CSC 561: Neural Networks and Deep Learning

Backpropagation

Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2024

PyTorch Autograd explained

https://www.youtube.com/watch?v=MswxJw-8PvE

Show me the code (PyTorch)

```
import torch

x = torch.tensor([-1.,-2.], requires_grad=True)
w = torch.tensor([2.,-3.], requires_grad=True)
b = torch.tensor(-3., requires_grad=True)

# forward pass
f = 1 / (1 + torch.exp(-(w@x + b)))

# backward pass
f.backward()

print(w.grad, x.grad, b.grad)
```

```
tensor([-0.1966, -0.3932]) tensor([ 0.3932, -0.5898]) tensor(0.1966)
```


Equivalent computational graph

```
import torch

x = torch.tensor([-1.,-2.], requires_grad=True)
w = torch.tensor([2.,-3.], requires_grad=True)
b = torch.tensor(-3., requires_grad=True)

# forward pass
f = torch.sigmoid(w @ x + b)

# backward pass
f.backward()

print(w.grad, x.grad, b.grad)
```

```
tensor([-0.1966, -0.3932]) tensor([ 0.3932, -0.5898]) tensor(0.1966)
```

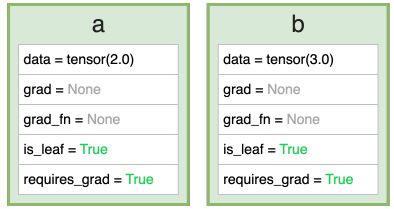
Computational graphs

- Pseudo-code for **forward** and **backward** passes

```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```


a = torch.tensor(2.0, requires_grad=True)

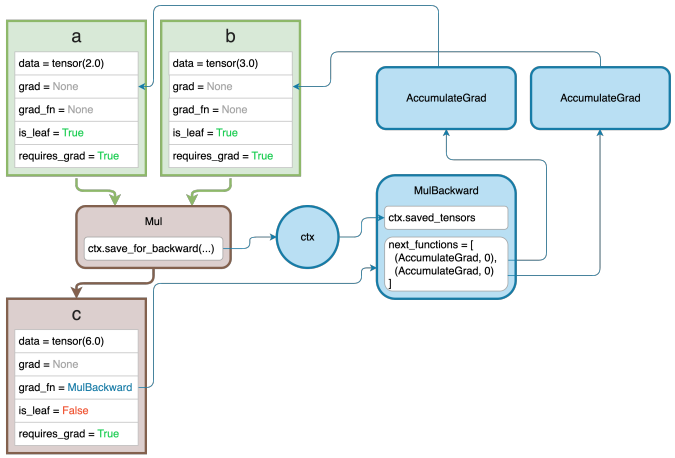
b = torch.tensor(3.0, requires_grad=True)



a = torch.tensor(2.0, requires_grad=True)

b = torch.tensor(3.0, requires_grad=True)

c = a * b

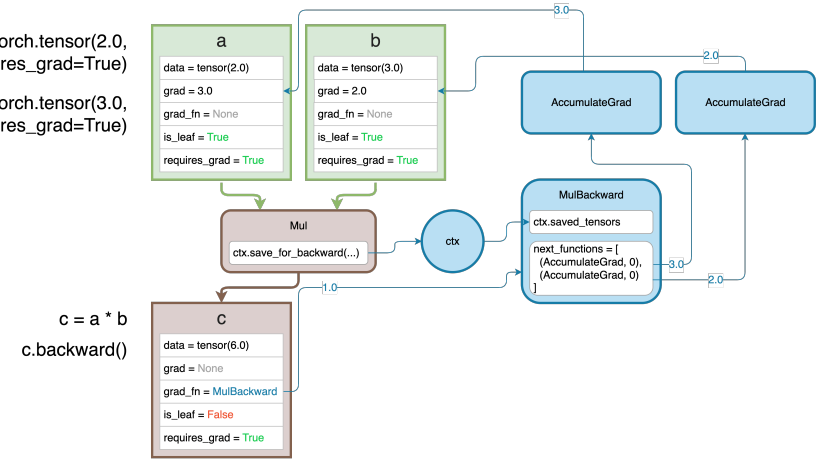


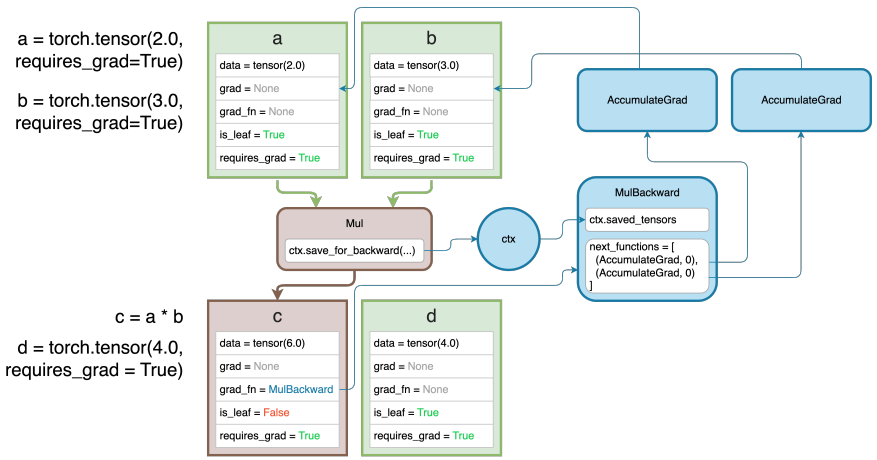
a = torch.tensor(2.0, requires_grad=True)

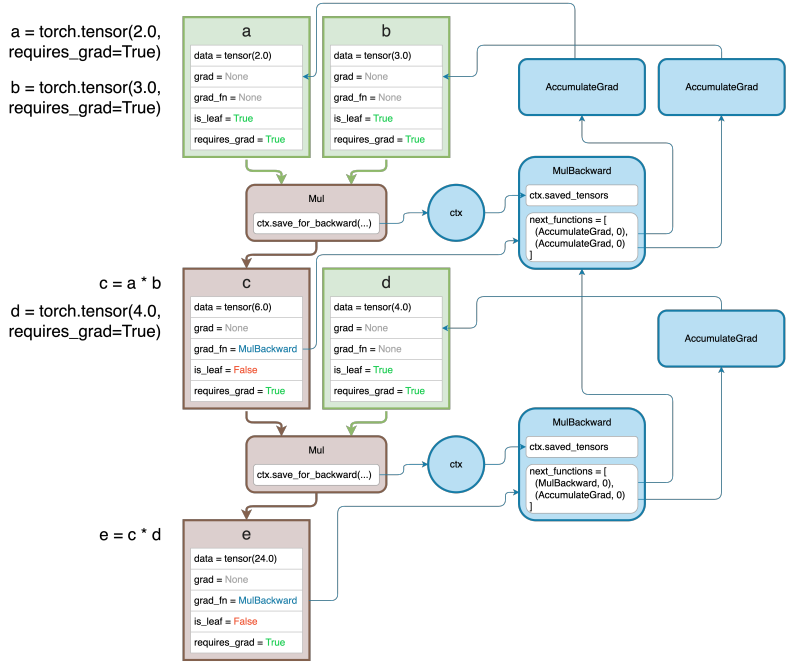
b = torch.tensor(3.0, requires_grad=True)

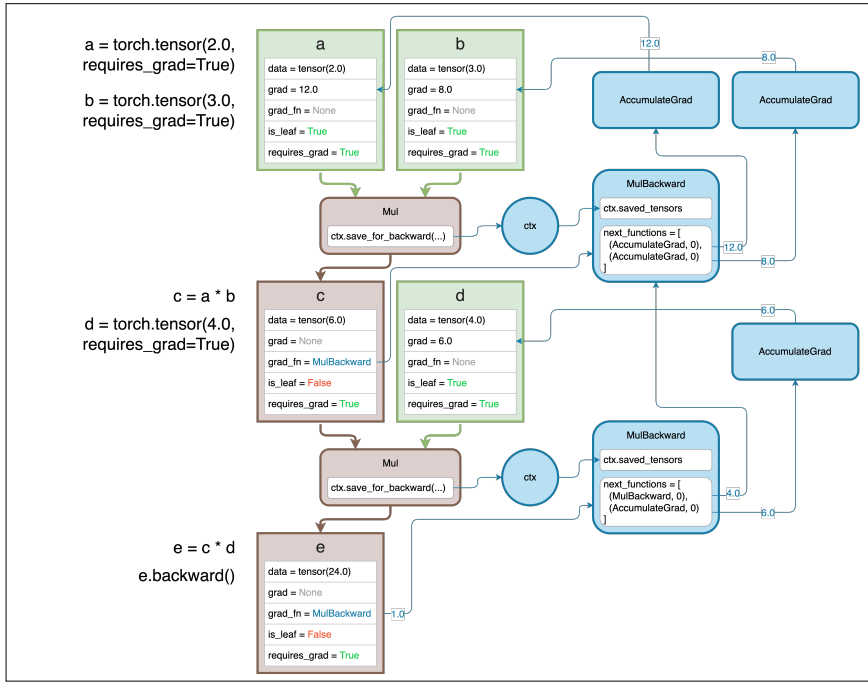
c = a * b

c.backward()









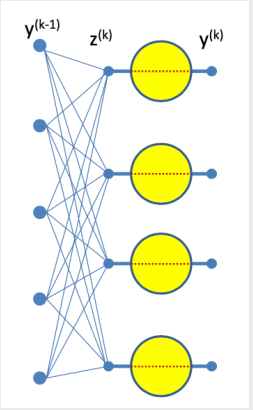
Defining new functions

