3.1 WHAT IS `autograd`

01 September 2025 12:37 AM Avinash Yadav

`autograd` is a core component of PyTorch that ***provides automatic differentiation for tensor operations***. It ***enables gradient computation***, which is essential for training machine learning models using optimization algorithms like gradient descent.

↳ with the help of it, we can perform automatic differentiation on any kind of tensor's operation

When training neural networks, the most frequently used algorithm is **back propagation**. In this algorithm, parameters (model weights) are adjusted according to the **gradient** of the loss function with respect to the given parameter. To compute those gradients, PyTorch has a built-in differentiation engine called `torch.autograd`. It supports automatic computation of gradient for any computational graph.

EXAMPLE 1.)

1.) $Y = x^2 \xrightarrow{\text{To find}} \frac{dy}{dx}$ for given 'x'

```
import torch
```
✓ 0.0s Python

Scalar Tensor

```
x = torch.tensor(3.0, requires_grad=True)
```
✓ 0.0s Python

we are setting requires gradient attribute as True. by default it is False; By this we are telling PyTorch that we want to calculate derivative of tensor 'x'. So it starts tracking it.

So whenever we want to calculate derivate of a tensor; then while creating, we have to set 'requires_grad' attribute as True.

```
y = x**2
```
✓ 0.0s Python

'y' calculated in forward dirxn

```
print(x)

print(y)
```
✓ 0.0s Python

```
tensor(3., requires_grad=True)
tensor(9., grad_fn=<PowBackward0>)
```

computed value

So when we set it True, PyTorch make computation graph internally

forward $y=x^2$

x → sq → y

backward

$\frac{dy}{dx}$

So while going backward PyTorch needs to know what operation was performed here while it went for forward pass. That why it remembers it as 'PowBackward0' for backward pass

```
y.backward()
```
✓ 0.0s Python

Now going backward, it will compute $\frac{dy}{dx}$ i.e. $2x$

```
x.grad
```
✓ 0.0s Python

this is $\frac{dy}{dx} = 2(x) = 2(3) = 6$

```
tensor(6.)
```

The .grad attribute of a Tensor that is not a leaf Tensor won't be populated during autograd.backward(). If you indeed want the .grad field to be populated for a non-leaf Tensor, use .retain_grad() on the non-leaf Tensor.

3 kinds of nodes in Computation Graph

input tensor

Last Node

output tensor

Leaf Node

intermediate nodes

Root Node

Computation Graph

By default gradients are not calculated around intermediate nodes

Conceptually, autograd keeps a record of data (tensors) and all executed operations (along with the resulting new tensors) in a directed acyclic graph (DAG) consisting of Function objects. In this DAG, ***leaves are the input tensors***, ***roots are the output tensors***. By tracing this graph from roots to leaves, you can automatically compute the gradients using the chain rule.

In a **forward pass**, autograd does two things simultaneously:

- run the requested operation to compute a resulting tensor
- maintain the operation's *gradient function* in the DAG.

The **backward pass** kicks off when ***.backward()*** is called on the DAG ***root.autograd*** then:

- computes the gradients from each ***.grad_fn***,
- accumulates them in the respective tensor's ***.grad attribute***
- using the chain rule, propagates all the way to the leaf tensors.

{EXAMPLE 2.}

$$\left.\begin{array}{l} y = x^2 \\ z = \sin(y) \end{array}\right\} \text{ we need } \frac{dz}{dx}$$

```python
x = torch.tensor(4.0, requires_grad=True)
x
```

✓ 0.0s Python

```
tensor(4., requires_grad=True)
```

```python
y = x ** 2
y
```

✓ 0.0s Python

```
tensor(16., grad_fn=<PowBackward0>)
```

Forward Pass

$x \rightarrow sq \rightarrow y \rightarrow sin \rightarrow z$

```python
z = torch.sin(y)
z
```

✓ 0.0s Python

```
tensor(-0.2879, grad_fn=<SinBackward0>)
```

$\frac{dz}{dx} \Leftarrow \frac{dy}{dx} \leftarrow * \leftarrow \frac{dz}{dy} \leftarrow$

Backward Pass

```python
z.backward()
z
```

✓ 0.0s Python

```
tensor(-0.2879, grad_fn=<SinBackward0>)
```

```python
x.grad
```

