

Lecture 06

Automatic Differentiation with PyTorch

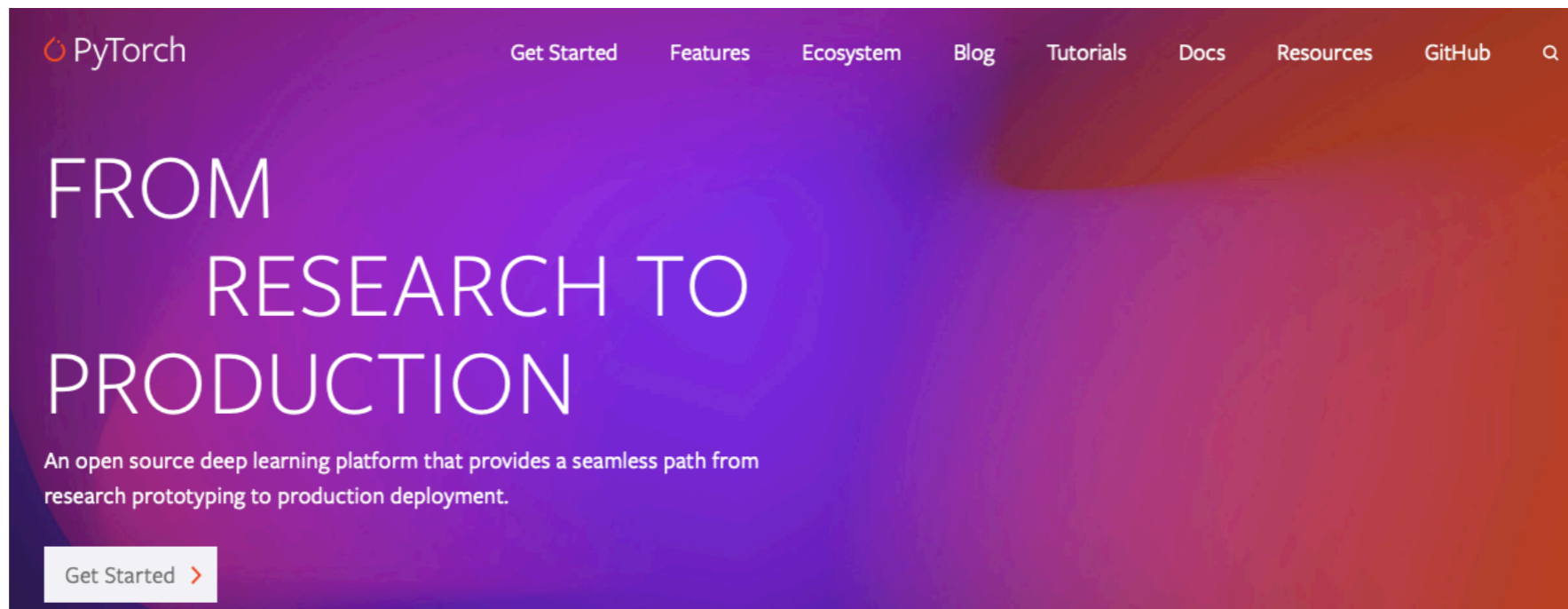
STAT 479: Deep Learning, Spring 2019

Sebastian Raschka

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



PyTorch

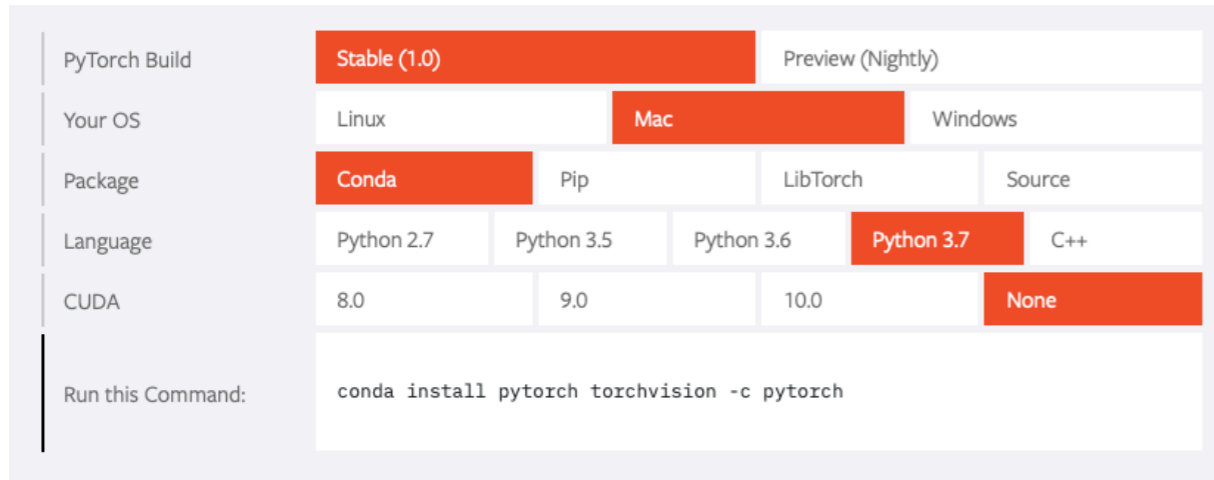


The screenshot shows the PyTorch website homepage. At the top left is the PyTorch logo. To its right is a navigation menu with links for 'Get Started', 'Features', 'Ecosystem', 'Blog', 'Tutorials', 'Docs', 'Resources', and 'GitHub'. The main content area has a purple-to-red gradient background. It features the text 'FROM RESEARCH TO PRODUCTION' in large white letters. Below this is a subtitle: 'An open source deep learning platform that provides a seamless path from research prototyping to production deployment.' At the bottom left of this section is a 'Get Started >' button.

<https://pytorch.org/>

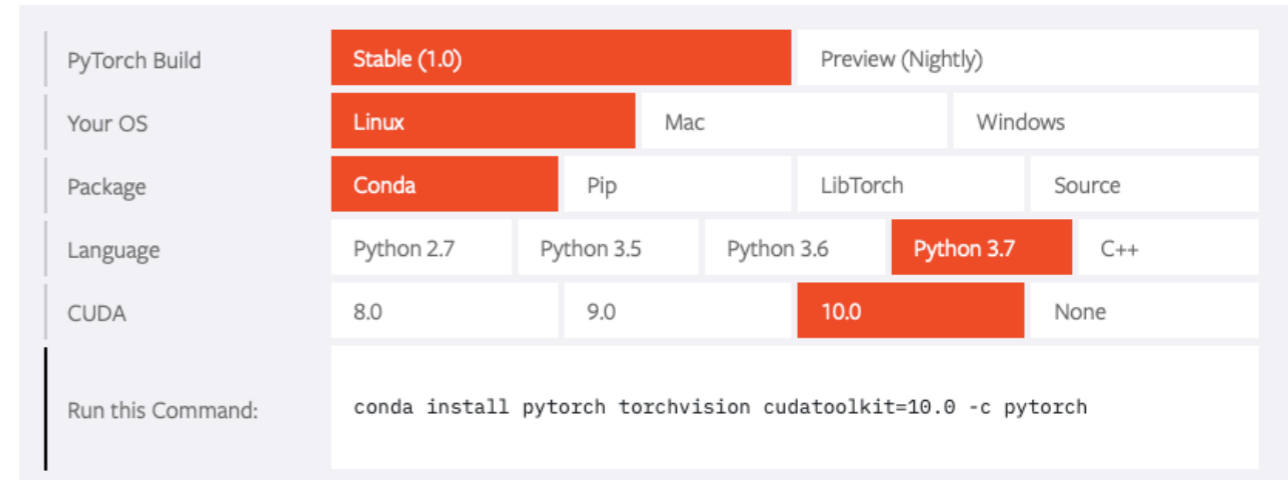
Installation

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	C++
CUDA	8.0	9.0	10.0	None	
Run this Command:	<code>conda install pytorch torchvision -c pytorch</code>				

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	C++
CUDA	8.0	9.0	10.0	None	
Run this Command:	<code>conda install pytorch torchvision cudatoolkit=10.0 -c pytorch</code>				

<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)

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.

