L3 Tensor Operations and Gradients

Calculating Derivatives

- Derivatives are one of the most fundamental concepts in calculus.
- They describe how changes in the variable inputs affect the function outputs

Approaches for Computing Derivatives

1. Symbolic differentiation: automatic manipulation of mathematical expressions to get derivatives

- Takes a math expression and returns a math expression:

$$f(x) = x^2 \to \frac{df(x)}{dx} = 2x$$

- Used in Mathematica, Maple, Sympy, etc.

2. Numeric differentiation: Approximating derivatives by finite differences

3. Automatic differentiation: Takes code that computes a function and returns code that computes the derivative of that function.

autograd is a Python package for automatic differentiation

autograd does not support GPUs, so we cannot use autograd for training big neural networks.

Pytorch is a gpu friendly framework on top of autograd.

PyTorch allows to dynamically define computational graphs that can be computed efficiently on GPUs

Approaches for Computing Derivatives

- autograd is a Python package for automatic differentiation
- autograd does not support GPUs, so we cannot use autograd for training big neural networks.
- Pytorch is a GPU friendly framework on top of autograd.
- PyTorch allows to dynamically define computational graphs that can be computed efficiently on GPUs. PyTorch rebuilds the graph every time we iterate or change it and hence uses a dynamic graph
- PyTorch generates derivatives by building a backwards graph behind the scenes

Methods and attributes for computing derivatives in PyTorch

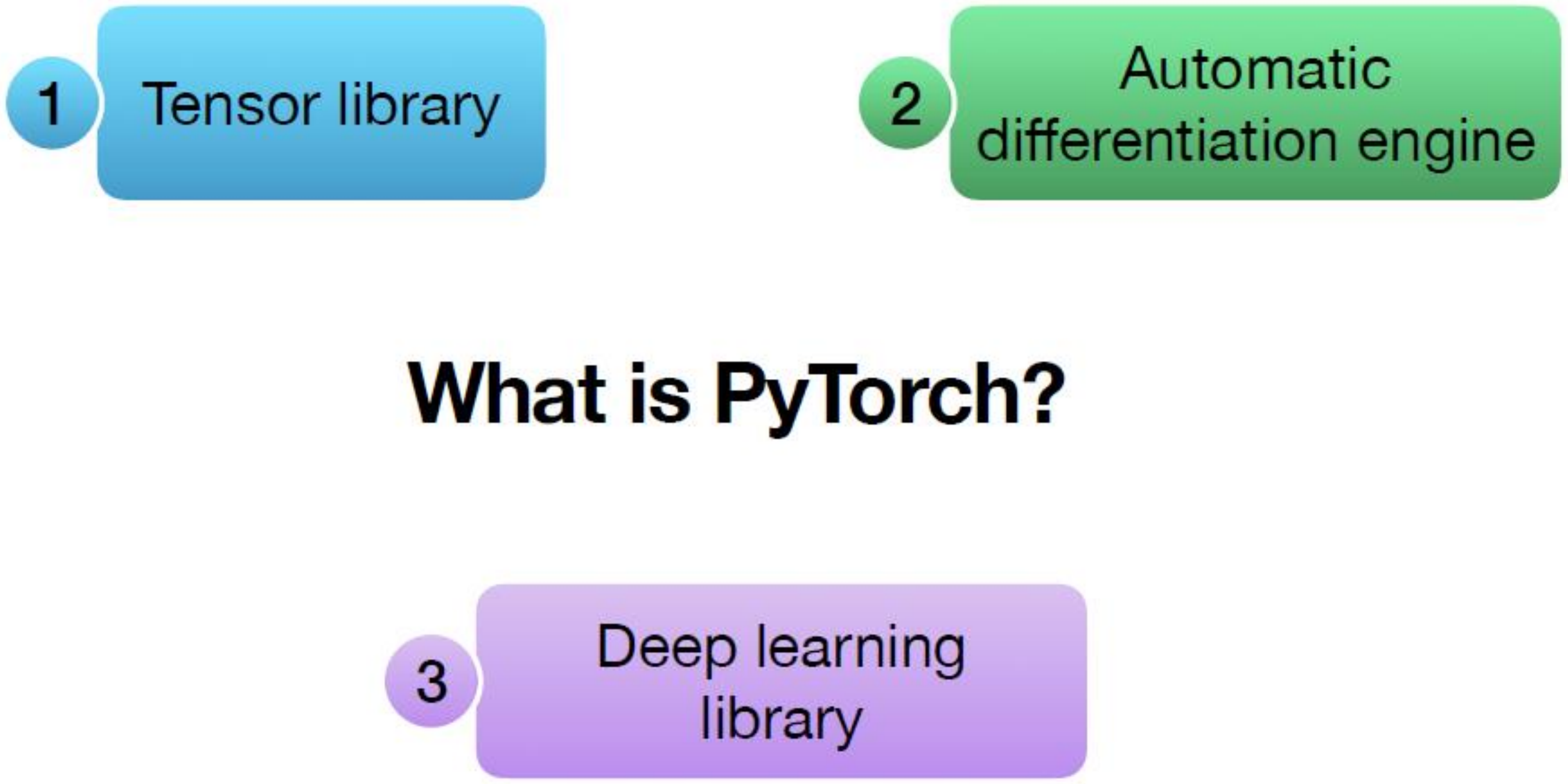
- requires_grad : to automatically compute the derivative of y w.r.t. the tensors that have requires_grad set to True
- requires_grad=True : ensures that any expression involving tensors with that attribute are differentiable with respect to these tensors
- .backward(): Perform a backward pass to compute gradients
- .grad: access the gradient - to track operations for gradient computation. The "grad" stands for gradient, which is another term for derivative
- gradients can only be calculated for floating point tensors so create a float type array before making it a gradient enabled PyTorch tensor
- In default case, the backward() is applied to scalar-valued function
- Gradients are calculated by tracing the graph from the root to the leaf and multiplying every gradient in the way using the chain rule.

Methods and attributes for computing derivatives in PyTorch

Multiple ways to initialize

- Define the requires_grad in the constructor
  - t1 = torch.randn((3,3), requires_grad = True)
- some require to set manually after creation of the Tensor
  - t2 = torch.FloatTensor(3,3) # No way to specify requires_grad while initiating
  - t2.requires_grad = True

PyTorch Library



PyTorch Library

Tensor library

torch.tensor ≈ numpy.array

+ GPU support `a.to('cuda:0')`

+ autodiff support

Computing derivative – Problem 1

Example 1: $y = 3x^2 + 2x + 1$

Calculate the derivative dy/dx and write Python code for the same

$$\frac{dy}{dx} = \frac{d}{dx}(3x^2 + 2x + 1) = 6x + 2$$

So, at $x = 2$, the gradient is $6 \times 2 + 2 = 14.$

Computing derivative – Problem 1

Example 1: $y = 3x^2 + 2x + 1$

- Calculate the derivative dy/dx and write Python code
- The derivates of y w.r.t the input tensor, x is stored in the .grad property of x : x.grad

$$\frac{dy}{dx} = \frac{d}{dx}(3x^2 + 2x + 1) = 6x + 2$$

So, at $x = 2$, the gradient is $6 \times 2 + 2 = 14$.

Input variable (x): 2.0
Result of the expression (y): 17.0
Gradient of y with respect to x: 14.0

```
import torch

# Create a variable with requires_grad=True
x = torch.tensor([2.0], requires_grad=True)

# Define the function
y = 3 * x**2 + 2 * x + 1

# Perform a backward pass to compute gradients
y.backward() # differentiation of y

# Access the gradient
gradient = x.grad #find the derivative dy/dx

# .item() - Gets the Python number within a tensor - only
#works with one-element tensors
print("Input variable (x):", x.item())
print("Result of the expression (y):", y.item())
print("Gradient of y with respect to x:", gradient.item())
```

Computational Graph in PyTorch

- PyTorch defines a computational graph as a Directed Acyclic Graph (DAG) where
  - nodes represent operations (e.g., addition, multiplication, etc.) and
  - edges represent the flow of data between the operations.
- In the context of deep learning (and PyTorch) it is helpful to think about neural networks as computation graphs
- When defining a PyTorch model, the computational graph is created by defining the forward function of the model.
- This function takes inputs and applies a sequence of operations to produce the outputs.
- During the forward pass, PyTorch creates the computational graph by keeping track of the operations and their inputs and outputs.
- In the backpropagation step, the graph is used to compute gradients and monitor the dependencies between computations.

Computational Graph in PyTorch

- A computational graph is a graphical representation of a mathematical function or algorithm, where the
  - nodes of the graph represent mathematical operations, and the
  - edges represent the input/output relationships between the operations.
- Computational graphs are widely used in deep learning
- In a neural network, each
  - node in the computational graph represents a neuron, and the
  - edges represent the connections between neurons.

