Lec = 3

The Why

28 November 2024 16:52

$y = x^2 \rightarrow$ program $\rightarrow (x) \rightarrow x \rightarrow \frac{dy}{dx}$

$\downarrow$

$\frac{dy}{dx} = \boxed{2x} \rightarrow$ code

$x \rightarrow 2 \quad \frac{dy}{dx} = 4$

$x \rightarrow 3 \quad \frac{dy}{dx} = 6$

$y = x^2 \rightarrow \frac{dy}{dx} = 2x$

$z = \sin(y) \rightarrow x \rightarrow \frac{dz}{dx}$ ② >> ①

$\frac{dz}{dy} = \cos y$

$\boxed{\frac{dz}{dx}} = \frac{dz}{dy}\frac{dy}{dx}$

$\rightarrow 2x\cos(y) = \boxed{2x\cos(x^2)}$

$$\begin{cases} y = x^2 \\ z = \sin y \\ u = e^z \end{cases}$$

$\frac{du}{dx} = \frac{du}{dz}\frac{dz}{dy}\frac{dy}{dx}$

$\downarrow$

code

[Nested function $\rightarrow$ complex $\rightarrow$ derivative $\rightarrow$ code difficult.]

Nested function $\rightarrow$ derivative

$\downarrow$

[Deep learning]

Summary

Every derivative can be easily calculated from programming code of nested function.

But if the function become a little more complex then finding the derivative by writing code ourselves become very difficult.

For example, I take the first example $y = x^2$

it derivative is easy to calculate by code.

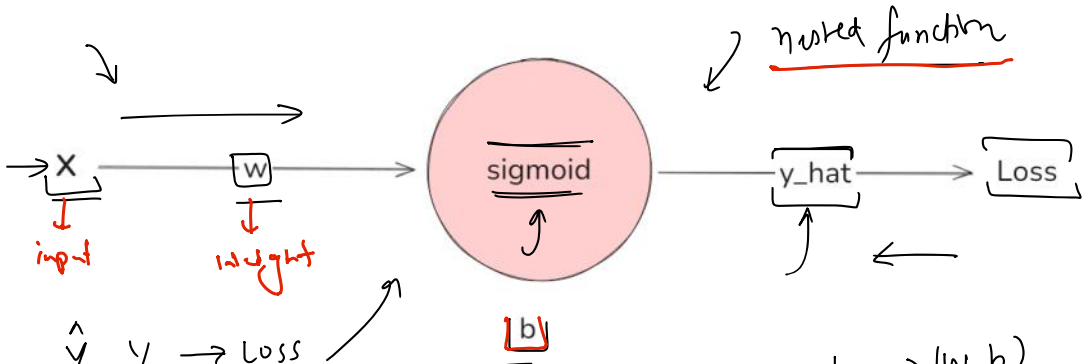
but when we increase a little complexity, like 2-3 nested functions, then finding their derivative become very difficult.

That is why in which field we use Autograd

↳ it is very easily and automatically calculated the derivative of more complex nested functions.

$\frac{\partial L}{\partial w}$, $\frac{\partial L}{\partial b}$

cgpa	placed
9.11	1
8.9	1
7	0
6.56	1
4.56	0



Training process

1. Forward pass - Compute the output of the network given an input.
2. Calculate loss - Calculate the loss function to quantify the error.
3. Backward pass - Compute gradients of the loss with respect to the parameters.
4. Update gradients - Adjust the parameters using an optimization algorithm (e.g., gradient descent).

