#1 PyTorch Fundamentals

AI Training

PyTorch & Training Performance

Why Performance matters

- **Bigger models = more computation**
- CPUs have **limited cores** (often 8–16), slowed further by Python's **Global Interpreter Lock (GIL)**
- Even a small MLP (e.g., 100 hidden units) can require **~80,000 parameters** to train — scaling up quickly becomes **infeasible** on a single CPU

The GPU Advantage:

- GPUs = **Mini supercomputers** inside your machine
- Designed for **massively parallel computation**
- Offer **better performance per cost** compared to top-end CPUs
- Perfect for **deep learning workloads** where thousands of operations run simultaneously

What is Pytorch?

- **Open-source ML framework** (released in 2016) developed by **Facebook AI Research (FAIR)** + community contributions
- Widely used in academia & industry — e.g., **Tesla Autopilot, Uber's Pyro, Hugging Face Transformers**
- Designed for **deep learning** with a **Python-first interface**

Pytorch: Performance & Device Support

- Runs on CPU, GPU, and XLA devices (e.g., TPUs)
- Best performance on CUDA (NVIDIA) and ROCm (AMD) GPUs
- Built on the Torch library foundation

PyTorch: Core Concepts

- **Computation Graph**:
  - Built **dynamically** during execution (imperative style)
  - Nodes = operations; edges = data flow
- **Tensors**:
  - Generalization of scalars, vectors, matrices
  - Similar to NumPy arrays but **GPU-ready** and **autograd-enabled**

PyTorch Auto Grad

Recap: MLP Learning Procedure

To compute the output of a Multilayer Perceptron (MLP), we follow a simple 3-step learning process:

1. **Forward Propagation**
   Feed input data through the network to generate predictions.
2. **Compute Loss**
   Compare the predictions with true labels using a loss function.
3. **Backpropagation & Update**
   Calculate gradients and adjust weights and biases to reduce loss.

Once trained over multiple epochs, we:

- Use forward propagation to make predictions
- Apply a threshold to convert outputs to **one-hot encoded** class labels

Recap: Forward Propagation in MLP

To generate predictions, we just forward-propagate the input features through the network:

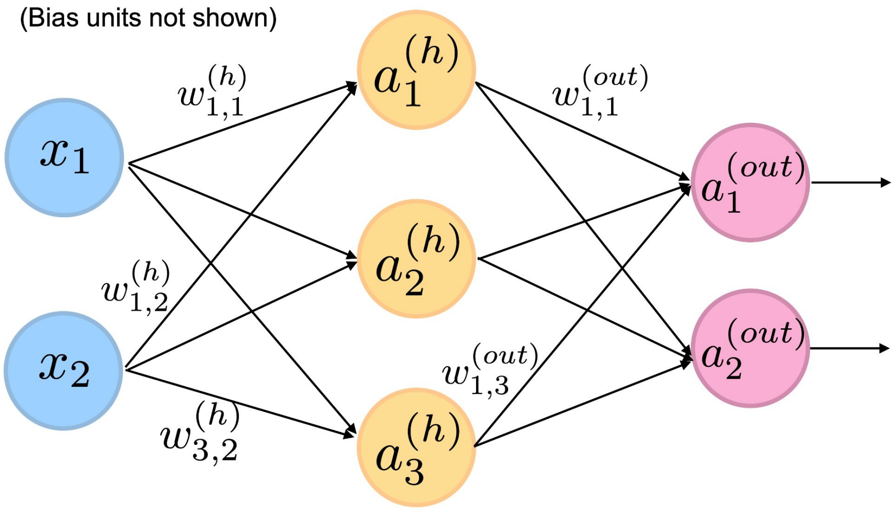


Figure 5.4: Forward-propagating the input features of an NN

Recap: Backpropagation & the Chain Rule

- A **computationally efficient** method for computing **partial derivatives**
- Helps optimize complex, **non-convex loss functions** in multilayer neural networks

Chain Rule Refresher

- The chain rule in calculus is used to compute the derivative of nested functions: $f(g(x))$