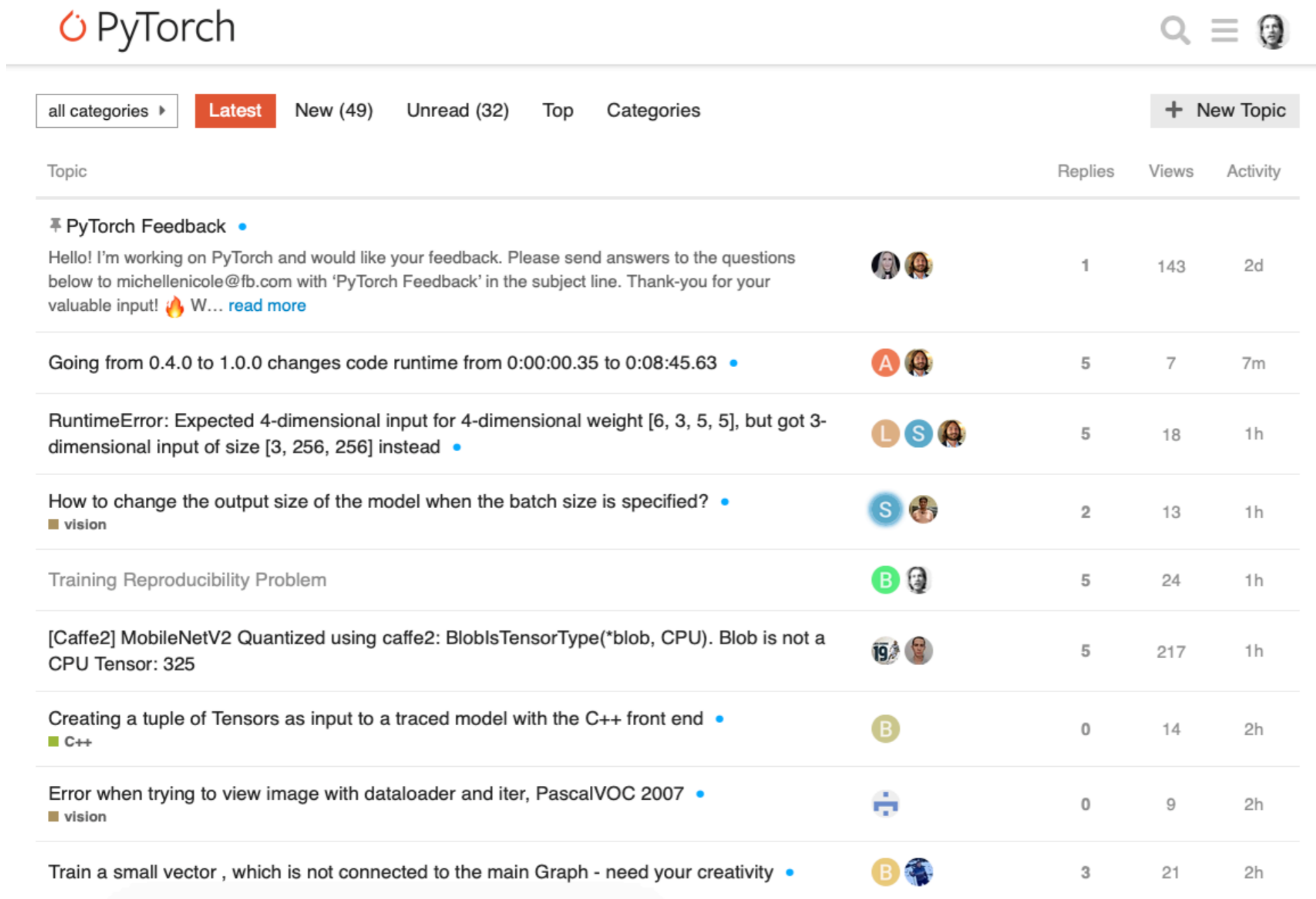


Examples

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

<https://pytorch.org/resources>

Very Active & Friendly Community and Help/Discussion Forum



The screenshot shows the PyTorch forum interface. At the top left is the PyTorch logo. On the right, there are search, menu, and profile icons. Below the header, there are navigation tabs: 'all categories', 'Latest' (highlighted in red), 'New (49)', 'Unread (32)', 'Top', and 'Categories'. A '+ New Topic' button is on the right. The main content is a table of forum posts.

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 •	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 •	5	18	1h
How to change the output size of the model when the batch size is specified? • ■ vision	2	13	1h
Training Reproducibility Problem	5	24	1h
[Caffe2] MobileNetV2 Quantized using caffe2: BlobIsTensorType(*blob, CPU). Blob is not a CPU Tensor: 325	5	217	1h
Creating a tuple of Tensors as input to a traced model with the C++ front end • ■ C++	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 •	3	21	2h

<https://pytorch.org/resources>



DEEP LEARNING WITH PYTORCH: A 60 MINUTE BLITZ

Author: Soumith Chintala

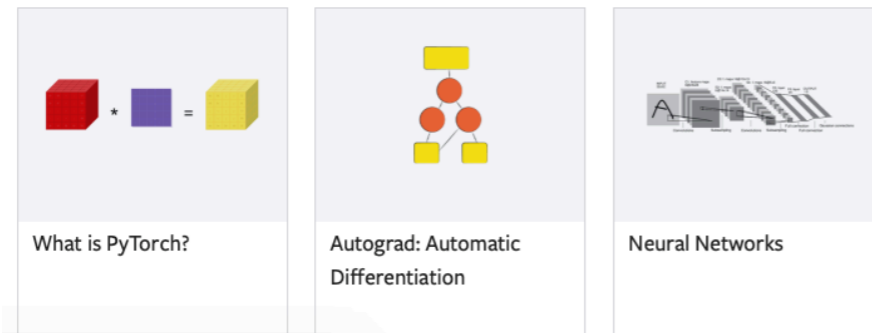
Goal of this tutorial:

- 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.



DEEP LEARNING WITH PYTORCH: A 60 MINUTE BLITZ

https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

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

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

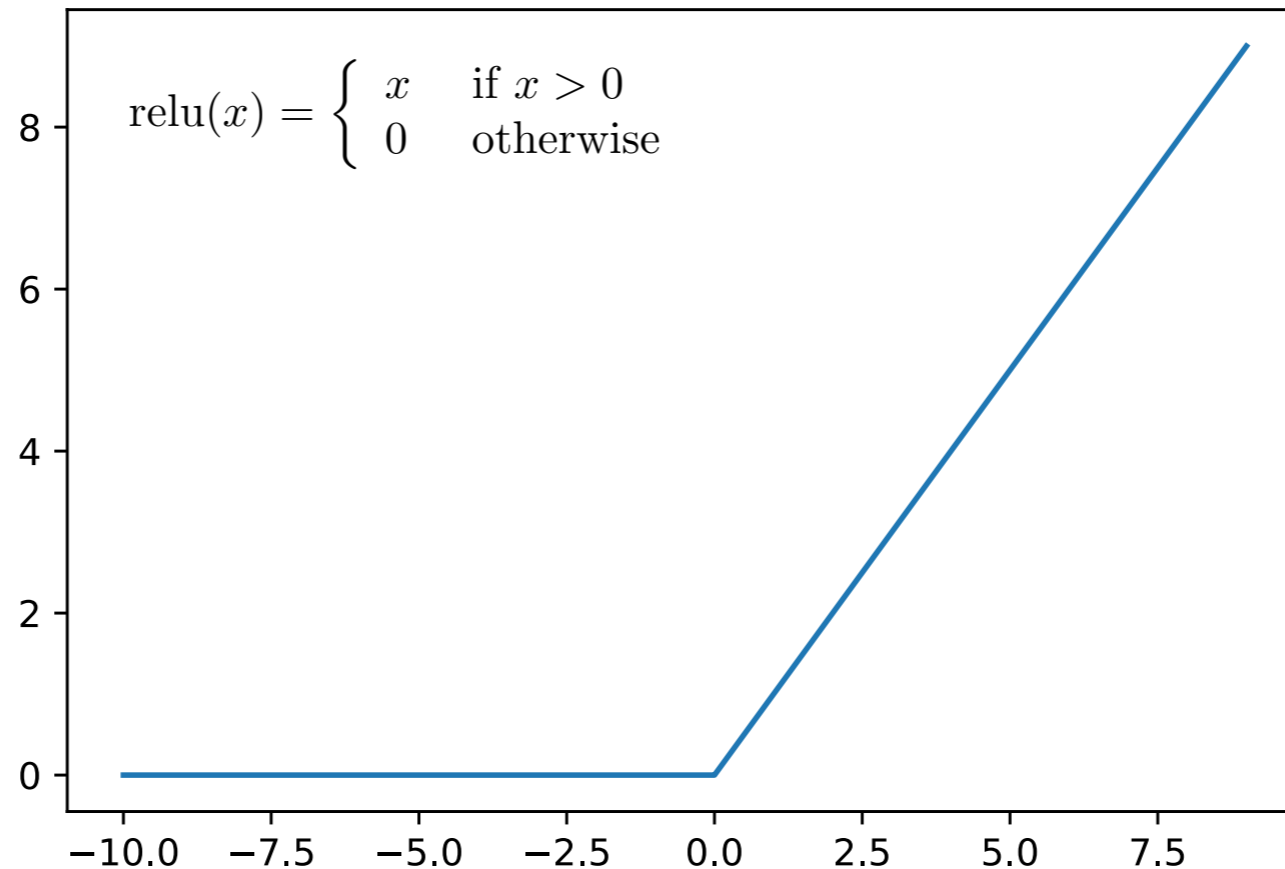
Text source: https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html#sphx-glr-beginner-blitz-autograd-tutorial-py

**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

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

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

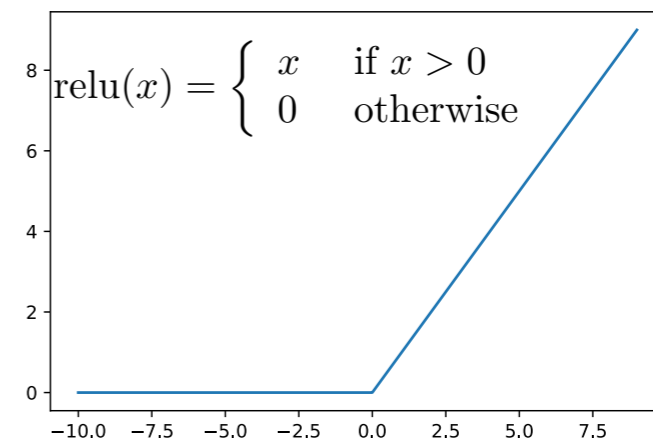
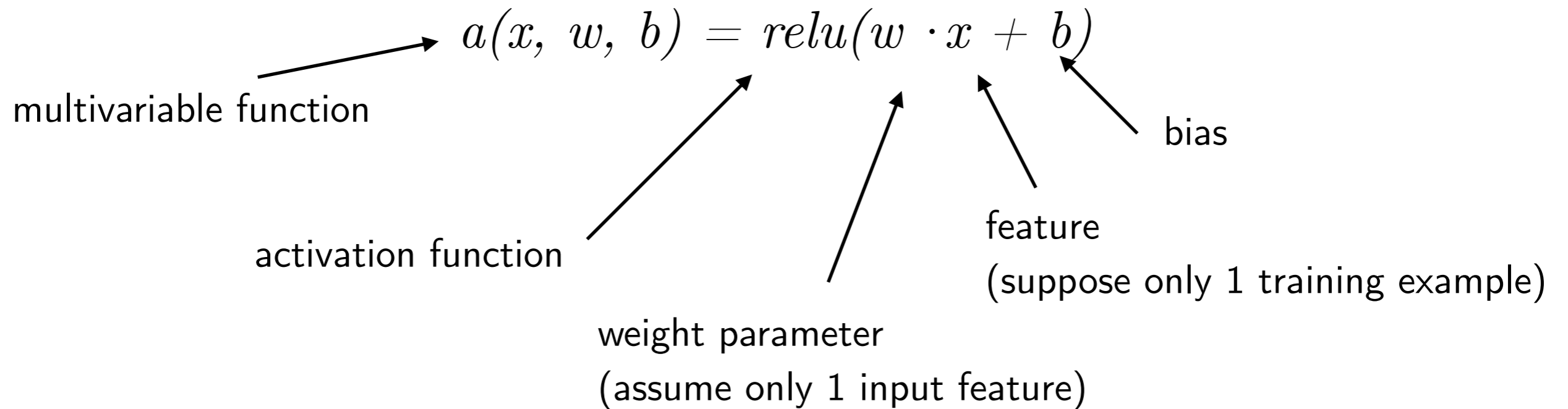
$$f'(x) = \lim_{x \rightarrow 0} \frac{\max(0, x + \Delta x) - \max(0, x)}{\Delta x}$$