Computing derivative – Problem 2

- Solve for $y=wx+b$, at $x=3$, $w=4$, $b=5$ and compute $\partial y/\partial w$ and $\partial y/ \partial b$

Computing derivative – Problem 2

- Create 3 tensors x, w and b, with values 3, 4 and 5
- w and b to have an additional parameter requires_grad set to True.
- Create a new tensor y : y=wx+b
- y is a tensor with the value 3 * 4 + 5 = 17.
- To compute the derivatives, call the .backward method on result y.
- The derivates of y w.r.t the input tensors, x, w and b are stored in the .grad property of the respective tensors, x, w and b: x.grad, w.grad, b.grad

Computing derivative – Problem 2

```
# Create tensors
import torch
x = torch.tensor(3.)
w = torch.tensor(4., requires_grad=True)
b = torch.tensor(5., requires_grad=True)
print(x, w, b)
# Arithmetic operations
y = w * x + b
print(y)
# Compute derivatives
y.backward()
# Display gradients
print('dy/dx:', x.grad)
print('dy/dw:', w.grad)
print('dy/db:', b.grad)
```

```
tensor(3.) tensor(4., requires_grad=True)
tensor(5., requires_grad=True)
tensor(17., grad_fn=<AddBackward0>)
dy/dx: None
dy/dw: tensor(3.)
dy/db: tensor(1.)
```

dy/dw has the same value as x i.e. 3,
dy/db has the value 1.
x.grad is None, because x does not have requires_grad set to True
Note: grad_fn=<AddBackward0> means the last operation on the output was Add operation.

Computational Graphs – Problem 3

- Computational Graphs represent symbolic expressions
- In the context of the Single layer Perceptron (SLP), the given equations for the forward- and backward-pass are symbolic expressions.
- In general, any expression of this type can be represented by a graph.
- Ex1: e=(a+b)*(b+1)
- Breaking down to atomic operations yields:

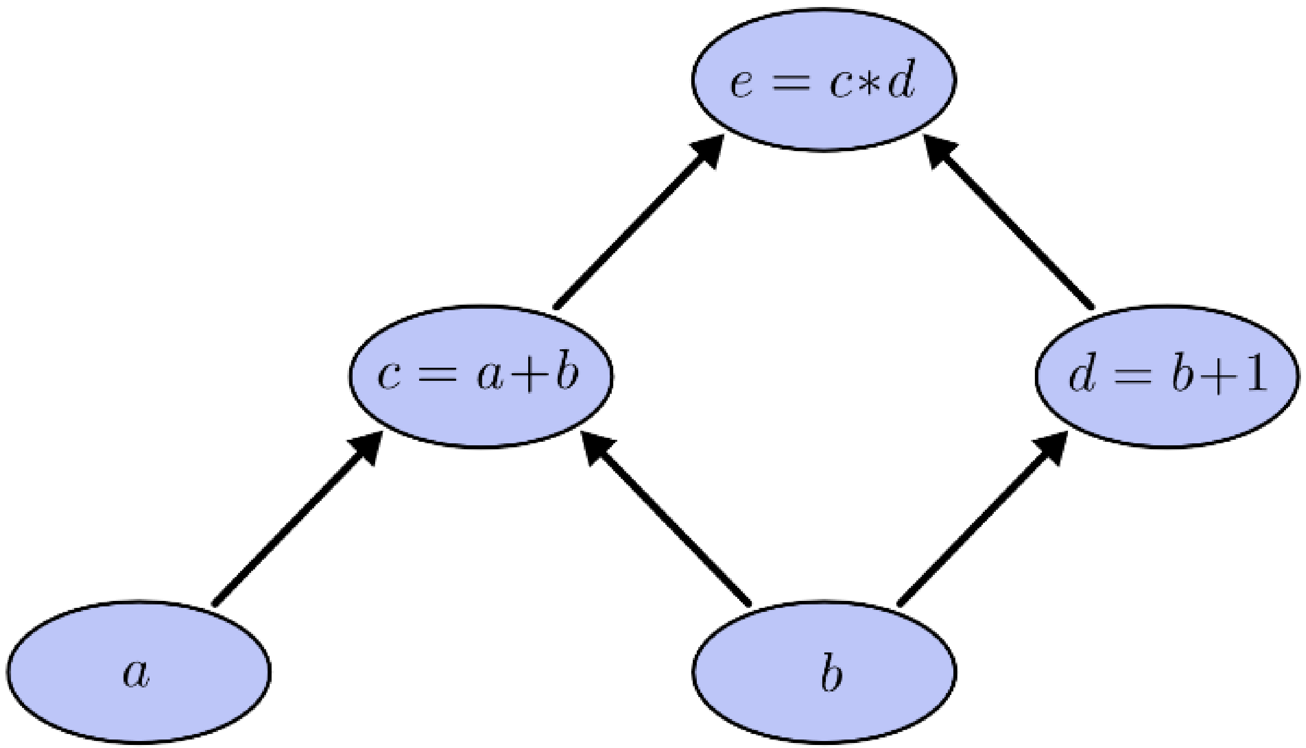
$$c = a + b$$
$$d = b + 1$$
$$e = c \cdot d$$

- Note: computational graphs are very closely related to dependency graphs and call graphs

Computational Graphs – Problem 3

Nodes in the compute graph represent either input variables or basic operations (or functions)

Edges indicate variables applied as operands in operation (arguments are applied for function)

