The Ultimate PyTorch Technical Reference

Deep Learning Notes

February 2026

Contents

1 Extensive Tensor Operations and "Weird" Logic

Tensors are the bedrock of PyTorch. Beyond simple addition, these functions allow for complex data manipulation.

1.1 Standard and Structural Initializers

- `torch.eye(n, m)`: Returns a 2D tensor with ones on the diagonal (Identity matrix).
- `torch.diag(input)`: Extracts a diagonal or constructs a diagonal matrix.
- `torch.tril(input) / torch.triu(input)`: Returns the lower or upper triangular part of a matrix (used for masking in Transformers).
- `torch.full((shape), value)`: Creates a tensor filled entirely with a specific scalar.
- `torch.linspace(start, end, steps)`: Creates a one-dimensional tensor of equally spaced points.

1.2 Advanced Indexing and Selection

These functions are the "weird" workhorses for dynamic neural architectures.

- **torch.einsum(equation, *operands)**: Einstein summation. Handles products, sums, and transposes in one string.
  - Batch MatMul: `'bij,bjk->bik'`
  - Inner product: `'i,i->'`
  - Trace: `'ii'`
- **torch.gather(input, dim, index)**: Gathers values along an axis.
- **torch.scatter_(dim, index, src)**: Writes values into a tensor at specific indices (in-place).
- **torch.where(condition, x, y)**: Returns a tensor of elements selected from either x or y based on the condition.
- **torch.roll(input, shifts, dims)**: Circularly shifts the tensor along given dimensions.
- **torch.unbind(input, dim)**: Removes a dimension, returning a tuple of all slices along that dimension.

1.3 Shape Manipulation and Memory Management

- `tensor.view(*shape)`: Changes shape without copying data (must be contiguous).
- `tensor.reshape(*shape)`: Changes shape, may copy data if necessary.
- `tensor.permute(*dims)`: Arbitrary reordering of dimensions.
- `tensor.squeeze() / tensor.unsqueeze(dim)`: Removes or adds dimensions of size 1.
- `tensor.expand(*sizes)`: Views a singleton dimension as larger without using extra memory.
- `tensor.repeat(*sizes)`: Copies the data to repeat the tensor.

—

2 Comprehensive Loss Function Catalog

Loss functions (`torch.nn`) define the objective the network tries to minimize.

Loss Function	Typical Task	Output Requirement
`nn.MSELoss`	Regression	Linear/Raw output
`nn.L1Loss`	Regression (MAE)	Robust to outliers
`nn.CrossEntropyLoss`	Multi-class	Raw Logits (No Softmax!)
`nn.BCEWithLogitsLoss`	Binary / Multi-label	Raw Logits (No Sigmoid!)
`nn.NLLLoss`	Multi-class	LogSoftmax outputs
`nn.KLDivLoss`	Distillation	Log-probabilities
`nn.HuberLoss`	Robust Regression	Combines MSE and L1

2.1 Deep Dive: When to use what?

1. **nn.CrossEntropyLoss**: Use for classification where each input belongs to exactly one class. It expects raw scores (logits) because it internally applies `LogSoftmax`.
2. **nn.BCEWithLogitsLoss**: Use for binary classification (Yes/No) or multi-label classification (e.g., tags in an image). It is more numerically stable than `Sigmoid` + `BCELoss`.
3. **nn.MSELoss**: Standard for predicting continuous values. If your data has many outliers, consider `nn.L1Loss` or `nn.HuberLoss`.

—

3 Implementation of the Training Pipeline

A training loop in PyTorch is explicit, giving the developer full control over the gradient flow.

3.1 The Anatomy of a Training Loop

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

# 1. Architecture Definition
class ResNetBlock(nn.Module):
    def __init__(self, in_dims, out_dims):
        super().__init__()
        self.linear = nn.Linear(in_dims, out_dims)
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.relu(self.linear(x))

# 2. Setup
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = ResNetBlock(784, 10).to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.CrossEntropyLoss()

# 3. Training Loop
```

```python
def train_one_epoch(epoch_index):
    model.train() # Enable dropout/batchnorm
    running_loss = 0.

    for i, (inputs, labels) in enumerate(train_loader):
        inputs, labels = inputs.to(device), labels.to(device)

        # MANDATORY: Clear gradients from previous step
        optimizer.zero_grad()

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, labels)

        # Backward pass (Autograd magic)
        loss.backward()

        # Update weights
        optimizer.step()

        running_loss += loss.item()

    return running_loss / len(train_loader)

# 4. Evaluation Loop
def validate():
    model.eval() # Disable dropout/batchnorm
    total_loss = 0
    with torch.no_grad(): # Disable gradient calculation (saves memory)
        for inputs, labels in val_loader:
            outputs = model(inputs.to(device))
            loss = criterion(outputs, labels.to(device))
            total_loss += loss.item()
    return total_loss / len(val_loader)
```

3.2 Critical Checklist

- **zero_grad()**: Without this, gradients accumulate (sum up) across batches, destroying training.
- **loss.backward()**: Populates the `.grad` attribute for every parameter.
- **optimizer.step()**: Adjusts parameters based on the `.grad` values.
- **torch.no_grad()**: Always use this during inference/validation to prevent the computation graph from being built, which saves a massive amount of VRAM.