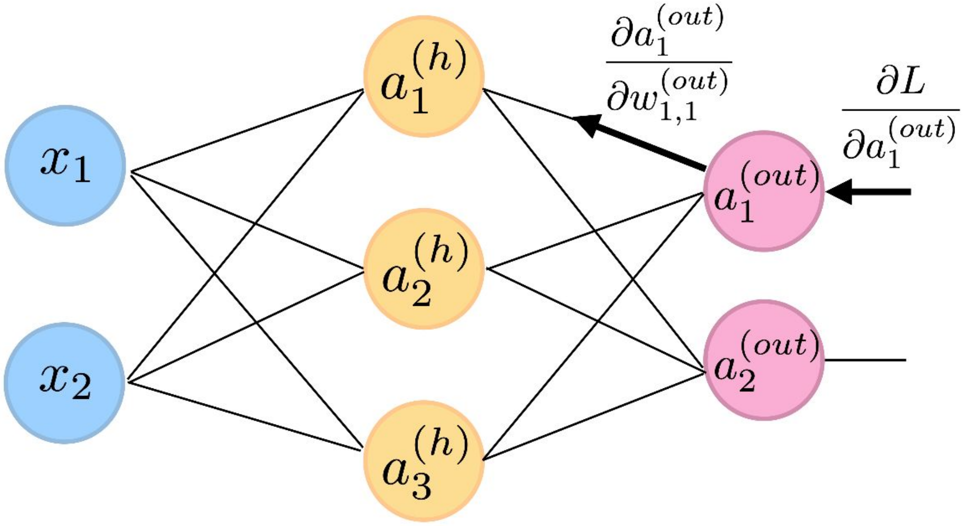
$$\frac{d}{dx}[f(g(x))] = \frac{df}{dg} \cdot \frac{dg}{dx}$$

- This principle generalizes to deeper function compositions: $F(x) = f(g(h(u(v(x)))))$

$$\frac{dF}{dx} = \frac{d}{dx}F(x) = \frac{d}{dx}f(g(h(u(v(x))))) = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx}$$

Recap: Backward Propagation in MLPs

- Applies the **chain rule** in reverse—from **output layer to input**
- Gradually computes the **gradient of the loss** with respect to each **weight (and bias)**
- Enables **efficient training** of deep neural networks via **gradient descent**



Gradient for output layer weight:

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial w_{1,1}^{(out)}}$$

Figure 5.5: Backpropagating the error of an NN

Recap: Backward Propagation in MLPs

```
class Linear:
    def __init__(self):
        ...
    def forward(self, x):
        ...
    def backward(self, grad_output):
        grad_input = np.dot(grad_output, self.weight.T)

        self.grad_weight[...] = np.dot(self.input.T, grad_output)
        self.grad_bias[...] = np.mean(grad_output, axis=0)
        return grad_input
```

- Deriving gradients manually (and from scratch) is **hard** and it's **easy to make mistakes**
- **How can we simplify and systematize our approach?**

PyTorch Computational Graph

Understanding the computational graph

- **Core idea**: PyTorch builds a **computation graph** to track how tensors are transformed from input to output.
- This graph records **operations** performed on tensors and their **dependencies**.
- Each **node** is an operation, which applies a function to its input tensor and returns an output
- PyTorch then uses this graph to automatically compute **gradients** during backpropagation.

Think of it as a **map** that traces every mathematical step from start to finish.