Forward Pass Computation

1. Linear Transformation:

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

2. Activation (Sigmoid Function):

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

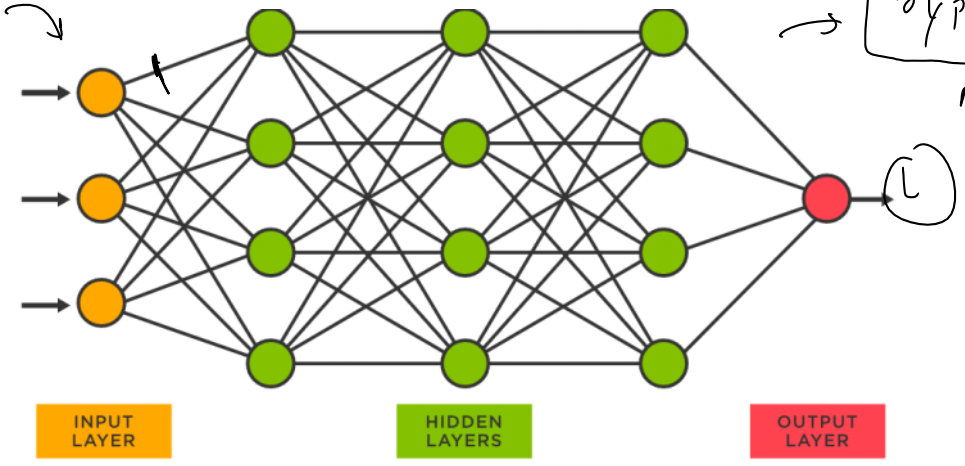
3. Loss Function (Binary Cross-Entropy Loss):

