Lecture 06

Automatic Differentiation with PyTorch

STAT 453: Deep Learning, Spring 2020

Sebastian Raschka

http://stat.wisc.edu/~sraschka/teaching/stat453-ss2020/

Today

Computing partial derivatives more easily (and automatically) with PyTorch

Overview

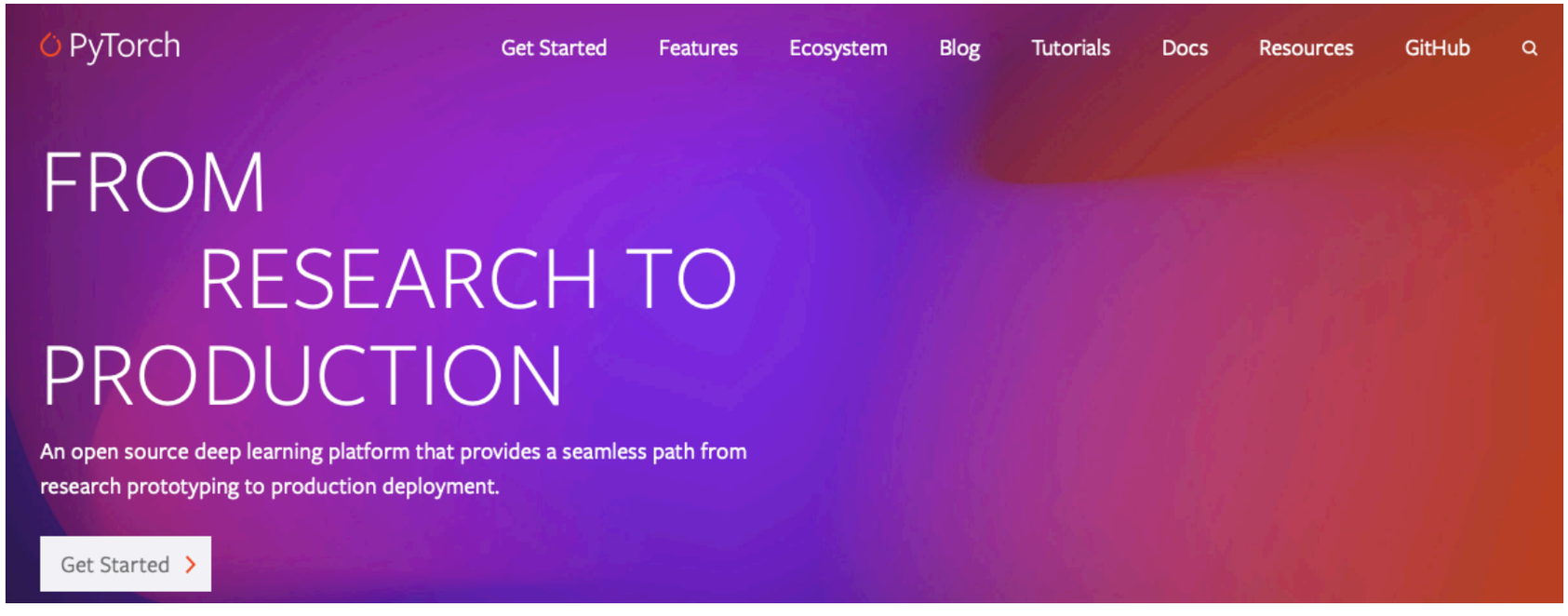
1/4 -- PyTorch Resources
2/4 -- Computation Graphs
3/4 -- Automatic Differentiation in PyTorch
4/4 -- PyTorch API

1/4 -- PyTorch Resources

2/4 -- Computation Graphs

3/4 -- Automatic Differentiation in PyTorch

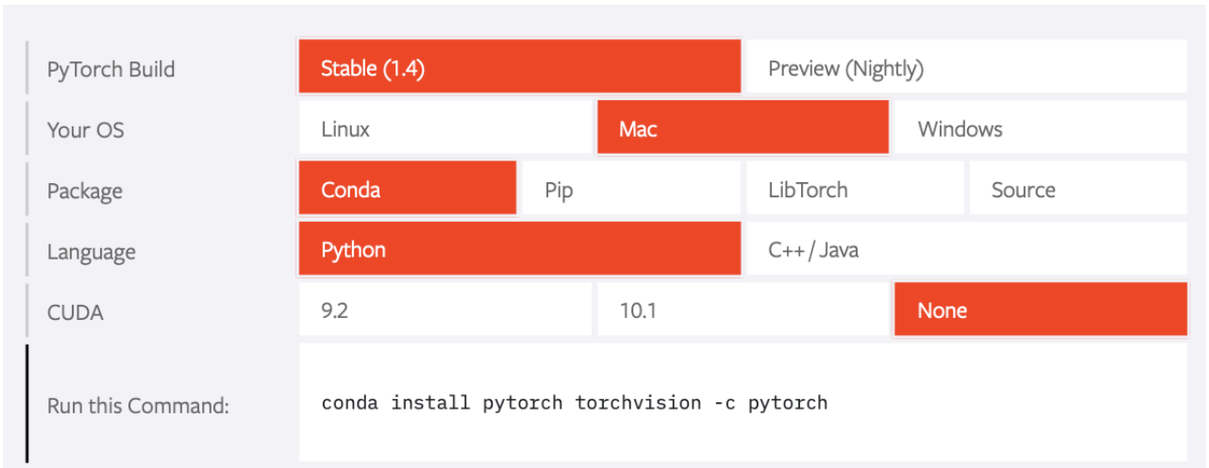
4/4 -- PyTorch API



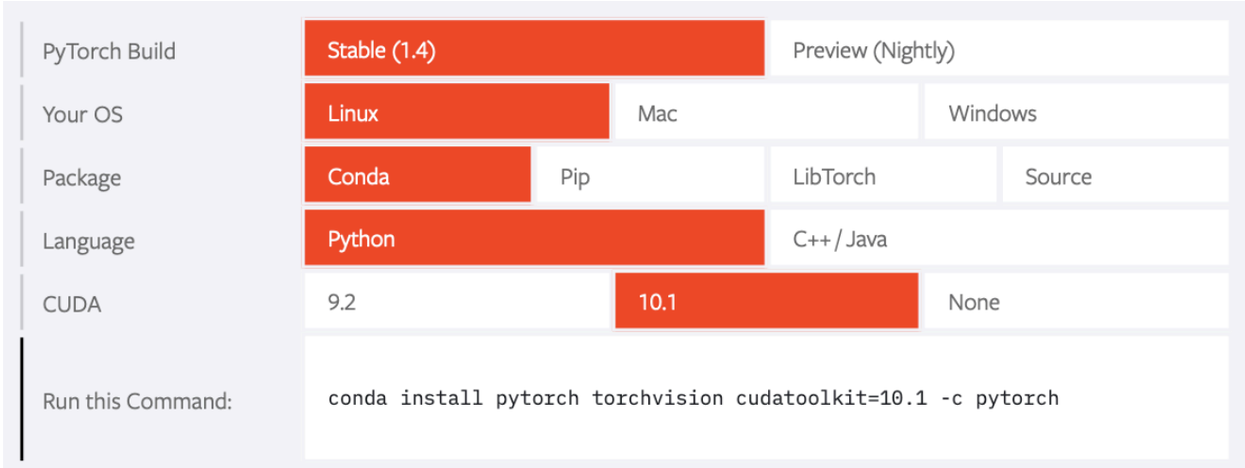
https://pytorch.org/

Installation

Recommendation for Laptop (e.g., MacBook)



Recommendation for Desktop (Linux) with GPU



https://pytorch.org/

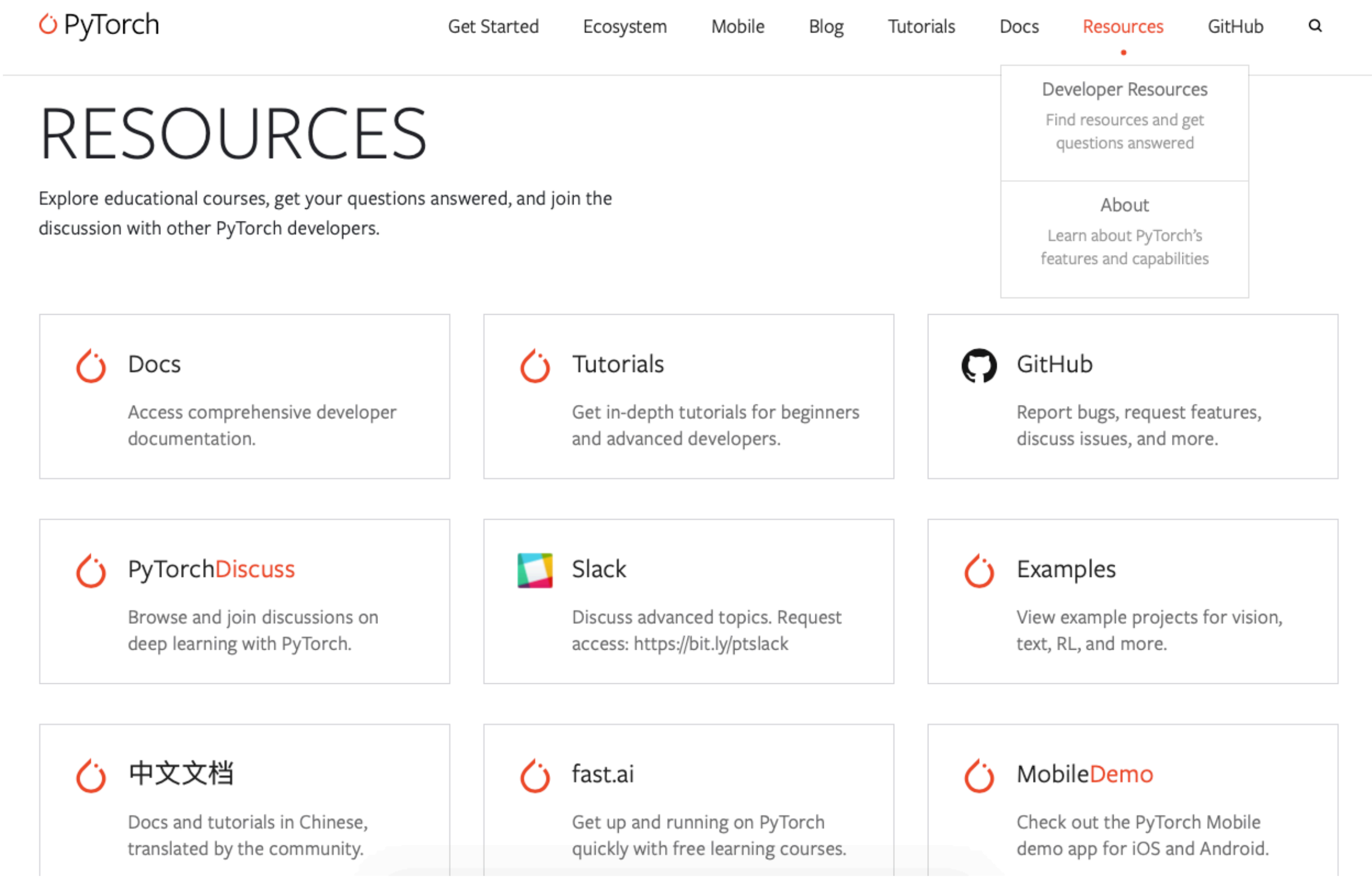
Installation Tips:

https://github.com/rasbt/stat453-deep-learning-ss20/tree/master/extras/installing-pytorch