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

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

Computation Graphs

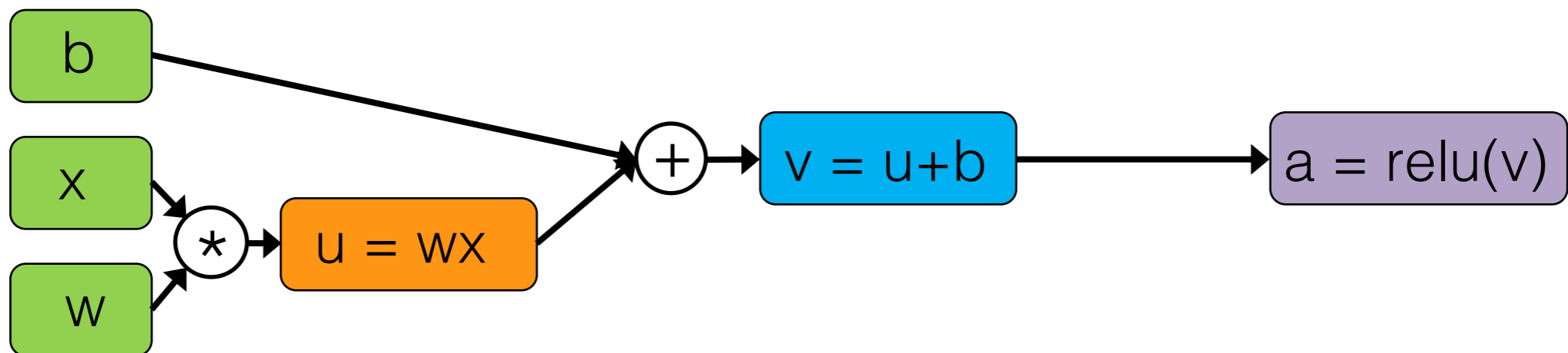
Suppose we have the following activation function:



Computation Graphs

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

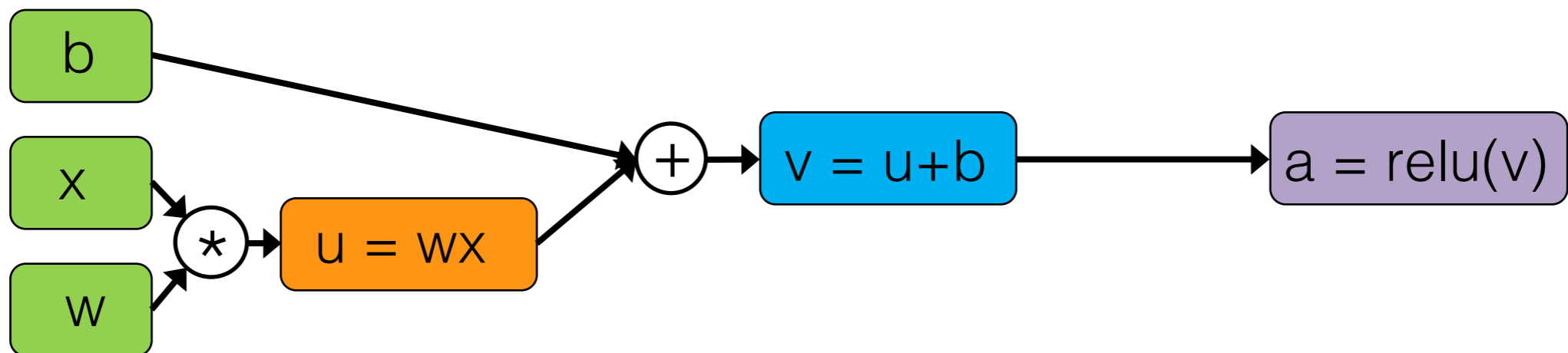
$\underbrace{\hspace{10em}}_u$
 $\underbrace{\hspace{15em}}_v$



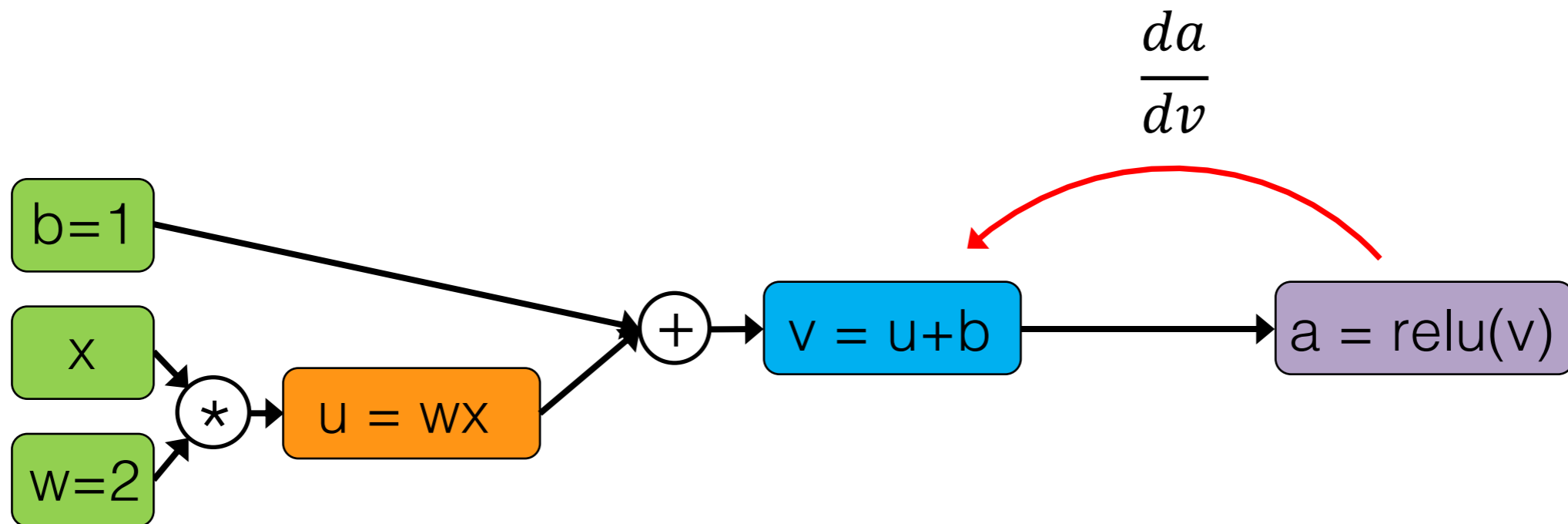
Computation Graphs

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

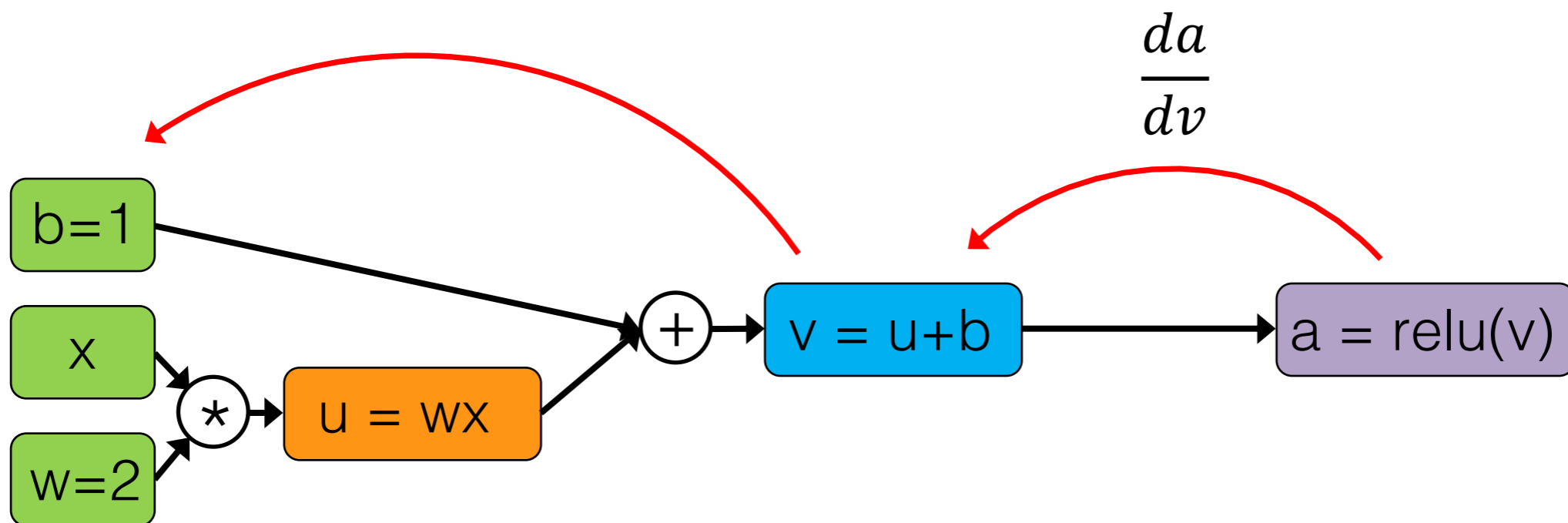
$\underbrace{\hspace{10em}}_u$
 $\underbrace{\hspace{15em}}_v$



