

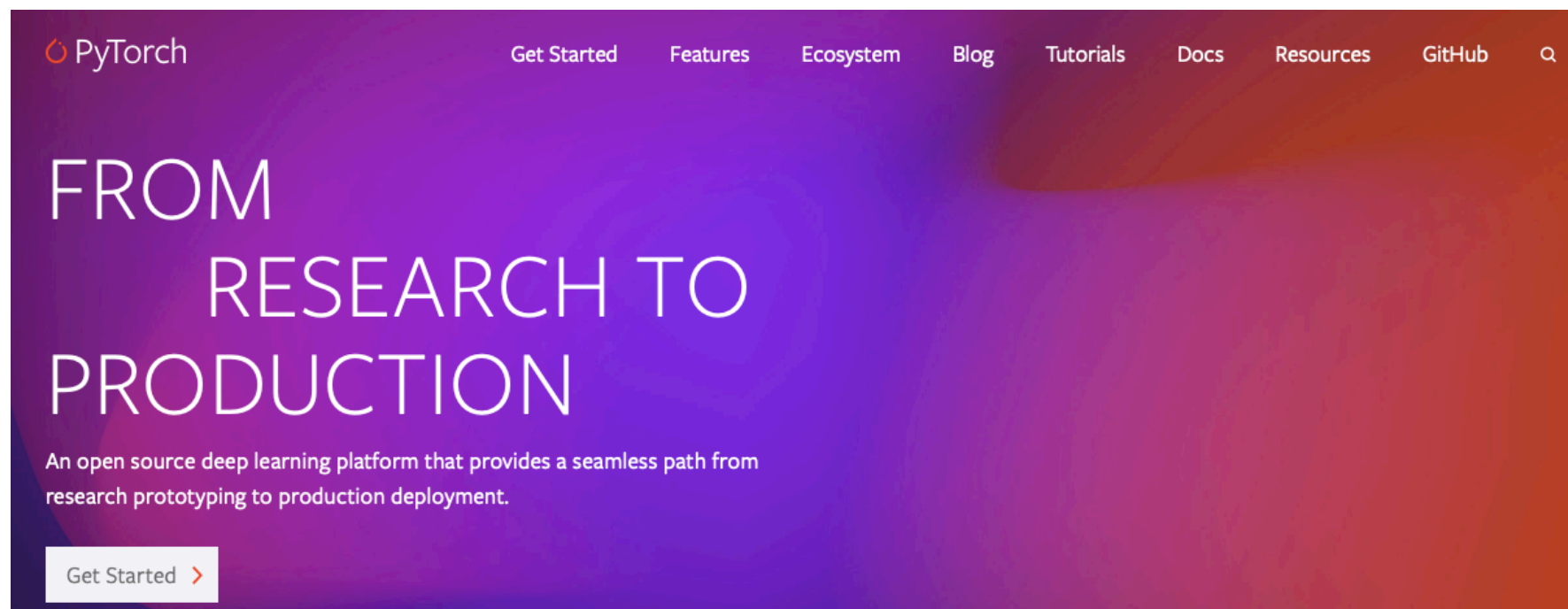
## Lecture 06

# Automatic Differentiation with PyTorch

STAT 479: Deep Learning, Spring 2019

Sebastian Raschka

<http://stat.wisc.edu/~sraschka/teaching/stat479-ss2019/>



<https://pytorch.org/>

# Installation

Recommendation for Laptop (e.g., MacBook)

PyTorch Build	Stable (1.0)		Preview (Nightly)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python 2.7	Python 3.5	Python 3.6	Python 3.7
CUDA	8.0	9.0	10.0	None
Run this Command:	conda install pytorch torchvision -c pytorch			

Recommendation for Desktop (Linux) with GPU

PyTorch Build	Stable (1.0)		Preview (Nightly)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python 2.7	Python 3.5	Python 3.6	Python 3.7
CUDA	8.0	9.0	10.0	None
Run this Command:	conda install pytorch torchvision cudatoolkit=10.0 -c pytorch			

<https://pytorch.org/>

## Installation Tips:

<https://github.com/rasbt/stat479-deep-learning-ss19/blob/master/other/pytorch-installation-tips.md>

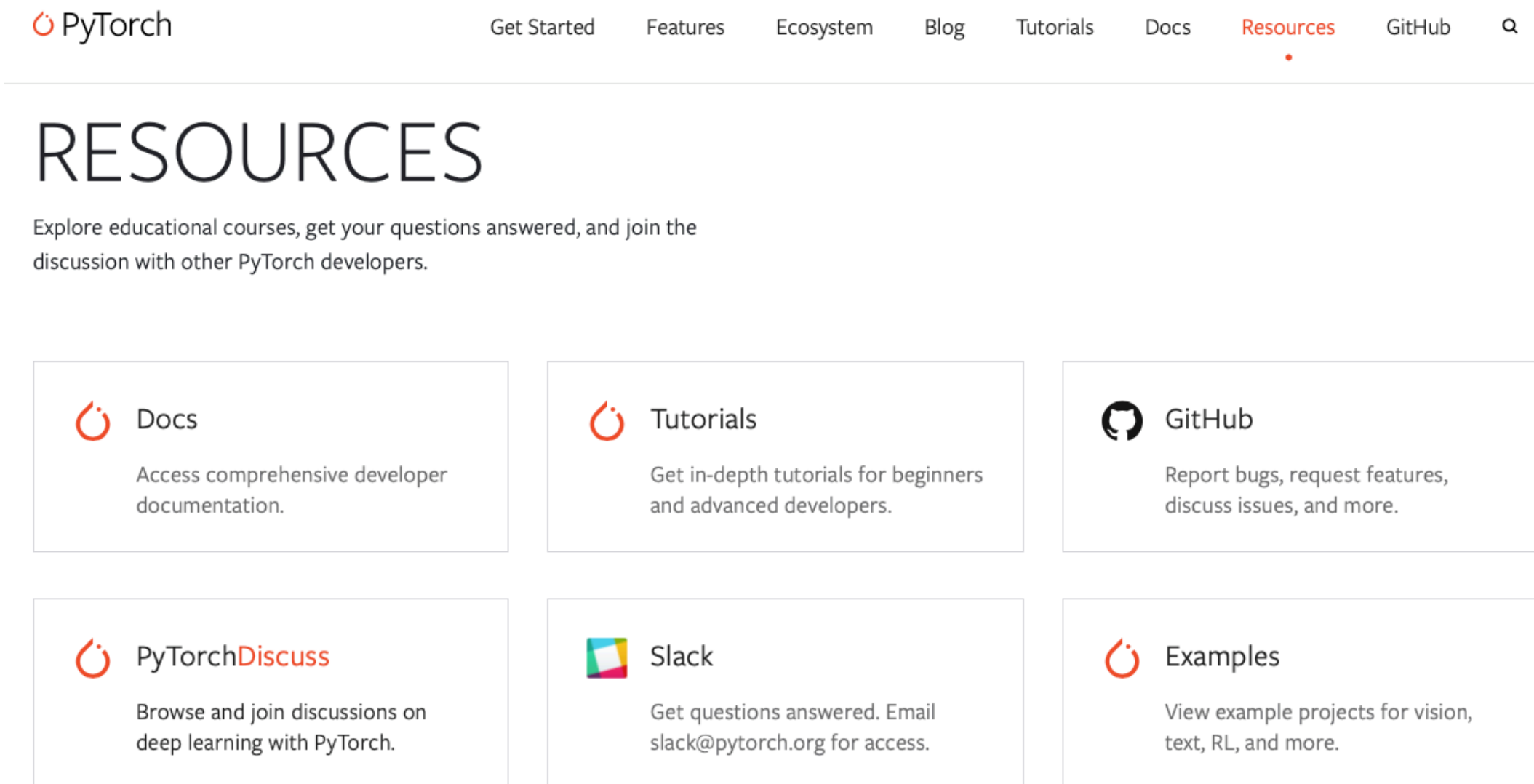
And don't forget that you import PyTorch as "import torch," not "import pytorch" :)

```
In [1]: import torch
```

```
In [2]: torch.__version__
```

```
Out[2]: '1.0.1'
```

# Many Useful Tutorials (recommend that you read some of them)




The screenshot shows the PyTorch website's 'Resources' page. At the top is a navigation bar with links: 'Get Started', 'Features', 'Ecosystem', 'Blog', 'Tutorials', 'Docs', 'Resources' (highlighted with a red dot), 'GitHub', and a search icon. Below the navigation bar is the heading 'RESOURCES' in large, bold, black letters. Underneath the heading is a subtext: 'Explore educational courses, get your questions answered, and join the discussion with other PyTorch developers.' Below this is a grid of six resource cards. Each card has an icon, a title, and a description. The cards are: 1. 'Docs' with a PyTorch logo icon, description: 'Access comprehensive developer documentation.' 2. 'Tutorials' with a PyTorch logo icon, description: 'Get in-depth tutorials for beginners and advanced developers.' 3. 'GitHub' with the GitHub logo icon, description: 'Report bugs, request features, discuss issues, and more.' 4. 'PyTorchDiscuss' with a PyTorch logo icon, description: 'Browse and join discussions on deep learning with PyTorch.' 5. 'Slack' with the Slack logo icon, description: 'Get questions answered. Email slack@pytorch.org for access.' 6. 'Examples' with a PyTorch logo icon, description: 'View example projects for vision, text, RL, and more.'


PyTorch


Get Started Features Ecosystem Blog Tutorials Docs **Resources** GitHub


## RESOURCES


Explore educational courses, get your questions answered, and join the discussion with other PyTorch developers.


 **Docs**  
Access comprehensive developer documentation.

 **Tutorials**  
Get in-depth tutorials for beginners and advanced developers.

 **GitHub**  
Report bugs, request features, discuss issues, and more.

 **PyTorchDiscuss**  
Browse and join discussions on deep learning with PyTorch.

 **Slack**  
Get questions answered. Email slack@pytorch.org for access.

 **Examples**  
View example projects for vision, text, RL, and more.

<https://pytorch.org/resources>

# Very Active & Friendly Community and Help/Discussion Forum

PyTorch

all categories ▸

Latest

New (49)

Unread (32)

Top

Categories

+ New Topic

Topic

Replies

Views

Activity

🚩 PyTorch Feedback •

Hello! I'm working on PyTorch and would like your feedback. Please send answers to the questions below to michellenicole@fb.com with 'PyTorch Feedback' in the subject line. Thank-you for your valuable input! 🔥 W... [read more](#)

1

143

2d

Going from 0.4.0 to 1.0.0 changes code runtime from 0:00:00.35 to 0:08:45.63 •

A

5

7

7m

RuntimeError: Expected 4-dimensional input for 4-dimensional weight [6, 3, 5, 5], but got 3-dimensional input of size [3, 256, 256] instead •

L

S

5

18

1h

How to change the output size of the model when the batch size is specified? •

■ vision

S

2

13

1h

Training Reproducibility Problem

B

5

24

1h

[Caffe2] MobileNetV2 Quantized using caffe2: BlobIsTensorType(\*blob, CPU). Blob is not a CPU Tensor: 325

19

5

217

1h

Creating a tuple of Tensors as input to a traced model with the C++ front end •

■ C++

B

0

14

2h

Error when trying to view image with dataloader and iter, PascalVOC 2007 •

■ vision

0

9

2h

Train a small vector , which is not connected to the main Graph - need your creativity •

B

3

21

2h

<https://pytorch.org/resources>



# DEEP LEARNING WITH PYTORCH: A 60 MINUTE BLITZ

**Author:** Soumith Chintala

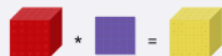
Goal of this tutorial:

- Understand PyTorch's Tensor library and neural networks at a high level.
- Train a small neural network to classify images

*This tutorial assumes that you have a basic familiarity of numpy*

## • NOTE

Make sure you have the `torch` and `torchvision` packages installed.