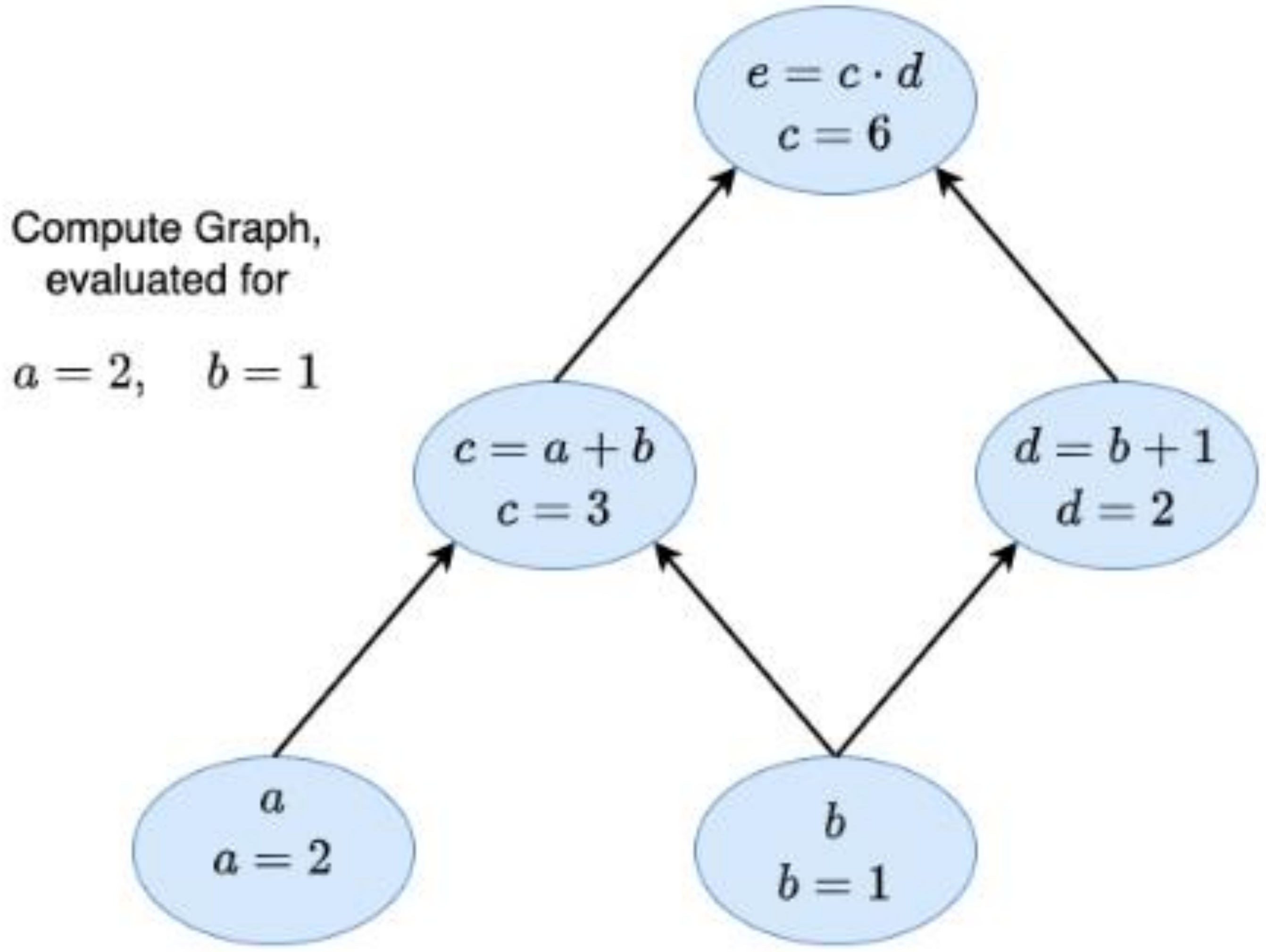


$$c = a + b$$
$$d = b + 1$$
$$e = c \cdot d$$

Computational Graphs -Evaluating Expressions- – Problem 3

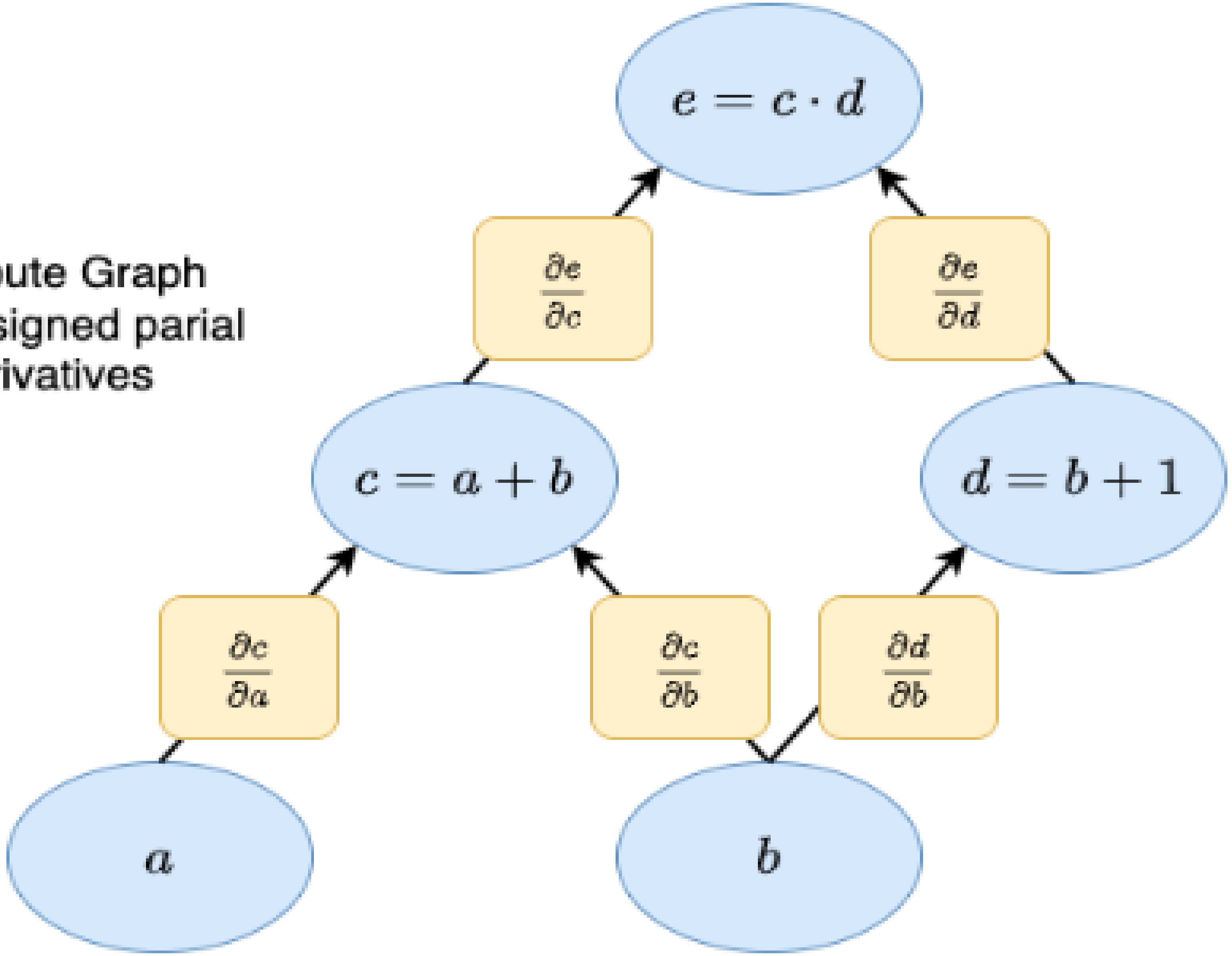
- By assigning values to the input variables the graph is evaluated and
- Values are produced at the output nodes.
- Ex: assigning the values a=2, b=1
- Other variables in the graph are evaluated and the result of this evaluation is 6

Computational Graphs -Evaluating Expressions- – Problem 3



Partial derivatives in Computational Graphs – Problem 3

Compute Graph
with assigned parial
derivatives



Evaluating the partial derivatives - – Problem 3

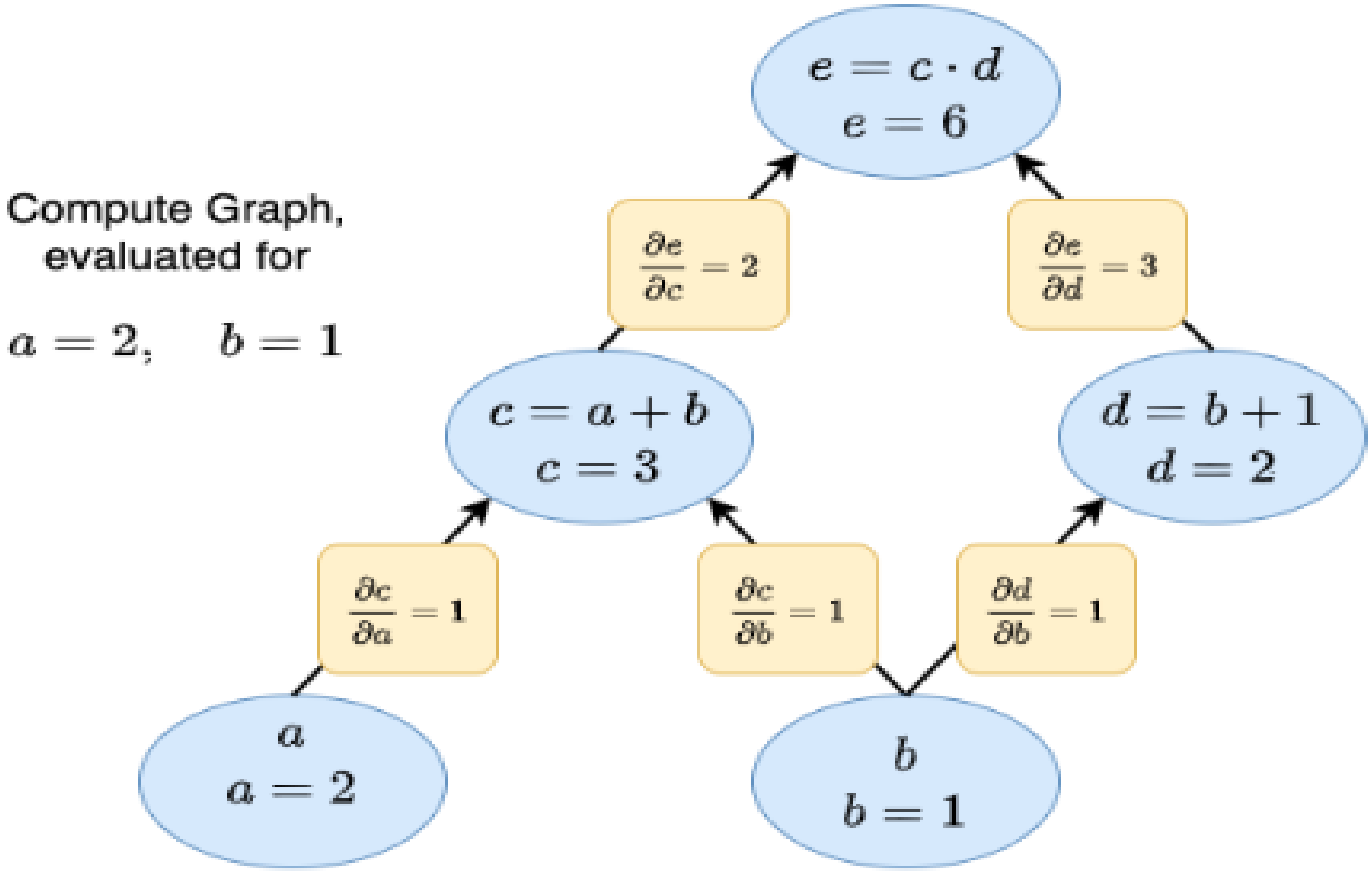
- To evaluate the partial derivatives in this graph, need the sum rule and the product rule:

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$

$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$

Partial derivatives in Computational Graphs - Problem 3

- The values for the partial derivatives for the assignment a=2, b=1
- Partial derivative on each edge labeled