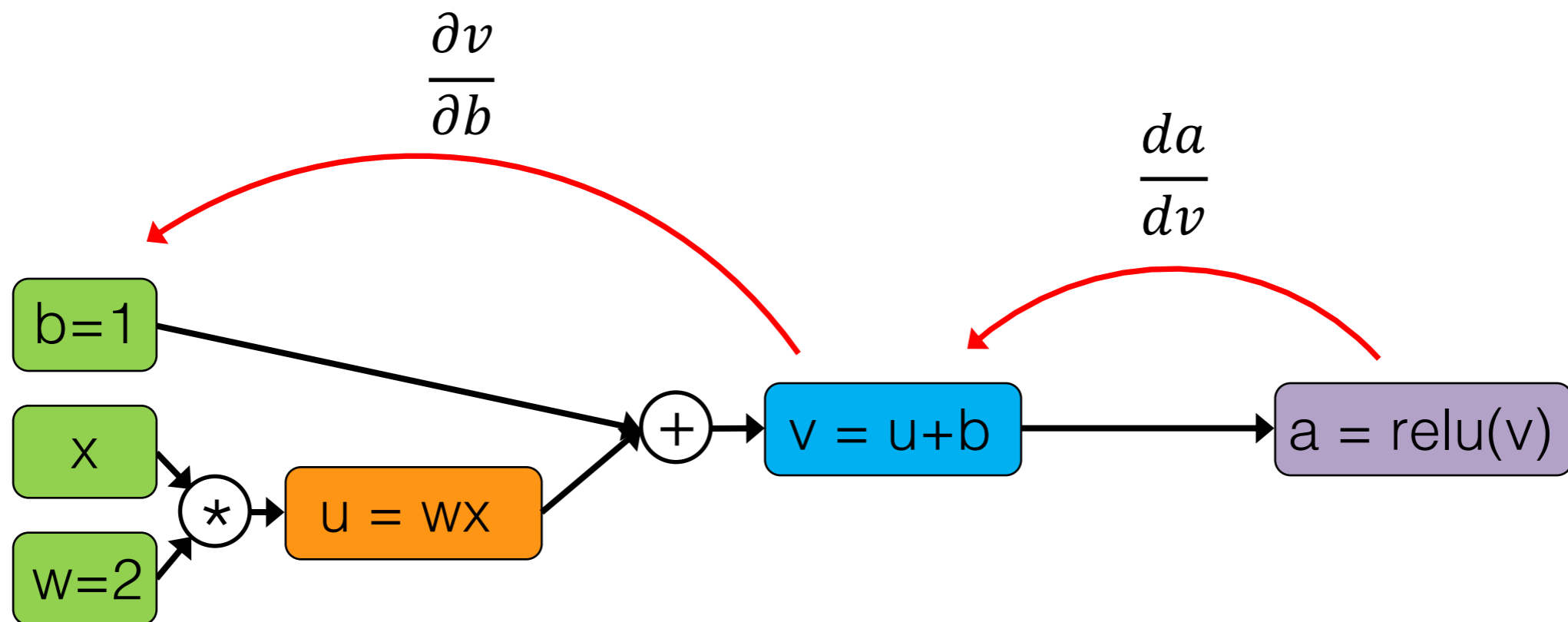
Computation Graphs



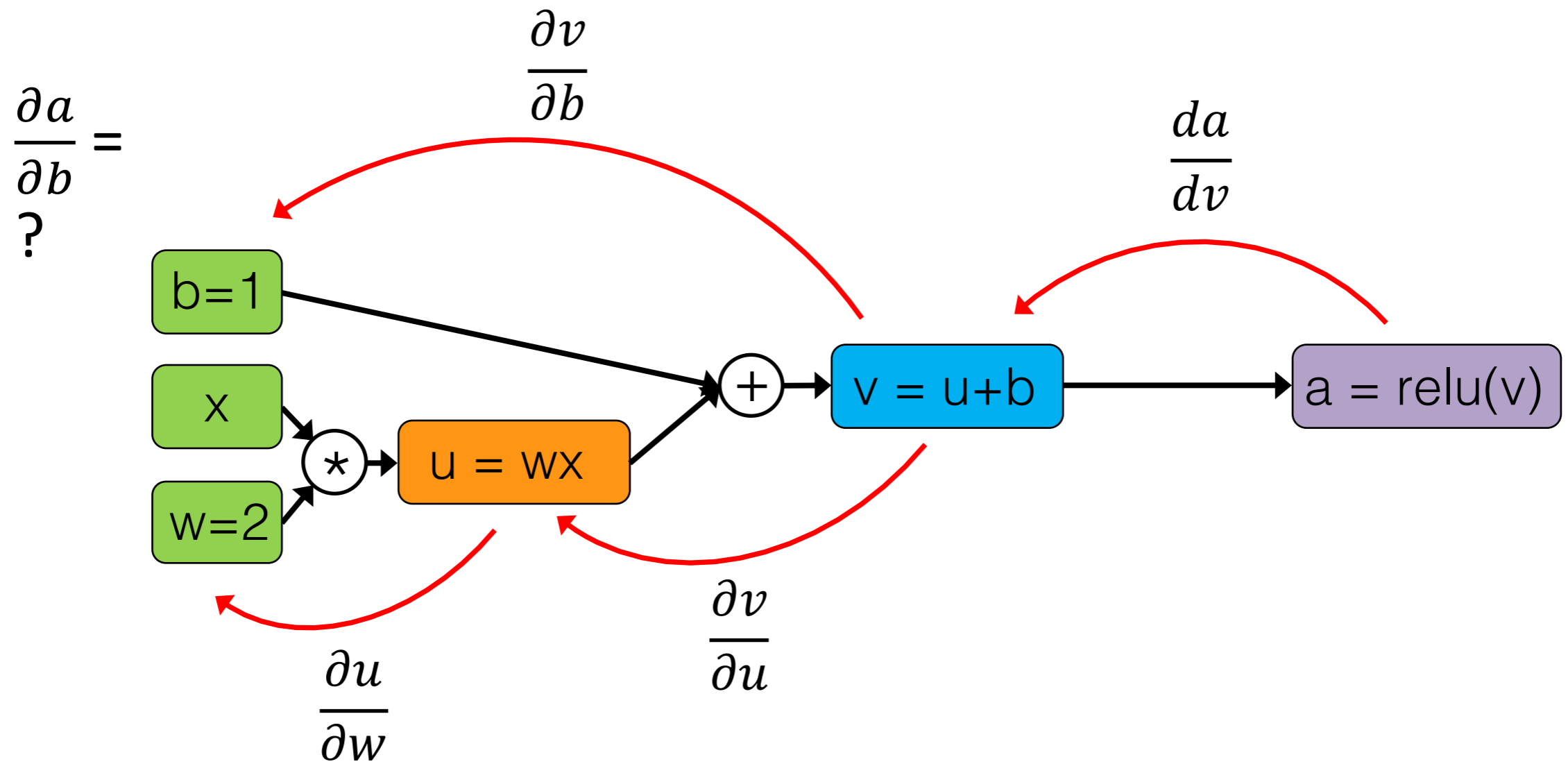
Computation Graphs



Computation Graphs

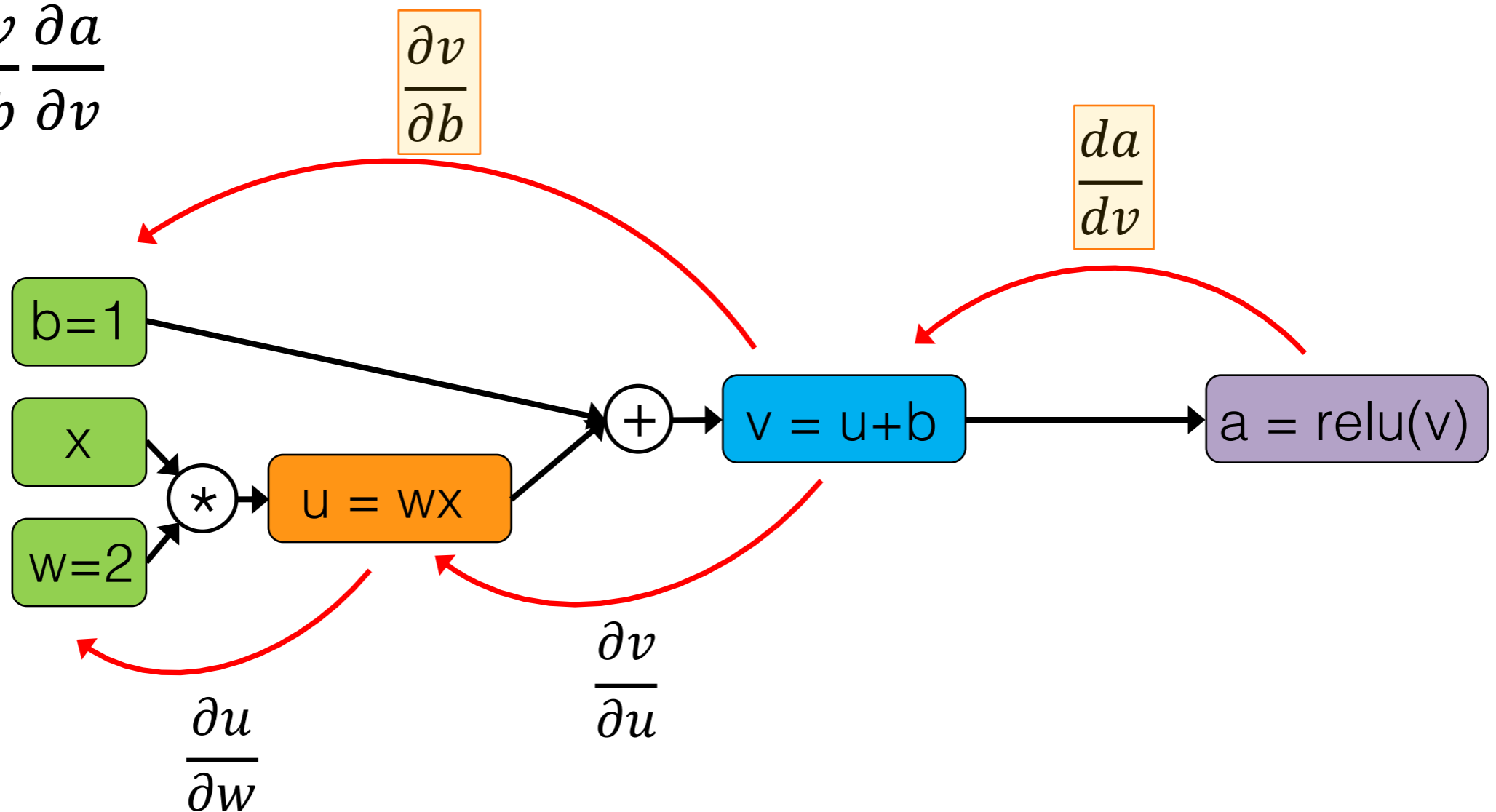


Computation Graphs

