# Understanding PyTorch Computational Graphs and Autograd

March 3, 2025

In our previous post, we explored building a custom model class in PyTorch. We briefly encountered the concept of computational graphs when we saw `grad_fn=<AddmmBackward0>` in our model's output tensor. Today, we'll dive deeper into computational graphs and PyTorch's autograd system, which together form the foundation of PyTorch's automatic differentiation capabilities.

## The Big Picture: How PyTorch Makes Neural Networks Work

Before we delve into the technical details, let's understand how these concepts fit together:

1. **The Neural Network Challenge**: Training neural networks requires calculating how to adjust thousands or millions of parameters to minimize error.
2. **The Mathematical Solution**: We use calculus - specifically, computing derivatives of the loss function with respect to each parameter (gradients).

3. **The Implementation Challenge**: Computing these gradients manually would be tedious and error-prone for complex networks.

4. **PyTorch's Solution**:

   - **Computational Graphs**: Track all operations performed on tensors
   - **Autograd**: Automatically compute gradients by traversing these graphs
   - **Dynamic Computation**: Build graphs on-the-fly during forward passes

This approach allows PyTorch to provide both flexibility (dynamic graphs) and efficiency (automatic gradient computation) - the best of both worlds.

# What are Computational Graphs?

At its core, a computational graph is a directed graph that represents a sequence of mathematical operations. In the context of deep learning:

- **Nodes** represent operations (like addition, multiplication, or activation functions) or variables (like model parameters or input data).
- **Edges** represent the flow of data between operations.

These graphs provide a structured way to represent complex mathematical expressions, making it easier to compute derivatives through the chain rule.

## A Simple Example

Let's build a computational graph for a simple expression, step by step:

$$y = (a + b) * (b + 1)$$

Here's how this expression can be represented as a computational graph:

1. We have two input variables: $a$ and $b$.
2. We compute an intermediate value $c = a + b$.

3. We compute another intermediate value $d = b + 1$.
4. Finally, we compute the output $y = c * d$.

Let's implement this in PyTorch and observe the computational graph:

```
import torch

# Enable gradient tracking for these tensors
a = torch.tensor(2.0, requires_grad=True)
b = torch.tensor(3.0, requires_grad=True)

# Forward pass: build the computational graph
c = a + b    # c = 5.0
d = b + 1    # d = 4.0
y = c * d    # y = 20.0

print(y)
print(f"Computational graph: {y.grad_fn}")
```

Output:

```
y = tensor(20., grad_fn=<MulBackward0>)
Computational graph: <MulBackward0 object at 0x7f12c4b9f340>
```

Notice how PyTorch automatically attaches `grad_fn` to tensors that are part of a computation. This records the operation that created the tensor, which is crucial for computing gradients later.

## Visualizing the Computational Graph

Let's take a closer look at our computational graph:

```
# Inspect backward graph structure
print(f"y was created by: {y.grad_fn}")
print(f"c was created by: {c.grad_fn}")
print(f"d was created by: {d.grad_fn}")
```

Output:

```
y was created by: <MulBackward0 object at 0x7f12c4b9f340>
c was created by: <AddBackward0 object at 0x7f12c4b9f1c0>
d was created by: <AddBackward0 object at 0x7f12c4b9f2c0>
```

We can further inspect the inputs to these operations:

```
# Look at the next nodes in the graph
print(f"Inputs to the multiplication: {y.grad_fn.next_functions}")
mul_next_a, mul_next_b = y.grad_fn.next_functions
print(f"Inputs to the first addition: {mul_next_a[0].next_functions}")
print(f"Inputs to the second addition: {mul_next_b[0].next_functions}")
```

Output:

```
Inputs to the multiplication: ((<AddBackward0 object at 0x7f12c4b9f1c0>,
Inputs to the first addition: ((<AccumulateGrad object at 0x7f12c4b9f400>
Inputs to the second addition: ((<AccumulateGrad object at 0x7f12c4b9f4c0
```