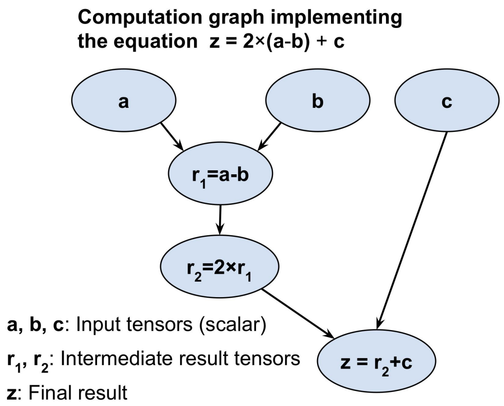


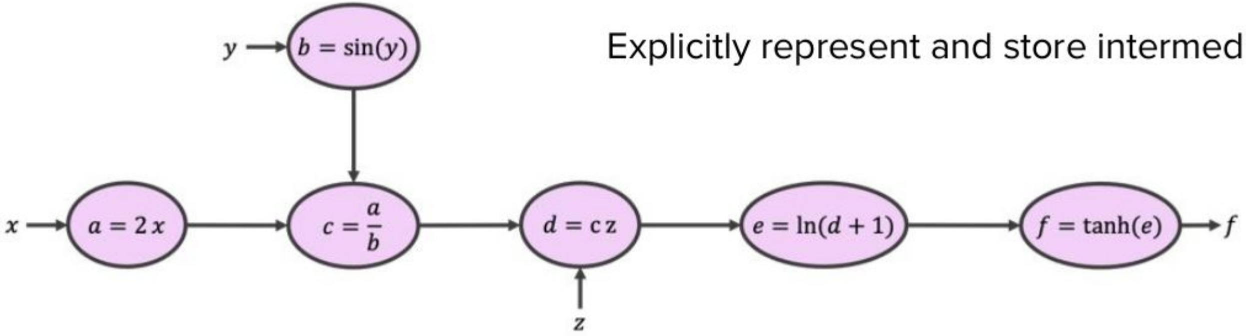
Figure 6.1: How a computation graph works

Computational Graph (forward)

$$f(x, y, z) = \tanh\left(\ln\left[1 + z\frac{2x}{sin(y)}\right]\right)$$

Computational Graph (forward)

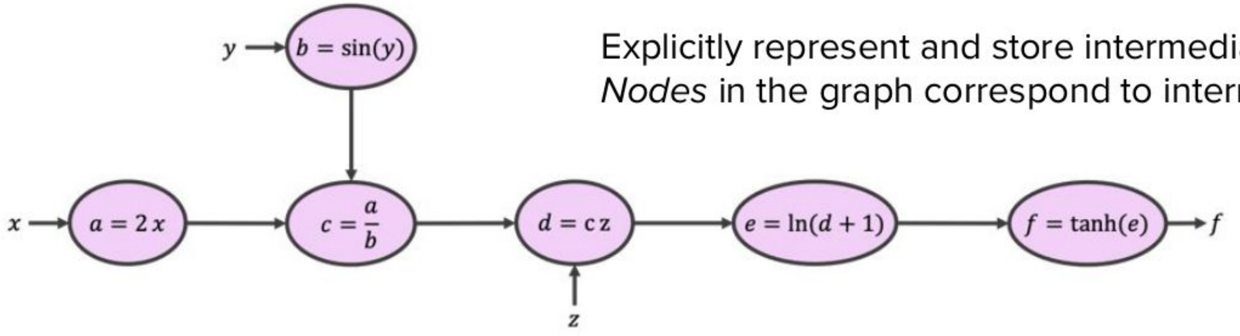
$$f(x, y, z) = \tanh\left(\ln\left[1 + z\frac{2x}{sin(y)}\right]\right)$$



Explicitly represent and store intermediate variables *a*,*b*,*c*,*d*,*e*

Computational Graph (forward)