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

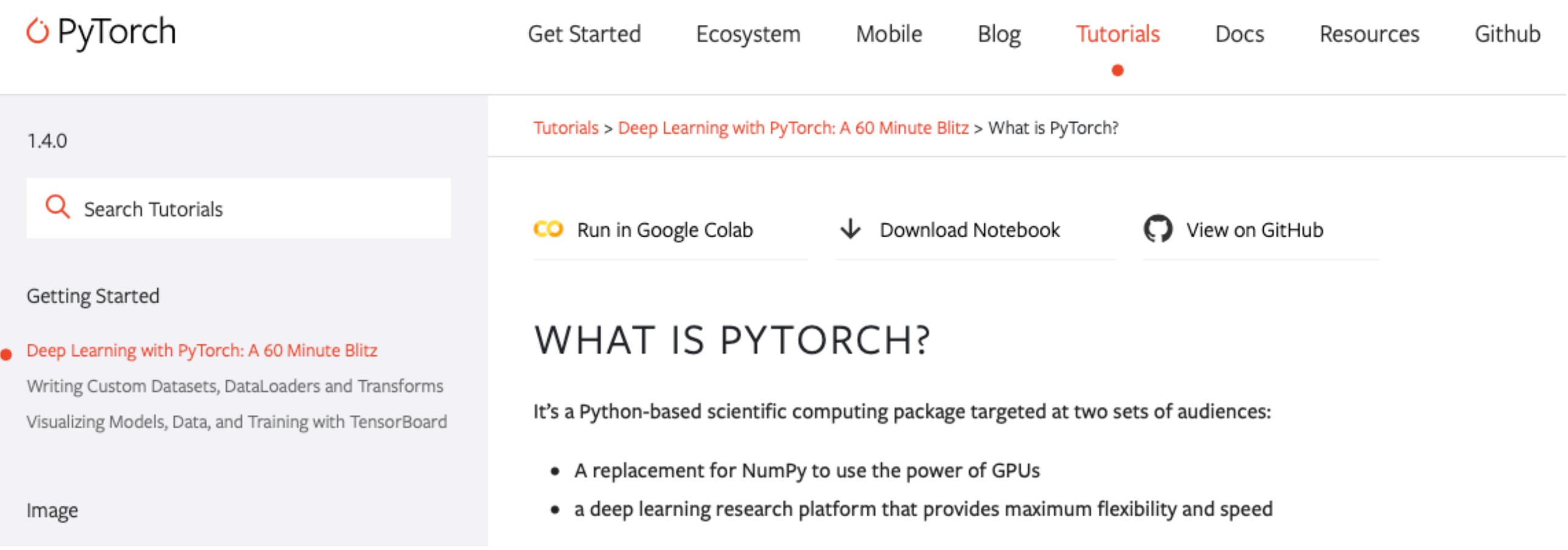
```
In [1]: import torch

In [2]: torch.__version__
Out[2]: '1.4.0'
```

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



https://pytorch.org/resources

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



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

Tutorials > Deep Learning with PyTorch: A 60 Minute Blitz

DEEP LEARNING WITH PYTORCH: A 60 MINUTE BLITZ

Author: Soumith Chintala

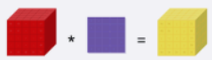
Goal of this tutorial:

- 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

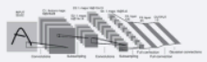
NOTE

Make sure you have the torch and torchvision packages installed.



What is PyTorch?

Autograd: Automatic Differentiation



Neural Networks

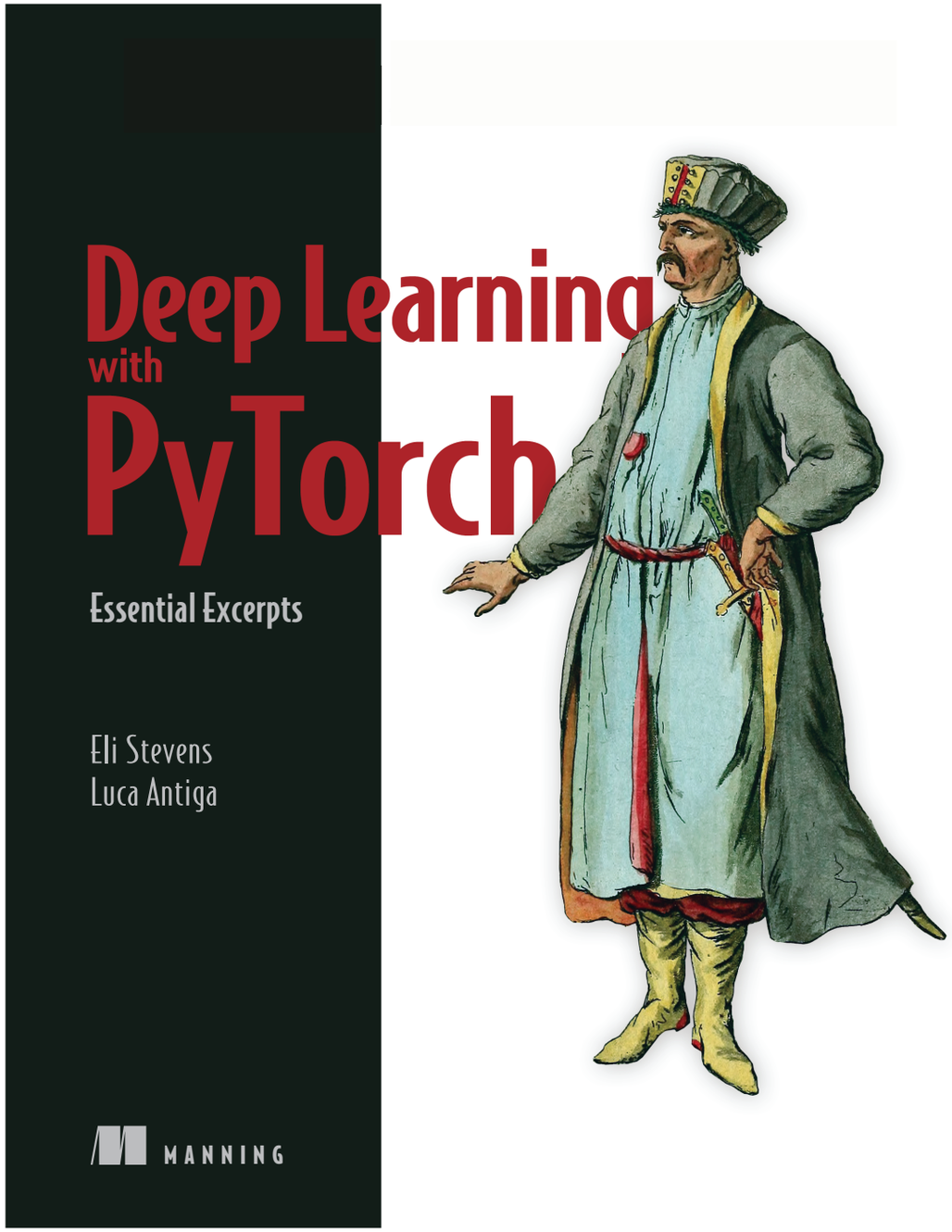
DEEP LEARNING WITH PYTORCH: A 60 MINUTE BLITZ

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 $v = \begin{pmatrix} v_1 & v_2 & \cdots & v_m \end{pmatrix}^T$, compute the product $v^T \cdot J$. If v happens to be the gradient of a scalar function $l = g\left(\vec{y}\right)$, that is, $v = \begin{pmatrix} \frac{\partial l}{\partial y_1} & \cdots & \frac{\partial l}{\partial y_m} \end{pmatrix}^T$, then by the chain rule, the vector-Jacobian product would be the gradient of l with respect to $\vec{x}$:

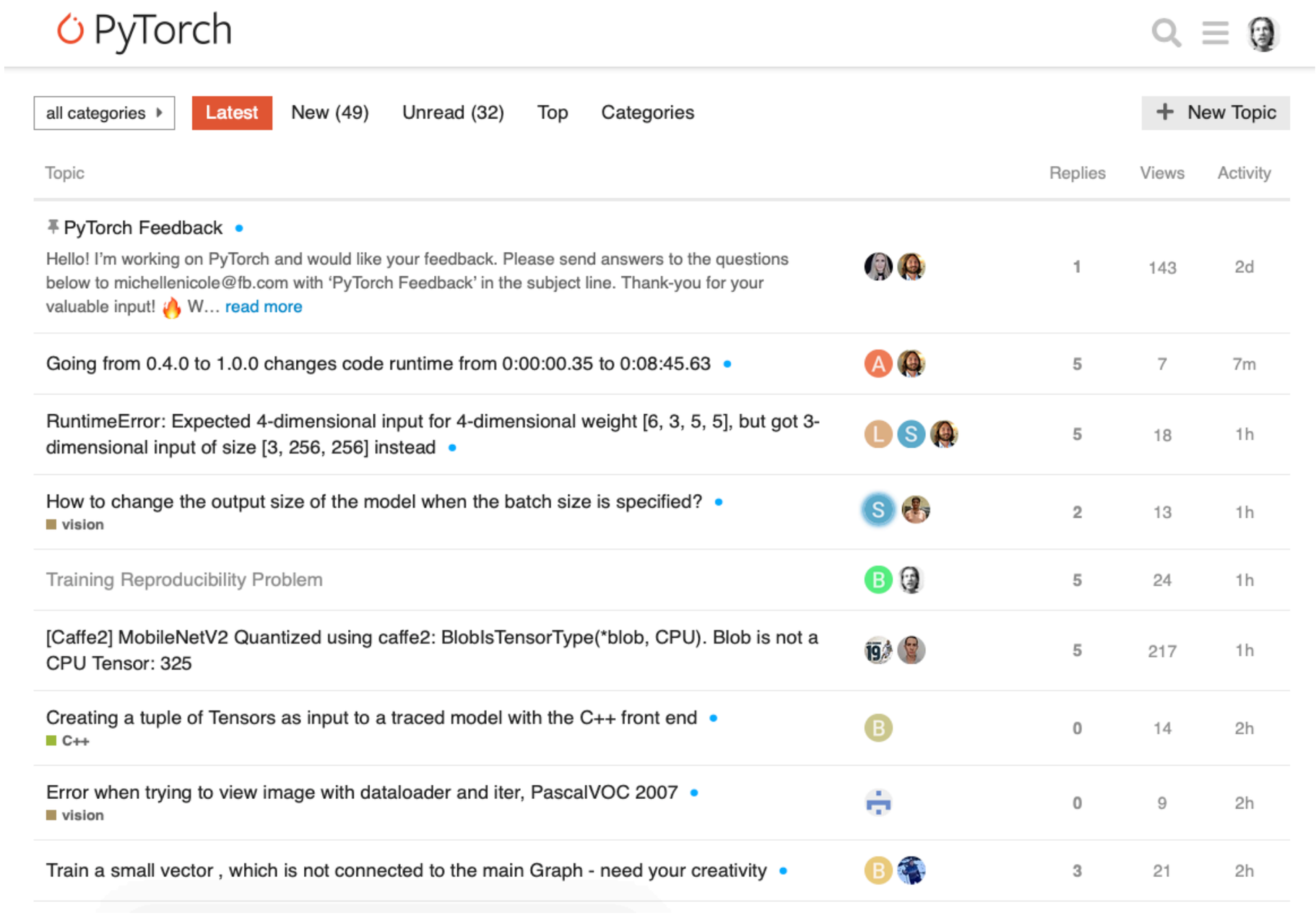
$$J^T \cdot v = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \begin{pmatrix} \frac{\partial l}{\partial y_1} \\ \vdots \\ \frac{\partial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

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



https://pytorch.org/assets/deep-learning/Deep-Learning-with-PyTorch.pdf

Very Active & Friendly Community and Help/Discussion Forum



https://pytorch.org/resources

My office hour this week

~~Thu 1:00 - 2:00 pm~~

Fri 2:30 - 3:30 pm

TA (Zhongjie Yu) office hour as usual:

Thu 2:30 - 3:30 pm (MSC 1130)

1/4 -- PyTorch Resources

2/4 -- Computation Graphs

3/4 -- Automatic Differentiation in PyTorch

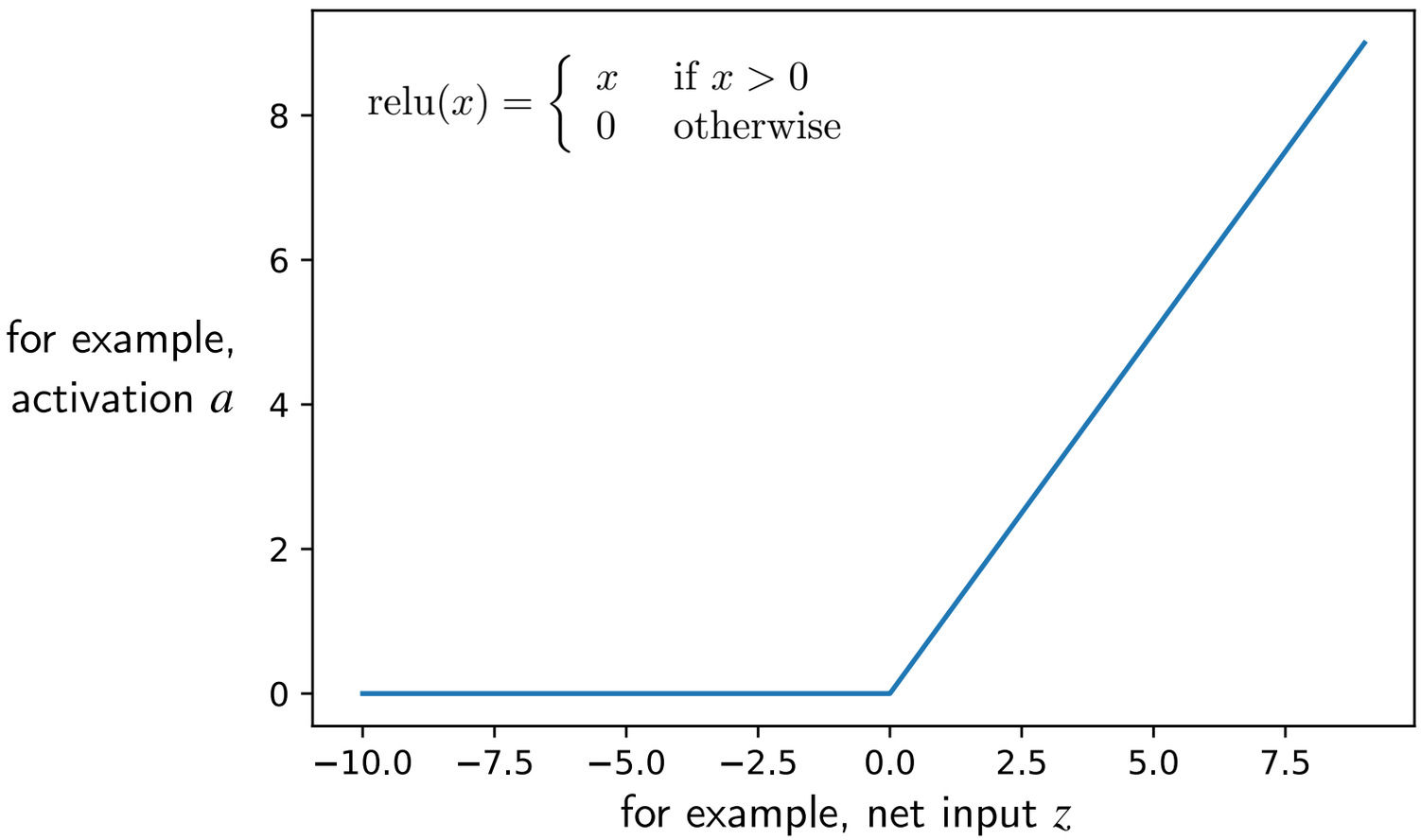
4/4 -- PyTorch API

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:

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



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 machine learning--computer science context, for convenience, we can just say

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

Why not differentiable?
Derivative does not exist (DNE) at 0, because the derivative is different if we approach the limit from the left or right:

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

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

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

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

Computation Graphs

Suppose we have the following activation function:

net input

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

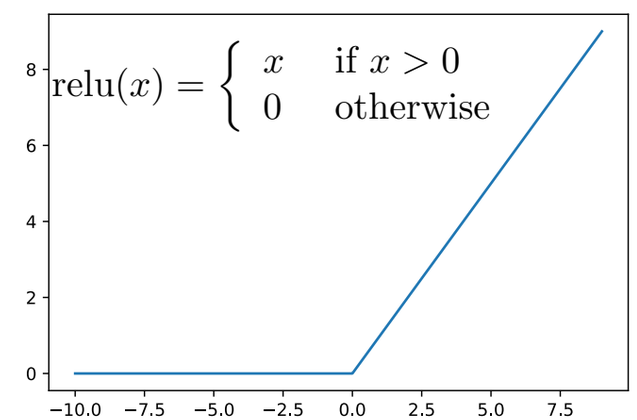
multivariable function

activation function

feature
(suppose only 1 training example)

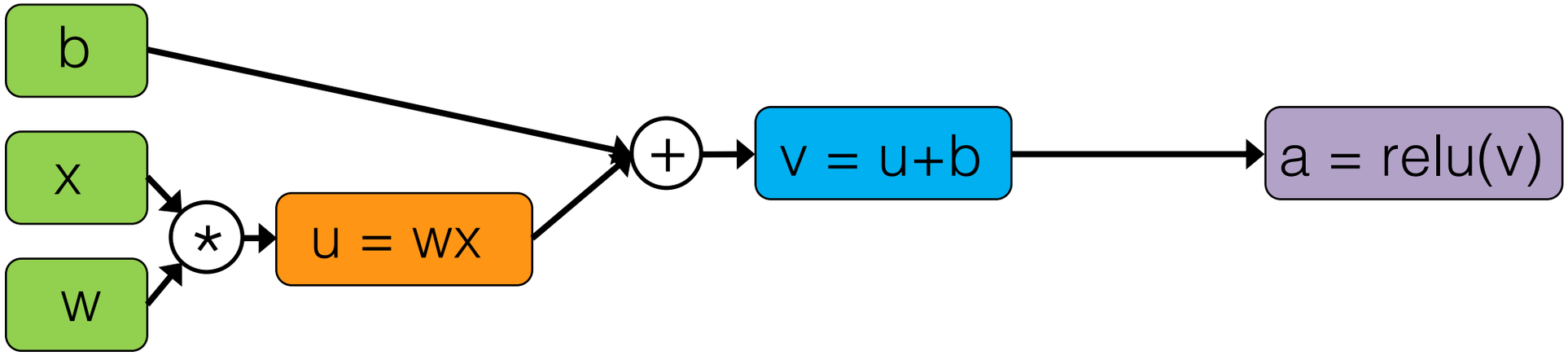
bias

weight parameter
(assume only 1 input feature)