```
class LegendrePolynomial3(torch.autograd.Function):
    '''
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    '''

    @staticmethod
    def forward(ctx, input):
        '''
        In the forward pass we receive a Tensor containing the input and return
        a Tensor containing the output. ctx is a context object that can be used
        to stash information for backward computation. You can cache arbitrary
        objects for use in the backward pass using the ctx.save_for_backward method.
        '''
        ctx.save_for_backward(input)
        return 0.5 * (5 * input ** 3 - 3 * input)

    @staticmethod
    def backward(ctx, grad_output):
        '''
        In the backward pass we receive a Tensor containing the gradient of the loss
        with respect to the output, and we need to compute the gradient of the loss
        with respect to the input.
        '''
        input, = ctx.saved_tensors
        return grad_output * 1.5 * (5 * input ** 2 - 1)
```

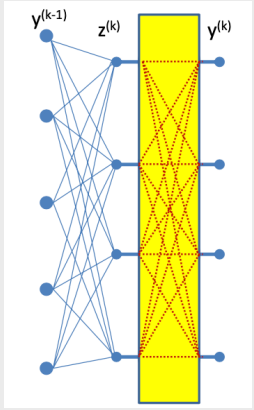
https://pytorch.org/tutorials/beginner/examples_autograd/two_layer_net_custom_function.html

Vectorized operations

Scalar and Vector Activations

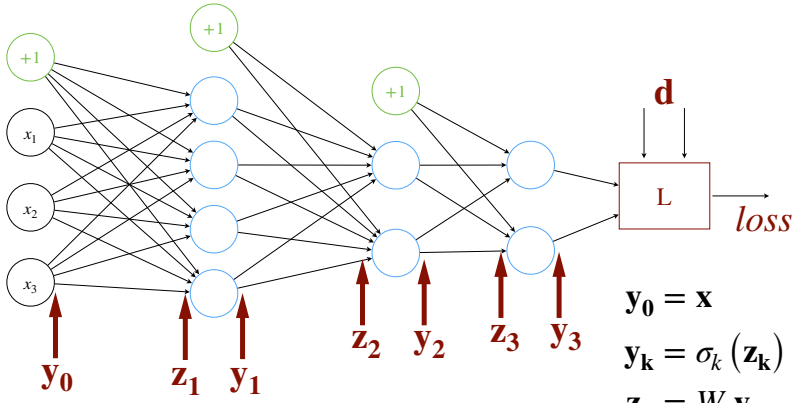


$$\frac{dL}{dz_i^{(k)}} = \frac{dL}{dy_i^{(k)}} \frac{dy_i^{(k)}}{dz_i^{(k)}}$$



$$\frac{dL}{dz_i^{(k)}} = \sum_j \frac{dL}{dy_j^{(k)}} \frac{dy_j^{(k)}}{dz_i^{(k)}}$$

Forward pass



$$\mathbf{y_0} = \mathbf{x}$$
$$\mathbf{y_k} = \sigma_k(\mathbf{z_k})$$
$$\mathbf{z_k} = W_k \mathbf{y_{k-1}} + \mathbf{b_k}$$

$$loss = L\left(\sigma_3\left(W_3\sigma_2\left(W_2\sigma_1\left(W_1\mathbf{y_0} + \mathbf{b_1}\right) + \mathbf{b_2}\right) + \mathbf{b_3}\right), \mathbf{d}\right)$$

Vector calculus (derivatives)

- Scalar function of a vector argument
  - $y = f(\mathbf{z})$ $\quad \Delta y = \boxed{\nabla_{\mathbf{z}} y} \Delta \mathbf{z}$ (row vector)
  - transpose of the derivative is the gradient of y w.r.t. $\mathbf{z}$
- Vector function of a vector argument
  - $\mathbf{y} = f(\mathbf{z})$ $\quad \Delta \mathbf{y} = \boxed{\nabla_{\mathbf{z}} y} \Delta \mathbf{z}$ (matrix $m \times n$)
  - derivative is called the Jacobian of y w.r.t. $\mathbf{z}$

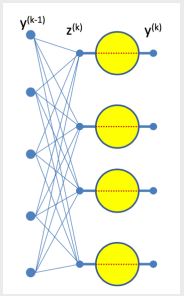
$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = f\left(\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix}\right)$$

Jacobian matrix