What is PyTorch?



Autograd: Automatic Differentiation



Neural Networks

Generally speaking, `torch.autograd` is an engine for computing vector-Jacobian product. That is, given any vector  $v = \begin{pmatrix} v_1 & v_2 & \dots & v_m \end{pmatrix}^T$ , compute the product  $v^T \cdot J$ . If  $v$  happens to be the gradient of a scalar function  $l = g(\vec{y})$ , that is,  $v = \begin{pmatrix} \frac{\partial l}{\partial y_1} & \dots & \frac{\partial l}{\partial y_m} \end{pmatrix}^T$ , then by the chain rule, the vector-Jacobian product would be the gradient of  $l$  with respect to  $x$ :

$$J^T \cdot v = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \dots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \begin{pmatrix} \frac{\partial l}{\partial y_1} \\ \vdots \\ \frac{\partial l}{\partial y_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial l}{\partial x_1} \\ \vdots \\ \frac{\partial l}{\partial x_n} \end{pmatrix}$$

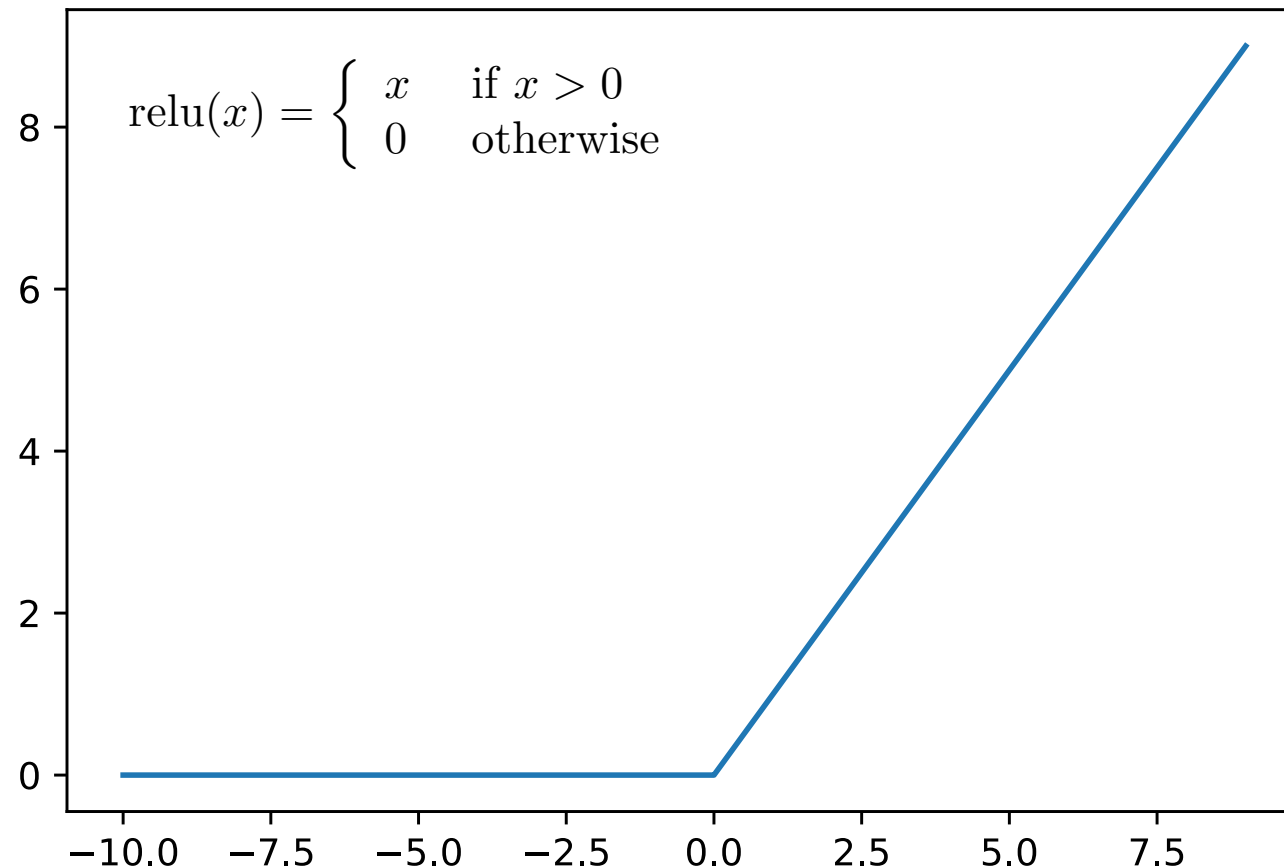
Text source: [https://pytorch.org/tutorials/beginner/blitz/autograd\\_tutorial.html#sphx-glr-beginner-blitz-autograd-tutorial-py](https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html#sphx-glr-beginner-blitz-autograd-tutorial-py)

**In the context of deep learning (and PyTorch)  
it is helpful to think about neural networks  
as computation graphs**

# Computation Graphs

Suppose we have the following activation function:

$$a(x, w, b) = \text{relu}(w \cdot x + b)$$



ReLU = Rectified Linear Unit

(prob. the most commonly used activation function in DL)



# Side-note about ReLU Function

You may note that

$$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x > 0 \\ \text{DNE} & \text{if } x = 0 \end{cases}$$

But in the computer science context, for convenience, we can just say

$$f'(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$

$$f'(x) = \lim_{x \rightarrow 0} \frac{\max(0, x + \Delta x) - \max(0, x)}{\Delta x}$$

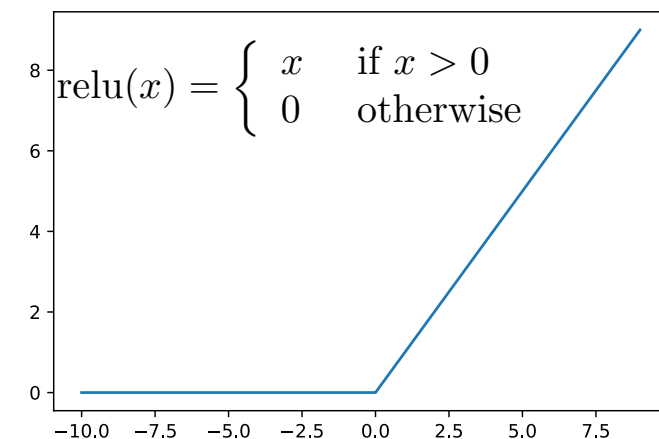
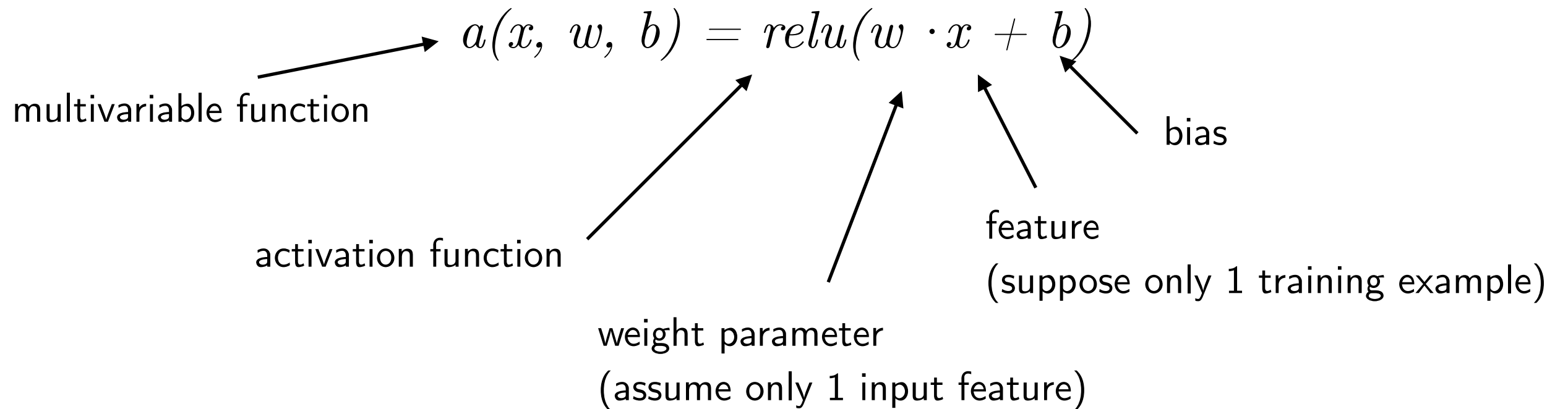
$$f'(x) = \lim_{x \rightarrow 0} \frac{\max(0, x + \Delta x) - \max(0, x)}{\Delta x}$$

$$f'(0) = \lim_{x \rightarrow 0^+} \frac{0 + \Delta x - 0}{\Delta x} = 1$$

$$f'(0) = \lim_{x \rightarrow 0^-} \frac{0 - 0}{\Delta x} = 0$$

# Computation Graphs

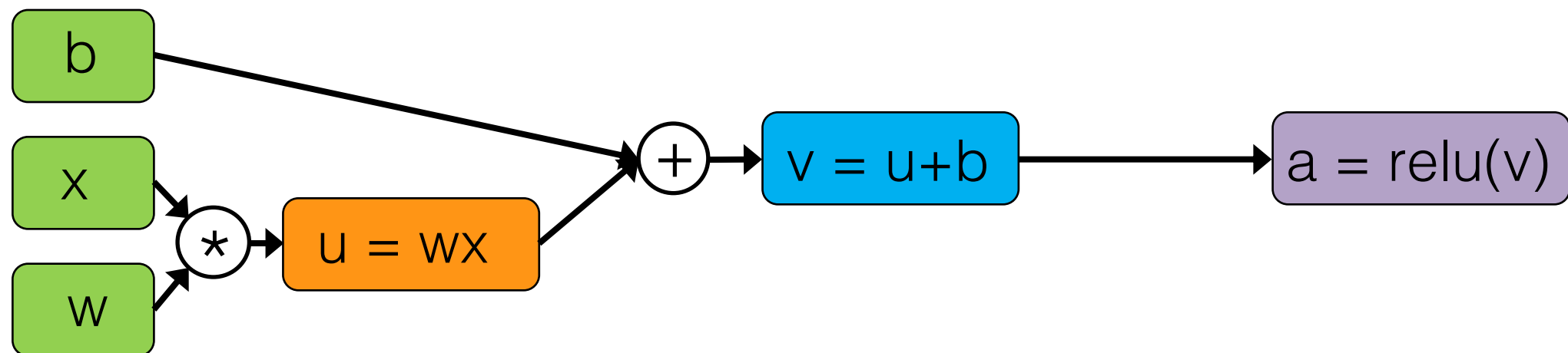
Suppose we have the following activation function:



# Computation Graphs

$$a(x, w, b) = \text{relu}(\underbrace{w \cdot x}_u + b)$$

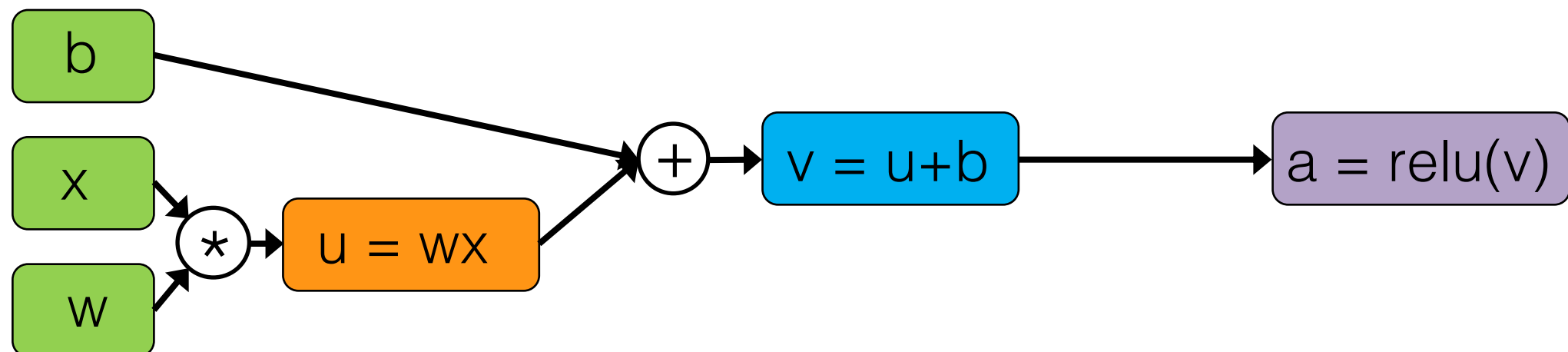
$v$



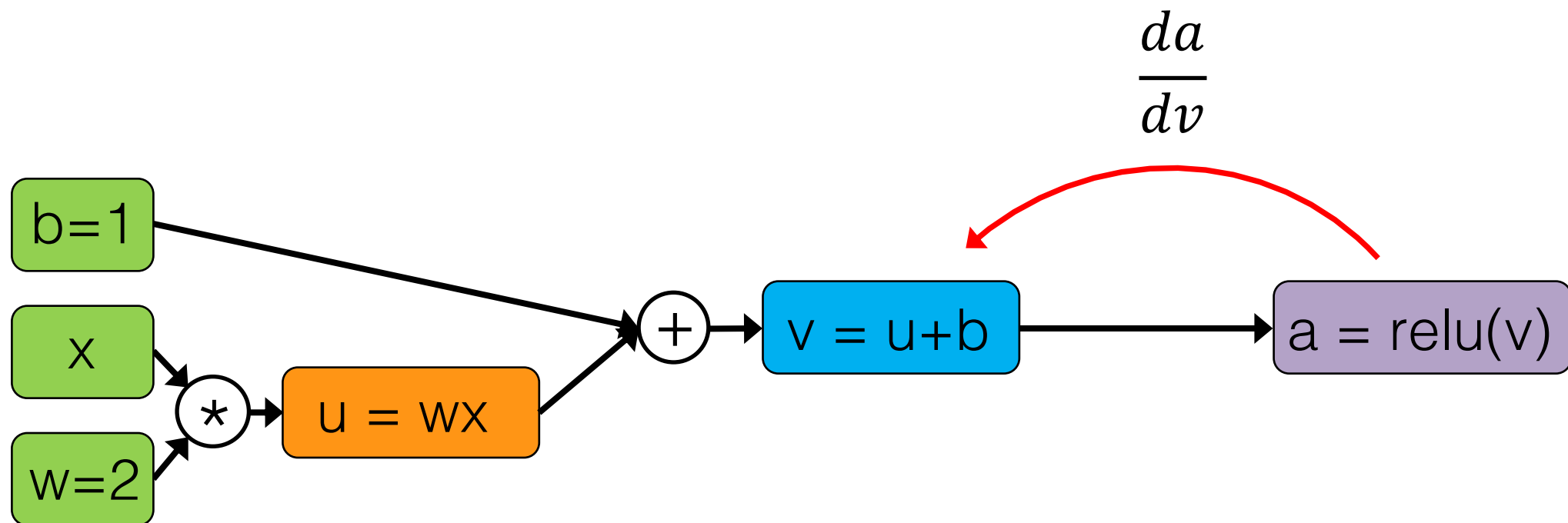
# Computation Graphs

$$a(x, w, b) = \text{relu}(\underbrace{w \cdot x}_u + b)$$

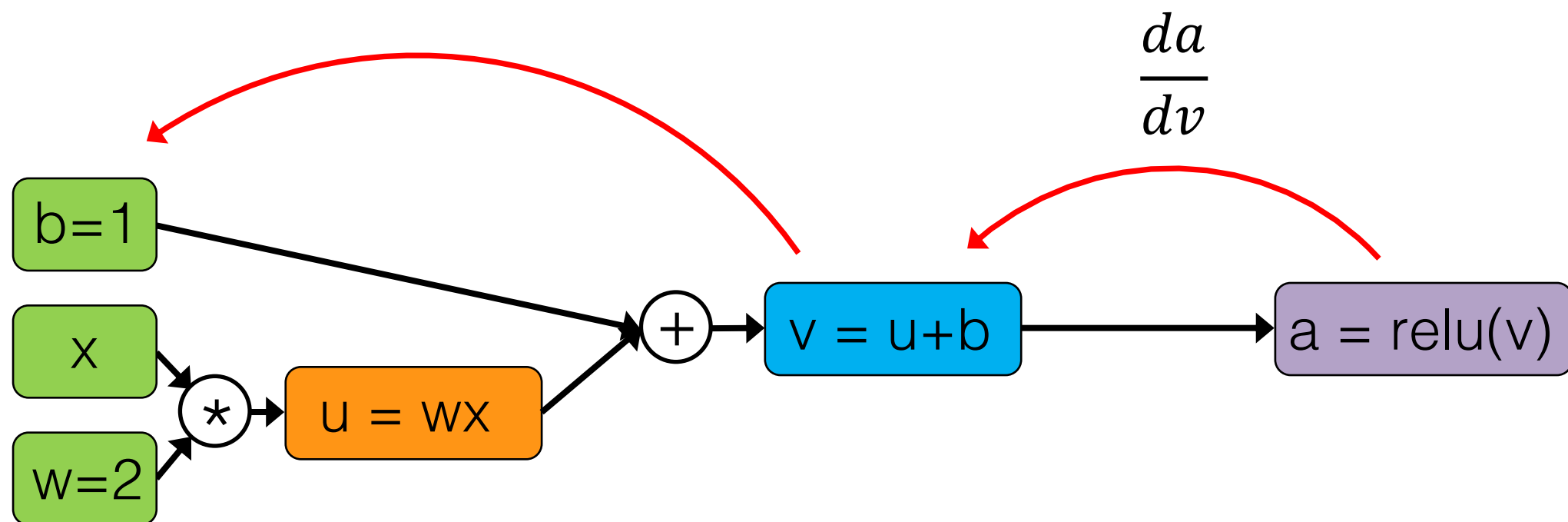
$v$



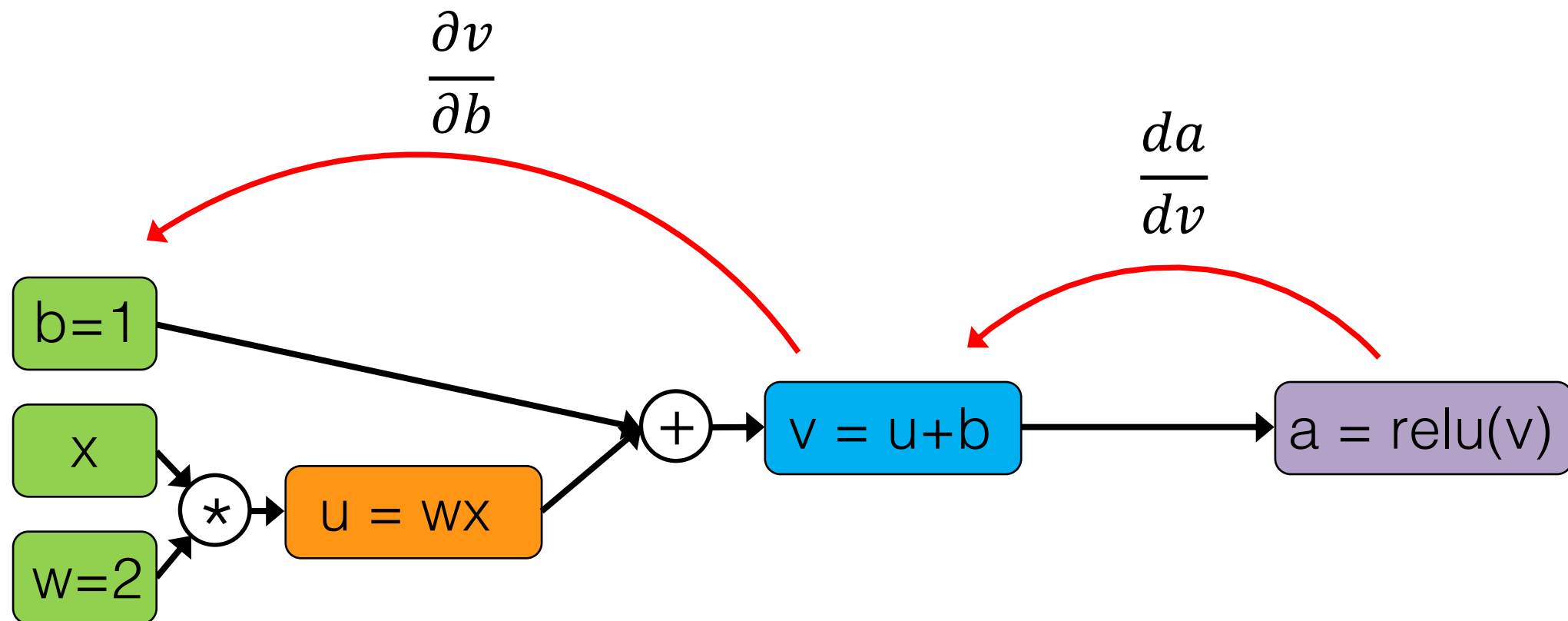