Computation Graphs

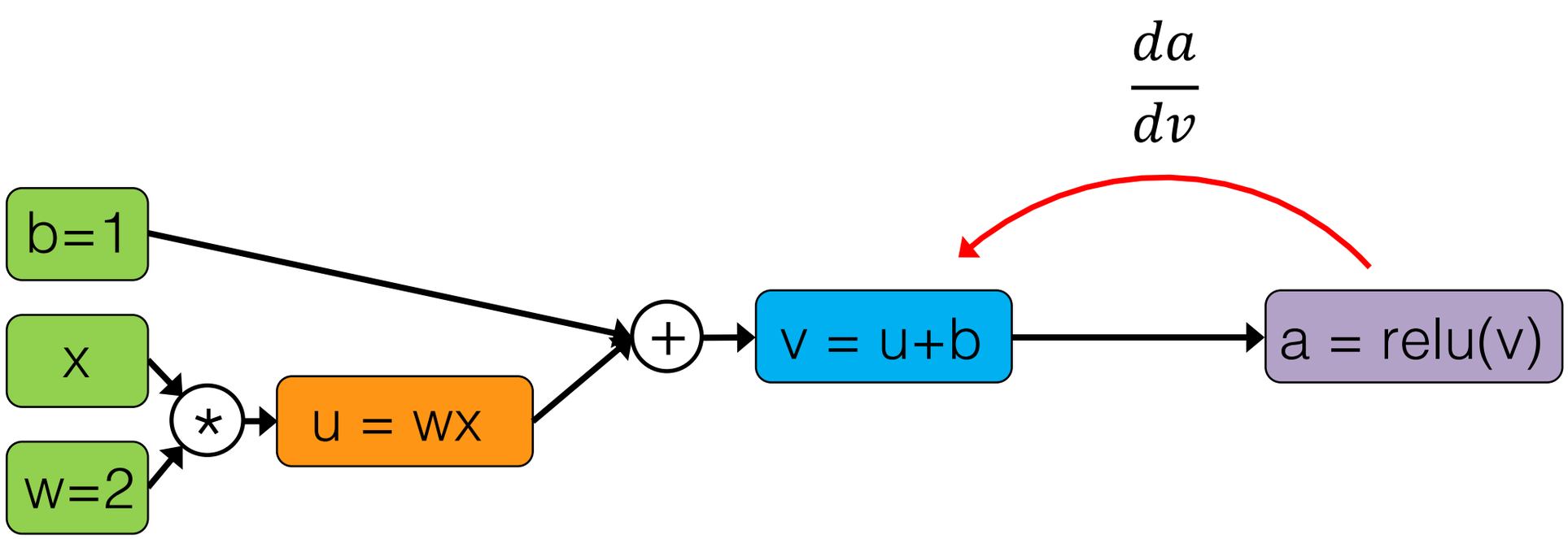
$$a(x,\ w,\ b) = relu(\underbrace{\underbrace{w \cdot x}_{} + b}_{v})$$



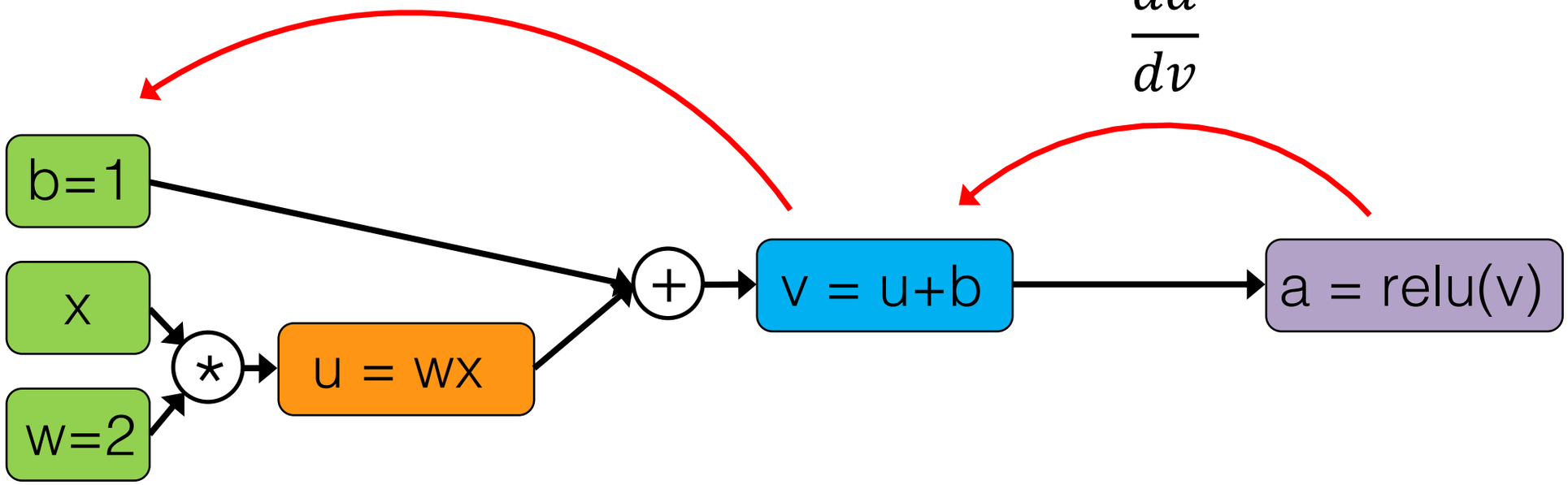
Computation Graphs

Usually, in deep learning (i.e., when we use gradient descent or backprop to optimize a loss function), we are interested in computing the derivative of the loss with respect to the parameters. When we do this, we apply the chain rule, and one component of it is computing the derivative of the activation with respect to the parameters, which we will be doing in the next couple of slides.

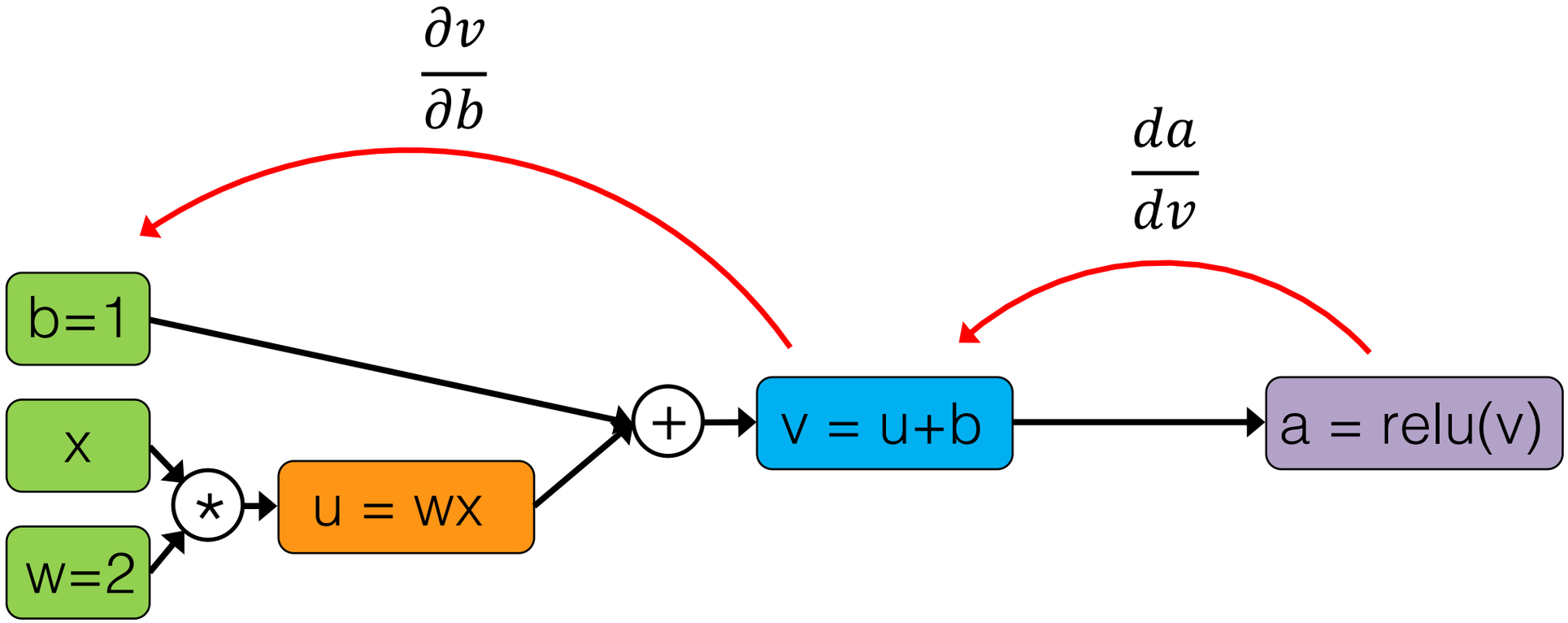
Assuming a "reverse mode" for computing the gradient of the activation with respect to the parameters w and b, we start as follows:



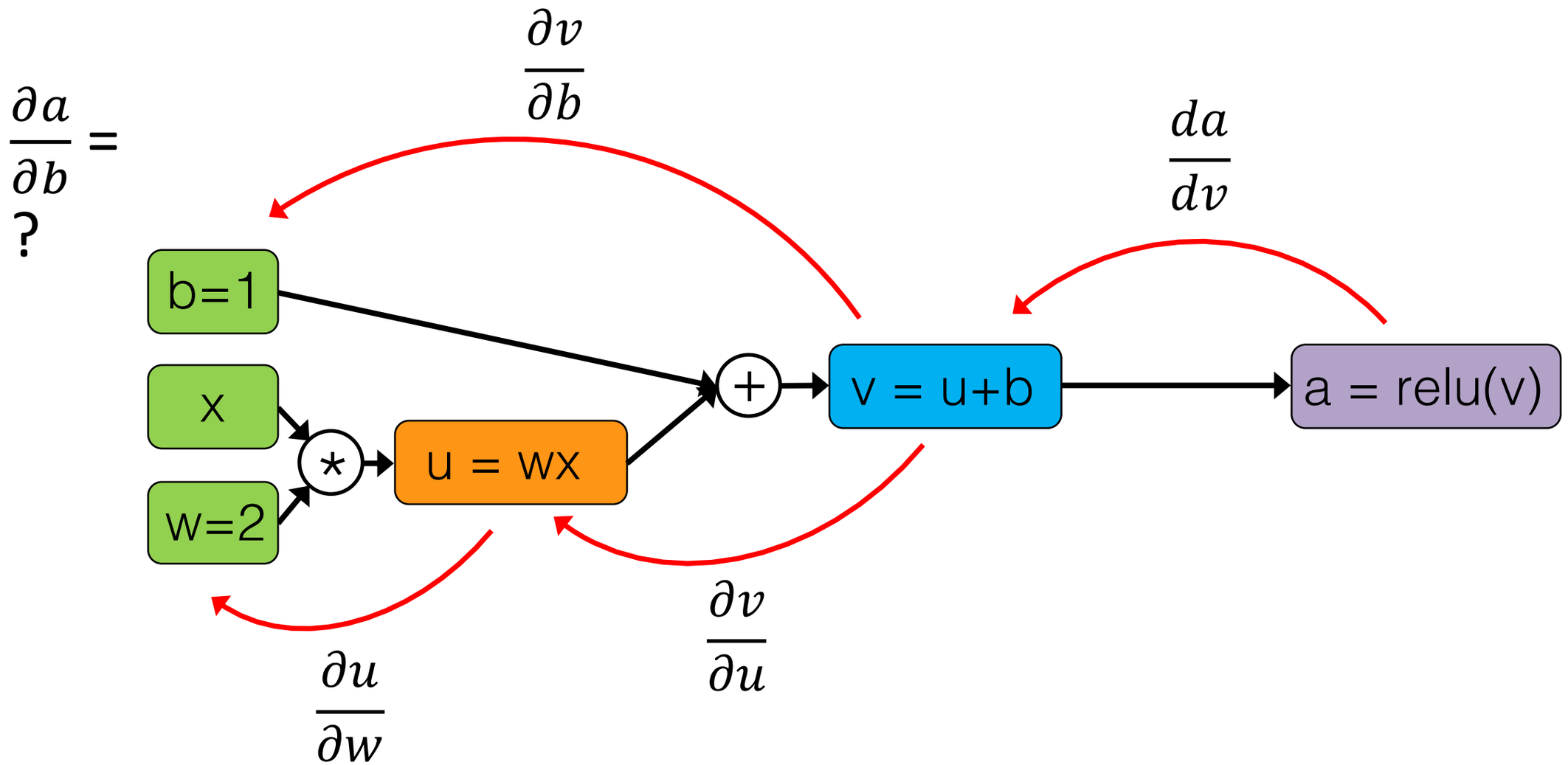
Computation Graphs



Computation Graphs

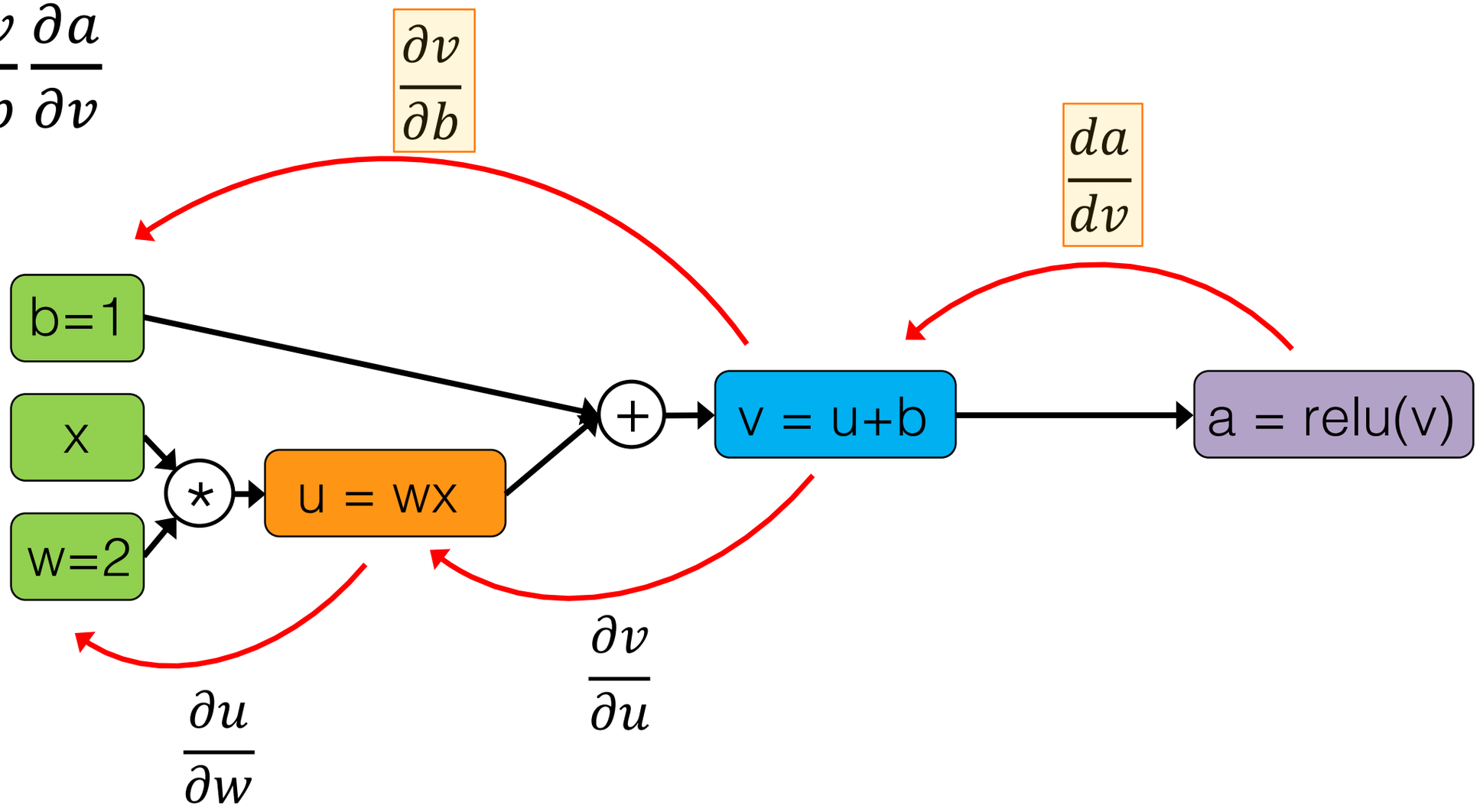


Computation Graphs



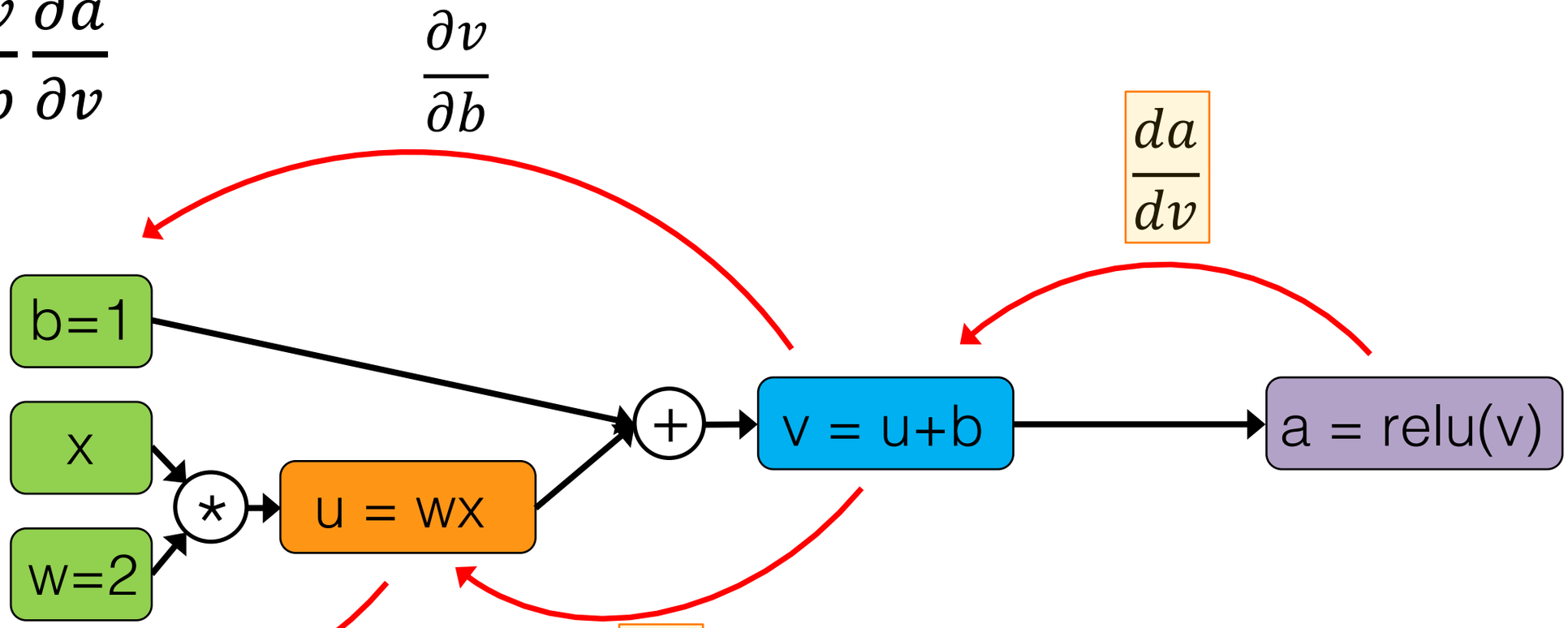
Computation Graphs

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



Computation Graphs

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

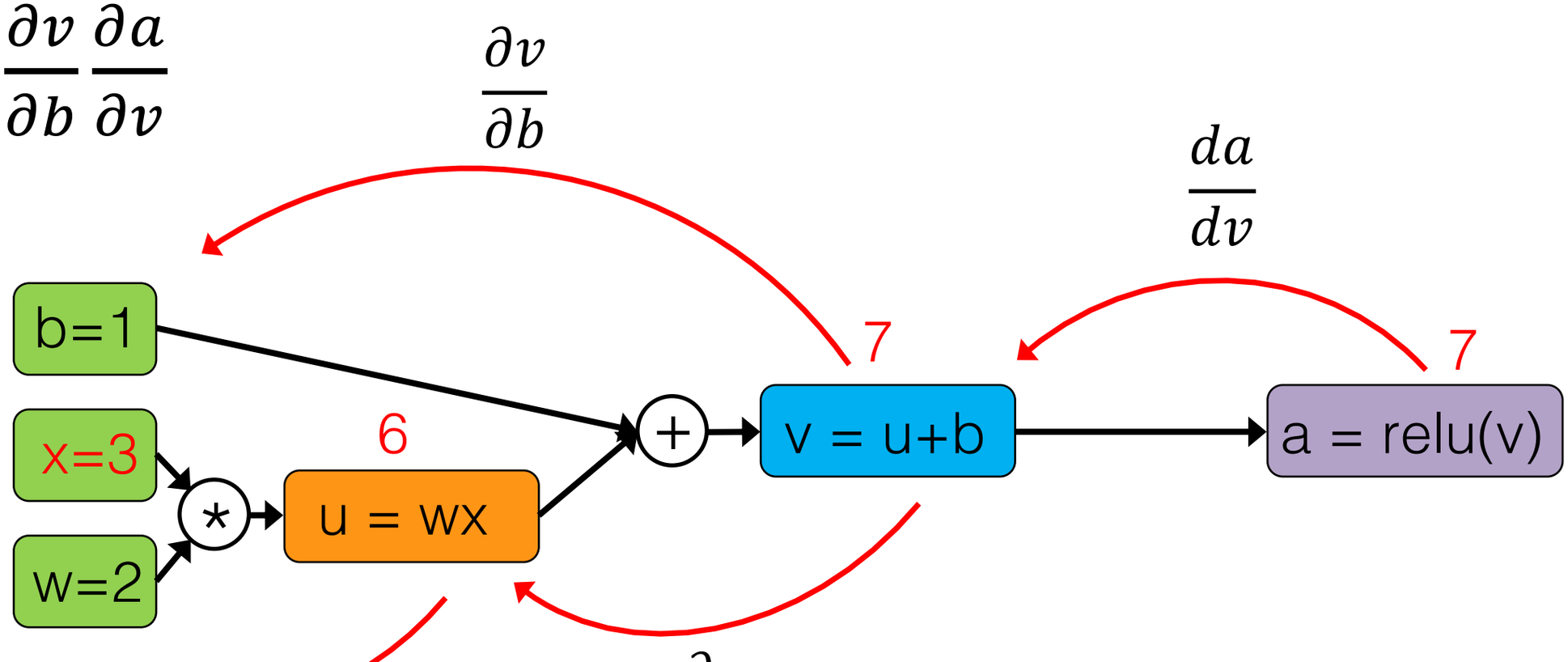


$$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w}\frac{\partial a}{\partial u}$$

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

Computation Graphs

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