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

chain rule

$\frac{\partial L}{\partial b}$ $\frac{\partial L}{\partial w}$

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



$\frac{\partial L}{\partial w}$

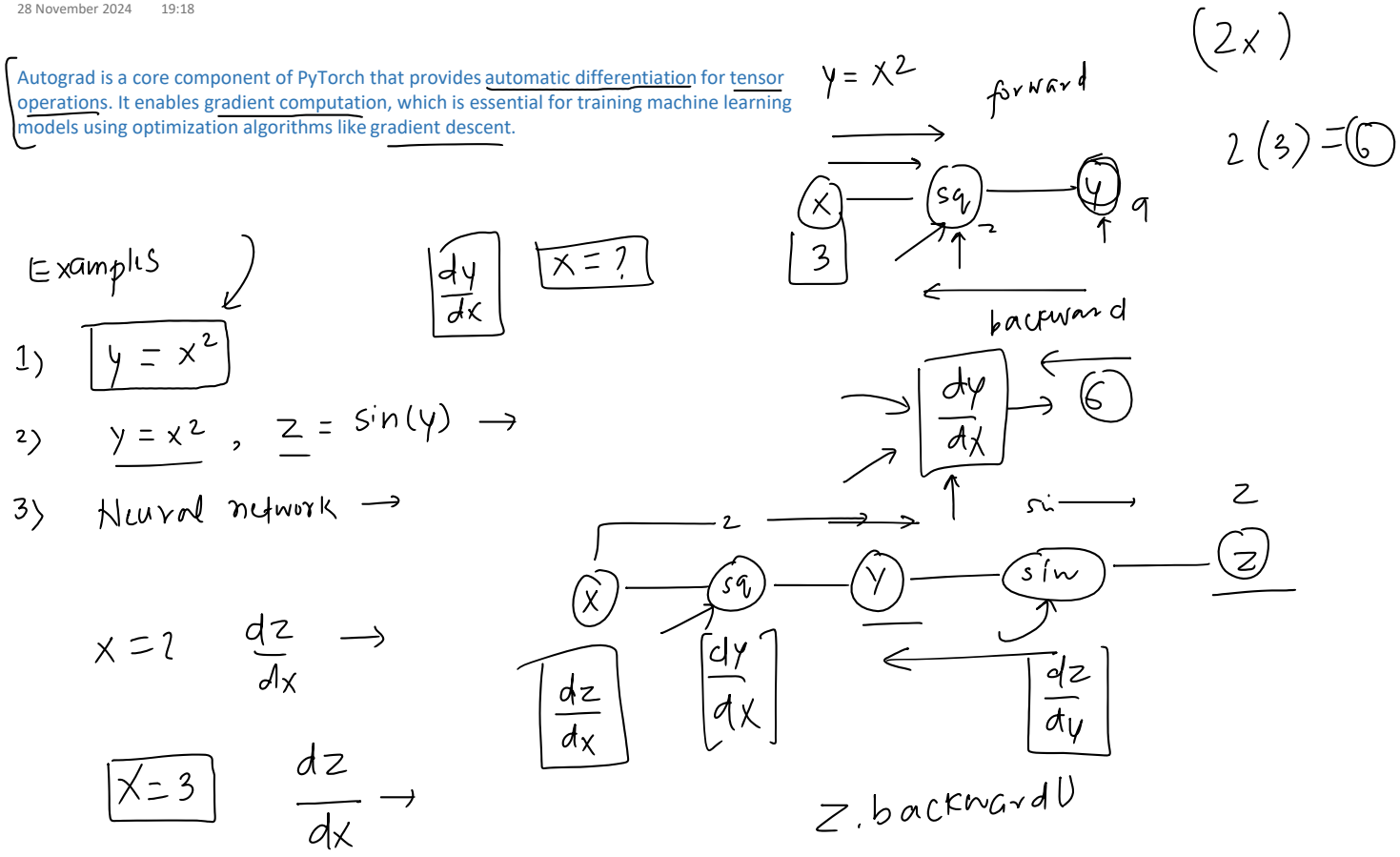
Autograd

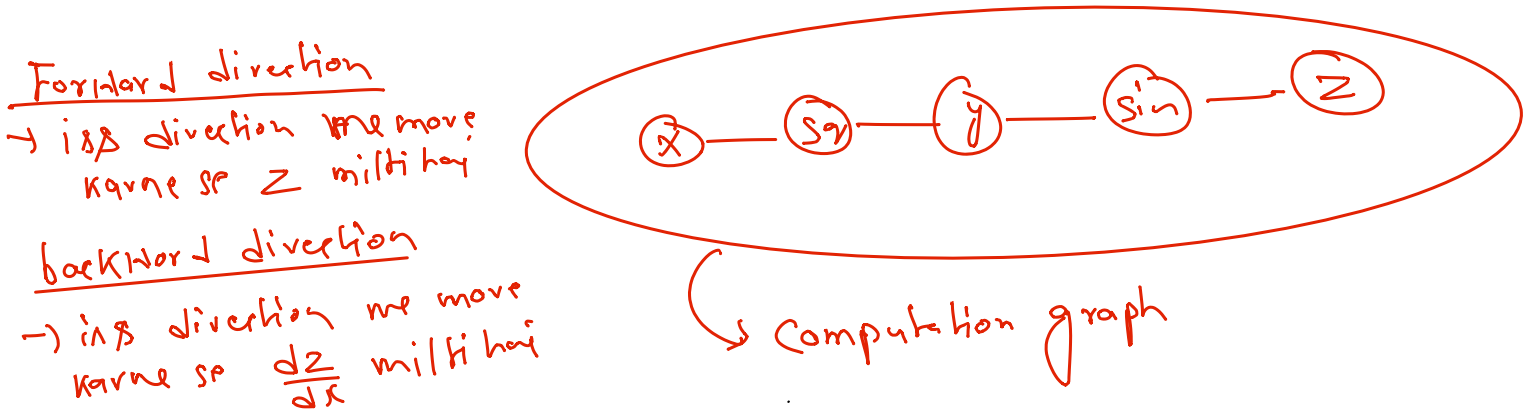
NN → nested function
↳ derivatives manually impossible

