Advanced PyTorch

Archita Jain

What is PyTorch?

- Open Source Deep Learning Library
- Facebook AI Research Lab
- Pytorch DDP and CUDA support
- Gradient computation via `torch.autograd`
  - Automatic differentiation for any operation

PyTorch vs. TensorFlow

VS

- Open-source Deep Learning Frameworks
- PyTorch
  - Facebook
  - Pythonic Interface
  - Dynamic Computation Graphs
- TensorFlow
  - Google
  - Suited for Scalability/Production
  - Static Computation Graphs
- Pytorch is widely used in Academia (which is our goal) and simply easier to play around with so: Pytorch!

Quick Foundations Recap

- Tensors
  - fundamental data structure in PyTorch.
  - multi-dimensional arrays, like NumPy's `ndarray`, but with two key advantages
    - Seamless GPU acceleration with (`.to('cuda')`).
    - Integration with the automatic differentiation engine
- Autograd (`torch.autograd`)
  - Gradient Calculation
    - Set `tensor.requires_grad=True` to track operations.
    - Call `loss.backward()` to compute gradients $\left(\frac{\partial \mathbf{loss}}{\partial \mathbf{tensor}}\right)$ via backpropagation.
    - Gradients are stored in the `.grad` attribute of each tensor.

Custom Architecture

- `torch.nn.Module`
  - base class for all neural network models in PyTorch
  - by subclassing it, you get critical functionality for free
    - Parameter Tracking (`nn.Parameter`)
    - State Management (`state_dict`)
  - Predefined Layers like `nn.Linear`, `nn.Conv2d`, `nn.RNN`, `nn.BatchNorm2d`, etc.
  - Custom Layers
    - You can easily create your own layers by combining existing ones or defining new operations within a custom `nn.Module`

Building from `nn.Module`

- `__init__(self)` method
  - define and initialize your model's layers (convolutional layers, linear layers, etc.)
    - similar to Python
  - Assigning a layer to an attribute automatically registers it, making its parameters available to the parent module
    - e.g. `self.conv1 = nn.Conv2d(...)`
- `forward(self, x)` method
  - defines the forward pass
    - how input `x` flows through the layers you defined in `__init__`

E.g. Custom ResNet Block

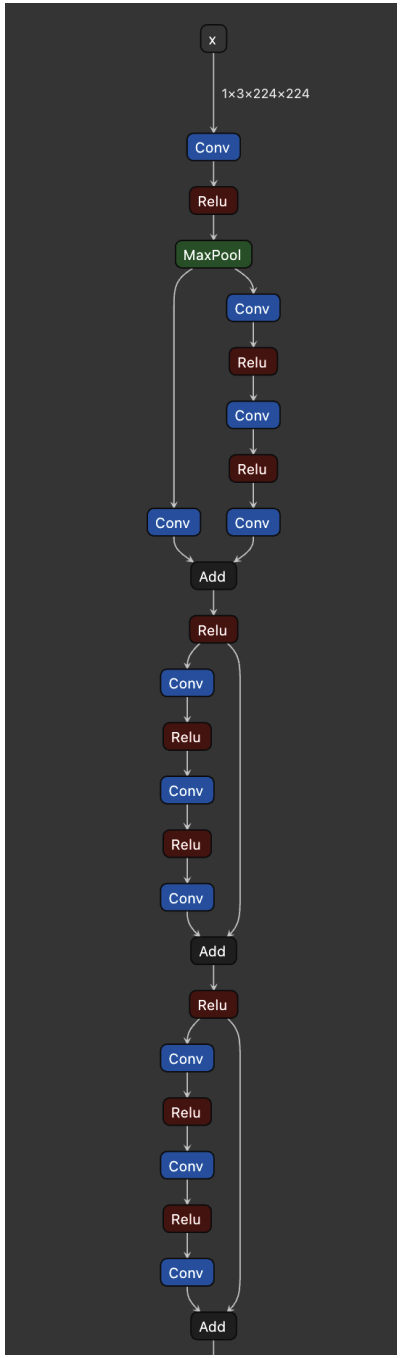
```python
import torch.nn as nn
import torch.nn.functional as F

class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()

        # Main path
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Shortcut path to handle dimension changes
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x) # skip connection
        out = F.relu(out)
        return out
```



Some Best Practices

- Reusing Submodules & Parameter Sharing (Siamese Networks)
  - Call submodule in `__init__` and multiple times in `forward()`
- `register_buffer()` vs. `register_parameter()`
  - Use `register_parameter()` to tell `nn.Module` to track a tensor that needs gradients but isn't part of a standard layer.
  - Use `register_buffer()` for state that should be part of the model's `state_dict` but doesn't require gradients (e.g. fixed positional encodings).

Custom Loss

- Standard losses like MSE or Cross-Entropy don't cover everything
- Method 1: Functional Style (Most Common)
  - Python function
    - Parameters: model predictions and targets
    - Output: scalar loss tensor
  - Works if you use differentiable `torch` operations
    - Autograd will handle the backward pass automatically.
- Method 2: Extending `nn.Module`
  - Useful if loss function has its own learnable parameters.
  - Implement loss like a layer
    - i.e. with an `__init__` to define parameters and a `forward()` method to perform the loss calculation.