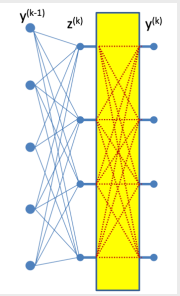
$$\mathbf{y} = f(\mathbf{z})$$

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = f\left(\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix}\right) \qquad J_{\mathbf{y}}(\mathbf{z}) = \begin{bmatrix} \frac{\partial y_1}{\partial z_1} & \frac{\partial y_1}{\partial z_2} & \cdots & \frac{\partial y_1}{\partial z_n} \\ \frac{\partial y_2}{\partial z_1} & \frac{\partial y_2}{\partial z_2} & \cdots & \frac{\partial y_2}{\partial z_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial z_1} & \frac{\partial y_m}{\partial z_2} & \cdots & \frac{\partial y_m}{\partial z_n} \end{bmatrix}$$

Jacobian matrices for activations



$$J_{\mathbf{y}}(\mathbf{z}) = \begin{bmatrix} \frac{\partial y_1}{\partial z_1} & 0 & \cdots & 0 \\ 0 & \frac{\partial y_2}{\partial z_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\partial y_m}{\partial z_n} \end{bmatrix}$$

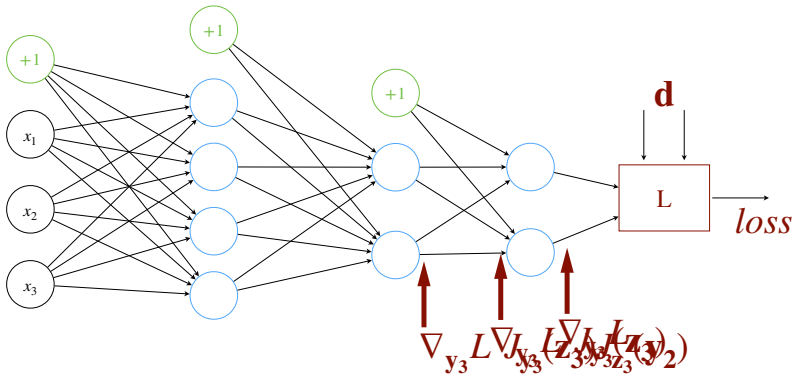


$$J_{\mathbf{y}}(\mathbf{z}) = \begin{bmatrix} \frac{\partial y_1}{\partial z_1} & \frac{\partial y_1}{\partial z_2} & \cdots & \frac{\partial y_1}{\partial z_n} \\ \frac{\partial y_2}{\partial z_1} & \frac{\partial y_2}{\partial z_2} & \cdots & \frac{\partial y_2}{\partial z_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial z_1} & \frac{\partial y_m}{\partial z_2} & \cdots & \frac{\partial y_m}{\partial z_n} \end{bmatrix}$$

Vector calculus (chain rule)

- Nested functions
  - $\mathbf{y} = f\left(g(\mathbf{z})\right)$ $\qquad J_{\mathbf{y}}(\mathbf{z}) = J_f(g) J_g(\mathbf{z})$
  - **note**: the derivative of the outer function comes first
- Jacobians can be combined with gradients
  - e.g., scalar functions of vector inputs

Backward pass



$\nabla_{\mathbf{y}_3} L \nabla_{\mathbf{y}_3} L J_{\mathbf{y}_3}(\mathbf{z}_3) J_{\mathbf{y}_3}(\mathbf{z}_3) J_{\mathbf{y}_3}(\mathbf{z}_3)$

$\nabla_{\mathbf{y}_3} L \; J_{\mathbf{y}_3}(\mathbf{z}_3) W_3$

$\nabla_{W_3} L = \mathbf{y}_2 \nabla_{\mathbf{y}_3} L \; J_{\mathbf{y}_3}(\mathbf{z}_3)$

$\nabla_{\mathbf{b}_3} L = \nabla_{\mathbf{y}_3} L \; J_{\mathbf{y}_3}(\mathbf{z}_3)$

Additional resources

- Backpropagation can also be extended to work with matrices (batches of data)
- Useful lectures:
  - CMU 11-785
    - https://www.youtube.com/watch?v=s8SHk_7koT4
  - Stanford CS 231n
    - https://www.youtube.com/watch?v=d14TUNcbn1k