# Computation Graphs



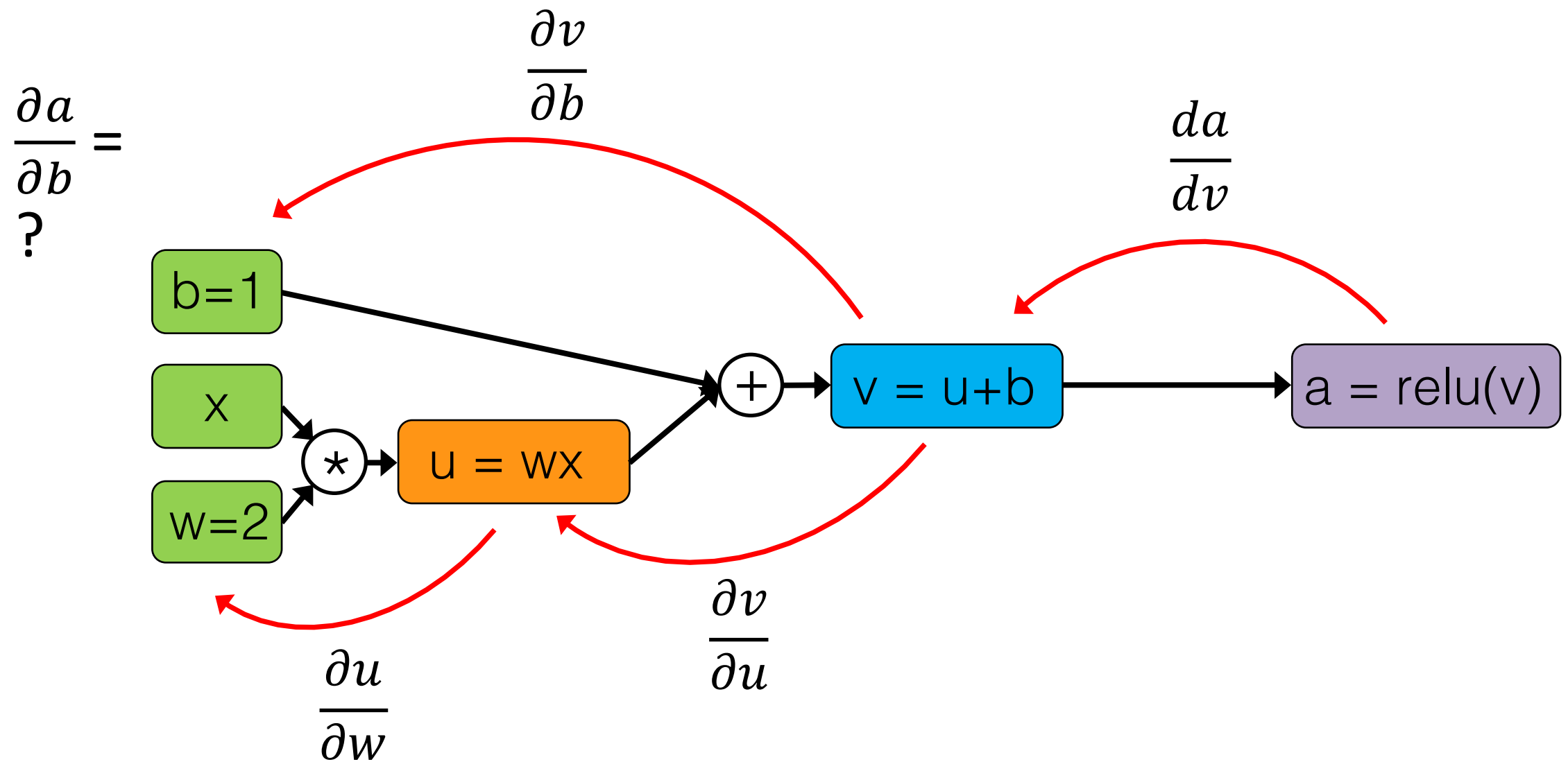
# Computation Graphs



# Computation Graphs



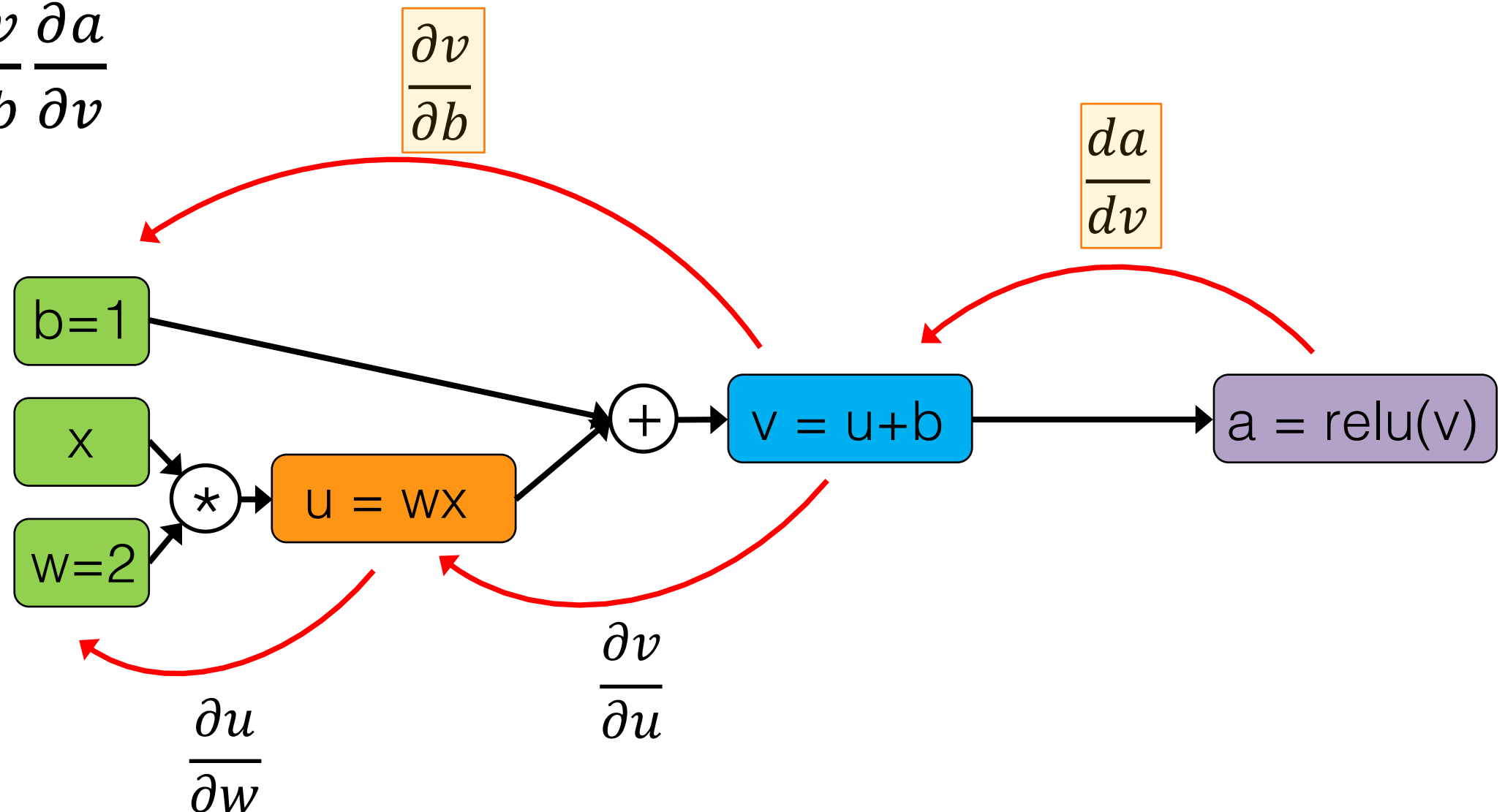
# Computation Graphs



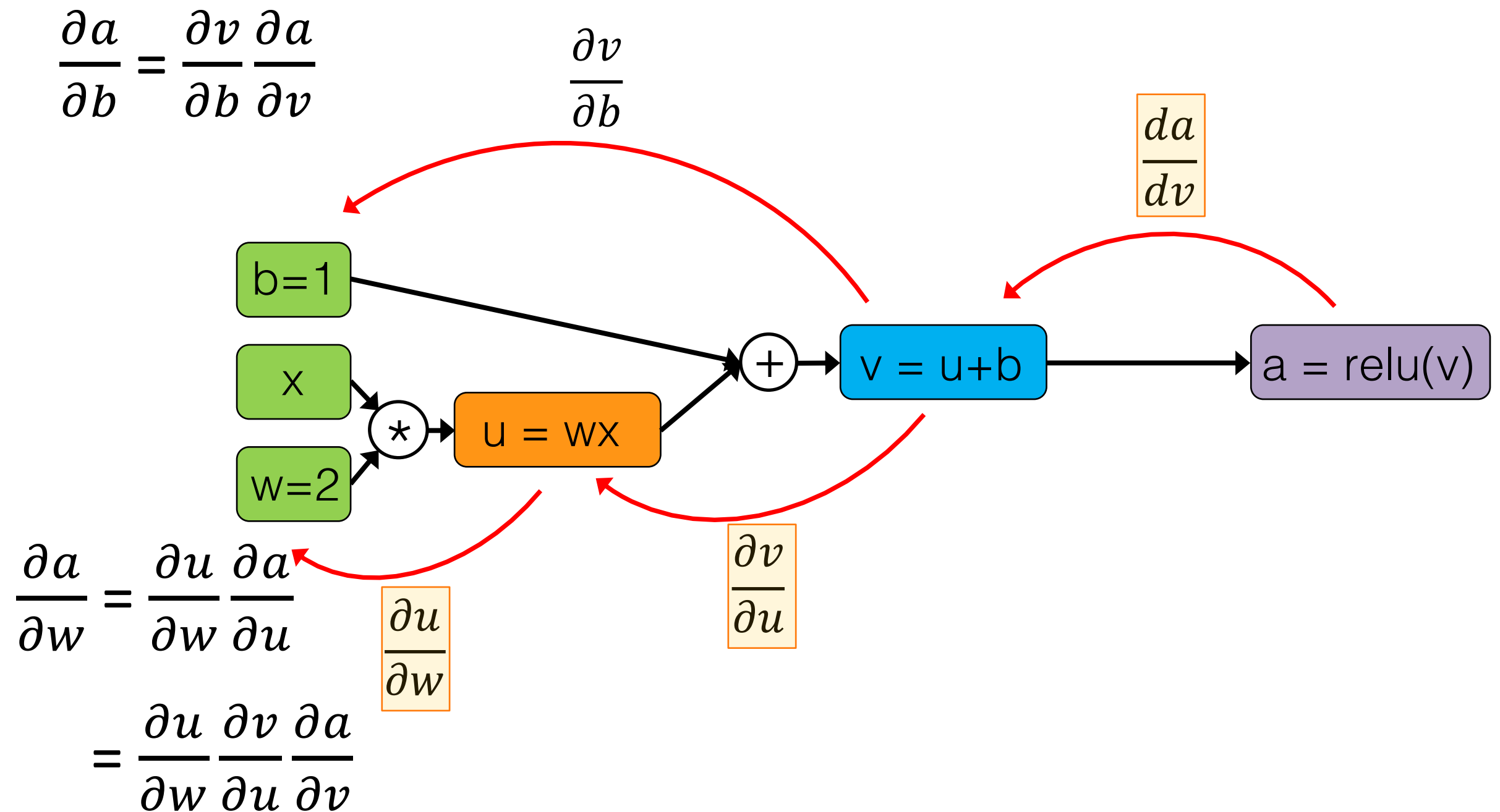


# Computation Graphs

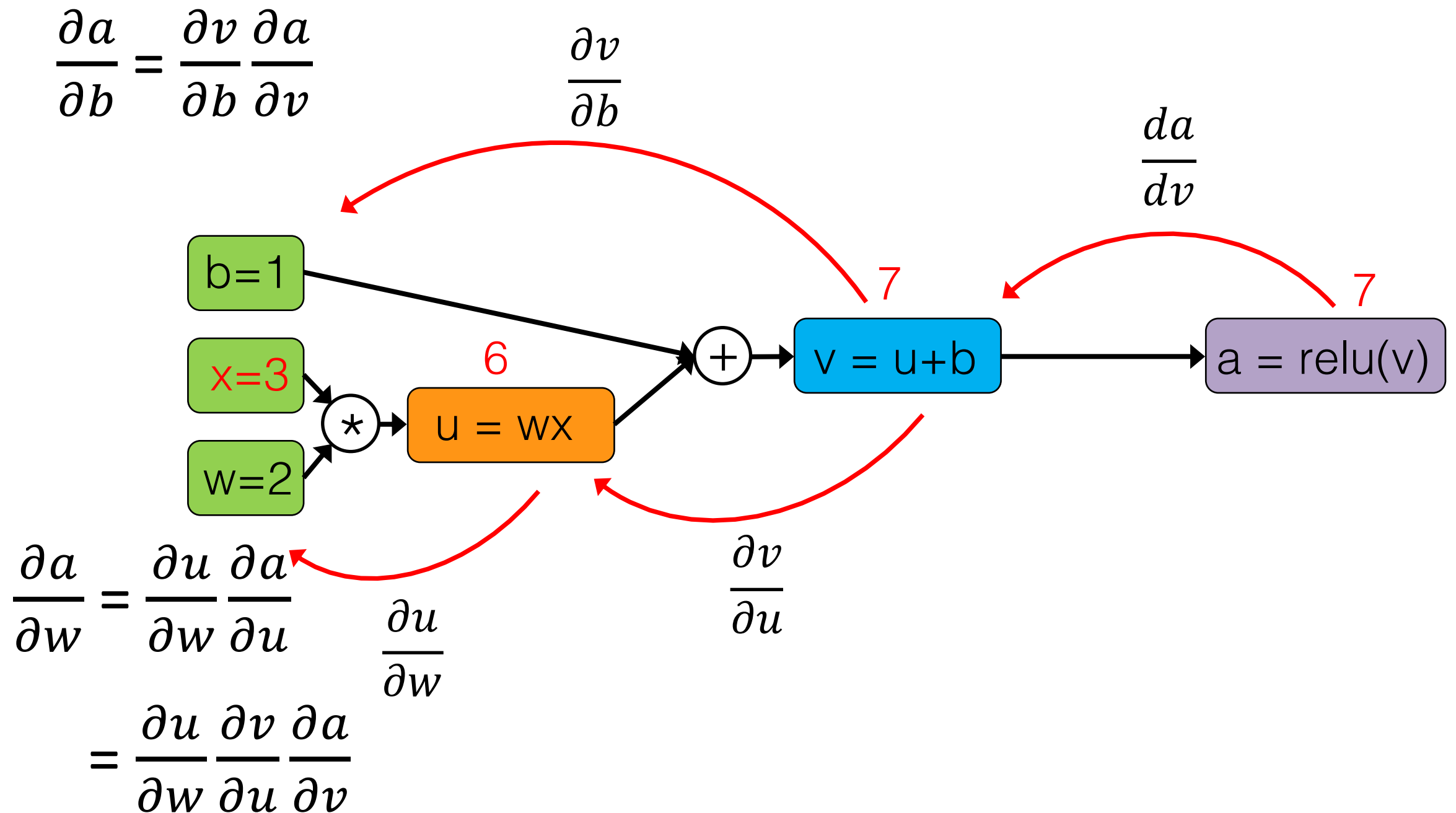
$$\frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \frac{\partial a}{\partial v}$$