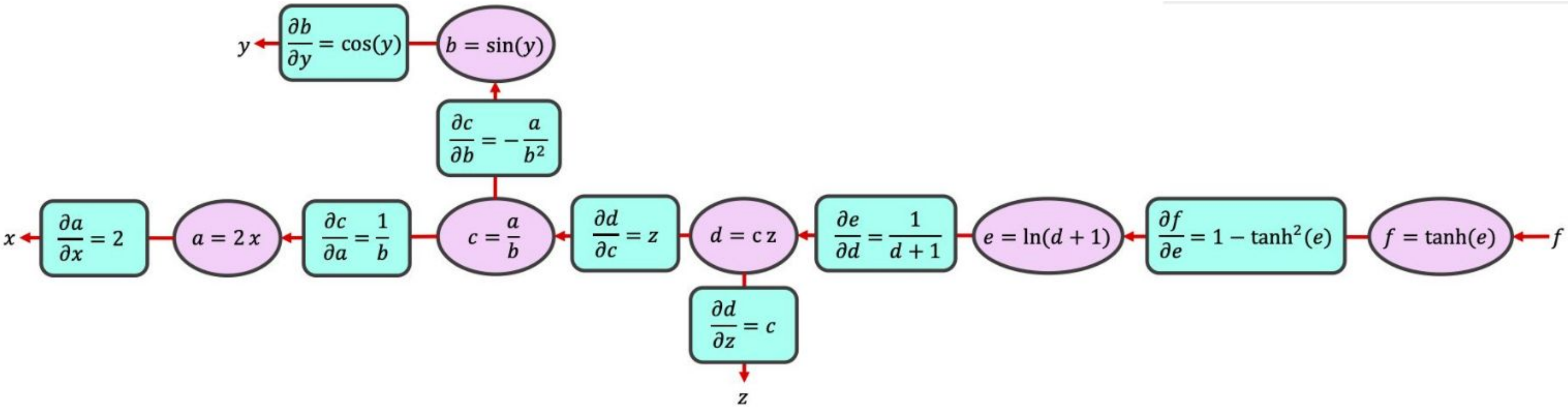
$$f(x,y,z) = \tanh\left(\ln\left[1 + z\frac{2x}{sin(y)}\right]\right)$$



Explicitly represent and store intermediate variables *a*,*b*,*c*,*d*,*e*.
Nodes in the graph correspond to intermediate variables.

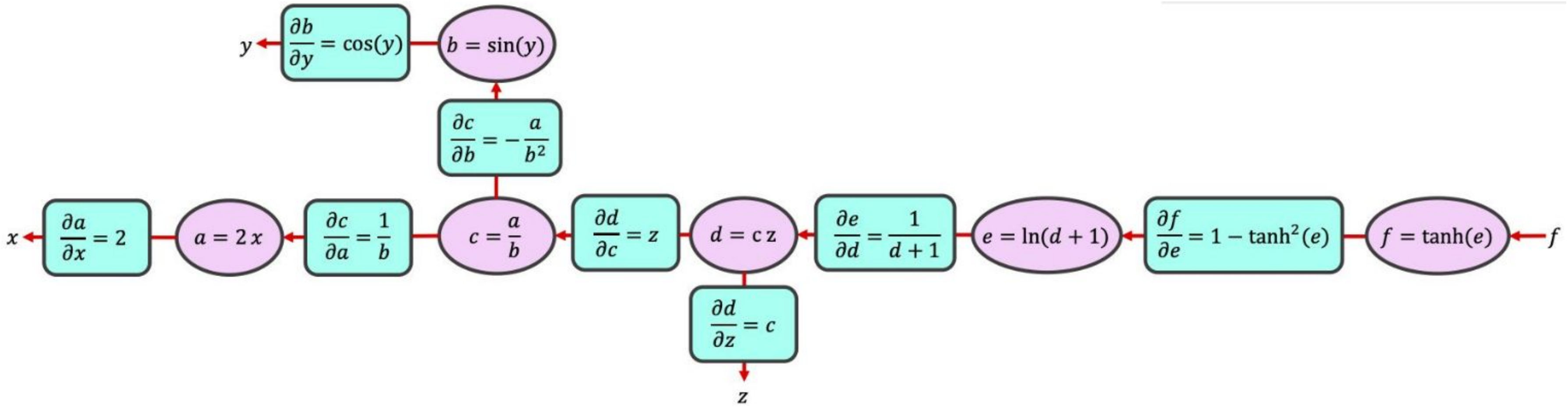
Computational Graph (backward)

Starting from the top, pass backward. Each edge stores partial derivative of the head of the edge with respect to the tail.