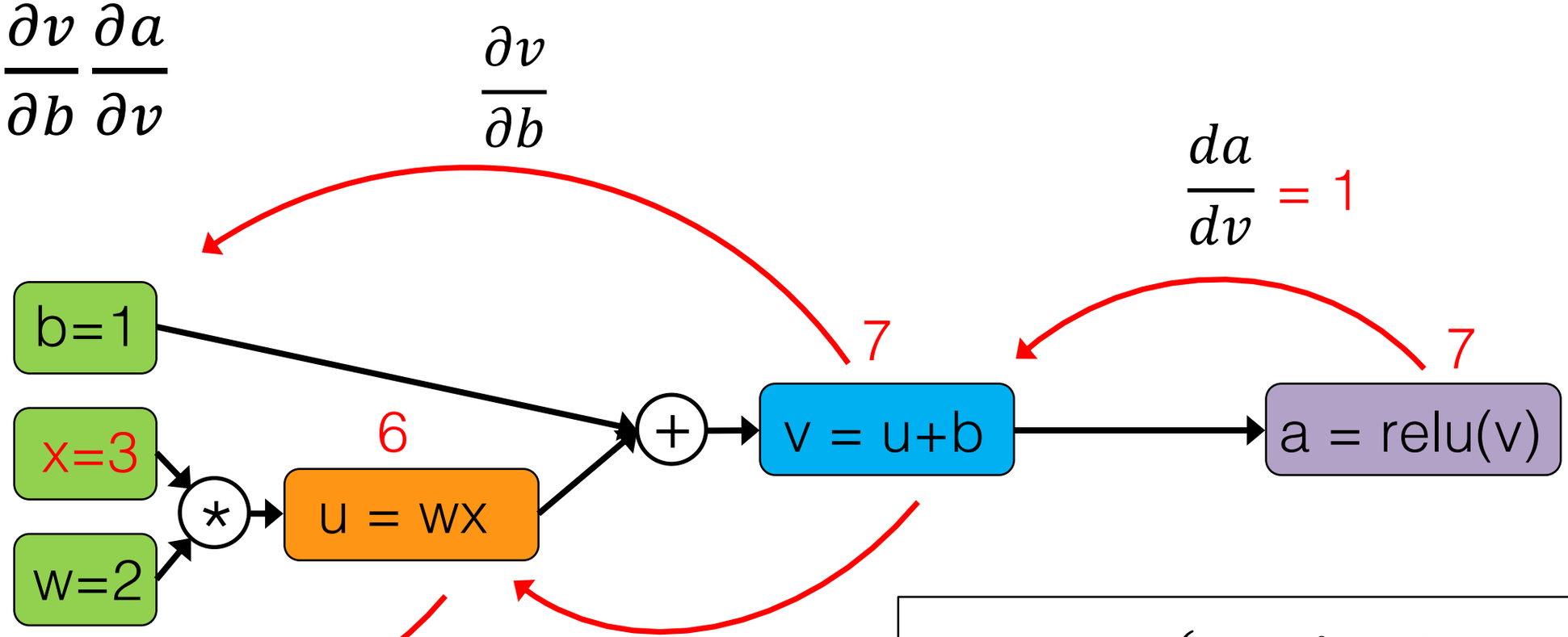


$$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w}\frac{\partial a}{\partial u}$$

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

Computation Graphs

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



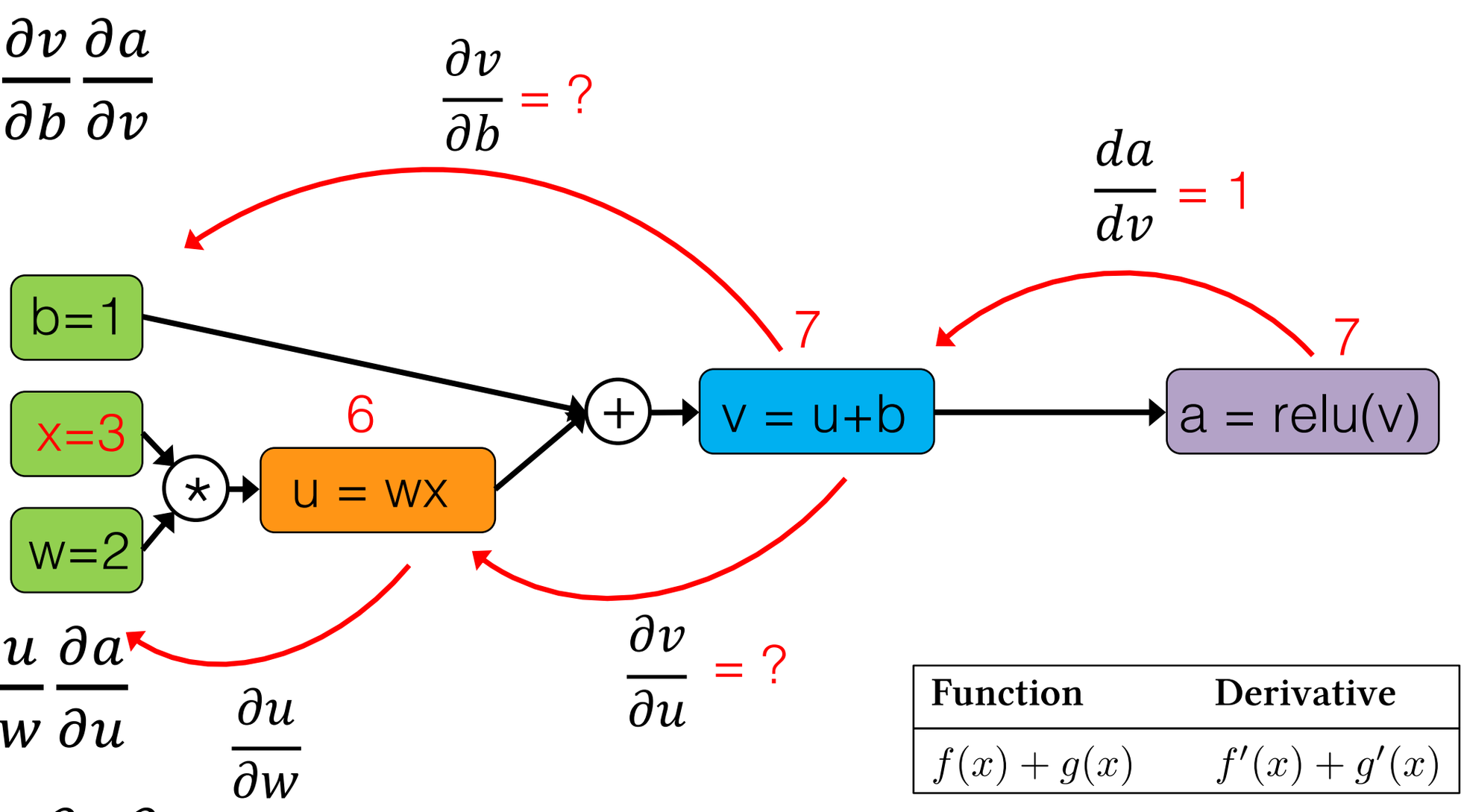
$$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w}\frac{\partial a}{\partial u} \qquad \frac{\partial u}{\partial w}$$

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

$$\text{relu}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Computation Graphs