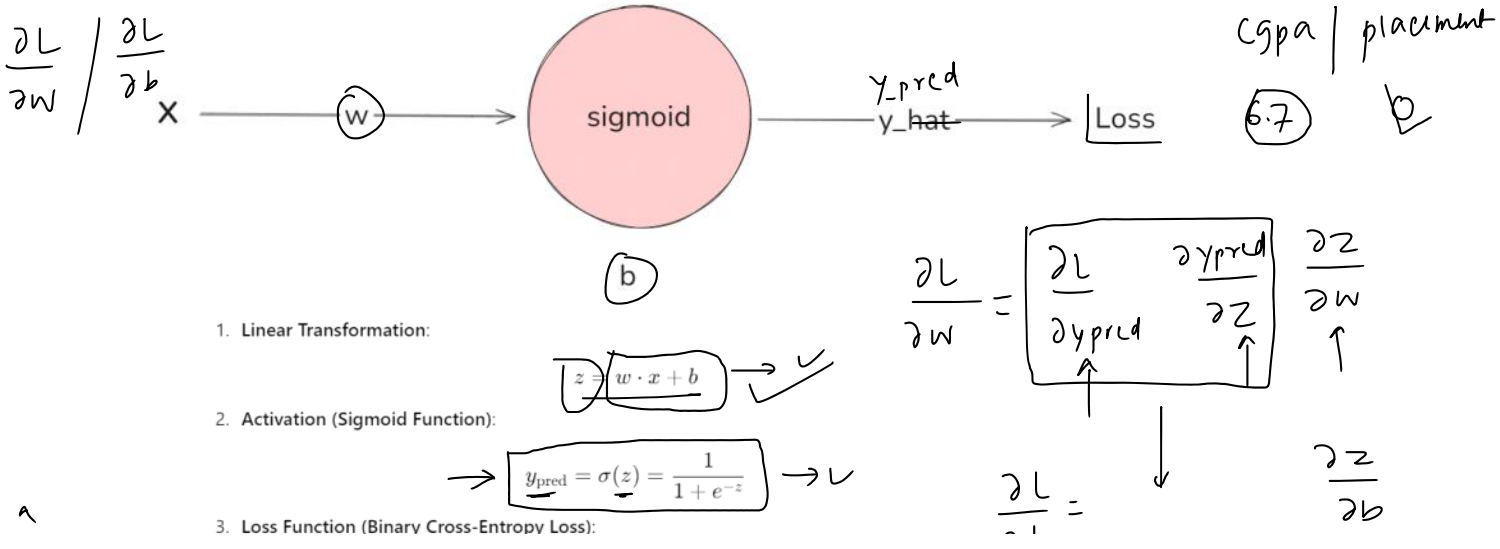
What is Autograd

28 November 2024 19:18

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.







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

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

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

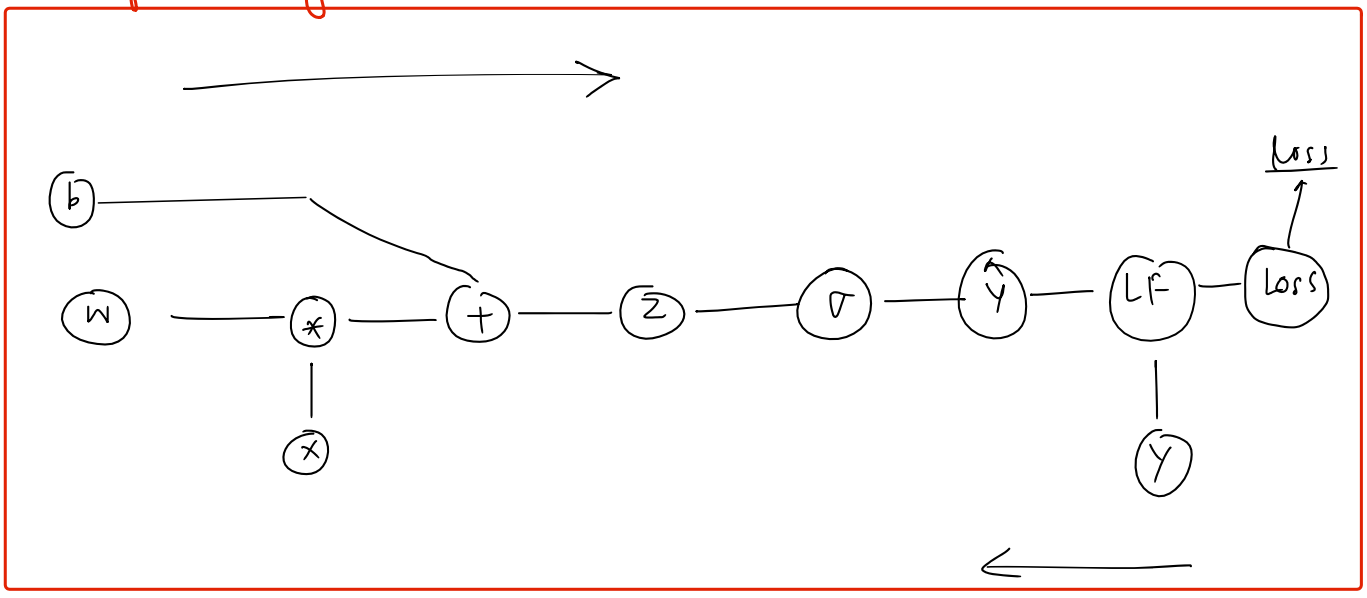
$\hat{y}$ y_pred

$\frac{\partial L}{\partial b} =$ $\frac{\partial z}{\partial b}$

$\frac{\partial L}{\partial y\_pred} = \frac{(\hat{y} - y)}{\hat{y}(1-\hat{y})}$ $\frac{\partial \hat{y}}{\partial z} = \hat{y}(1-\hat{y})$ $\frac{\partial z}{\partial w} = x$ $\frac{\partial z}{\partial b} = 1$