This shows us the complete computational graph structure:

- `y` was created by a multiplication operation ( `MulBackward0` )
- The multiplication had two inputs: `c` and `d` , both created by addition operations ( `AddBackward0` )
- The first addition had inputs `a` and `b` (represented by `AccumulateGrad` objects, which are leaf nodes)
- The second addition had inputs `b` and the constant `1` (which has no gradient history)

## Computational Graph in Neural Networks

Now, let's look at how computational graphs apply to a simple neural network. Consider a basic network with one hidden layer:

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

# Define a simple neural network
class SimpleNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(2, 3)  # 2 inputs, 3 hidden units
        self.activation = nn.ReLU()
        self.linear2 = nn.Linear(3, 1)  # 3 inputs, 1 output

    def forward(self, x):
        x = self.linear1(x)
        x = self.activation(x)
        x = self.linear2(x)
        return x

# Create an instance and examine a forward pass
model = SimpleNN()
x = torch.tensor([[0.5, 0.3]], requires_grad=True)
output = model(x)

print(f"Output: {output}")
print(f"Output grad_fn: {output.grad_fn}")
```

Output:

```
Output: tensor([[-0.1435]], grad_fn=<AddmmBackward0>)
Output grad_fn: <AddmmBackward0 object at 0x7f12c4b9f880>
```

The `grad_fn=<AddmmBackward0>` indicates that the last operation was a matrix multiplication with bias addition (from the final linear layer). If we were to trace backward, we'd find the complete computational graph representing all operations in our neural network.

# How Computational Graphs Help with Backpropagation

The computational graph is the key to efficient backpropagation. In deep learning, we need to compute gradients of the loss with respect to model parameters to update them during training.

Think of the computational graph as a roadmap that shows exactly how your input data transforms into the final output through a series of operations. When we need to update our model parameters, we need to know how each parameter influenced the final result - and the computational graph provides this information by enabling us to trace the impact of each parameter backward through the network.

## Understanding Backpropagation

Backpropagation is an algorithm that efficiently computes gradients through the chain rule of calculus. It works by:

1. Performing a forward pass through the network, building the computational graph
2. Computing the loss at the output
3. Computing gradients of the loss with respect to each parameter by traversing the graph backward

Let's see backpropagation in action using our simple expression:

```
# Forward pass (same as before)
a = torch.tensor(2.0, requires_grad=True)
b = torch.tensor(3.0, requires_grad=True)
c = a + b    # c = 5.0
d = b + 1    # d = 4.0
y = c * d    # y = 20.0
```

```
# Backward pass - compute gradients
y.backward()

# Print the gradients
print(f"Gradient of y with respect to a: {a.grad}")
print(f"Gradient of y with respect to b: {b.grad}")
```

Output:

```
Gradient of y with respect to a: 4.0
Gradient of y with respect to b: 9.0
```

Let's verify these gradients analytically:

- $y = (a + b) * (b + 1)$
- $\frac{\partial y}{\partial a} = (b + 1) = 4$
- $\frac{\partial y}{\partial b} = a + 2b + 1 = 2 + 6 + 1 = 9$

The computed gradients match our analytical calculation!

## Backpropagation in Neural Networks

For neural networks, the process is similar but more complex due to the larger number of parameters. Let's demonstrate with our simple neural network:

```
# Create a new network instance
model = SimpleNN()

# Define inputs and targets
x = torch.tensor([[0.5, 0.3]], requires_grad=True)
target = torch.tensor([[1.0]])

# Forward pass
output = model(x)

# Compute loss
loss_fn = nn.MSELoss()
```

```
loss = loss_fn(output, target)
print(f"Loss: {loss.item()}")