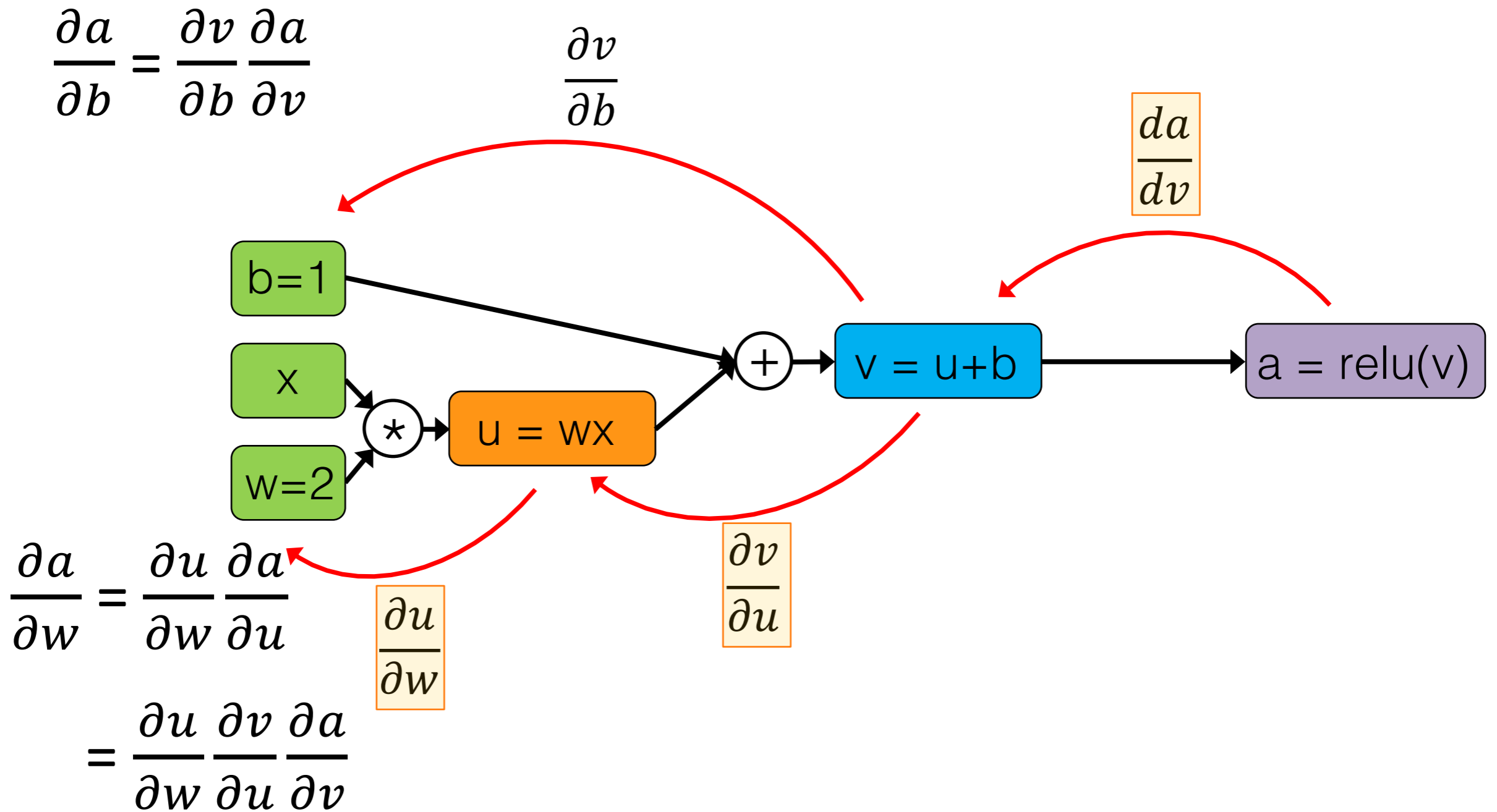


Computation Graphs

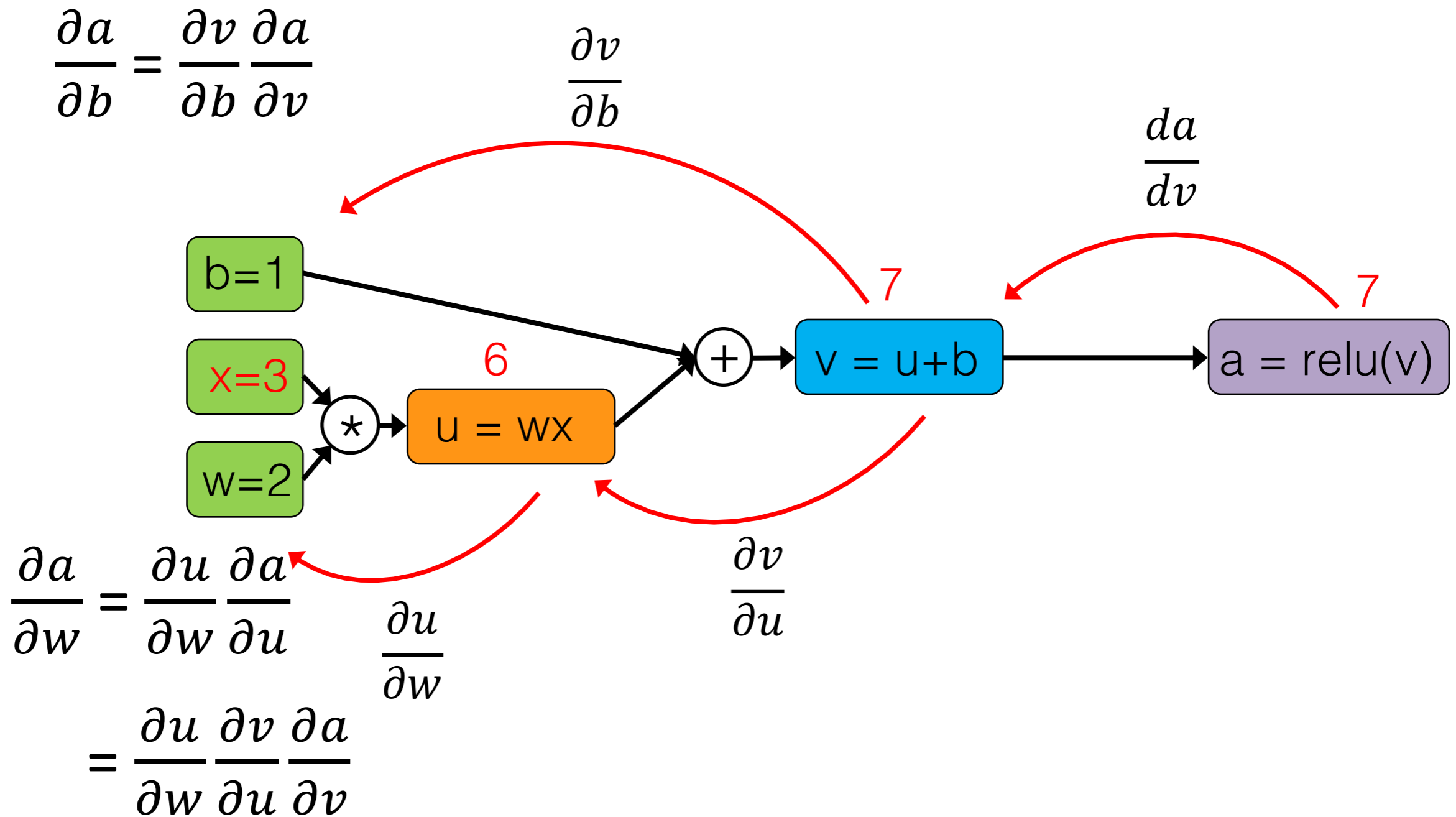
$$\frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \frac{\partial a}{\partial v}$$



