_sum_2d, tf_sum_2d, torch_sum_3d, tf_sum_3d]
})

display(df)
```

	Tensor Type	Shape	Example Index [1][0][2]	Sum Result
0	1D Torch	torch.Size([5])	-	60.0
1	1D TensorFlow	(5,)	-	30.0
2	2D Torch	torch.Size([3, 4])	-	156.0
3	2D TensorFlow	(3, 4)	-	156.0
4	3D Torch	torch.Size([2, 2, 3])	9	156.0
5	3D TensorFlow	(2, 2, 3)	90	1560.0

Next steps: Generate code with df View recommended plots New interactive sheet

Part 2 of the Assignment: The Role of Broadcasting in Tensor Operations

Describe how broadcasting works in tensor operations, providing a simple example to illustrate your explanation.

Answer

```
**Broadcasting** is a mechansim that allows operations between
different shapes to be performed efficiently **without explici
smaller tensor to match the larger one**.

Broadcasting automatically expands the smaller tensor along di
1 or missing dimensions to match the shape of the larger tenso
element-wise operations without unnecessary memory duplication

**Rules of Broadcasting**
Two tensors are "broadcastable" if the following rules hold (a
recursively from the trailing dimension):

**Dimension Compatibility:**

The dimensions of the two tensors are compared from right to l
leading).

For each corresponding dimension pair, one of the following mu

The dimensions are equal, or

One of the dimensions is 1, or

One of the dimensions does not exist (the tensor has fewer dim

**Expansion:**

The smaller tensor is virtually "stretched" (repeated) along t
where its size is 1 or where it is missing a dimension to matc
tensor.
```

Broadcasting is a mechansim that allows operations between tensors of different shapes to be performed efficiently **without explicitly expanding the smaller tensor to match the larger one**.

Broadcasting automatically expands the smaller tensor along dimensions of size 1 or missing dimensions to match the shape of the larger tensor, enabling element-wise operations without unnecessary memory duplication.

Rules of Broadcasting Two tensors are "broadcastable" if the following rules hold (applied recursively from the trailing dimension):

Dimension Compatibility:

The dimensions of the two tensors are compared from right to left (trailing to leading).

For each corresponding dimension pair, one of the following must be true:

The dimensions are equal, or

One of the dimensions is 1, or

One of the dimensions does not exist (the tensor has fewer dimensions).

Expansion:

The smaller tensor is virtually "stretched" (repeated) along the dimensions where its size is 1 or where it is missing a dimension to match the larger tensor.

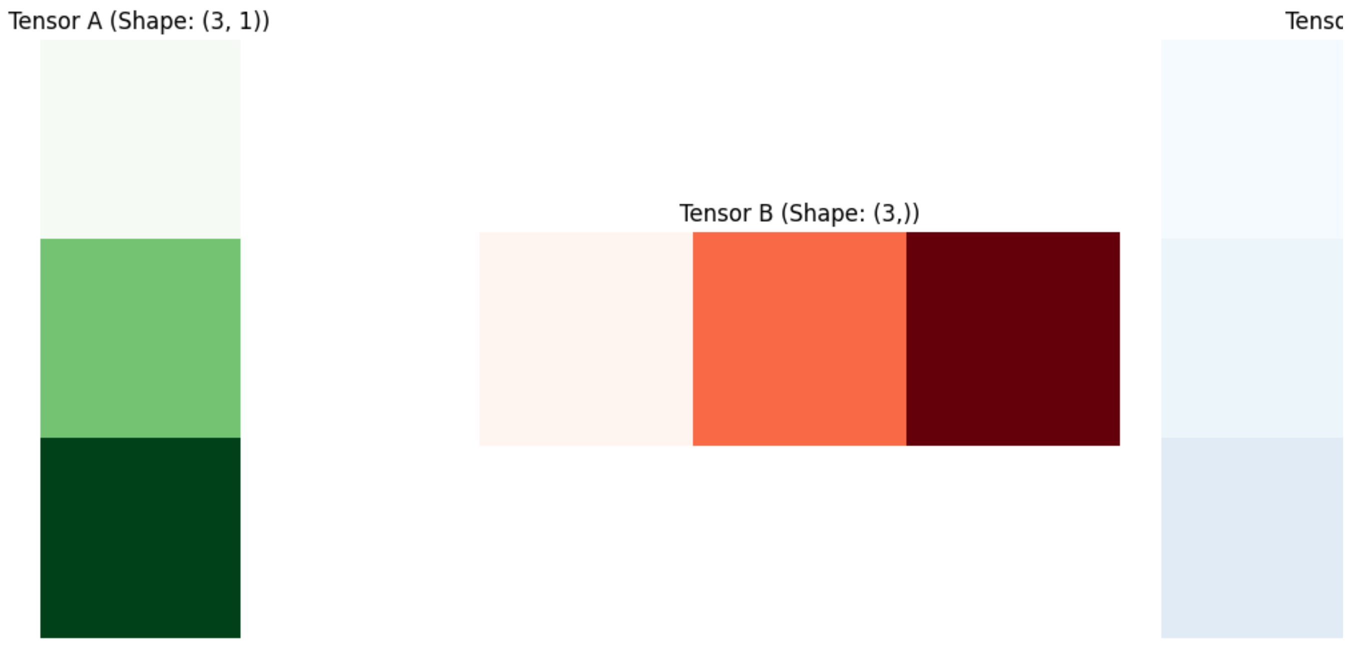
```
# Broadcasting example using PyTorch
import torch
A = torch.tensor([[1], [2], [3]])       # Shape: (3, 1)
B = torch.tensor([10, 20, 30])          # Shape: (3,) or (1, 3)