Computational Graph (backward)

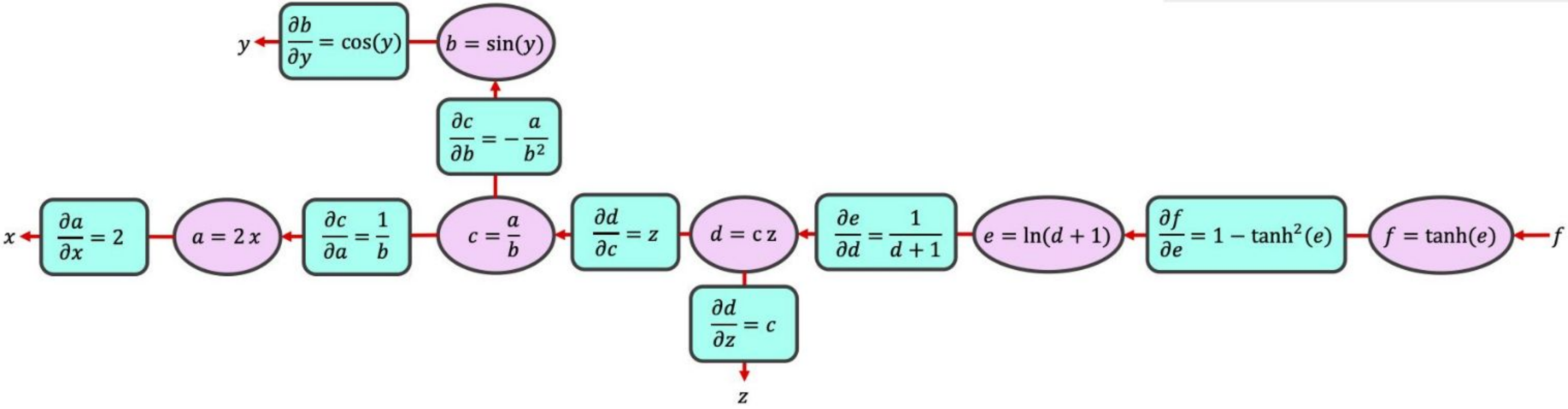
Starting from the top, pass backward. Each edge stores partial derivative of the head of the edge with respect to the tail.



Conveniently, the partial derivatives can often be expressed using the intermediate variables calculated in the forward pass (a,b,c,d,e).

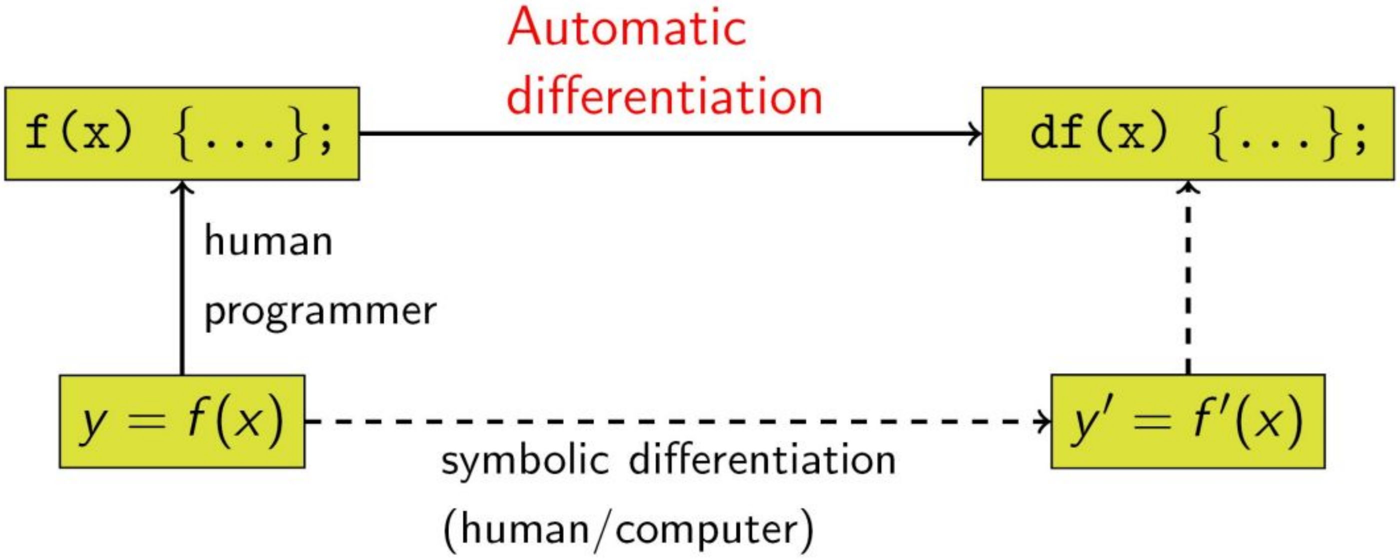
Computational Graph (gradients)