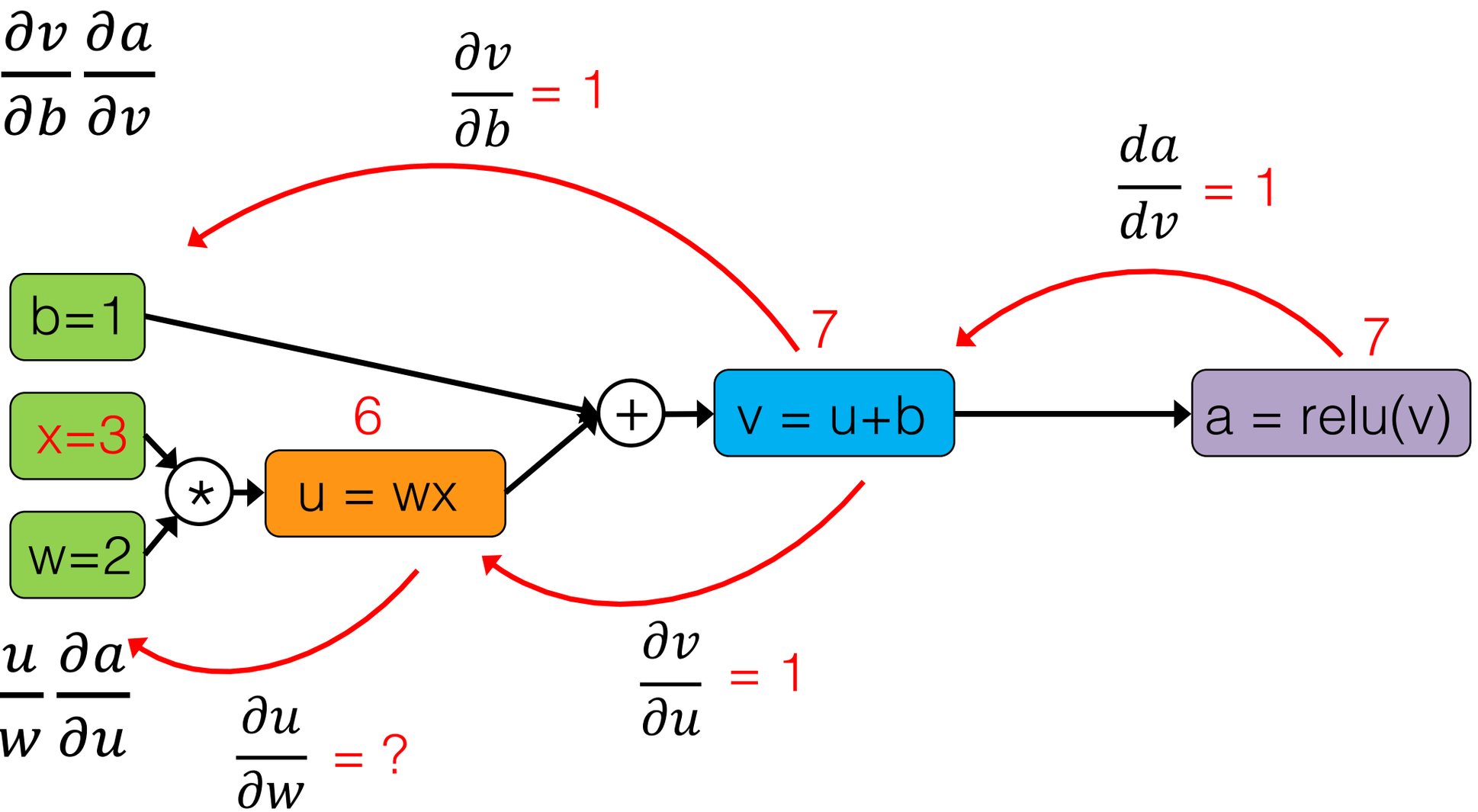
$$\frac{\partial a}{\partial b} = \frac{\partial v}{\partial b}\frac{\partial a}{\partial v}$$



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

Computation Graphs

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

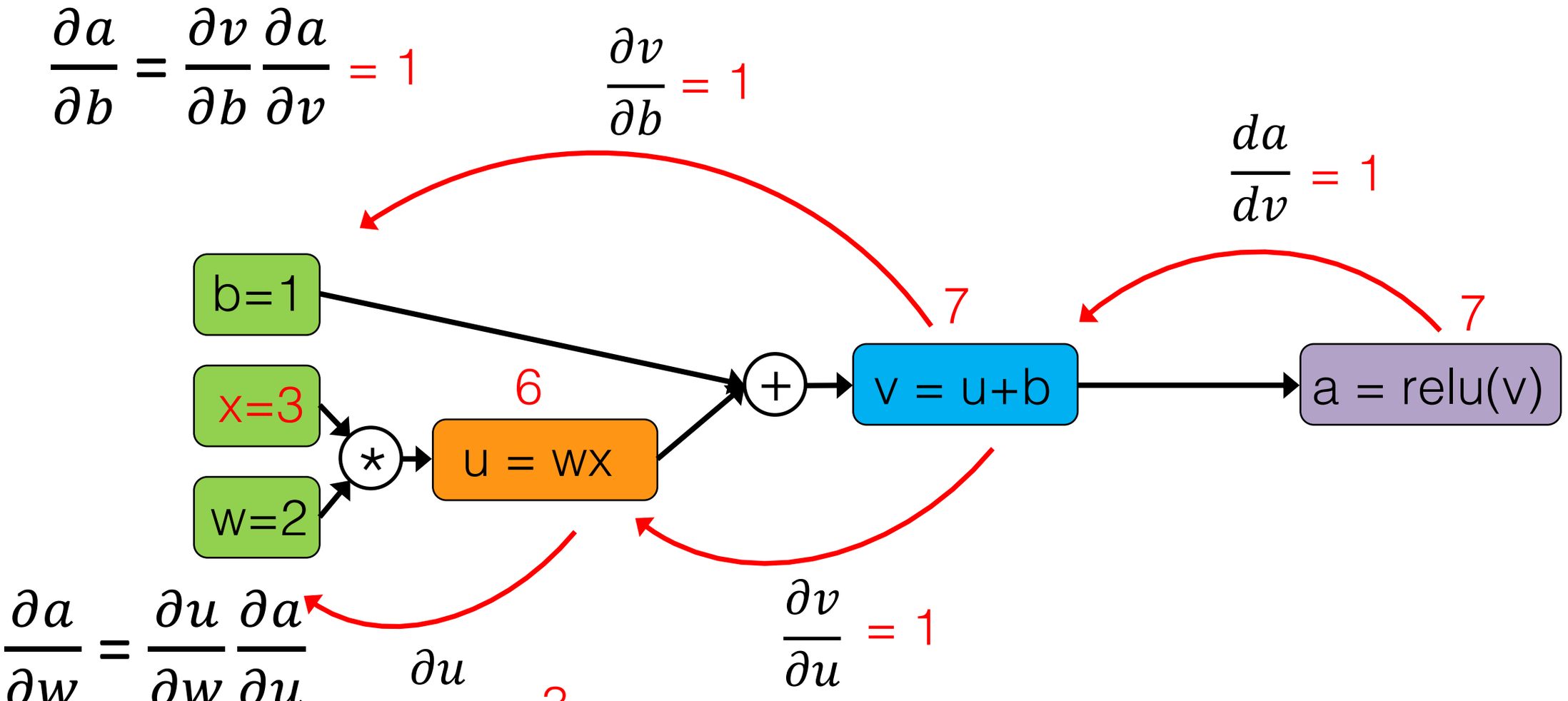


$$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w}\frac{\partial a}{\partial u}$$

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

Computation Graphs

$$\frac{\partial a}{\partial b} = \frac{\partial v}{\partial b}\frac{\partial a}{\partial v} = 1$$



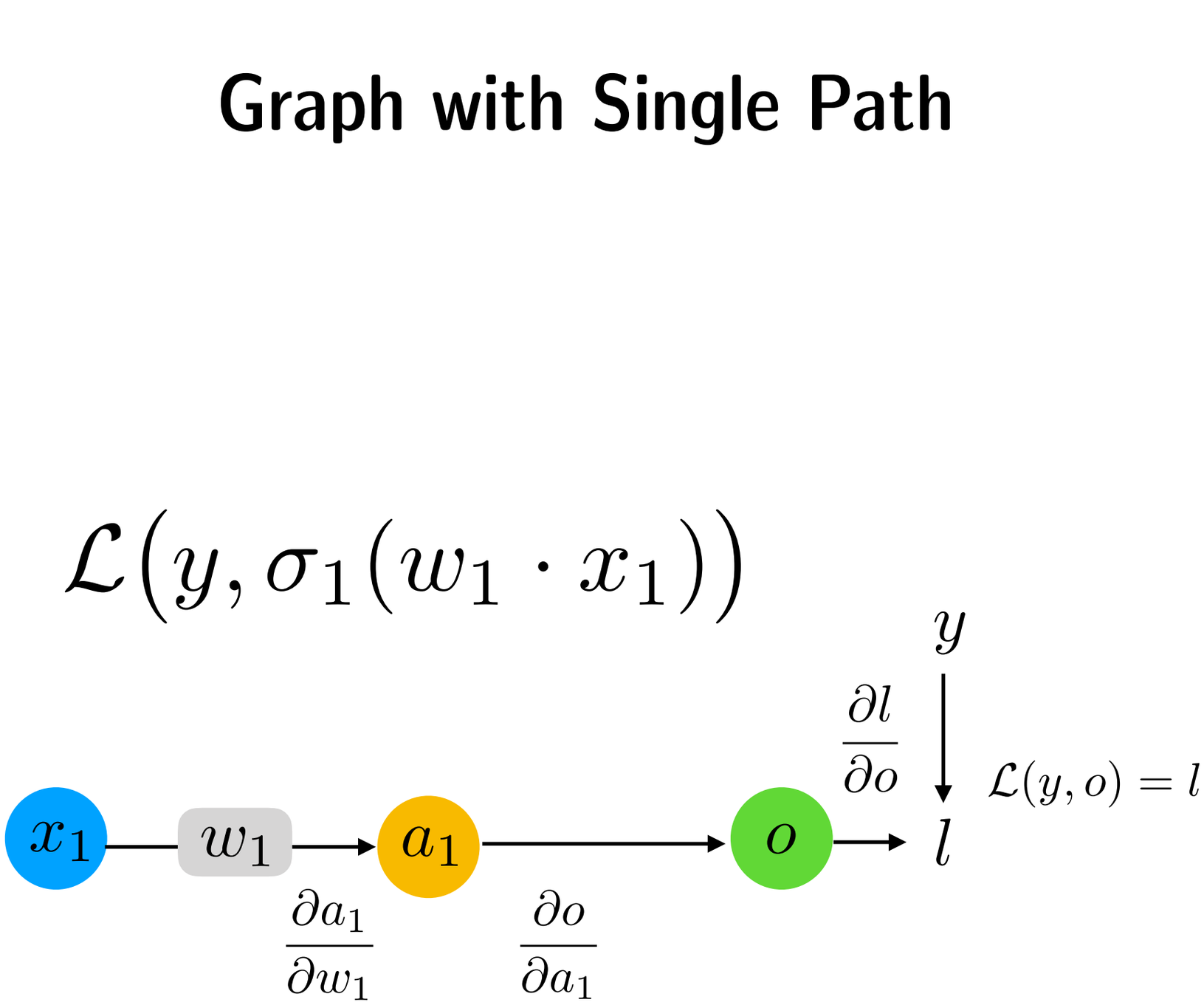
$$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w}\frac{\partial a}{\partial u}$$

$$\frac{\partial u}{\partial w} = 3$$

$$= \frac{\partial u}{\partial w}\frac{\partial v}{\partial u}\frac{\partial a}{\partial v} = 3*1*1 = 3$$