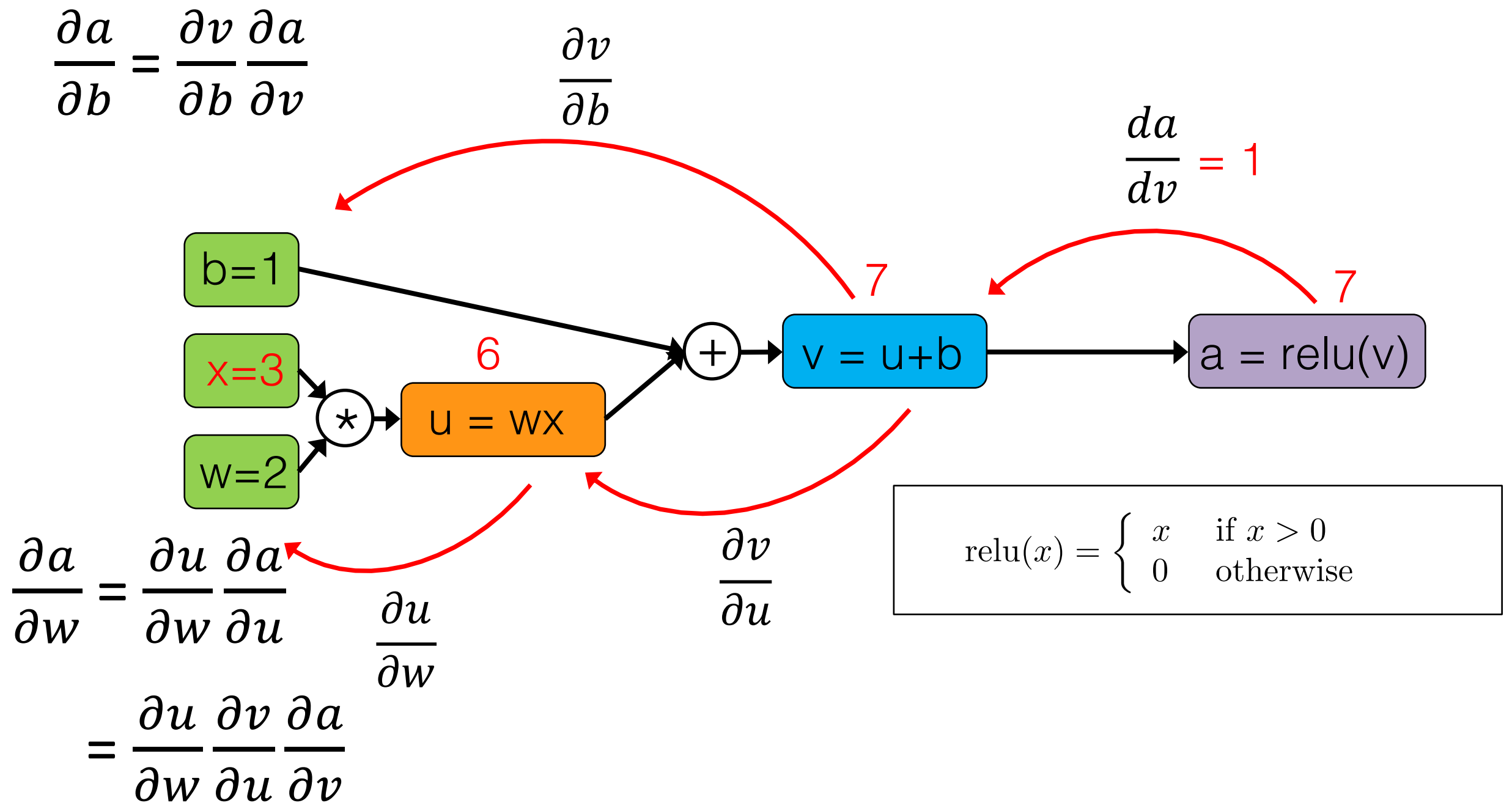
# Computation Graphs



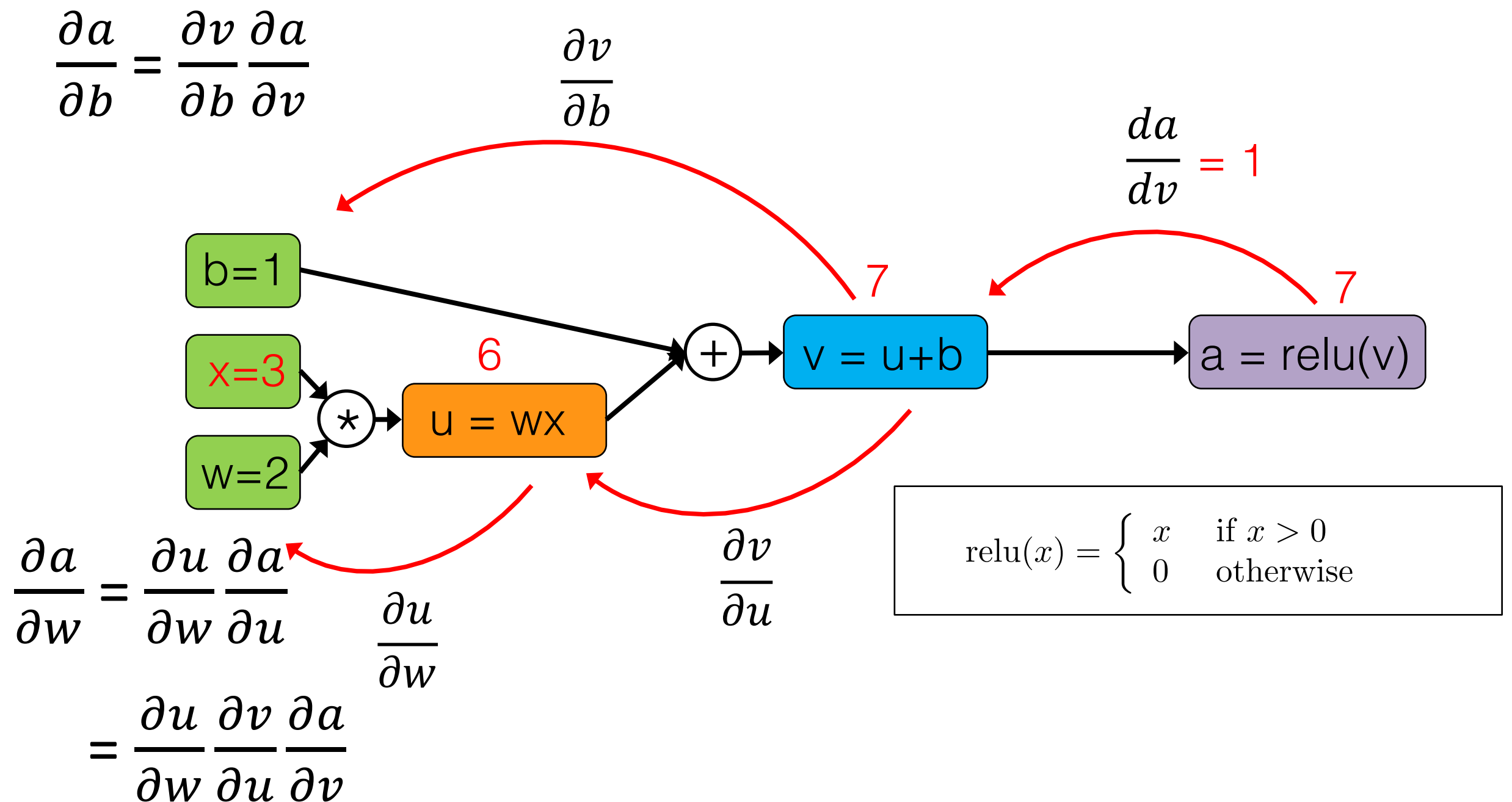
# Computation Graphs



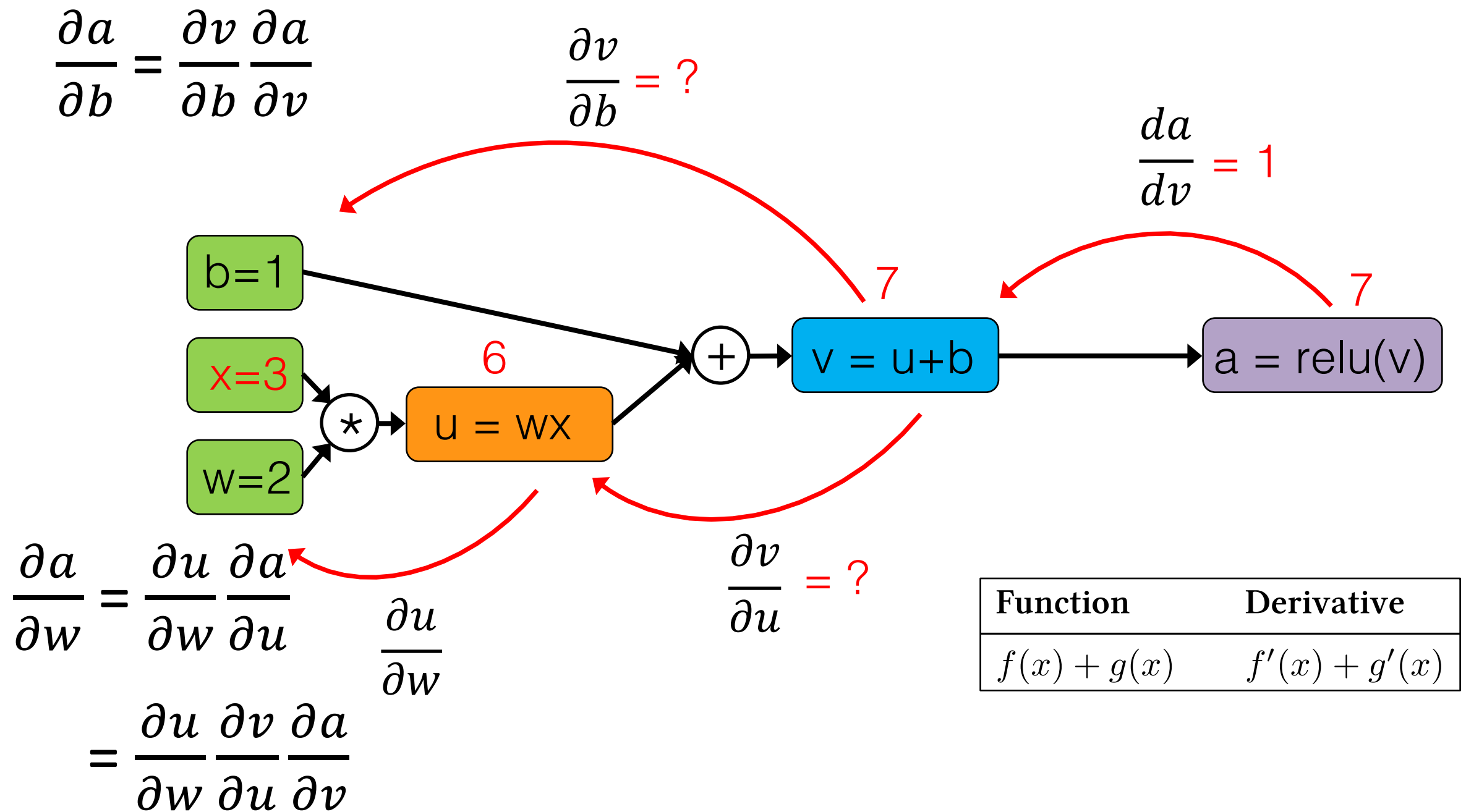
# Computation Graphs



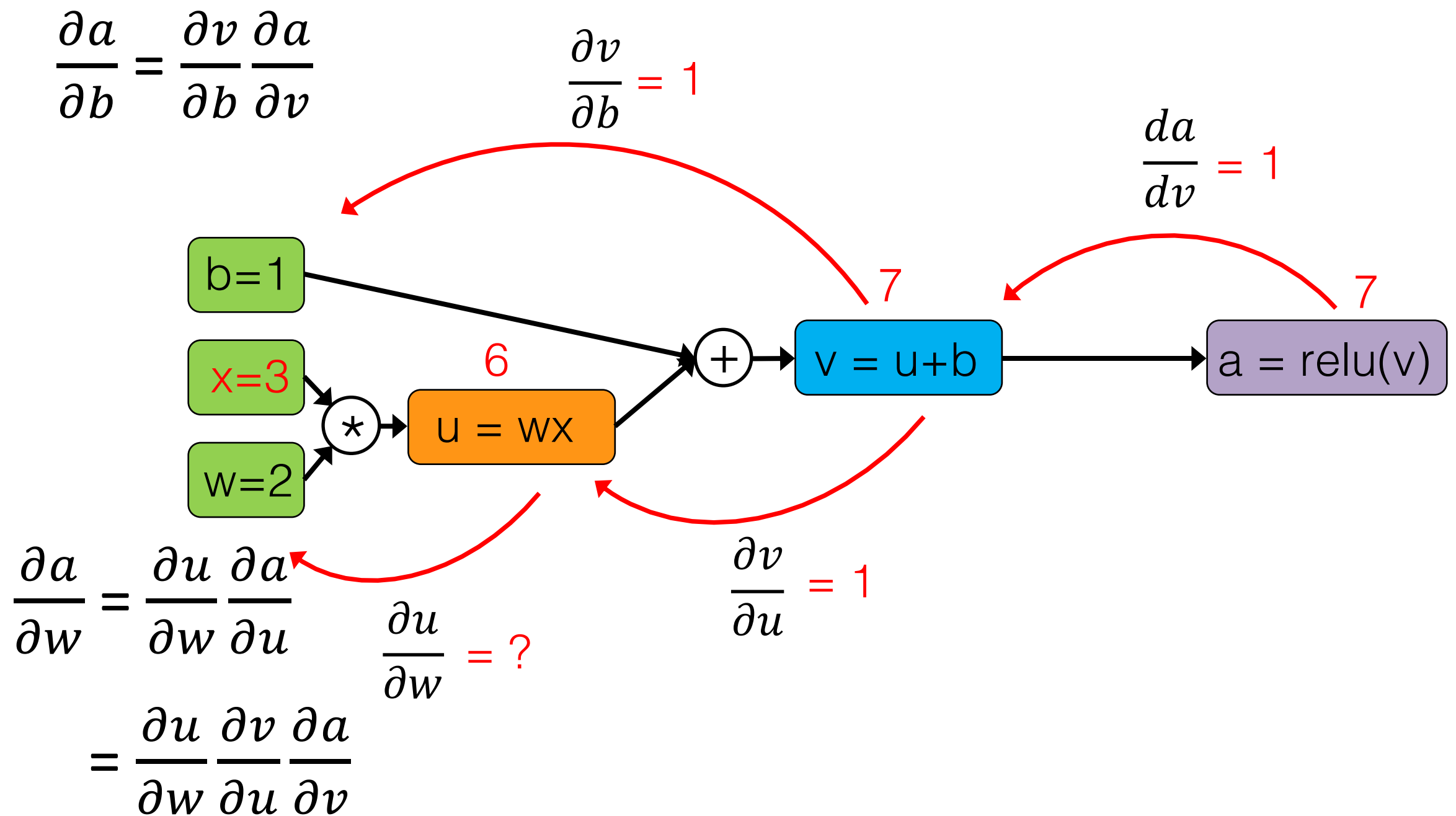
# Computation Graphs



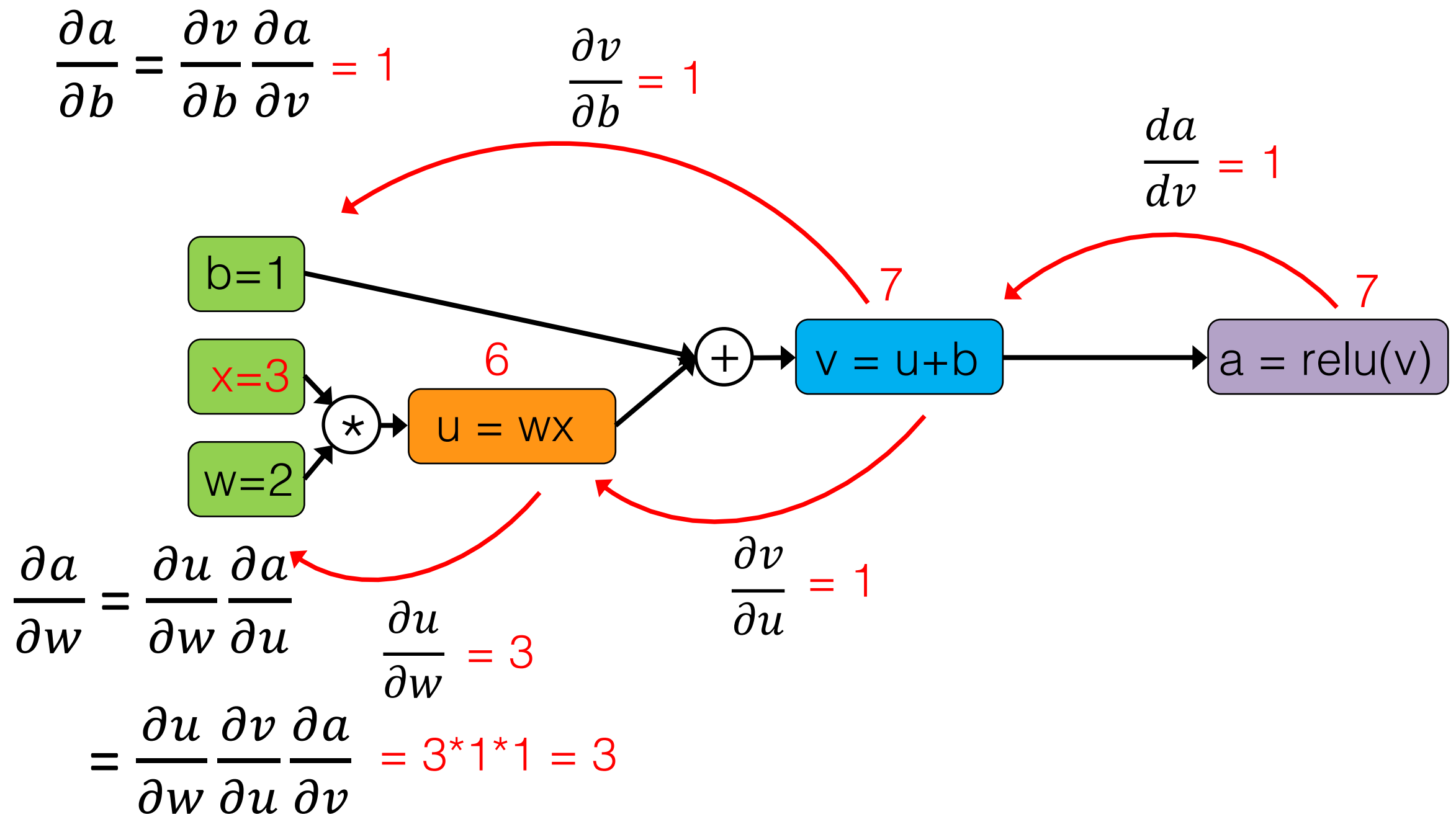
# Computation Graphs



# Computation Graphs



# Computation Graphs





# PyTorch Autograd Example

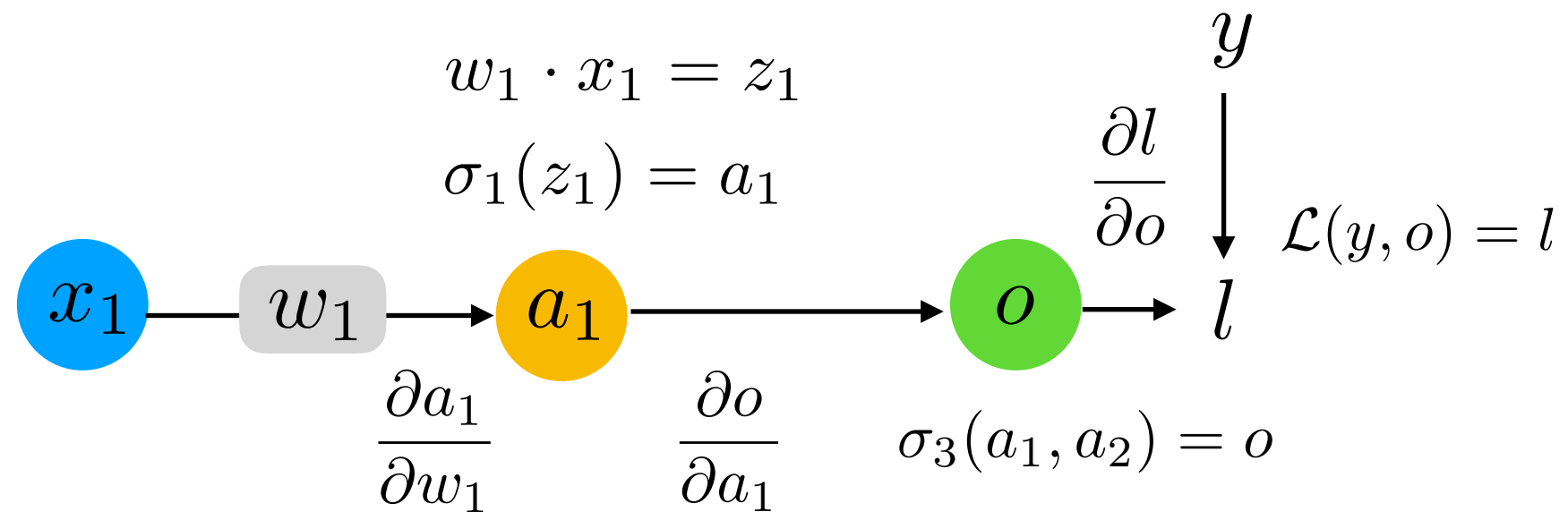