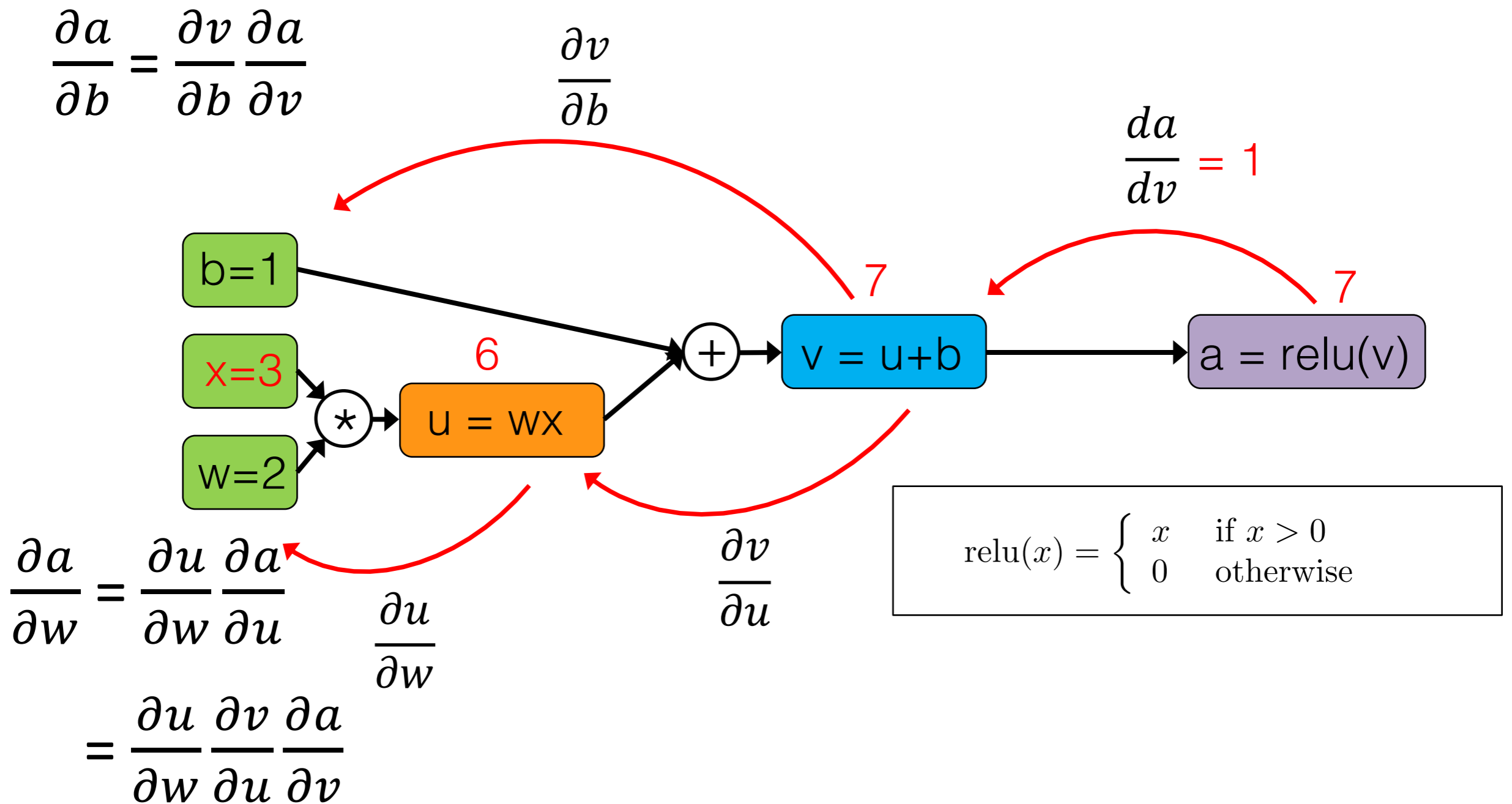
Computation Graphs



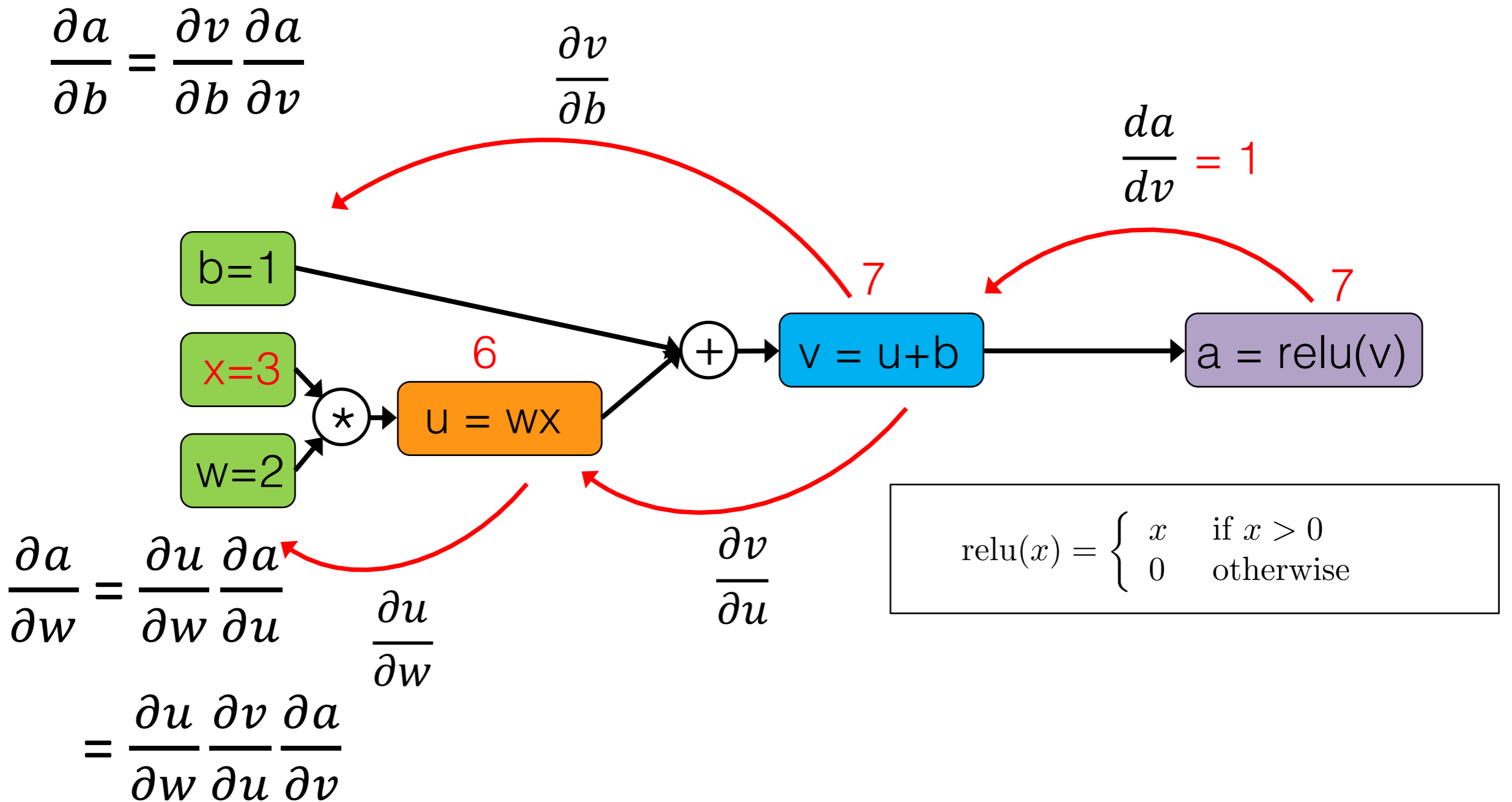
Computation Graphs



Computation Graphs



Computation Graphs

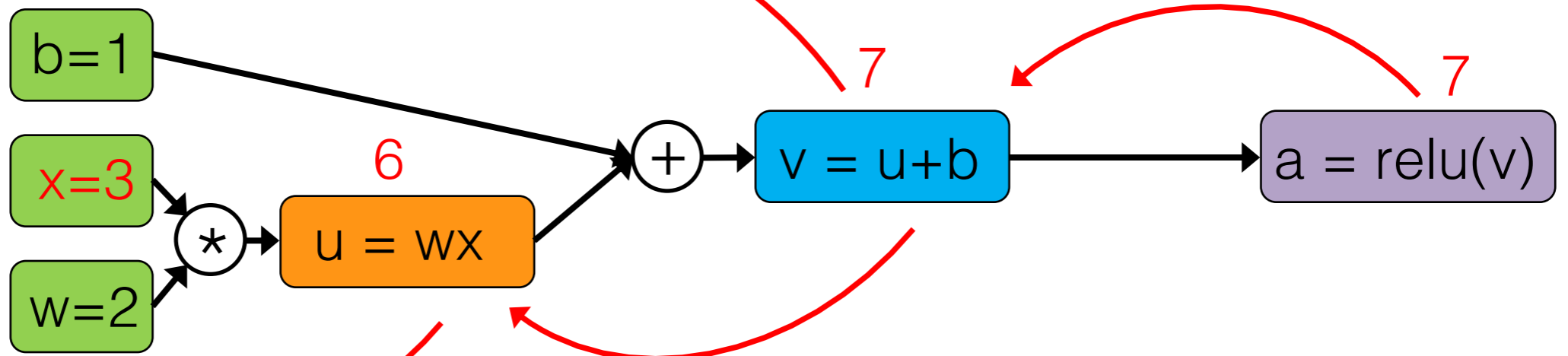


Computation Graphs

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

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

$$\frac{da}{dv} = 1$$



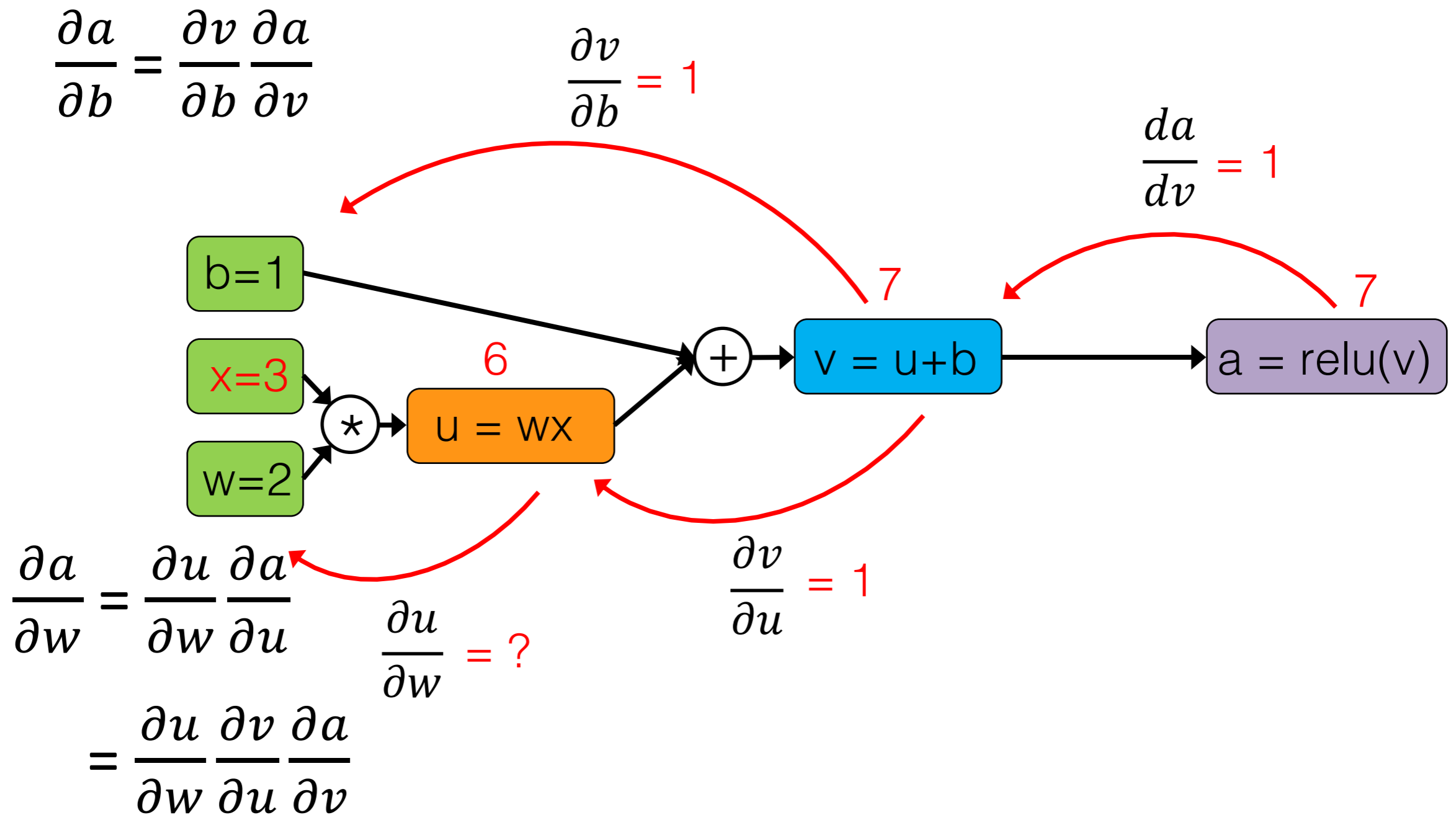
$$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w} \frac{\partial a}{\partial u}$$