C = A + B



print(C)
```

```
tensor([[11, 21, 31],
        [12, 22, 32],
        [13, 23, 33]])
```

```
pd.DataFrame([[1], [2], [3]]) + pd.DataFrame([10, 20, 30])
```

```
      0
0    11
1    22
2    33
```

```
# Plotting the Broadcasting concept

import matplotlib.pyplot as plt
import numpy as np
# plot the graph of A, B, and C
A_np = A.numpy()
B_np = B.numpy()
C_np = C.numpy()

fig, axes = plt.subplots(1, 3, figsize=(15, 5))

axes[0].imshow(A_np,cmap='Greens')
axes[0].set_title("Tensor A (Shape: {})".format(A_np.shape))

axes[1].imshow(B_np.reshape(1,-1),cmap='Reds')
axes[1].set_title("Tensor B (Shape: {})".format(B_np.shape))

axes[2].imshow(C_np,cmap='Blues')
axes[2].set_title("Tensor C (A + B) (Shape: {})".format(C_np.shape))

for ax in axes:
  ax.axis('off')

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



Interpretation

Tensor Shapes Tensor A: Shape (3, 1) → A column vector with 3 rows and 1 column.

Tensor B: Shape (3,) → A 1D array with 3 values.

Broadcasting allows arrays of different shapes to be combined in operations like addition (+) by automatically expanding one or both arrays to a compatible shape.

Tensor C = A + B (Shape: (3, 3))

To compute C = A + B, broadcasting happens like this:

Step 1: Expand B Tensor B (shape (3,)) is broadcast to shape (1, 3) to behave like a row vector.

Step 2: Expand A Tensor A (shape (3,1)) is broadcast to shape (3,3) by repeating its single column across all 3 columns.

Step 3: Element-wise Addition The final result C is a (3, 3) matrix

Discuss why broadcasting is important and how it enhances the flexibility of tensor operations in building neural network models.

Answer

Broadcasting is a set of rules used in TensorFlow and PyTorch that allow arithmetic operations on tensors with different shapes by automatically expanding the smaller tensor's shape to match the larger one — without copying data.

Reasons why Broadcasting is Important

Efficiency in Memory and Computation

Avoids explicitly replicating smaller tensors to match larger ones, saving memory and reducing redundant computations.

Example: In Neural Networks, Adding a bias vector (D,) to a batch of activations (B, D) doesn't require manually expanding the bias to (B, D)— broadcasting handles it implicitly.