[https://github.com/rasbt/stat479-deep-learning-ss19/blob/master/L06\\_pytorch/code/pytorch-autograd.ipynb](https://github.com/rasbt/stat479-deep-learning-ss19/blob/master/L06_pytorch/code/pytorch-autograd.ipynb)

# Gradients of intermediate variables (usually not required in practice outside research)

[https://github.com/rasbt/stat479-deep-learning-ss19/blob/master/L06\\_pytorch/code/grad-intermediate-var.ipynb](https://github.com/rasbt/stat479-deep-learning-ss19/blob/master/L06_pytorch/code/grad-intermediate-var.ipynb)

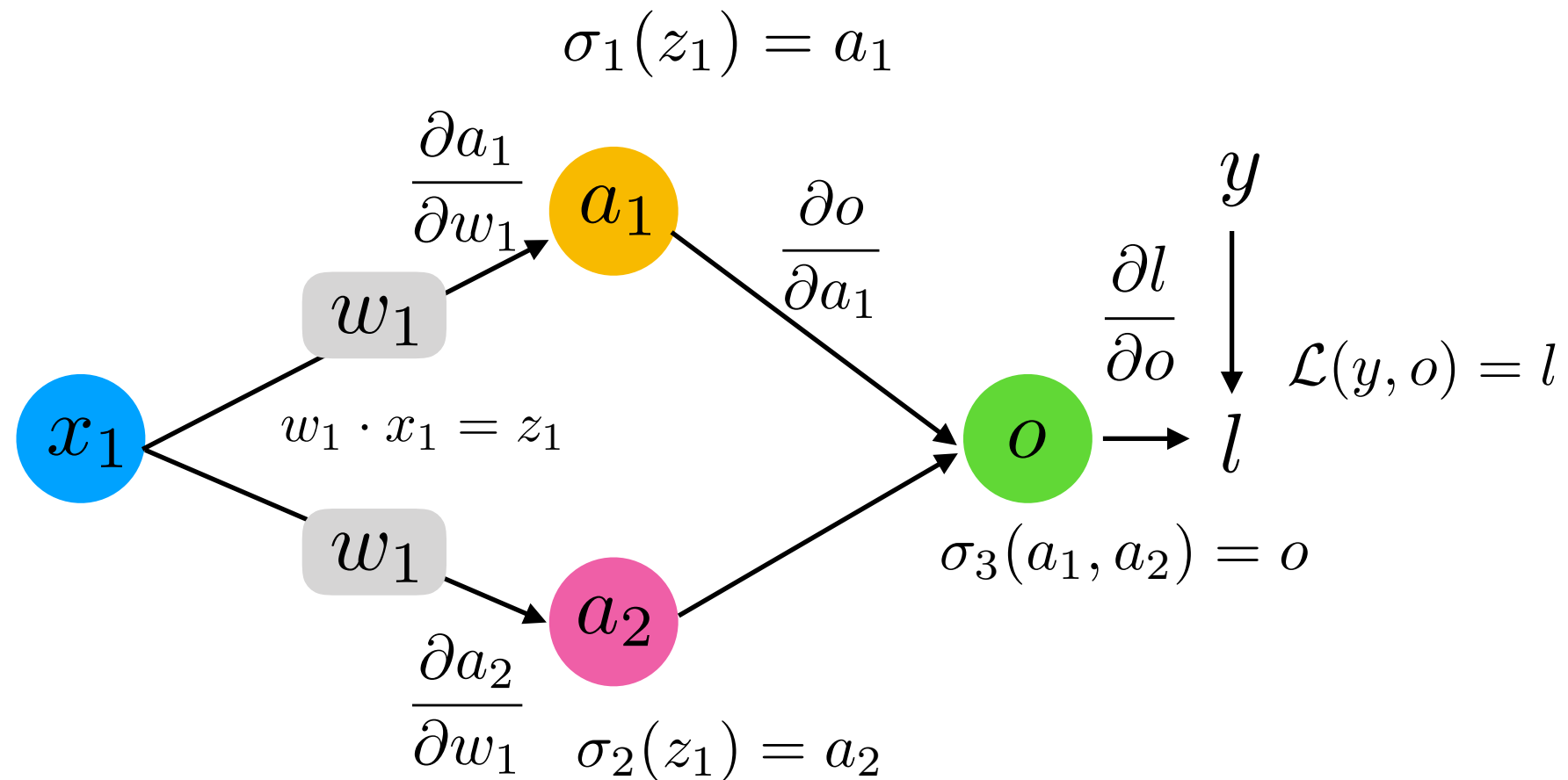
# Some More Computation Graphs

# Graph with Single Path



$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1} \quad (\text{univariate chain rule})$$

# Graph with Weight Sharing

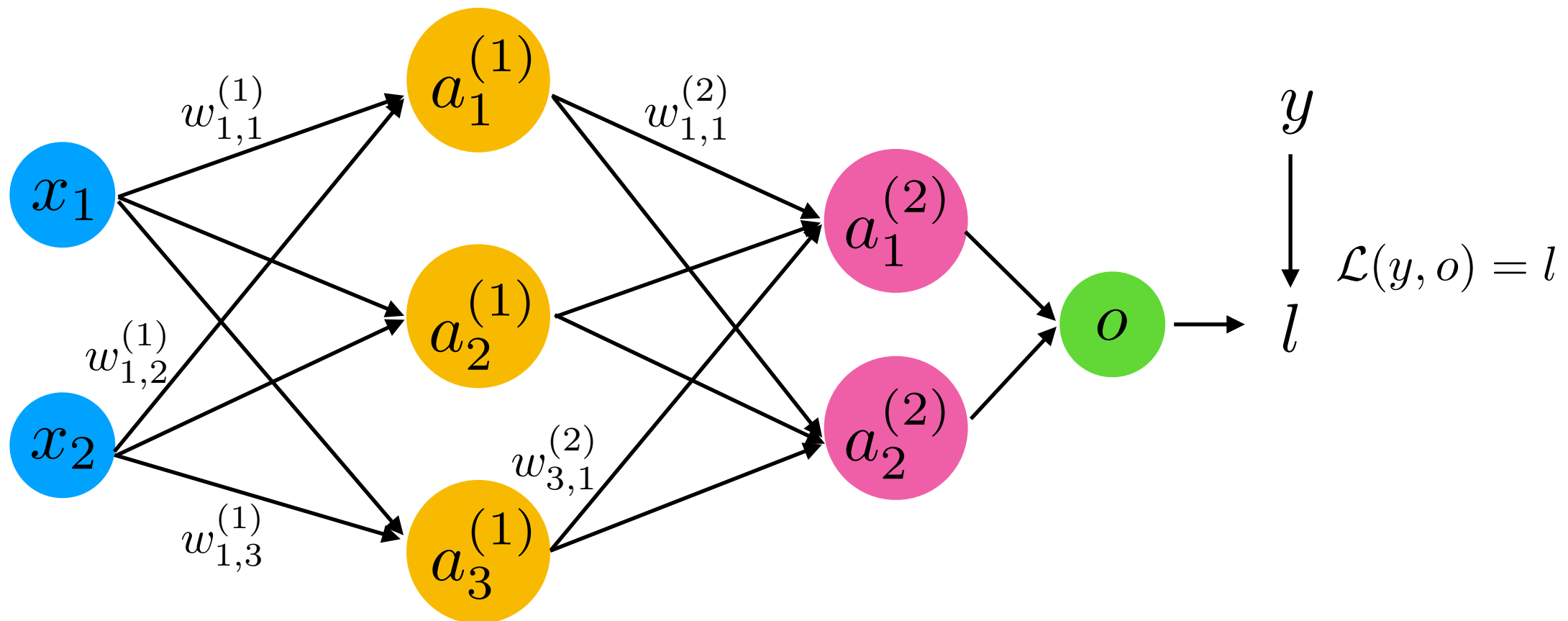


Upper path

$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2} \cdot \frac{\partial a_2}{\partial w_1} \quad (\text{multivariable chain rule})$$

