Andrew Taylor, atayl136 EN.705.743 September 1, 2025

Tensors, Parameters, and Gradients in PyTorch

Tensors: the raw material

A tensor in PyTorch is the most fundamental building block, the generalization of scalars, vectors, and matrices into arbitrary dimensions. It holds numerical data and, if configured with requires_grad=True, it can also carry gradient history, making it suitable for participation in backpropagation. Conceptually, you can think of a tensor as a container for numbers that can be manipulated by mathematical operations, all while potentially recording those operations in a computation graph. However, a tensor by itself is not inherently "learnable." It is simply data, flexible and powerful, but without any guarantee that PyTorch's optimization machinery will pay attention to it during training.

Parameters: tensors with a role

A parameter in PyTorch is a tensor with a special status. Defined as a subclass of Tensor, it signals to PyTorch that the values it contains are intended to be optimized. This is how weights and biases of neural network layers are stored in models. When a tensor is wrapped as a Parameter and assigned inside an nn.Module, it is automatically registered by the framework so that optimizers such as SGD or Adam can update it. Without this wrapping, the optimizer will ignore the tensor entirely, even if gradients flow through it. This subtle distinction is one of the easiest traps to fall into in PyTorch: a tensor may appear to be part of the model, but unless it is declared as a parameter, the learning algorithm won't adjust it.

Gradients: the shadows of change

Gradients are not objects you create directly but are instead produced by PyTorch's automatic differentiation system when you call .backward(). They are stored in the .grad attribute of tensors or parameters and represent the derivative of the loss with respect to those values. Gradients quantify how sensitive the loss is to small changes in parameters, and they form the basis for optimization steps that update model weights. A crucial quirk of PyTorch is that gradients accumulate by default; if you forget to reset them with optimizer.zero_grad(), each call to .backward() will stack new information on top of the old, effectively distorting your training dynamics.

The interplay of tensors, parameters, and gradients

The training process only works when tensors, parameters, and gradients are aligned properly. A parameter without a gradient will never be updated, a gradient attached to a plain tensor won't matter if the optimizer does not know about it, and a tensor that never participates in backpropagation remains static data. Together, these three concepts form the foundation of learning in PyTorch: tensors as the raw materials, parameters as the designated targets of optimization, and gradients as the guiding signals that inform how to adjust them. The secret is that this interplay is fragile—get one link wrong, and your model will appear to run but silently fail to learn.

Putting it together

- **Tensor** = any block of data, maybe differentiable.
- **Parameter** = a tensor that's officially marked as part of a model's anatomy.
- **Gradient** = the derivative tensor that autograd leaves behind, used for learning.