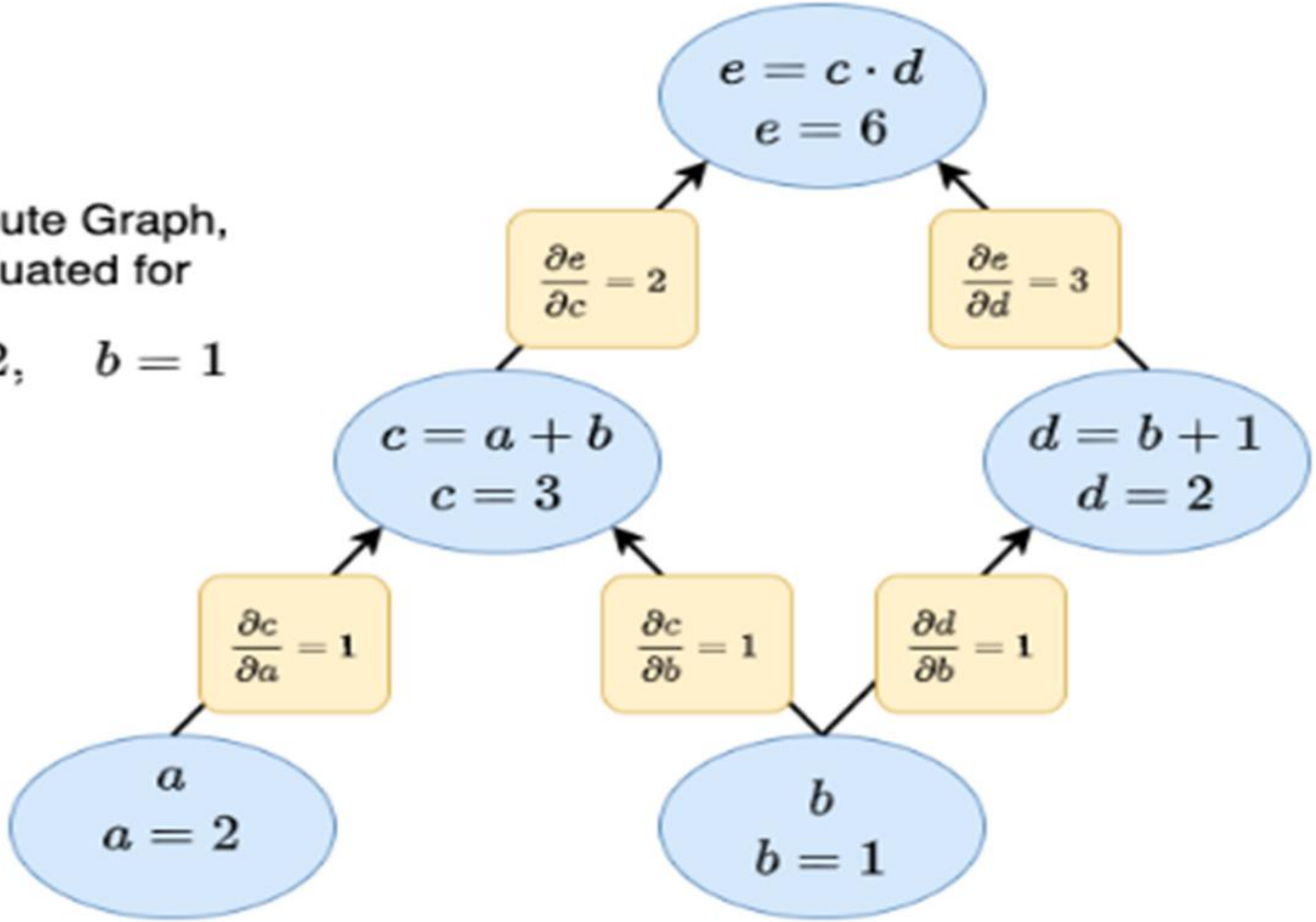


Partial derivatives in Computational Graphs - Problem 3

- With this representation we can easily calculate any partial derivation

Compute Graph, evaluated for

$a = 2, \quad b = 1$



$$\frac{\partial e}{\partial c} = d$$

$$\frac{\partial e}{\partial a} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial a}$$

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b} + \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b}$$

- For calculating the partial derivatives, we just apply the well known chain rule.
- However, if we have visualized our expression in a computational graphs, we can easily determine which partial-derivative-factors must be multiplied and for which paths the products must be added

Partial derivatives in Computational Graphs - Problem 3

- The general rule is to sum over all possible paths from one node to the other, multiplying the derivatives on each edge of the path together.
- Ex: to get the derivative of e with respect to b

$$\frac{\partial e}{\partial b} = 1 * 2 + 1 * 3$$

- This accounts for how b affects e through c and also how b affects e through d.

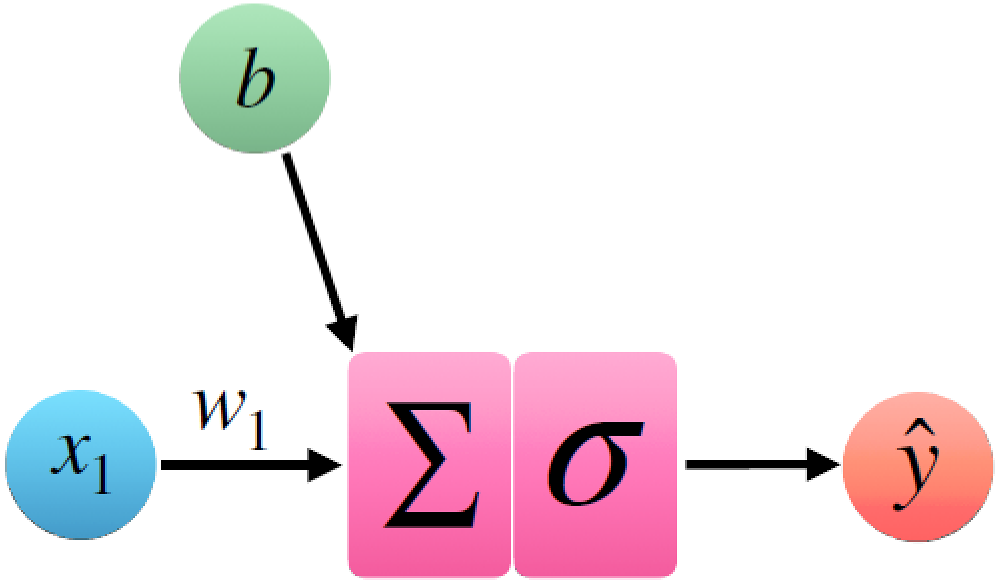
- This general “sum over paths” rule is the multivariate chain rule
- Back propagation is same as chain rule

Partial derivatives in Computational Graphs - Problem 3- Complete the program

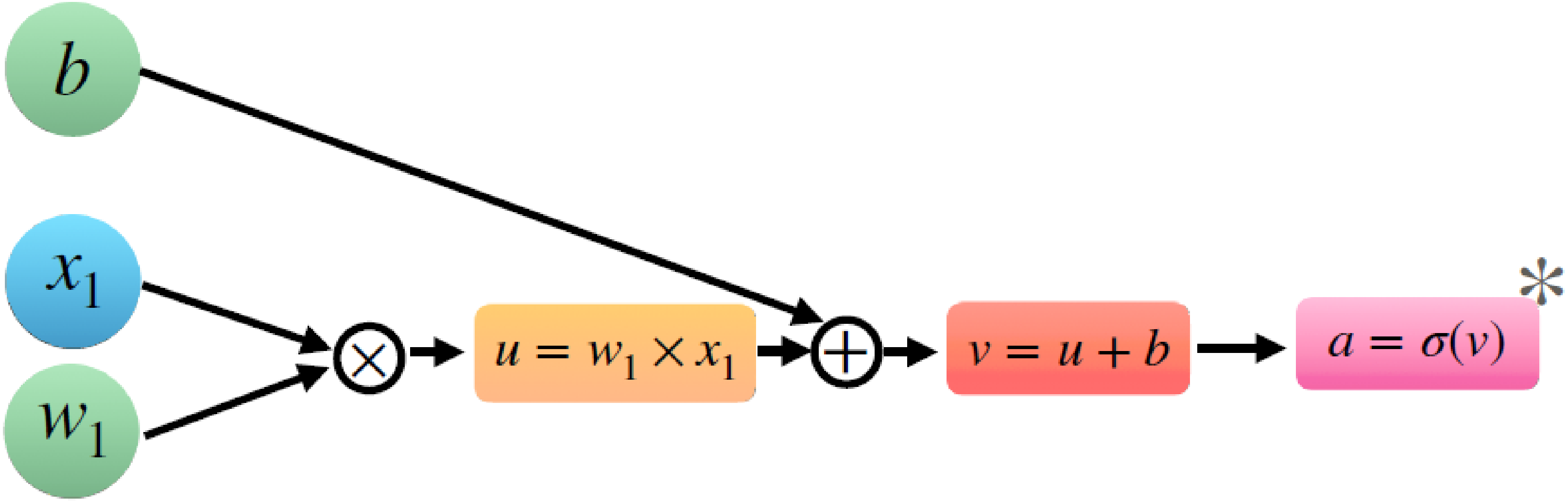
```
a = torch.tensor(2.0, requires_grad=True) # we set requires_grad=True to let PyTorch know to keep the graph

b = torch.tensor(1.0, requires_grad=True)

c = a + b

d = b + 1

e = c * d

print('c', c)

print('d', d)

print('e', e)
```

```
c tensor(3., grad_fn=<AddBackward0>)
d tensor(2., grad_fn=<AddBackward0>)
e tensor(6., grad_fn=<MulBackward0>)
```