} Applying the computed differentiation on the given scalar tensor

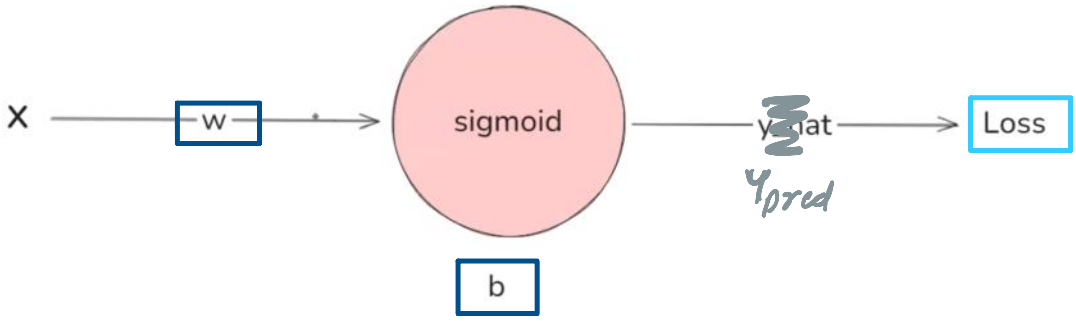
✓ 0.0s Python

```
tensor(-7.6613)
```

How Does autograd works?

INPUT →

CGPA	PLACED
6.7	0 → NO



1. **Linear Transformation:**

$$z = w \cdot x + b$$

2. **Activation (Sigmoid Function):**

$$y_{\text{pred}} = \sigma(z) = \frac{1}{1 + e^{-z}}$$

3. **Loss Function (Binary Cross-Entropy Loss):**

$$L = -\left[y_{\text{target}} \cdot \ln(y_{\text{pred}}) + (1 - y_{\text{target}}) \cdot \ln(1 - y_{\text{pred}})\right]$$

Since we have 2 parameters 'w' & 'b'. Therefore we'll need to find 2 derivates.

1.) derivative of loss wrt 'w' ⇒ $\partial L / \partial w$

2.) derivative of loss wrt 'b' ⇒ $\partial L / \partial b$

$$\therefore \frac{\partial L}{\partial w} = \underbrace{\frac{\partial L}{\partial y_{pred}}}_{\text{from eq.3}} \times \underbrace{\frac{\partial y_{pred}}{\partial z}}_{\text{from eq.2}} \times \underbrace{\frac{\partial z}{\partial w}}_{\text{from eq.1}}$$

Similarly;

$$\frac{\partial L}{\partial b} = \underbrace{\frac{\partial L}{\partial y_{pred}}}_{\text{from eq.3}} \times \underbrace{\frac{\partial y_{pred}}{\partial z}}_{\text{from eq.2}} \times \underbrace{\frac{\partial z}{\partial b}}_{\text{from eq.1}}$$

So now computing derivates:

$$\frac{\partial L}{\partial y_{pred}} = \frac{(y_{pred} - y)}{y_{pred}(1 - y_{pred})} \qquad \frac{\partial z}{\partial w} = x$$

$$\frac{\partial y_{pred}}{\partial z} = y_{pred}(1 - y_{pred}) \qquad \frac{\partial z}{\partial b} = 1$$

$$\therefore \frac{\partial L}{\partial w} = \frac{\partial L}{\partial y_{pred}} \times \frac{\partial y_{pred}}{\partial z} \times \frac{\partial z}{\partial w}$$

$$= \frac{(y_{pred} - y)}{\cancel{y_{pred}(1 - y_{pred})}} * \cancel{y_{pred}(1 - y_{pred})} * X$$

$$= (y_{pred} - y) * X$$

$$\therefore \frac{\partial L}{\partial w} = X(y_{pred} - y)$$

Similarly

$$\frac{\partial L}{\partial b} = (y_{pred} - y)$$

```python
import torch

# Inputs
x = torch.tensor(6.7)  # Input feature
y = torch.tensor(0.0)  # True label (binary)

w = torch.tensor(1.0)  # Weight
b = torch.tensor(0.0)  # Bias
```

→ x
→ y
initial setup with Random value

✓ 0.0s Python

$$L = -\left[y_{\text{target}} \cdot \ln(y_{\text{pred}}) + (1 - y_{\text{target}}) \cdot \ln(1 - y_{\text{pred}})\right]$$

Calculating loss

```python
# Binary Cross-Entropy Loss for scalar
def binary_cross_entropy_loss(prediction, target):
    epsilon = 1e-8  # To prevent log(0)
    prediction = torch.clamp(prediction, epsilon, 1 - epsilon)
    return -(target * torch.log(prediction) + (1 - target) * torch.log(1 - prediction))
```