Starting from the top, pass backward. Each edge stores partial derivative of the head of the edge with respect to the tail.



$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{\partial a}\frac{\partial a}{\partial x} = (1 - \tanh^2(e)) \cdot \frac{1}{d+1} \cdot z \cdot \frac{1}{b} \cdot 2$$

The magic of automatic differentiation



Building the computational graph

A data structure for storing intermediate values and partial derivatives needed to compute gradients.

- Node **v** represents variable
  - Stores value
  - Gradient
  - The function that created the node
- Directed edge from v to u represents the partial derivative of u w.r.t. v
- To compute the gradient $\partial L/\partial v$, find the unique path from L to v and multiply the edge weights, where L is the overall loss

Building the computational graph

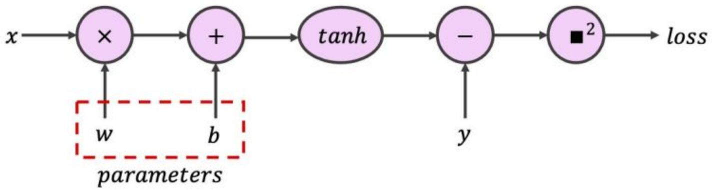
When we perform operations on PyTorch Tensors, PyTorch does not simply calculate the output

- Instead, **each operation** is added to the **computational graph**
- PyTorch can then do a forward and backward pass through the graph, **storing necessary intermediate variables**, and yield any **gradients** we need

Building the computational graph

- Often only some parameters are trainable and require gradients.
- We indicate tensors that require gradients by setting **`requires_grad=True`**

$$(y - \tanh(wx + b))^2$$



Building the computational graph

PyTorch can keep adding to the graph as your code winds through functions and classes

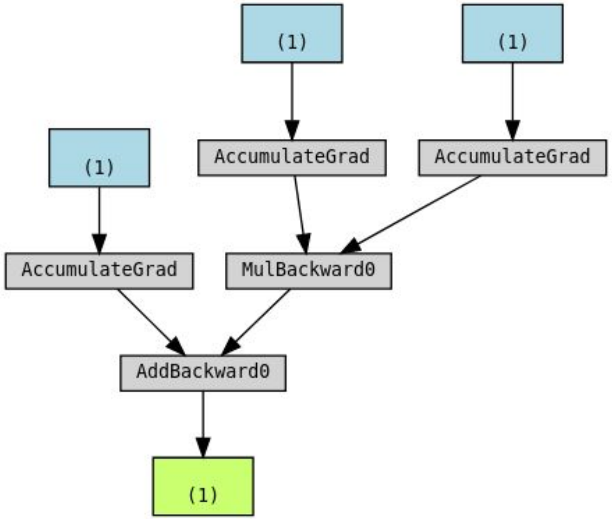
In essence, you write code to compute the loss L; AutoGrad does the rest