$$\frac{\partial v}{\partial u} = ?$$

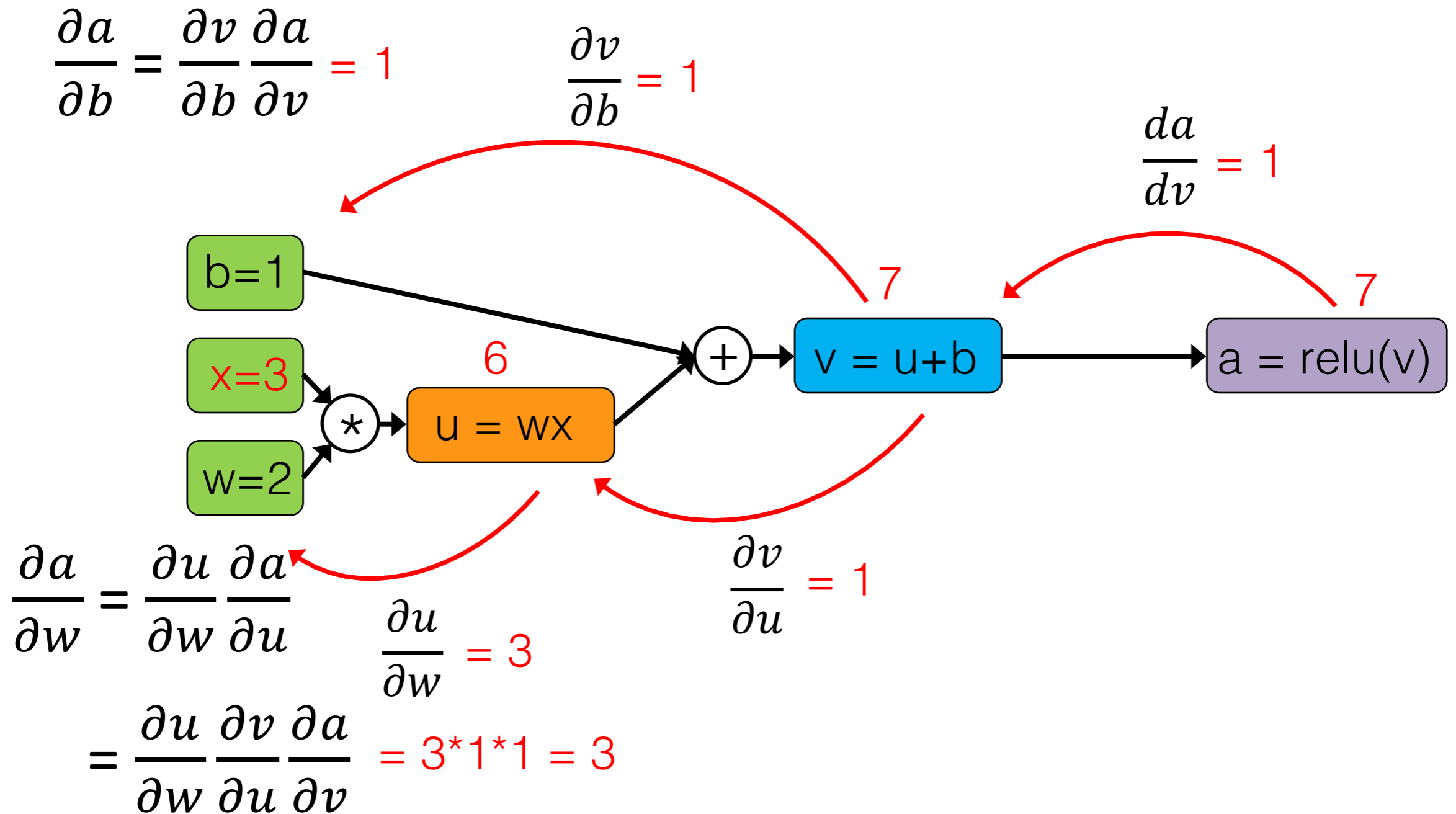
$$= \frac{\partial u}{\partial w} \frac{\partial v}{\partial u} \frac{\partial a}{\partial v}$$

Function	Derivative
$f(x) + g(x)$	$f'(x) + g'(x)$

Computation Graphs



Computation Graphs



PyTorch Autograd Example

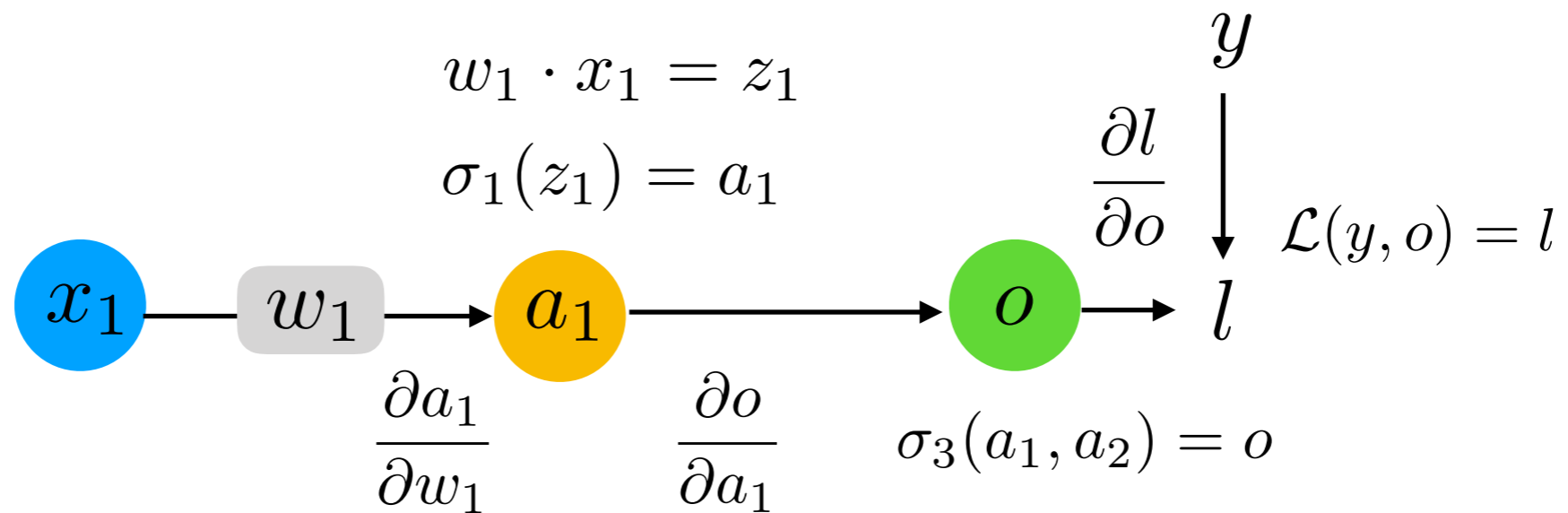
https://github.com/rasbt/stat479-deep-learning-ss19/blob/master/L06_pytorch/code/pytorch-autograd.ipynb