Lower path

# Graph with Fully-Connected Layers (later in this course)



$$\begin{aligned} \frac{\partial l}{\partial w_{1,1}^{(1)}} &= \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} \\ &\quad + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} \end{aligned}$$

# PyTorch Usage: Step 1 (Definition)

```
class MultilayerPerceptron(torch.nn.Module):
```

Backward will be inferred automatically if we use the nn.Module class!

```
    def __init__(self, num_features, num_classes):  
        super(MultilayerPerceptron, self).__init__()
```

```
        ### 1st hidden layer
```

```
        self.linear_1 = torch.nn.Linear(num_feat, num_h1)
```

```
        ### 2nd hidden layer
```

```
        self.linear_2 = torch.nn.Linear(num_h1, num_h2)
```

```
        ### Output layer
```

```
        self.linear_out = torch.nn.Linear(num_h2, num_classes)
```

Define model parameters that will be instantiated when created an object of this class

```
    def forward(self, x):
```

```
        out = self.linear_1(x)
```

```
        out = F.relu(out)
```

```
        out = self.linear_2(out)
```

```
        out = F.relu(out)
```

```
        logits = self.linear_out(out)
```

```
        probas = F.log_softmax(logits, dim=1)
```

```
        return logits, probas
```

Define how and in what order the model parameters should be used in the forward pass

# PyTorch Usage: Step 2 (Creation)

```
torch.manual_seed(random_seed)
model = MultilayerPerceptron(num_features=num_features,
                             num_classes=num_classes)
model = model.to(device)
optimizer = torch.optim.SGD(model.parameters(),
                             lr=learning_rate)
```

Instantiate model  
(creates the model parameters)

Define an optimization method



# PyTorch Usage: Step 2 (Creation)

```
torch.manual_seed(random_seed)
model = MultilayerPerceptron(num_features=num_features,
                             num_classes=num_classes)
```

```
model = model.to(device)
```

```
optimizer = torch.optim.SGD(model.parameters(),
                             lr=learning_rate)
```

Optionally move model to GPU, where  
device e.g. `torch.device('cuda:0')`



# PyTorch Usage: Step 3 (Training)

```
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        cost = F.cross_entropy(probas, targets)
        optimizer.zero_grad()

        cost.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

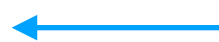
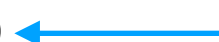



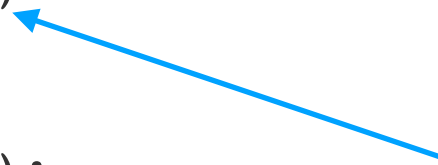
    model.eval()
    with torch.no_grad():
        # compute accuracy
```

Run for a specified number of epochs

Iterate over minibatches in epoch

If your model is on the GPU, data should also be on the GPU

# PyTorch Usage: Step 3 (Training)

```
for epoch in range(num_epochs):  
    model.train()  
    for batch_idx, (features, targets) in enumerate(train_loader):  
  
        features = features.view(-1, 28*28).to(device)  
        targets = targets.to(device)  
  
        ### FORWARD AND BACK PROP  
        logits, probas = model(features)  This will run the forward() method  
        loss = F.cross_entropy(probas, targets)  Define a loss function to optimize  
        optimizer.zero_grad()  Set the gradient to zero  
                                (could be non-zero from a previous forward pass)  
        loss.backward()   
                                 Compute the gradients, the backward is automatically  
                                constructed by "autograd" based on the forward()  
                                method and the loss function  
        ### UPDATE MODEL PARAMETERS  
        optimizer.step()   
                                Use the gradients to update the weights according to  
                                the optimization method (defined on the previous slide)  
                                E.g., for SGD,  $w := w + \text{learning\_rate} \times \text{gradient}$   
  
    model.eval()  
    with torch.no_grad():  
        # compute accuracy
```

# PyTorch Usage: Step 3 (Training)

```
for epoch in range(num_epochs):  
    model.train()  
    for batch_idx, (features, targets) in enumerate(train_loader):
```

```
        features = features.view(-1, 28*28).to(device)  
        targets = targets.to(device)
```

```
        ### FORWARD AND BACK PROP  
        logits, probas = model(features)  
        loss = F.cross_entropy(probas, targets)  
        optimizer.zero_grad()
```

```
        loss.backward()
```

```
        ### UPDATE MODEL PARAMETERS  
        optimizer.step()
```

```
model.eval()
```

```
with torch.no_grad():  
    # compute accuracy
```

For evaluation, set the model to eval mode (will be relevant later when we use Dropout or BatchNorm)

This prevents the computation graph for backpropagation from automatically being build in the background to save memory

# PyTorch ADALINE (neuron model) Example

[https://github.com/rasbt/stat479-deep-learning-ss19/blob/master/L06\\_pytorch/code/adaline-with-autograd.ipynb](https://github.com/rasbt/stat479-deep-learning-ss19/blob/master/L06_pytorch/code/adaline-with-autograd.ipynb)

# Objected-Oriented vs Functional\* API

\*Note that with "functional" I mean "functional programming" (one paradigm in CS)

```
import torch.nn.functional as F
```

```
class MultilayerPerceptron(torch.nn.Module):  
  
    def __init__(self, num_features, num_classes):  
        super(MultilayerPerceptron, self).__init__()  
  
        ### 1st hidden layer  
        self.linear_1 = torch.nn.Linear(num_features,  
                                         num_hidden_1)  
  
        ### 2nd hidden layer  
        self.linear_2 = torch.nn.Linear(num_hidden_1,  
                                         num_hidden_2)  
  
        ### Output layer  
        self.linear_out = torch.nn.Linear(num_hidden_2,  
                                           num_classes)  
  
    def forward(self, x):  
        out = self.linear_1(x)  
        out = F.relu(out)  
        out = self.linear_2(out)  
        out = F.relu(out)  
        logits = self.linear_out(out)  
        probas = F.log_softmax(logits, dim=1)  
        return logits, probas
```

Unnecessary because these functions don't need to store a state but maybe helpful for keeping track of order of ops (when implementing "forward")

```
class MultilayerPerceptron(torch.nn.Module):
```

```
    def __init__(self, num_features, num_classes):  
        super(MultilayerPerceptron, self).__init__()  
  
        ### 1st hidden layer  
        self.linear_1 = torch.nn.Linear(num_features,  
                                         num_hidden_1)  
        self.relu1 = torch.nn.ReLU()  
  
        ### 2nd hidden layer  
        self.linear_2 = torch.nn.Linear(num_hidden_1,  
                                         num_hidden_2)  
        self.relu2 = torch.nn.ReLU()  
  
        ### Output layer  
        self.linear_out = torch.nn.Linear(num_hidden_2,  
                                           num_classes)  
        self.softmax = torch.nn.Softmax()  
  
    def forward(self, x):  
        out = self.linear_1(x)  
        out = self.relu1(out)  
        out = self.linear_2(out)  
        out = self.relu2(out)  
        logits = self.linear_out(out)  
        probas = self.softmax(logits, dim=1)  
        return logits, probas
```

# Objected-Oriented vs Functional API

Using "Sequential"

```
import torch.nn.functional as F
```

```
class MultilayerPerceptron(torch.nn.Module):  
  
    def __init__(self, num_features, num_classes):  
        super(MultilayerPerceptron, self).__init__()  
  
        ### 1st hidden layer  
        self.linear_1 = torch.nn.Linear(num_features,  
                                         num_hidden_1)  
  
        ### 2nd hidden layer  
        self.linear_2 = torch.nn.Linear(num_hidden_1,  
                                         num_hidden_2)  
  
        ### Output layer  
        self.linear_out = torch.nn.Linear(num_hidden_2,  
                                           num_classes)  
  
    def forward(self, x):  
        out = self.linear_1(x)  
        out = F.relu(out)  
        out = self.linear_2(out)  
        out = F.relu(out)  
        logits = self.linear_out(out)  
        probas = F.log_softmax(logits, dim=1)  
        return logits, probas
```

```
class MultilayerPerceptron(torch.nn.Module):  
  
    def __init__(self, num_features, num_classes):  
        super(MultilayerPerceptron, self).__init__()  
  
        self.my_network = torch.nn.Sequential(  
            torch.nn.Linear(num_features, num_hidden_1),  
            torch.nn.ReLU(),  
            torch.nn.Linear(num_hidden_1, num_hidden_2),  
            torch.nn.ReLU(),  
            torch.nn.Linear(num_hidden_2, num_classes)  
        )  
  
    def forward(self, x):  
        logits = self.my_network(x)  
        probas = F.softmax(logits, dim=1)  
        return logits, probas
```

Much more compact and clear, but "forward" may be harder to debug if there are errors (we cannot simply add breakpoints or insert "print" statements)

**More PyTorch features will be introduced step-by-step later in this course when we start working with more complex networks, including**

- Running code on the GPU
- Using efficient data loaders
- Splitting networks across different GPUs



# Reading Assignments

- What is PyTorch

[https://pytorch.org/tutorials/beginner/blitz/tensor\\_tutorial.html#sphx-glr-beginner-blitz-tensor-tutorial-py](https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#sphx-glr-beginner-blitz-tensor-tutorial-py)

- Autograd: Automatic Differentiation

[https://pytorch.org/tutorials/beginner/blitz/autograd\\_tutorial.html#sphx-glr-beginner-blitz-autograd-tutorial-py](https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html#sphx-glr-beginner-blitz-autograd-tutorial-py)