✓ 0.0s Python

```python
# Forward pass
z = w * x + b  # Weighted sum (linear part)

y_pred = torch.sigmoid(z)  # Predicted probability

# Compute binary cross-entropy loss
loss = binary_cross_entropy_loss(y_pred, y)
loss
```

→ Eq.1
→ Eq.2
→ Eq.3

✓ 0.0s Python

```
tensor(6.7012)
```

BACK-PROPAGATION

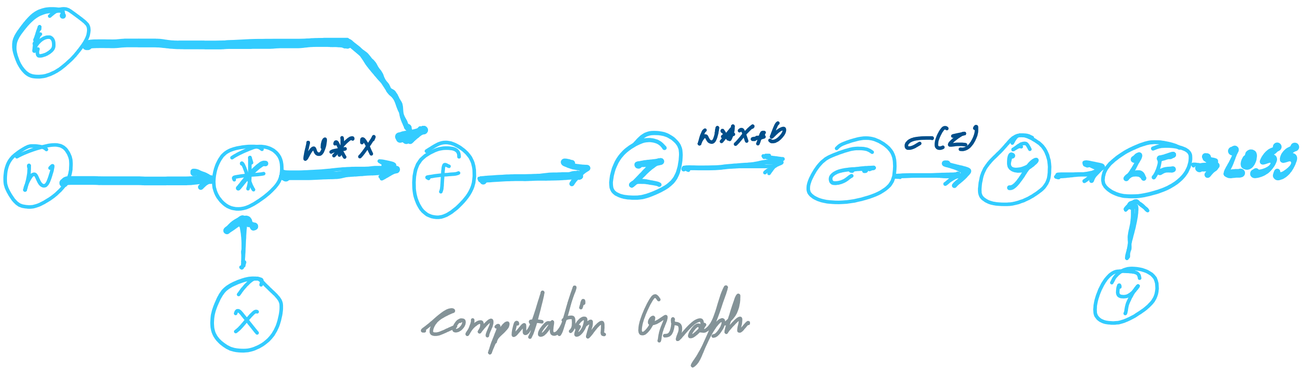
```python
# Derivatives:
# 1. dL/d(y_pred): Loss with respect to the prediction (y_pred)
dloss_dy_pred = (y_pred - y)/(y_pred*(1-y_pred))

# 2. dy_pred/dz: Prediction (y_pred) with respect to z (sigmoid derivative)
dy_pred_dz = y_pred * (1 - y_pred)

# 3. dz/dw and dz/db: z with respect to w and b
dz_dw = x  # dz/dw = x
dz_db = 1  # dz/db = 1 (bias contributes directly to z)

dL_dw = dloss_dy_pred * dy_pred_dz * dz_dw
dL_db = dloss_dy_pred * dy_pred_dz * dz_db
```

✓ 0.0s Python

```python
print(f"Manual Gradient of loss w.r.t weight (dw): {dL_dw}")
print(f"Manual Gradient of loss w.r.t bias (db): {dL_db}")
```

✓ 0.0s Python

```
Manual Gradient of loss w.r.t weight (dw): 6.691762447357178
Manual Gradient of loss w.r.t bias (db): 0.998770534992218
```



Computation Graph

Using autograd

```python
x = torch.tensor(6.7)
y = torch.tensor(0.0)
print(x)
print(y)
```

We are not using 'requires_grad = True' because we do not have to calculate the derivatives wrt to x & y

✓ 0.0s Python

```
tensor(6.7000)
tensor(0.)
```

```python
w = torch.tensor(1.0, requires_grad=True)
b = torch.tensor(0.0, requires_grad=True)
print(w)
print(b)
```

✓ 0.0s Python

```
tensor(1., requires_grad=True)
tensor(0., requires_grad=True)
```

STARTING FORWARD PASS :

```python
z = w*x + b
z
```

Eq.1.

✓ 0.0s Python

```
tensor(6.7000, grad_fn=<AddBackward0>)
```

```python
y_pred = torch.sigmoid(z)
y_pred
```

Eq.2.

✓ 0.0s Python

```
tensor(0.9988, grad_fn=<SigmoidBackward0>)
```

```python
loss = binary_cross_entropy_loss(y_pred, y)
loss
```

Eq.3.