Generalization Across Dimensions

Enables operations to work seamlessly across batches, channels, or other grouped dimensions without manual intervention.

Example: A weight matrix (C_out, C_in) can multiply a batched input (B, C_in, H, W) using broadcasting rules.

let batched input is shaped (2, 3) — 2 samples, 3 features each

Let weights are shaped (3,) — one weight per feature import torch

X = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]) # Shape: (2, 3) W = torch.tensor([0.1, 0.2, 0.3]) # Shape: (3,)

Y = X * W # Broadcasting W to shape (2, 3)

tensor([[0.1, 0.4, 0.9], [0.4, 1.0, 1.8]]) No need to reshape W to (2, 3) — broadcasting will do it

Broadcasting Enhances Neural Network Flexibility

1. Handling Batch Operations Naturally In neural networks, inputs are typically batched (e.g., shape (B, D)). Broadcasting allows operations like adding biases or applying activations to work independently of batch size.

Example:

Bias (D,) + Batch (B, D) → Automatically broadcasts bias to (B, D) outputs = inputs + bias # No need to reshape bias to (B, D)

2. Efficient Parameter Sharing Broadcasting enables parameters (e.g., weights, biases) to apply across spatial/temporal dimensions without explicit replication.

Example in CNNs:

A kernel (C_out, C_in, 3, 3) convolves with input (B, C_in, H, W) by broadcasting across batch and spatial dimensions.

Input Tensor: (B, C_in, H, W)

B: Batch size (e.g., 32 images)

C_in: Input channels (e.g., 3 for RGB)

H, W: Height and Width

Weight Tensor (Kernel): (C_out, C_in, K_h, K_w)

C_out: Number of output channels (filters)

C_in: Matches input channels

K_h, K_w: Kernel height and width

3. Scalar and Tensor Interactions Scalars or small tensors can modify large tensors naturally (e.g., learning rate adjustments, masking).

Example:

updated_weights = weights - lr * gradients # lr is a scalar

4. Dynamic Shape Compatibility Models can handle variable input shapes (e.g., different sequence lengths in RNNs) as long as broadcasting rules are satisfied.

Example:

Attention scores (B, T1, T2) + Mask (T1, T2) = Broadcasts mask to batch scores = scores + mask # Mask is automatically aligned

5. Simplified Loss Functions and Regularization Broadcasting simplifies operations like per-sample loss weighting or layer-wise regularization.

Example:

weighted_loss = weights * loss # weights shape (B,) broadcasts to loss (B,)

Practical Scenarios in Neural Networks Transformer Implementation Bias Addition in Fully Connected Layers

A bias vector (D,) is broadcast to match the batch dimension (B, D) during forward passes.

Normalization Layers (BatchNorm, LayerNorm)

Statistics (mean/variance) are computed per-channel and broadcast across spatial/batch dimensions.

Attention Mechanisms

Masking tensors (e.g., (T, T)) broadcast to match batched attention scores (B, H, T, T).

Multi-Head Operations in Transfromers

Query/Key/Value projections in transformers use broadcasting to parallelize across heads ((B, H, T, D)).

Conclusion Broadcasting is a cornerstone of modern tensor libraries, making neural network implementations more concise, efficient, and expressive. By automating shape alignment, it allows developers to focus on model logic rather than tedious tensor manipulations, accelerating experimentation and reducing errors. Its seamless integration into frameworks like PyTorch and TensorFlow is a key reason why these libraries are so widely adopted for deep learning.

Explain what limitations or challenges might arise when relying on broadcasting, particularly in the context of ensuring model correctness and efficiency.

Broadcasting is a powerful tool, but it introduces several challenges and limitations that can impact model correctness, efficiency, and debugging. Below are key issues to consider when relying on broadcasting in tensor operations, particularly in neural networks.

1. Silent Shape Mismatches Leading to Bugs Problem: Broadcasting can silently succeed when shapes are misaligned, producing incorrect results without raising errors.

Example:

Intended: Add per-channel bias (C,) to a batch of images (B, C, H, W) bias = torch.randn(C)
output = activations + bias # Correct if activations.shape = (B, C, H, W)

Bug: If activations.shape = (B, H, W, C), broadcasting still works but adds bias to the wrong dimension! The operation runs but produces mathematically wrong gradients or outputs.