# Backward pass
loss.backward()

# Print gradients for one parameter
print(f"Gradient for first layer weights:\n{model.linear1.weight.grad}")
```

PyTorch traverses the computational graph from the loss to all parameters, computing gradients along the way. This is the essence of training neural networks: update parameters in the direction that minimizes the loss.

# What is Autograd and How Does it Work?

PyTorch's autograd system is a powerful engine for automatic differentiation. It handles all the complex gradient calculations so you don't have to implement backpropagation manually.

Now that we understand computational graphs and backpropagation, let's see how PyTorch implements these concepts through its autograd system. Think of autograd as the engine that powers PyTorch's ability to compute gradients - it's the practical implementation of the theoretical concepts we've discussed so far.

## Key Components of Autograd

### 1. Tensor Tracking

PyTorch tracks operations on tensors with `requires_grad=True` and builds a dynamic computational graph:

```
# Tensors with gradient tracking
x = torch.tensor([2.0, 3.0], requires_grad=True)
y = torch.tensor([4.0, 5.0], requires_grad=True)
```

```
# Operation that will be tracked
z = x * y + torch.sum(x)

# Verify tracking
print(f"requires_grad: {z.requires_grad}")
print(f"grad_fn: {z.grad_fn}")
```

Output:

```
requires_grad: True
grad_fn: <AddBackward0 object at 0x7f12c4b9fb80>
```

## 2. Automatic Graph Building

PyTorch dynamically builds the graph during the forward pass:

```
# Dynamic graph - different operations can be performed each time
def compute(x, flag):
    if flag:
        return x * x
    else:
        return x + x

a = torch.tensor(2.0, requires_grad=True)
b = compute(a, True)   # b = a * a = 4.0
c = compute(a, False)  # c = a + a = 4.0

print(f"b grad_fn: {b.grad_fn}")  # MulBackward0
print(f"c grad_fn: {c.grad_fn}")  # AddBackward0
```

Output:

```
b grad_fn: <MulBackward0 object at 0x7f12c4b9fc40>
c grad_fn: <AddBackward0 object at 0x7f12c4b9fd00>
```

This dynamic nature allows PyTorch to handle arbitrary code paths, control flows, and recursion.

## 3. Detaching from Graph

Sometimes you need to stop gradient tracking, which you can do with `.detach()`:

```
x = torch.tensor([2.0, 3.0], requires_grad=True)
y = x * 2

# Detach y from the graph
z = y.detach()
w = z * 3

# y requires gradient, but z and w don't
print(f"y requires_grad: {y.requires_grad}")
print(f"z requires_grad: {z.requires_grad}")
print(f"w requires_grad: {w.requires_grad}")
```

Output:

```
y requires_grad: True
z requires_grad: False
w requires_grad: False
```

This is useful for inference or when implementing techniques like gradient clipping.

## 4. No-Gradient Mode

For code blocks where you don't need gradients, use `torch.no_grad()`:

```
x = torch.tensor([2.0, 3.0], requires_grad=True)

with torch.no_grad():
    y = x * 2
```

```
    print(f"Inside no_grad, y requires_grad: {y.requires_grad}")

z = x * 2
print(f"Outside no_grad, z requires_grad: {z.requires_grad}")
```

Output:

```
Inside no_grad, y requires_grad: False
Outside no_grad, z requires_grad: True
```

This is widely used during model evaluation to save memory and computation.

### 5. Retaining Computation Graph

By default, the graph is freed after `.backward()` is called. To retain it for multiple backward passes (like in RNNs), use `retain_graph=True`:

```
x = torch.tensor(2.0, requires_grad=True)
y = x * x

# First backward pass
y.backward(retain_graph=True)
print(f"After first backward: {x.grad}")

# Reset gradients
x.grad.zero_()

# Second backward pass (would error without retain_graph=True)
y.backward()
print(f"After second backward: {x.grad}")
```

Output:

```
After first backward: tensor(4.)
After second backward: tensor(4.)
```

## Under the Hood: How Autograd Computes Gradients

Autograd uses the chain rule to compute gradients. For each operation, PyTorch defines both the forward function and a corresponding backward function that calculates gradients.

For a simple example, consider $y = x^2$:

1. **Forward pass**: Compute $y = x^2$ and store in the graph that $x$ was squared
2. **Backward pass**: Use the chain rule to compute $\frac{dy}{dx} = 2x$

When you call `.backward()` on a tensor, PyTorch:

1. Starts at the output node
2. Computes gradients with respect to each input of the operation
3. Propagates these gradients backward through the graph
4. Accumulates gradients for leaf nodes (like parameters)

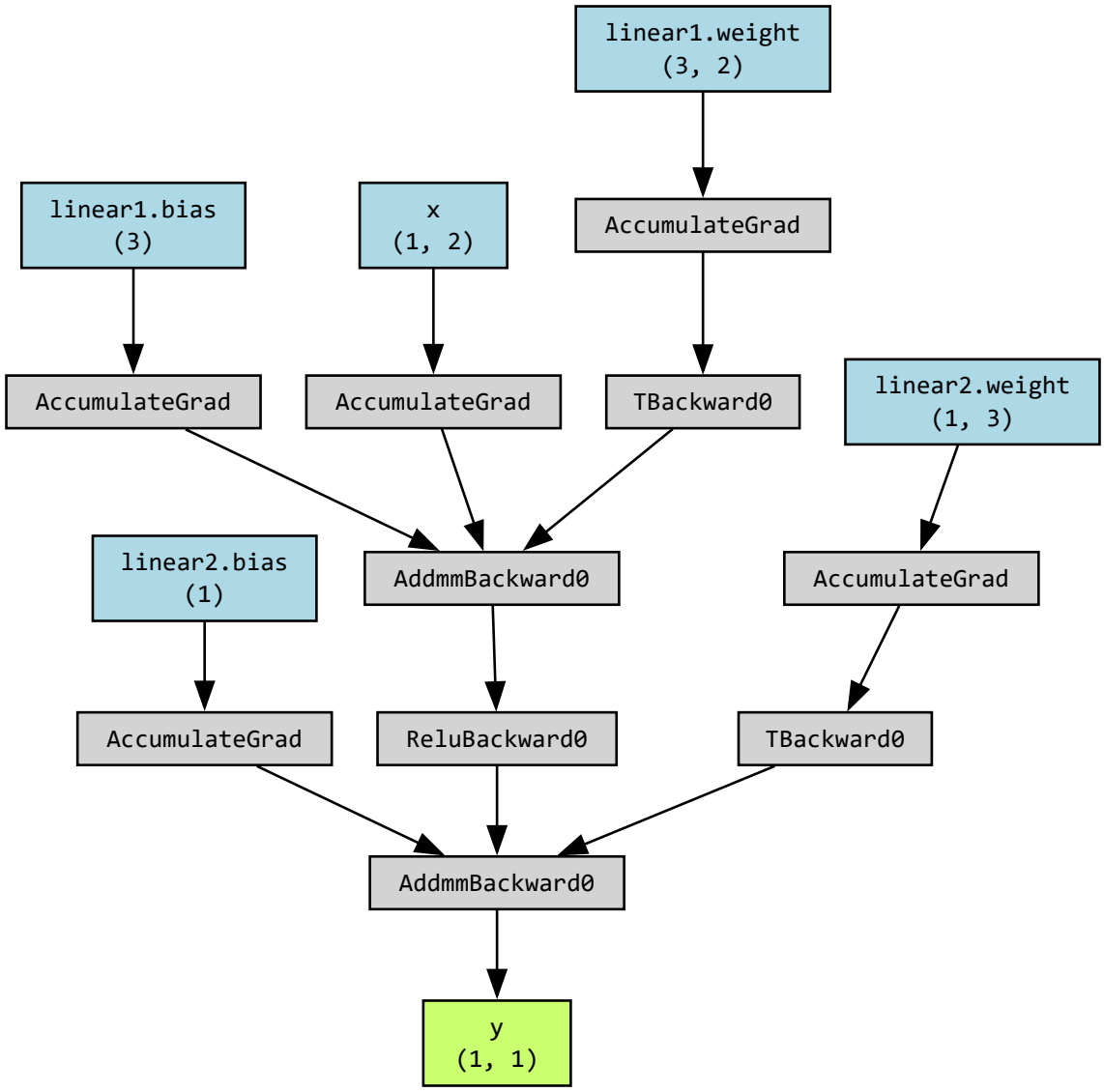
# Visualizing a Computational Graph for a Neural Network

Let's visualize a more comprehensive computational graph for a neural network. We'll use the `torchviz` package to render the graph (note: in real usage, you'd need to install this package):