Computation graphs – Problem 4



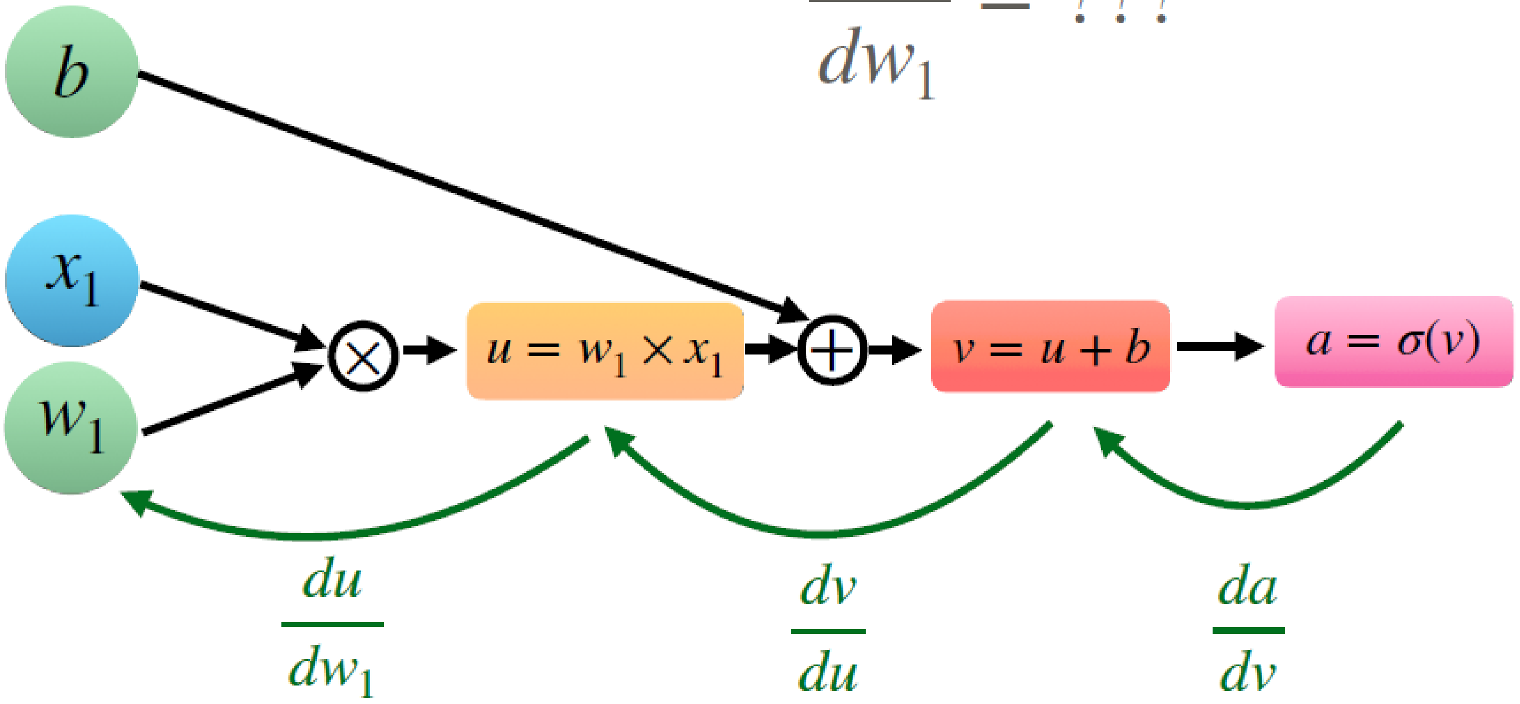
Computation graphs – Problem 4



* $\sigma(z) = \dfrac{1}{1 + e^{-z}}$

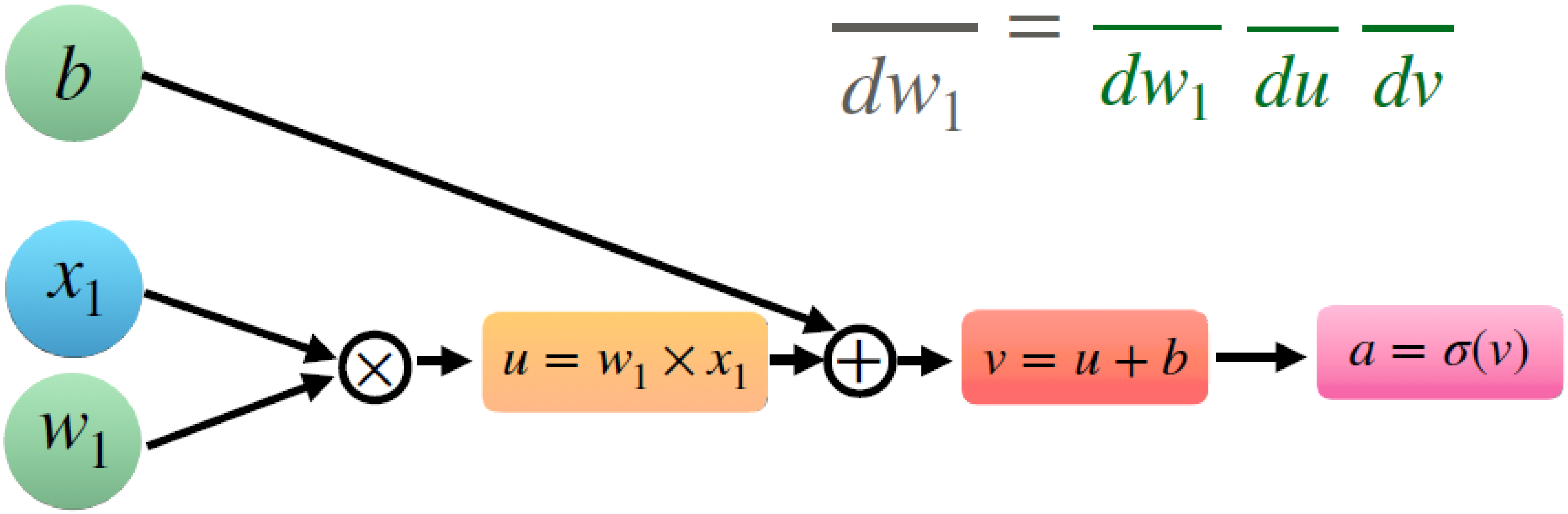
Computation graphs – Problem 4

$$\frac{da}{dw_1} = ???$$



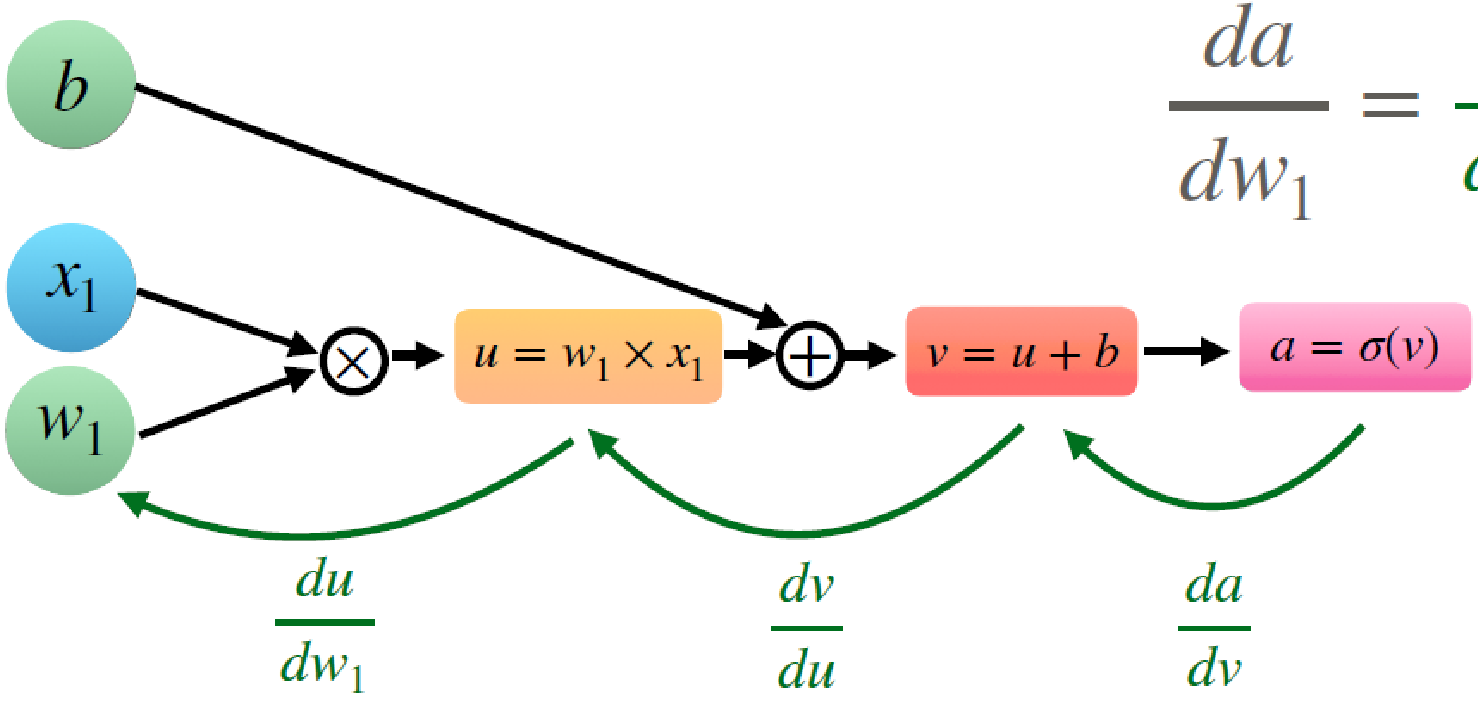
Computation graphs – Problem 4

$$\frac{da}{dw_1} = \frac{du}{dw_1}\frac{dv}{du}\frac{da}{dv}$$



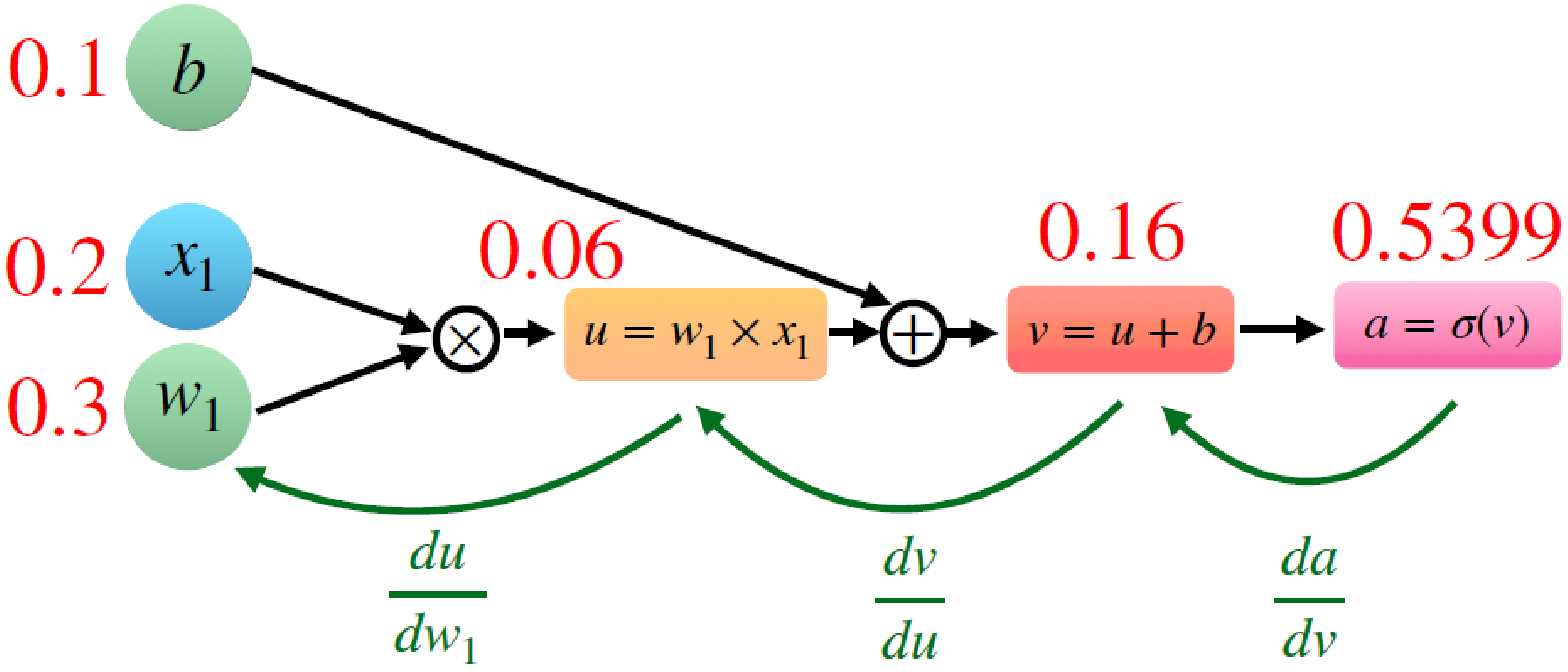
Computation graphs – Problem 4

$$\frac{da}{dw_1} = \frac{du}{dw_1}\frac{dv}{du}\frac{da}{dv}$$



x_1 $\quad$ 1 $\quad$ $\sigma(v)(1-\sigma(v))$

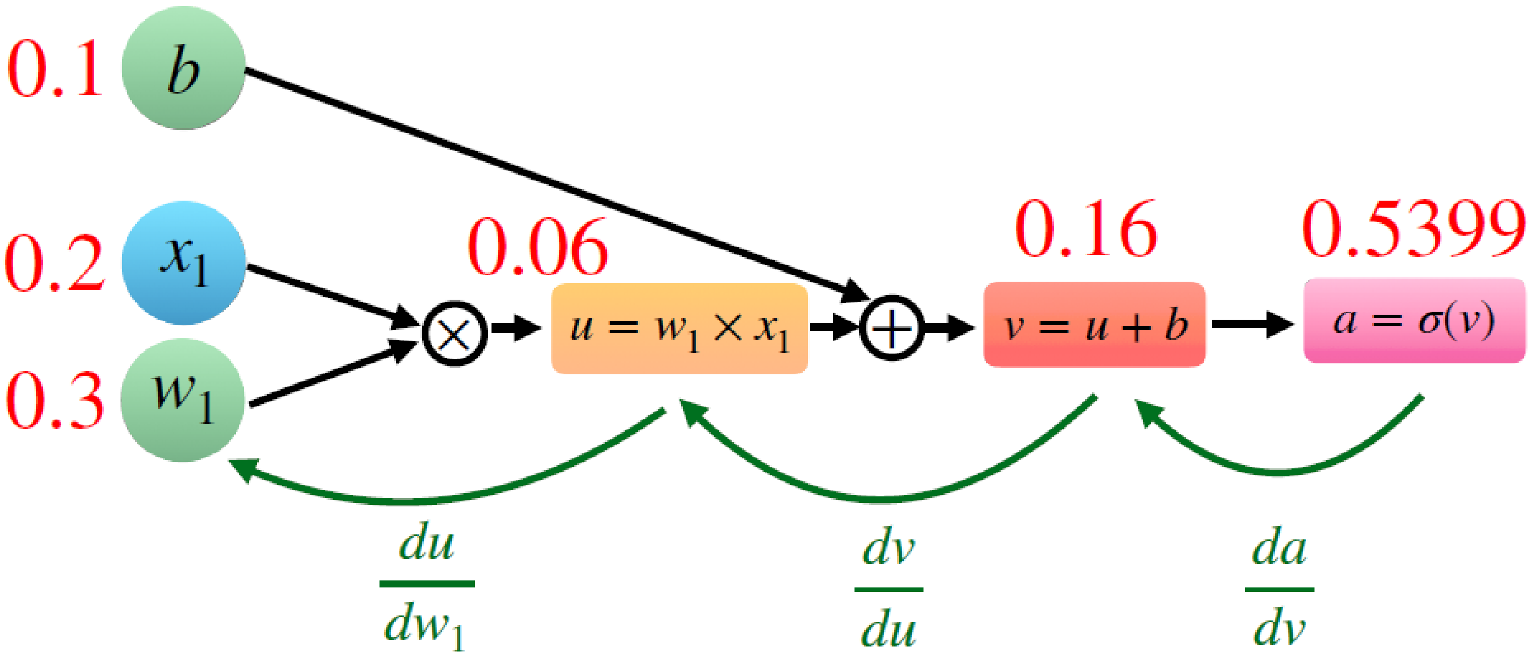
Computation graphs – Problem 4



$$\frac{da}{dw_1} = \frac{du}{dw_1}\,\frac{dv}{du}\,\frac{da}{dv}$$

x_1 $\quad$ 1 $\quad$ $\sigma(v)(1-\sigma(v))$

Computation graphs – Problem 4

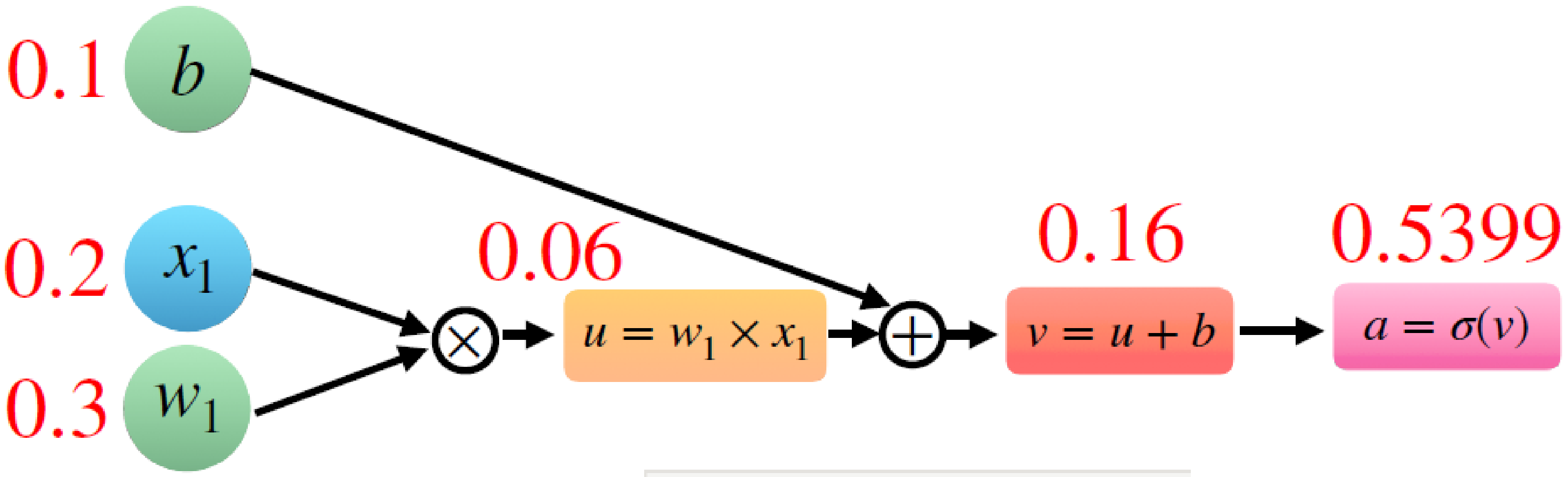


$$\frac{da}{dw_1} = \frac{du}{dw_1}\frac{dv}{du}\frac{da}{dv}$$

$$\frac{da}{dw_1} = x_1 \times \sigma(v)(1 - \sigma(v))$$

0.0497

Computation graphs – Problem 4

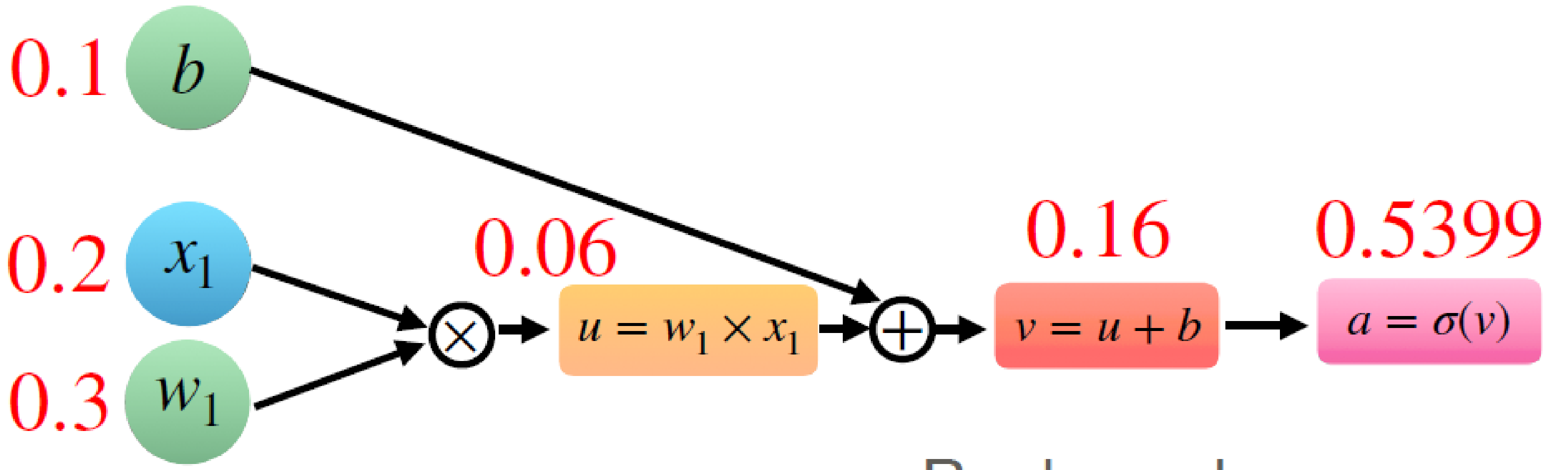


Forward pass
in PyTorch

```
b = torch.tensor(0.1)
x1 = torch.tensor(0.2)
w1 = torch.tensor(0.3)

u = w1*x1
v = u + b
a = torch.sigmoid(v)
a
```

```
tensor(0.5399)
```

Computation graphs – Problem 4




```python
b = torch.tensor(0.1)
x1 = torch.tensor(0.2)
w1 = torch.tensor(0.3)

u = w1*x1
v = u + b
a = torch.sigmoid(v)
a
```

```
tensor(0.5399)
```

Backward pass
in PyTorch

$$\frac{da}{dw_1} =$$

```python
a * (1-a) * x1
```

```
tensor(0.0497)
```

Computing derivative – Problem 5

- z=log(3x2+5xy)- compute dz/dx and dz/dy
- x=2, y=3

Computing derivative – Problem 5

z=log(3x^2+5xy)- compute dz/dx and dz/dy