✓ 0.0s Python

```
tensor(6.7012, grad_fn=<NegBackward0>)
```

```python
loss.backward()
```

✓ 0.0s Python

```python
loss.backward()
```

✓ 0.0s Python

```python
print(f"By using autograd function (dw): {w.grad}")
print(f"By using autograd function (db): {b.grad}")
```

✓ 0.0s Python

```
By using autograd function (dw): 6.6917619705200195
By using autograd function (db): 0.9987704753875732
```

— x — x — x — x — x — x —

Clearing Gradients

The key concept behind `autograd` in PyTorch is **gradient accumulation**. Each time you call `.backward()` on a tensor, the gradients for all tensors with `requires_grad=True` are added (accumulated) to their `.grad` attributes. This means if you run the backward pass multiple times without clearing the gradients, the values in `.grad` will keep increasing, reflecting the sum of all computed gradients.

Why does this happen?
This behavior is useful when training neural networks using mini-batches. You might want to accumulate gradients over several batches before updating the model parameters.

How to manage gradient accumulation:

- To avoid unwanted accumulation, always clear gradients before a new backward pass using `.zero_()` on the `.grad` attribute:

  ```python
  x.grad.zero_()
  ```

- Alternatively, use `optimizer.zero_grad()` when working with optimizers.

Summary:

- `.backward()` accumulates gradients in `.grad`.
- Always clear gradients before a new backward pass unless you intentionally want to accumulate them.

FIRST RUN

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

✓ 0.0s Python

```
tensor(2., requires_grad=True)
```

FP

```python
y = x ** 2
y
```

✓ 0.0s Python

```
tensor(4., grad_fn=<PowBackward0>)
```

BP

```python
y.backward()
```

✓ 0.0s Python

SECOND RUN

FP

```python
y = x ** 2
y
```

✓ 0.0s Python

```
tensor(4., grad_fn=<PowBackward0>)
```

BP

```python
y.backward()
```

✓ 0.0s Python

THIRD RUN

FP

```python
y = x ** 2
y
```

✓ 0.0s Python

```
tensor(4., grad_fn=<PowBackward0>)
```

BP

```python
y.backward()
```

✓ 0.0s Python

BP

```
1 y.backward()
✓ 0.0s                                   Python
```

GRAD

```
1 x.grad
✓ 0.0s                                   Python
```

tensor(4.)

BP

```
1 y.backward()
✓ 0.0s                                   Python
```

GRAD

```
1 x.grad
✓ 0.0s                                   Python
```

tensor(8.)

BP

```
1 y.backward()
✓ 0.0s                                   Python
```

GRAD

```
1 x.grad
✓ 0.0s                                   Python
```

tensor(12.)

```
x = torch.tensor(2.0, requires_grad=True)
x
✓ 0.0s                                   Python
```

tensor(2., requires_grad=True)

```
y = x ** 2
y
✓ 0.0s                                   Python
```

tensor(4., grad_fn=<PowBackward0>)

```
y.backward()
✓ 0.0s                                   Python
```

```
x.grad
✓ 0.0s                                   Python
```

tensor(4.)

```
x.grad.zero_()
✓ 0.0s                                   Python
```

tensor(0.)

Disabling Gradient Tracking

Sometimes, we may need to perform computations without tracking gradients or calculating derivatives.

Scenarios Where Disabling Gradient Tracking is Useful

- **Model Inference:**
  When making predictions with a trained model, gradients are not needed.
- **Model Evaluation:**
  During validation or testing phases, to save memory and computation.
- **Feature Extraction:**
  When using a model to extract features from data without updating weights.
- **Saving/Loading Model Outputs:**
  When storing intermediate results for later use.
- **Visualizations:**
  When plotting or analyzing outputs that do not require gradients.
- **Deployment:**
  In production environments where only forward passes are performed.

In such cases, PyTorch provides several ways to disable gradient tracking:

- **Set `requires_grad` to `False`:**
  You can turn off gradient tracking for a tensor by setting its `requires_grad` attribute to `False` using `requires_grad_(False)`.
- **Detach a tensor:**
  Use `.detach()` to create a new tensor that does not require gradients and is disconnected from the computation graph.
- **Use `torch.no_grad()` context:**
  Wrap your code inside a `with torch.no_grad():` block to temporarily disable gradient tracking for all operations within the block.

Disabling gradient tracking is useful for inference, evaluation, or any scenario where derivatives are not needed, as it reduces memory usage and speeds up computations.