Some More Computation Graphs

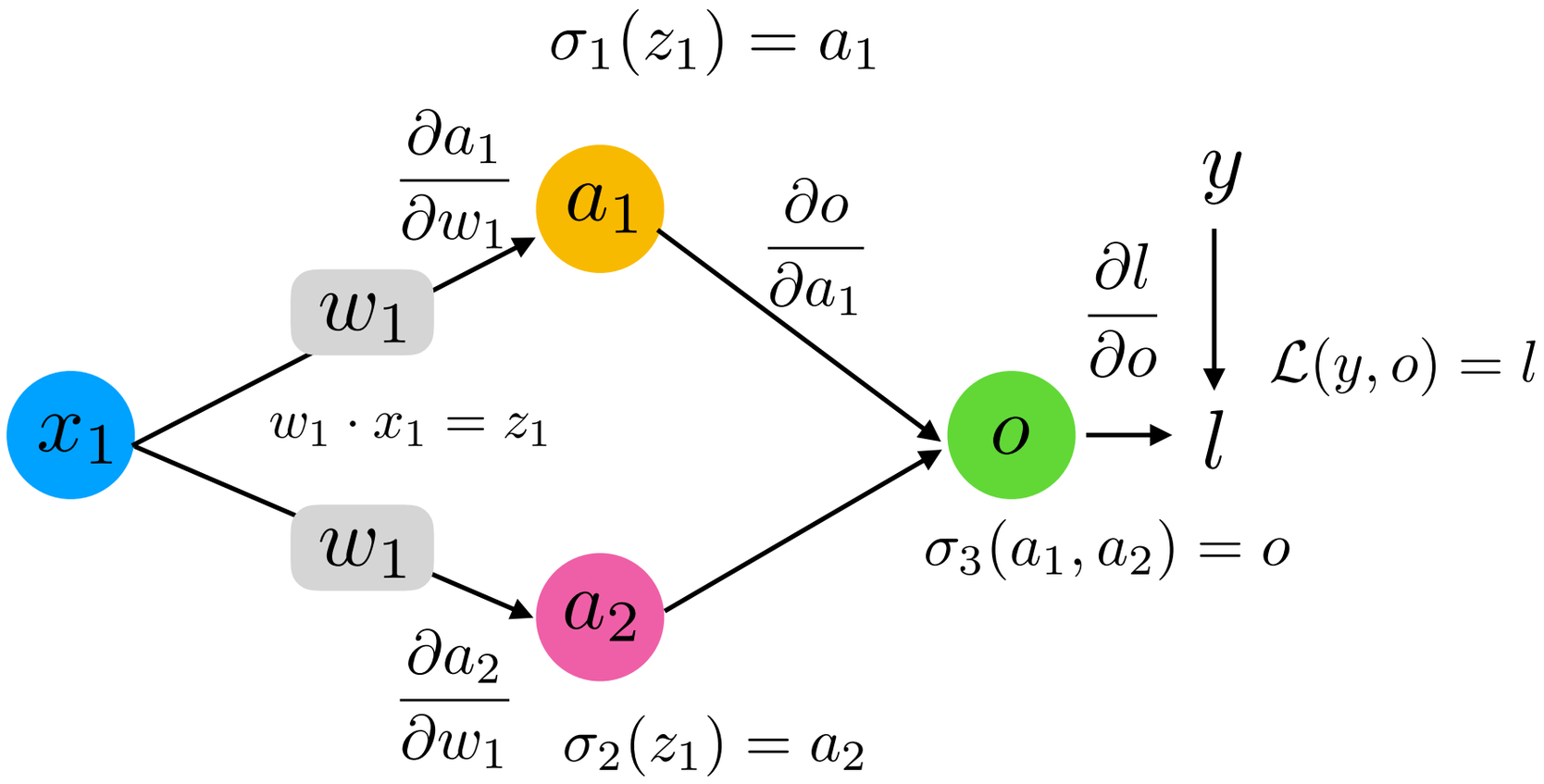
Graph with Single Path



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

Graph with Weight Sharing

$$\mathcal{L}\big(y, \sigma_3\big[\sigma_1(w_1 \cdot x_1), \sigma_2(w_1 \cdot x_1)\big]\big)$$

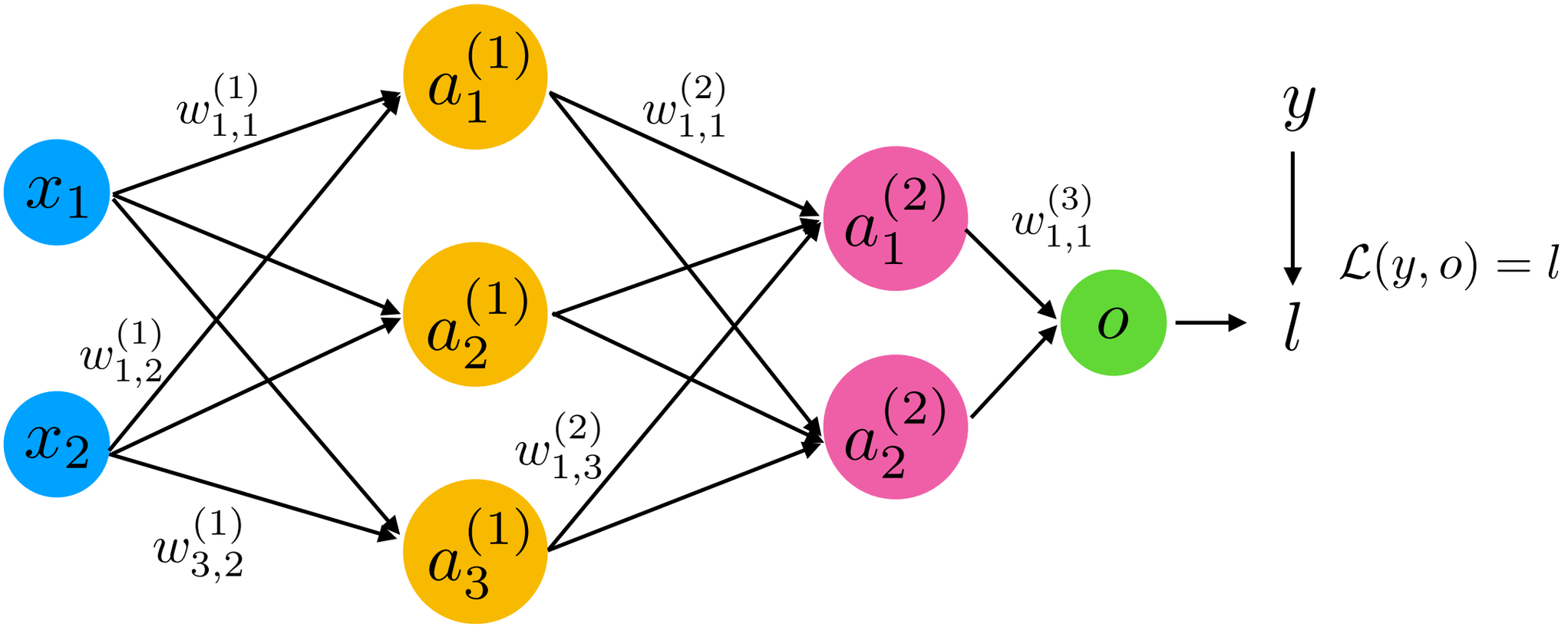


Upper path

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

Lower path

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



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

1/4 -- PyTorch Resources

2/4 -- Computation Graphs

3/4 -- Automatic Differentiation in PyTorch

4/4 -- PyTorch API

PyTorch Autograd Example

https://github.com/rasbt/stat453-deep-learning-ss20/tree/master/L06-pytorch/code/pytorch-autograd.ipynb

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

https://github.com/rasbt/stat453-deep-learning-ss20/tree/master/L06-pytorch/code/grad-intermediate-var.ipynb

1/4 -- PyTorch Resources

2/4 -- Computation Graphs

3/4 -- Automatic Differentiation in PyTorch

4/4 -- PyTorch API

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 it 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)

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

```
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
```

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
```

This will run the forward() method

Define a loss function to optimize

Set the gradient to zero
(could be non-zero from a previous forward pass)

Compute the gradients, the backward is automatically constructed by "autograd" based on the forward() method and the loss function

Use the gradients to update the weights according to the optimization method (defined on the previous slide)
E.g., for SGD, $w := w + \text{learning\_rate} \times \text{gradient}$

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/stat453-deep-learning-ss20/tree/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_
            torch.nn.ReLU(),
            torch.nn.Linear(num_hidden_2, num_classe
        )

    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

- PyTorch Book: https://pytorch.org/assets/deep-learning/Deep-Learning-with-PyTorch.pdf
  Chapter 1: Introducing deep learning and the PyTorch library
  Chapter 2: It starts with a tensor

HW2