```
import torch
x = torch.tensor([2.0], requires_grad=True)
y = torch.tensor([3.0], requires_grad=True)
z = torch.log(3*x**2 + 5*x*y)
print("z = ", z)
z.backward()
print("x.grad = ", x.grad)
print("y.grad = ", y.grad)
```

```
z =  tensor([3.7377],grad_fn=<LogBackward0>)
x.grad =  tensor([0.6429])
y.grad =  tensor([0.2381])
```

log here is ln or log e – natural log

d/dx(ln x) = 1/x * d(x)/dx

Computing derivative – Problem 5

- For the same problem, use intermediate variables and draw computation graph
- $z=\log(3x^2+5xy)$

```
a = x**2

b = 3 * a

c = x * y

d = 5 * c

e = b + d

z = torch.log(e)
```

Computing derivative – Problem 5– retain_grad()

- While we declare x and y with requires_grad=True, x.grad and y.grad computations are available.
- x = torch.tensor([2.0], requires_grad=True)
- y = torch.tensor([3.0], requires_grad=True)
- But a.grad, b.grad, c.grad, d.grad, e.grad are not stored by default.
- a.requires_grad=True # RuntimeEror
- RuntimeError: you can only change requires_grad flags of leaf variables.
- To store all intermediate values in the backward propagation, explicitly encode all intermediate steps to remember (retain) their derivative values by using retain_grad()
- a.retain_grad() computes a.grad and the same way, all intermediate variables are to be set.
- z.backward() is same as z.backward(retain_graph=False)

Computing derivative – Problem 5

```
x = torch.tensor([2.0], requires_grad=True)
y = torch.tensor([3.0], requires_grad=True)
a = x**2; a.retain_grad()
b = 3 * a; b.retain_grad()
c = x * y; c.retain_grad()
d = 5 * c; d.retain_grad()
e = b + d; e.retain_grad()
z = torch.log(e); z.retain_grad() #dz/dz=1