$$(y - \tanh(wx + b))^2$$

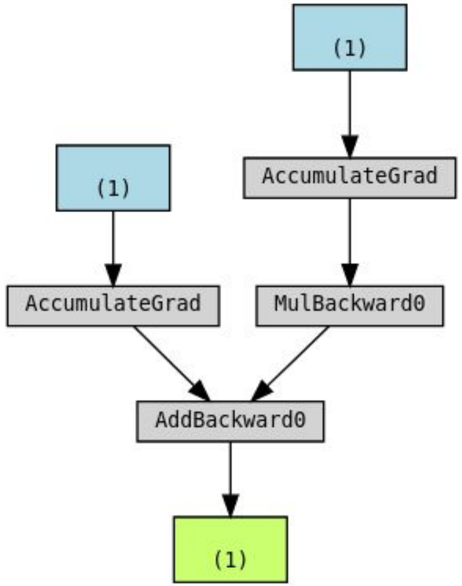
TYU

If b and w are trainable parameters and x is a feature vector, which computation graph best represents the operation yhat = b + w * x?

A)

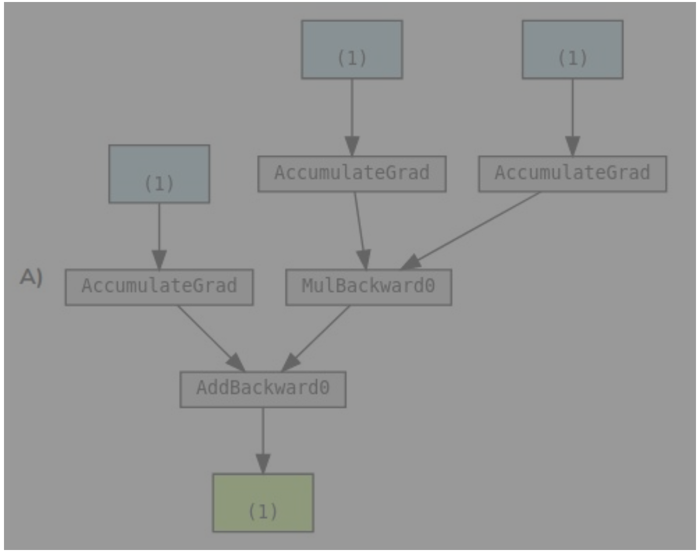


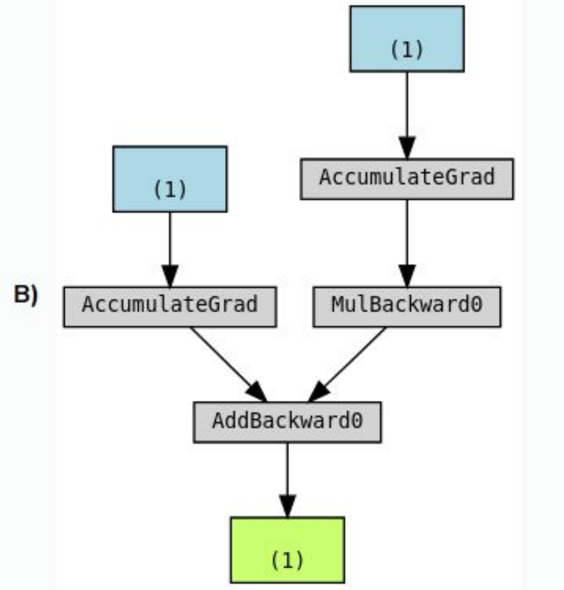
B)



TYU

If b and w are trainable parameters and x is a feature vector, which computation graph best represents the operation yhat = b + w * x?





Learning PyTorch: Our Roadmap

1. **Tensors & Basics**
    - Create and manipulate tensors
    - Understand PyTorch's programming model
2. **Working with Data**
    - Load datasets using `torch.utils.data`
    - Efficiently iterate through data
    - Explore built-in datasets in `torch.utils.data.Dataset`
3. **Building Neural Networks**
    - Introduce the `torch.nn` module
    - Compose and train models
    - Save trained models for future evaluation

Practice

Neural Network with PyTorch

Parallelizing Neural Network Training with PyTorch

First Steps with PyTorch

Any questions?

THANKS