```
# This code is for illustration - in a real notebook you'd need:
# pip install torchviz

import torch
import torch.nn as nn
from torchviz import make_dot

# Define a simple network
class TinyNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(2, 3)
```

```
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(3, 1)

    def forward(self, x):
        x = self.linear1(x)
        x = self.relu(x)
        x = self.linear2(x)
        return x

# Create model and input
model = TinyNet()
x = torch.randn(1, 2, requires_grad=True)

# Forward pass
y = model(x)

# Visualize the graph
dot = make_dot(y, params=dict(list(model.named_parameters()) + [('x', x),
dot.format = 'png'
dot.render('computation_graph')
```

Here's the visualization of a computational graph for our neural network:




This visualization shows:

1. Input tensor `x` at the bottom
2. Model parameters (weights and biases)
3. Operations like matrix multiplication and ReLU activation
4. The output tensor `y` at the top
5. Connections showing how data flows through the network

Each node represents an operation or parameter, and the edges show how data flows through the network during the forward pass. When we run

backpropagation, gradients flow in the reverse direction along these same edges.

# Putting It All Together: The Training Loop

Now that we understand computational graphs, backpropagation, and autograd, let's see how these concepts come together in a practical training loop. This is where the magic happens - all the theoretical concepts we've discussed translate into a working neural network.

```
# Training loop with autograd
model = TinyNet()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
criterion = nn.MSELoss()

# Training data
x = torch.randn(10, 2)
y_true = torch.randn(10, 1)

# Training loop
for epoch in range(100):
    # Forward pass - builds the computational graph
    y_pred = model(x)
    loss = criterion(y_pred, y_true)

    # Backward pass
    optimizer.zero_grad()  # Clear previous gradients
    loss.backward()        # Compute gradients via autograd
    optimizer.step()       # Update weights using gradients

    if (epoch+1) % 10 == 0:
        print(f"Epoch {epoch+1}, Loss: {loss.item():.4f}")
```

Let's break down what's happening in each step and how our concepts connect:

1. **Forward Pass**: When we call `model(x)`, PyTorch builds a computational graph tracking all operations.

2. **Loss Calculation**: The `criterion(y_pred, y_true)` adds more nodes to our computational graph, connecting our predictions to the loss value.
3. **Gradient Clearing**: `optimizer.zero_grad()` resets gradients from the previous iteration.
4. **Backward Pass**: `loss.backward()` triggers autograd to traverse the computational graph backward, computing gradients for all parameters using the chain rule.
5. **Parameter Update**: `optimizer.step()` uses the computed gradients to update the model parameters.

The beauty of PyTorch is that these complex operations are handled automatically. You don't need to manually implement backpropagation or track the computational graph - PyTorch does it all for you!

# Conclusion

Computational graphs are the backbone of modern deep learning frameworks. They enable automatic differentiation, making it possible to build and train complex neural networks without manually deriving gradients.

PyTorch's dynamic computational graph and autograd system provide:

1. **Flexibility**: Dynamic graphs allow for complex, data-dependent network architectures
2. **Efficiency**: Only operations on tensors with `requires_grad=True` are tracked
3. **Simplicity**: Automatic differentiation happens behind the scenes
4. **Transparency**: You can inspect the graph and gradients when needed

In the next post, we'll explore the training loop in more detail, including optimization algorithms, learning rate schedules, and regularization techniques.

## References


- PyTorch Autograd Mechanics
- PyTorch Tensors and Autograd Tutorial