z.backward(retain_graph=False) #same as z.backward()
```

```
print(z)
print("x grad = ", x.grad)
print("y grad = ", y.grad)
print("a grad = ", a.grad)
print("b grad = ", b.grad)
print("c grad = ", c.grad)
print("d grad = ", d.grad)
print("e grad = ", e.grad)
print("z grad = ", z.grad)
```

```
tensor([3.7377],
grad_fn=<LogBackward0>)
x grad =  tensor([0.6429])
y grad =  tensor([0.2381])
a grad =  tensor([0.0714])
b grad =  tensor([0.0238])
c grad =  tensor([0.1190])
d grad =  tensor([0.0238])
e grad =  tensor([0.0238])
z grad =  tensor([1.])
```

backward() in detail

- tensor.backward()- parameters:
- backward(gradient=None, retain_graph=None, create_graph=False) : Computes the gradient of current tensor w.r.t. graph leaves.
- The graph is differentiated using the chain rule.
- If the tensor is non-scalar (i.e. its data has more than one element) and requires gradient, the function additionally requires specifying gradient.
  - It should be a tensor of matching type and location, that contains the gradient of the differentiated function w.r.t. self.
- This function accumulates gradients in the leaves
- retain_graph (bool, optional) – If False, the graph used to compute the grads will be freed
- create_graph (bool, optional) – If True, graph of the derivative will be constructed, allowing to compute higher order derivative products. Defaults to False
- Note: By default, pytorch expects backward() to be called for the last output of the network - the loss function.
- The loss function always outputs a scalar and therefore, the gradients of the scalar loss w.r.t all other variables/parameters is well defined (using the chain rule)