Mitigation: Explicitly check shapes with assertions:

assert bias.shape == (activations.shape[1],) # Ensure bias aligns with channel dim Use keepdim=True in ops like sum()/mean() to preserve dimensions.

2. Performance Overheads from Implicit Expansion Problem: Broadcasting creates temporary virtual tensors, but some frameworks may materialize them in memory, hurting efficiency.

Example:

Inefficient: Broadcasting a large tensor (e.g., mask of shape (1, H, W) to (B, C, H, W))May allocate unnecessary memory for repeated values This can lead to high memory usage or slow operations on accelerators (GPU/TPU).

Mitigation: Prefer expand() over broadcasting when possible (explicit control).

Use in-place operations (e.g., tensor.add_(bias)) to avoid temporary copies.

3. Gradient Computation Issues Problem: Broadcasting can cause unintended gradient propagation if not handled carefully.

Example:

If a scalar is broadcast to a tensor, gradients may scatter unexpectedly. Example

x = torch.randn(3, requires_grad=True)

y = x * 2 # Scalar 2 is broadcast; gradients w.r.t. `x` are correct.

z = y.sum() # But if `y` is used in a larger graph, gradients may get messy.

In complex ops, gradients may not flow as intended due to implicit expansion.

4.Ambiguity in Higher Dimensions Problem: Broadcasting rules align dimensions right-to-left, which can be confusing for high-rank tensors.

Example:

Shape (5, 4, 3) + (4, 3) ===> Valid (trailing dims match) Shape (5, 4, 3) + (4, 1) ==> Valid (dim=1 is 1) Shape (5, 4, 3) + (3,)==> Valid (aligns with last dim) Shape (5, 4, 3) + (5,) ==> **Fails** (dim=0 doesn't match and isn't 1) Debugging becomes hard when shapes are misaligned in non-trailing dimensions.

Mitigation: Use unsqueeze() or reshape() to make shapes explicit:

tensor = tensor.unsqueeze(1) # Add a dimension for clarity Visualize shapes with tools like torchviz or logging.

5. Device and Framework Compatibility Problem: Broadcasting behavior may differ across frameworks (PyTorch vs. TensorFlow) or devices (CPU vs. GPU).

Example:

Some GPU kernels optimize broadcasted ops, while others fall back to slower paths.

Edge cases (e.g., empty tensors) may behave inconsistently.

Mitigation: Test on all target devices.

Avoid "clever" broadcasting tricks that aren't portable.

6. Debugging Difficulty Problem: Errors caused by broadcasting are often non-intuitive and surface late (e.g., during loss computation).

Example:

A misbroadcasted tensor might only fail in backprop or produce NaN gradients. Mitigation: Enable anomaly detection:

torch.autograd.set_detect_anomaly(True) # Catches broadcast-related NaN gradients Use shape-checking libraries like torchtyping or beartype.

Best Practices to Avoid Pitfalls bold text

Explicit over Implicit:

Use reshape(), expand(), or repeat() instead of relying on broadcasting when clarity matters.

Shape Assertions:

Validate shapes at runtime:

assert x.shape == (B, C, H, W), f"Expected (B,C,H,W), got {x.shape}" Gradient Checks:

Verify custom ops with torch.autograd.gradcheck.

Memory Profiling:

Monitor GPU memory usage to catch unintended expansions.

When to Avoid Broadcasting Critical Performance Paths: Use pre-allocated buffers for repeated ops (e.g., in RL loops).

Custom CUDA Kernels: Explicit shapes are often clearer for low-level optimizations.

Distributed Training: Broadcasting across devices can introduce synchronization overhead.

Conclusion While broadcasting simplifies code and boosts productivity, it requires careful attention to shapes, gradients, and performance. By combining defensive programming (assertions, explicit reshaping) with debugging tools, you can harness its power while avoiding pitfalls. Always validate tensor shapes during development—especially in neural networks, where silent bugs can corrupt training dynamics.

END