$$\frac{\partial L}{\partial w} = (\hat{y} - y) * x \qquad \frac{\partial L}{\partial b} = (\hat{y} - y) * 1$$

Computation graph



$$x = [x_1 \; x_2 \; x_3] = [1 \; 2 \; 3]$$

$y = x^2.\text{mean}()$ multivar func.

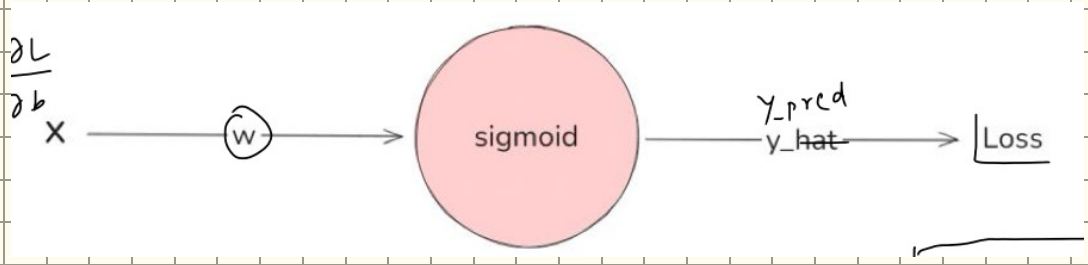
$$y = \frac{x_1^2 + x_2^2 + x_3^3}{3}$$

$y = f(x_1, x_2, x_3)$ $\frac{dy}{dx_1} = \frac{2x_1}{3}$ $\frac{\partial y}{dx_2} = \frac{2x_2}{3}$ $\frac{\partial y}{\partial x_3} = \frac{2x_3}{3}$

Summary

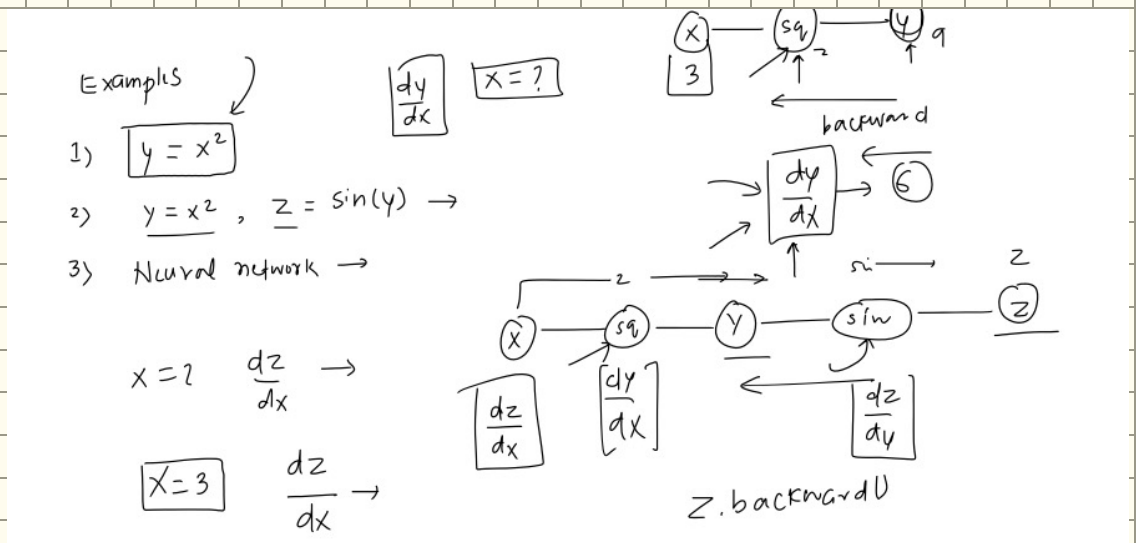
* How Autograd works

The training process of autograd happens in 4 steps.
First, Autograd create a computation graph.