Computing derivative – retain_graph()

- To reduce memory usage, during the .backward() call, all the intermediary results are deleted when they are not needed anymore.
- the part of graph will be freed by default to save memory.
- Hence if we try to call .backward() again, the intermediary results do not exist and the backward pass cannot be performed (and we get the error).
- We can call .backward(retain_graph=True) to make a backward pass that will not delete intermediary results, and so we will be able to call .backward() again.
- Illustrated in 3 different variations

Computing derivative – Problem 5– Variation 1

```
x = torch.tensor([2.0], requires_grad=True)
y = torch.tensor([3.0], requires_grad=True)
a = x**2; a.retain_grad()
b = 3 * a; b.retain_grad()
c = x * y; c.retain_grad()
d = 5 * c; d.retain_grad()
e = b + d; e.retain_grad()
z = torch.log(e); z.retain_grad()
z.backward(retain_graph=True)
#z.backward(retain_graph=True)
#z.backward(retain_graph=True)
#z.backward()
```

```
print(z)
print("x grad = ", x.grad)
print("y grad = ", y.grad)
print("a grad = ", a.grad)
print("b grad = ", b.grad)
print("c grad = ", c.grad)
print("d grad = ", d.grad)
print("e grad = ", e.grad)
print("z grad = ", z.grad)
```

```
tensor([3.7377], grad_fn=<LogBackward0>)
x grad =  tensor([0.6429])
y grad =  tensor([0.2381])
a grad =  tensor([0.0714])
b grad =  tensor([0.0238])
c grad =  tensor([0.1190])
d grad =  tensor([0.0238])
e grad =  tensor([0.0238])
z grad =  tensor([1.])
```

Computing derivative – Problem 5 – Variation 2

```
x = torch.tensor([2.0], requires_grad=True)
y = torch.tensor([3.0], requires_grad=True)
a = x**2; a.retain_grad()
b = 3 * a; b.retain_grad()
c = x * y; c.retain_grad()
d = 5 * c; d.retain_grad()
e = b + d; e.retain_grad()
z = torch.log(e); z.retain_grad()
z.backward(retain_graph=True)
z.backward(retain_graph=True)
#z.backward(retain_graph=True)
#z.backward()
```

Note: For subsequent ".backward(retain_graph=True)" call, as the previous grad results are not freed up in memory, x. grad = tensor([0.6429]) value is accumulated again, due to which x.grad+x.grad is summed up as shown for all intermediate variables.

```
print(z)
print("x grad = ", x.grad)
print("y grad = ", y.grad)
print("a grad = ", a.grad)
print("b grad = ", b.grad)
print("c grad = ", c.grad)
print("d grad = ", d.grad)
print("e grad = ", e.grad)
print("z grad = ", z.grad)
tensor([3.7377], grad_fn=<LogBackward0>)
x grad =  tensor([1.2857])
y grad =  tensor([0.4762])
a grad =  tensor([0.1429])
b grad =  tensor([0.0476])
c grad =  tensor([0.2381])
d grad =  tensor([0.0476])
e grad =  tensor([0.0476])
z grad =  tensor([2.])
```

Computing derivative – Problem 5– Variation 3

```
x = torch.tensor([2.0], requires_grad=True)
y = torch.tensor([3.0], requires_grad=True)
a = x**2; a.retain_grad()
b = 3 * a; b.retain_grad()
c = x * y; c.retain_grad()
d = 5 * c; d.retain_grad()
e = b + d; e.retain_grad()
z = torch.log(e); z.retain_grad()
z.backward(retain_graph=True)
z.backward(retain_graph=True)
z.backward(retain_graph=True)
#z.backward()
```

Note: For subsequent ".backward(retain_graph=True)" call, as the previous grad results are not freed up in memory, x. grad = tensor([0.6429]) value is accumulated again, due to which x.grad+x.grad+x.grad is summed up as shown for all intermediate variables.

```
print(z)
print("x grad = ", x.grad)
print("y grad = ", y.grad)
print("a grad = ", a.grad)
print("b grad = ", b.grad)
print("c grad = ", c.grad)
print("d grad = ", d.grad)
print("e grad = ", e.grad)
print("z grad = ", z.grad)
```

```
tensor([3.7377], grad_fn=<LogBackward0>)
x grad =  tensor([1.9286])
y grad =  tensor([0.7143])
a grad =  tensor([0.2143])
b grad =  tensor([0.0714])
c grad =  tensor([0.3571])
d grad =  tensor([0.0714])
e grad =  tensor([0.0714])
z grad =  tensor([3.])
```

Gradient for non-scalar input – Problem 6

- Compute gradient dr/dx and draw computation graph. Implement the same and compare the results with analytical gradient.
- x=[1., 1., 1.]
- y=x^2
- z=x^3
- r=y+z #does not work - grad can be implicitly created only for scalar outputs

Gradient for non-scalar input – Problem 6

```
x=torch.ones(3, requires_grad=True)
print("x=", x)
y=x**2
z=x**3
print("y=", y)
print("z=", z)
#r=(y+z).sum()
r=(y+z)
print("r=", r)
r.backward()
print(x.grad)
```

```
x= tensor([1., 1., 1.],
requires_grad=True)
y= tensor([1., 1., 1.],
grad_fn=<PowBackward0>)
z= tensor([1., 1., 1.],
grad_fn=<PowBackward0>)
r= tensor([2., 2., 2.],
grad_fn=<AddBackward0>)
```

RuntimeError: grad can be implicitly created only for scalar outputs

Re-write with r=(y+z).sum()

```
x= tensor([1., 1., 1.], requires_grad=True)
y= tensor([1., 1., 1.], grad_fn=<PowBackward0>)
z= tensor([1., 1., 1.], grad_fn=<PowBackward0>)
r= tensor(6., grad_fn=<SumBackward0>)
tensor([5., 5., 5.])
```

with torch.no_grad():

- To stop PyTorch from tracking the history and forming the backward graph, the code can be wrapped inside with torch.no_grad():
- It will make the code run faster whenever gradient tracking is not needed.
- We use torch.no_grad() so that the optimizer does not calculate gradients for the lines of code that follow. This reduces memory usage and speeds up computation.
- We are able to use w.grad (w is tensor enabled input with requires_grad=True) because the gradients were calculated before when we called loss.backward() (where loss is the output).

with torch.no_grad():

- Disabling gradient calculation is useful for inference, when we are sure that there will not be call Tensor.backward().
- It will reduce memory consumption for computations that would otherwise have requires_grad=True.
- In this mode, the result of every computation will have requires_grad=False, even when the inputs have requires_grad=True.

with torch.no_grad():

```
#without using no_grad
x = torch.ones(3, requires_grad=True)
y = x**2
z = x**3
r = (y+z).sum()
print(r.requires_grad)
True
#when using no_grad
x = torch.ones(3, requires_grad=True)
with torch.no_grad():
    y = x**2
    z = x**3
    r = (y+z).sum()
print(r.requires_grad)
False
```

In the above example, "with torch.no_grad():" will make all the operations in the block have no gradients. Hence, we cannot use backward() on the computed variable.

```
x = torch.ones(3, requires_grad=True)
with torch.no_grad():
    y = x**2
    z = x**3
    r = (y+z).sum()
    r.backward()
print(r.requires_grad)
RuntimeError: element 0 of tensors does not require grad and does not have a grad_fn
```

Training a neural network

- Creating and training a neural network involves the following essential steps:

1. Define the architecture
2. Forward propagate on the architecture using input data
3. Calculate the loss
4. Backpropagate to calculate the gradient for each weight
5. Update the weights using a learning rate

The change in the loss for a small change in an input weight is called the gradient of that weight and is calculated using backpropagation.

The gradient is then used to update the weight using a learning rate to overall reduce the loss and train the neural net

Loss function – Problem 7

- Compute gradient d(loss)/dw and draw computation graph. Implement the same and compare the results with analytical gradient.

- $y^{\wedge}$=wx
- loss=(y^-y)**2
- x and w are input tensors with x=1, w=1
- y is initialized with 2

Loss function - Problem 7

```
import torch
x = torch.tensor(1.0)
y = torch.tensor(2.0)
# This is the parameter we want to optimize ->
#requires_grad=True
w = torch.tensor(1.0, requires_grad=True)
# forward pass to compute loss
y_predicted = w * x
loss = (y_predicted - y)**2
print("loss=", loss)
# backward pass to compute gradient
#dLoss/dw
loss.backward()
print("w.grad=", w.grad)
# update weights, this operation
should not be part of the
computational graph
with torch.no_grad():
    w -= 0.01 * w.grad
    print("w=", w, "w.grad=", w.grad)
# zero the gradients
w.grad.zero_()
print("w.grad=", w.grad)
```

Loss function – Problem 7

loss= tensor(1., grad_fn=<PowBackward0>)

w.grad= tensor(-2.)

w= tensor(1.0200, requires_grad=True) w.grad= tensor(-2.)

w.grad= tensor(0.)