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

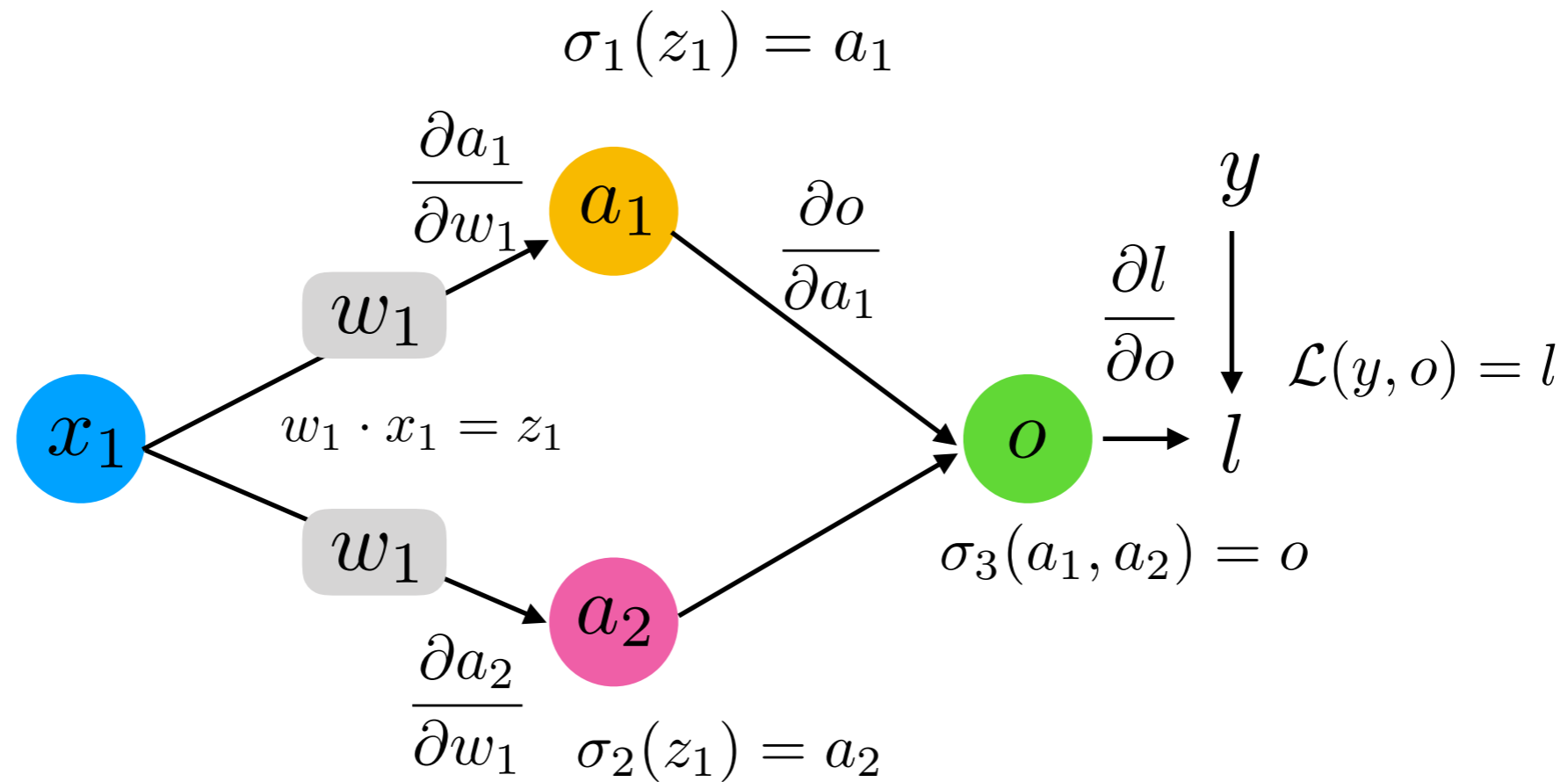
Some More Computation Graphs

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

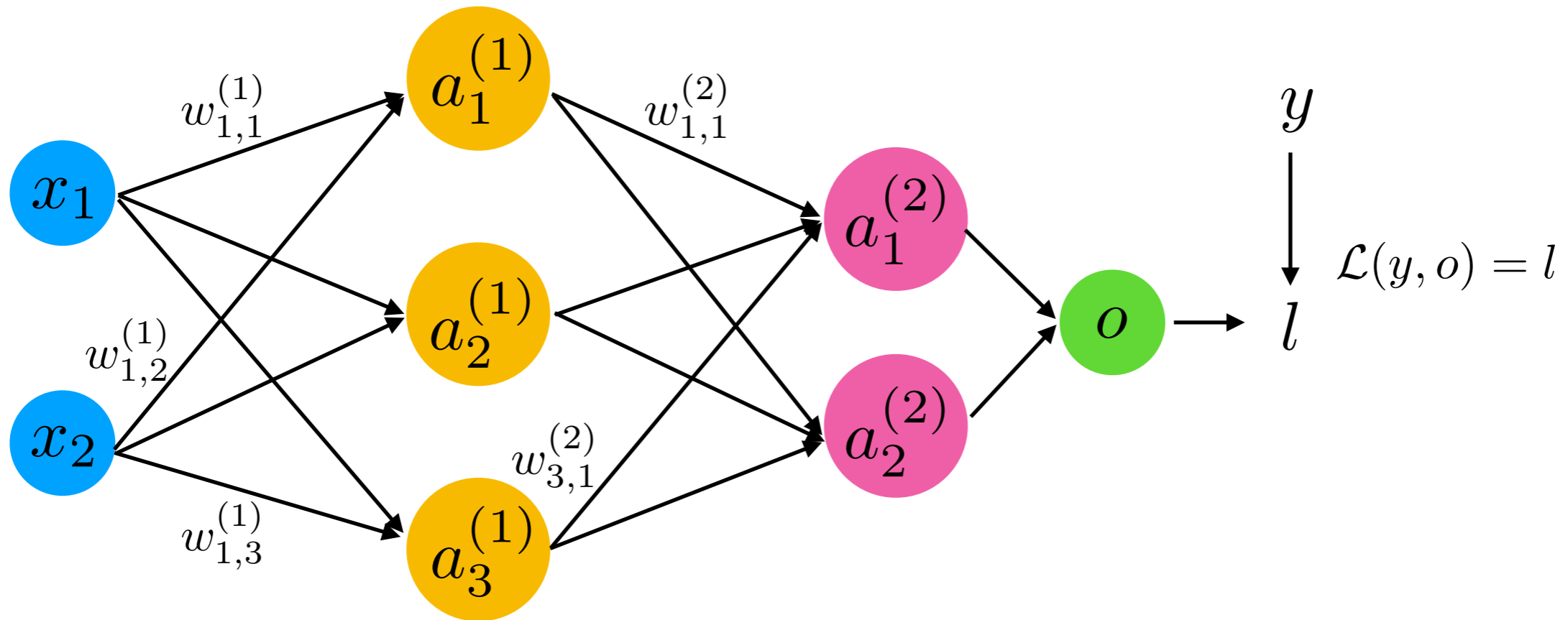


Upper path

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

Lower path

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



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

PyTorch Usage: Step 1 (Definition)

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

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

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

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(logits, 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(logits, 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 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(logits, targets)
        optimizer.zero_grad()

        loss.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

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



logits because of computational efficiency.
Basically, it internally uses a `log_softmax(logits)` function that is more stable than `log(softmax(logits))`.
More on logits ("net inputs" of the last layer) in the next lecture. Please also see <https://github.com/rasbt/stat479-deep-learning-ss19/blob/master/other/pytorch-lossfunc-cheatsheet.md>

PyTorch ADALINE (neuron model) Example

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)

Objected-Oriented vs Functional API

Using "Sequential"

1)

```
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),  
            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)

2)

However, if you use Sequential, you can define "hooks" to get intermediate outputs.
For example:

```
[7]: model.net  
[7]: Sequential(  
  (0): Linear(in_features=784, out_features=128, bias=True)  
  (1): ReLU(inplace)  
  (2): Linear(in_features=128, out_features=256, bias=True)  
  (3): ReLU(inplace)  
  (4): Linear(in_features=256, out_features=10, bias=True)  
)  
[ ]: If we want to get the output from the 2nd layer during the forward pass, we can register a hook as follows:  
[8]: outputs = []  
      def hook(module, input, output):  
          outputs.append(output)  
      model.net[2].register_forward_hook(hook)  
[8]: <torch.utils.hooks.RemovableHandle at 0x7f659c6685c0>  
      Now, if we call the model on some inputs, it will save the intermediate results in the "outputs" list:  
[9]: _ = model(features)  
      print(outputs)  
      [tensor([[0.5341, 1.0513, 2.3542, ..., 0.0000, 0.0000, 0.0000],  
              [0.0000, 0.6676, 0.6620, ..., 0.0000, 0.0000, 2.4056],  
              [1.1520, 0.0000, 0.0000, ..., 2.5860, 0.8992, 0.9642],  
              ...,  
              [0.0000, 0.1076, 0.0000, ..., 1.8367, 0.0000, 2.5203],  
              [0.5415, 0.0000, 0.0000, ..., 2.7968, 0.8244, 1.6335],  
              [1.0710, 0.9805, 3.0103, ..., 0.0000, 0.0000, 0.0000]],  
             device='cuda:3', grad_fn=<ThresholdBackward1>)]
```

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

- Autograd: Automatic Differentiation

https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html#sphx-glr-beginner-blitz-autograd-tutorial-py