Custom Loss cont.

- Handling Imbalanced Datasets using `nn.CrossEntropyLoss`
  - Weighted BCE/CE
  - Focal Loss
- Metric Learning
  - Contrastive Loss
  - Cosine Embedding Loss
- Multi-task Learning
  - Combine multiple losses
    - e.g. `total_loss = w1 * classification_loss + w2 * regression_loss`
  - Learn the weights w1 and w2 as parameters to allow the model to dynamically balance tasks.

Optimizers

- Learning Rate Schedules using `torch.optim.lr_scheduler`
  - `CosineAnnealingLR`
    - Smoothly anneals the learning rate following a cosine curve.
  - `OneCycleLR`
    - Warms up the learning rate to a maximum value and then anneals it down
    - Can lead to super-convergence and faster training
- Gradient Clipping using `torch.nn.utils.clip_grad_norm_`
  - Prevents exploding gradients by scaling them down if their norm exceeds a threshold
  - Essential for training RNNs and Transformers.
- Gradient Accumulation
  - Simulates a larger batch size on a memory-constrained GPU
  - Perform forward/backward passes for several small batches
  - call `optimizer.step()` after accumulating gradients from all batches

Optimizers cont.

- SGD with Momentum
  - The classic workhorse (`torch.optim.SGD`).
  - Accumulates past gradients to accelerate convergence and navigate ravines.
  - Still state-of-the-art for many vision tasks when tuned well.
- Adam (Adaptive Moment Estimation)
  - Excellent general-purpose optimizer (`torch.optim.Adam`).
  - Maintains per-parameter learning rates based on estimates of first (mean) and second (variance) moments of the gradients.
  - Works well out-of-the-box for a wide variety of problems.
- AdamW
  - `torch.optim.AdamW` fixes how weight decay is handled in Adam.
  - It decouples the L2 regularization from the adaptive learning rate update, which often leads to better model generalization.
  - The recommended default for Transformers and other modern architectures.

Batching Techniques

- The Role of Batch Size
  - Large Batch – stable gradients, higher learning rates, faster training
  - Small Batch – noisy gradients, generalizes better
- Normalization Layers
  - Batch Normalization (`nn.BatchNorm2d`):
    - Normalizes activations across the batch dimension.
    - Stabilizes training immensely.
  - Layer Normalization (`nn.LayerNorm`):
    - Normalizes activations across the feature dimension for a single training example.

Advanced Training

- Mixed Precision Training with `torch.cuda.amp`
  - Using a mix of `float32` (full precision) and `float16` (half precision) during training.
- `nn.DataParallel` (DP) [DEPRECATED]
- `nn.DistributedDataParallel` (DDP)
  - Industry standard. Faster and more efficient.
  - Spawns a separate process for each GPU. model copied once at the beginning.
  - During the backward pass, gradients are efficiently averaged across all processes using an all-reduce operation.
  - Requires more setup (initializing a process group) but provides superior performance

Hooks

- Callback functions that you can attach to a `nn.Module` or a Tensor
- Forward Hooks (`register_forward_hook`):
  - Runs after the forward pass of a module.
  - Good for Feature Extraction
- Backward Hooks (`register_full_backward_hook`):
  - Run after gradients have been computed for a module.
  - Good for Gradient Analysis & Interpretability.
    - Saliency Maps
    - Debugging

Custom Autograd

- Subclassing torch.autograd.Function and implementing two static methods:
  - `forward(context_tensors, *args)`
    - Define the operation itself
  - `backward(context_tensors, *grad_outputs)`
    - mathematically derive and implement the gradient formula for your operation.
- When to use this?
  - Integrating highly optimized custom C++ or CUDA kernels.
  - Implementing operations that are non-differentiable, but for which you want to define a "surrogate" gradient.
  - Physics-Informed Neural Networks (PINNs) where derivatives are part of the model's logic.

Advanced Architecture

- Transformers
  - Hugging Face transformers Library – access to most opensource models (GPT, BERT, ViT, etc.)
  - Native PyTorch - `nn.Transformer`, `nn.TransformerEncoderLayer`, and `nn.MultiheadAttention`
- GNNs - PyTorch Geometric (PyG)
  - vast collection of optimized GNN layers (GCN, GAT, GraphSAGE), graph-based datasets, and utilities
- MAML: Requires computing gradients of gradients (higher-order derivatives).
  - PyTorch's autograd supports this out of the box by calling `loss.backward(create_graph=True)`.

Thank You!

Q&A

Regularization??

- Weight Decay (`AdamW`)
- Stochastic Depth
  - During training, randomly bypass entire residual blocks by setting their output to the identity.
  - Acts as a powerful regularizer for very deep networks.
- Data Augmentation
  - Mixup
    - Create new training samples by taking weighted linear interpolations of existing images and their labels: $\tilde{x} = \lambda x_i + (1 - \lambda) x_j$.
  - CutMix: Cut a patch from one image and paste it over another. The labels are mixed in proportion to the area of the patch.