① Forward pass :- In this process, the value of z is calculated, then the value of ypred is calculated, and at the end the value of Loss is calculated.

② after that come backword pass :- In this process, the derivative is calculated.



Autograd Kya Hai? (What is Autograd?)

Autograd PyTorch ka automatic differentiation engine hai. Neural networks mein, humein loss function ke respect mein weights aur biases ka gradient (derivative) nikalna padta hai taki hum unhe update kar sakein. Autograd yeh saara kaam hamare liye automatically kar deta hai.

Autograd Kaise Use Karein? (How to Use Autograd?)

1. **Gradient Tracking Shuru Karna (requires_grad=True)**: Jab aap ek tensor banate hain, agar aapko uske respect mein gradient chahiye, to requires_grad=True set karein.
    - x = torch.tensor(3.0, requires_grad=True)
    - PyTorch ab x par hone wale saare operations ka ek graph banayega.
2. **Operations Track Karna (grad_fn)**: Jab aap requires_grad=True wale tensor par koi operation karte hain, to naye bane tensor mein grad_fn attribute jud jaata hai. Yeh batata hai ki yeh tensor kis operation se bana hai (jaise PowBackward0 power operation ke liye).
    - y = x**2
    - y will have grad_fn=
3. **Gradients Calculate Karna (.backward())**: Final output (jaise loss ya z) par .backward() call karne se PyTorch pure graph mein piche jaata hai (backpropagation) aur har requires_grad=True wale tensor ke liye gradient calculate karta hai.
    - z.backward()
4. **Gradient Access Karna (.grad)**: .backward() call karne ke baad, aap original tensor ke .grad attribute se uska gradient dekh sakte hain.
    - x.grad

Manual Calculation vs. Autograd

Aapne dekha ki binary cross-entropy loss ke liye gradients (derivatives) manually nikalna (using chain rule) kitna lamba kaam hai.

- **Manual**: dL_dw = dloss_dy_pred * dy_pred_dz * dz_dw
- **Autograd**: Sirf loss.backward() call karo, aur PyTorch w.grad aur b.grad mein automatically correct values daal dega.

Yeh Autograd ka sabse bada fayda hai, especially jab models bahut bade aur complex ho jaate hain.

Gradients ko Manage Karna (Managing Gradients)

- **Gradients Accumulate Hote Hain**: PyTorch by default gradients ko add karta jaata hai. Har baar .backward() call karne par, naye gradients purane gradients mein jud jaate hain. Training loop mein yeh galat results de sakta hai.
- **Gradients Clear Karna (.grad.zero_())**: Isliye, har training step (iteration) mein naye gradients calculate karne se pehle, purane gradients ko zero karna zaroori hai.
    - x.grad.zero_()

Gradient Tracking ko Kab aur Kaise Rokein?

Kabhi-kabhi humein gradient tracking ki zaroorat nahi hoti, jaise:

- Model ko evaluate karte waqt (inference).
- Jab hum sirf forward pass karna chahte hain.

Gradient tracking rokne se memory kam use hoti hai aur computations fast hote hain. Iske teen tarike hain:

1. **x.requires_grad_(False)**: Yeh ek in-place function hai jo tensor ke liye gradient tracking hamesha ke liye band kar deta hai.
2. **z = x.detach()**: Yeh ek naya tensor z banata hai jo x ke jaisa hi hota hai, lekin computation graph se alag (detached) hota hai. x par gradient tracking chalti rehti hai, lekin z par nahi.

3. **with torch.no_grad():**: Yeh ek context manager hai. Is block ke andar kiye gaye saare operations ke liye gradient tracking temporarily band ho jaati hai.

Error Explanation (RuntimeError): Jab aap y.backward() aise tensor par call karte hain jiske liye gradient tracking on nahi thi (yaani uska requires_grad False tha aur koi grad_fn nahi tha), to PyTorch yeh error deta hai: RuntimeError: element 0 of tensors does not require grad and does not have a grad_fn. Yeh error upar bataye gaye teeno methods ka use karne par aayega, kyunki unka kaam hi gradient tracking ko rokna hai.