STAT 453: Introduction to Deep Learning and Generative Models

Sebastian Raschka

http://stat.wisc.edu/~sraschka/teaching

Lecture 06

# Automatic Differentiation with PyTorch

# Today

Computing partial derivatives more easily
(and automatically) with PyTorch

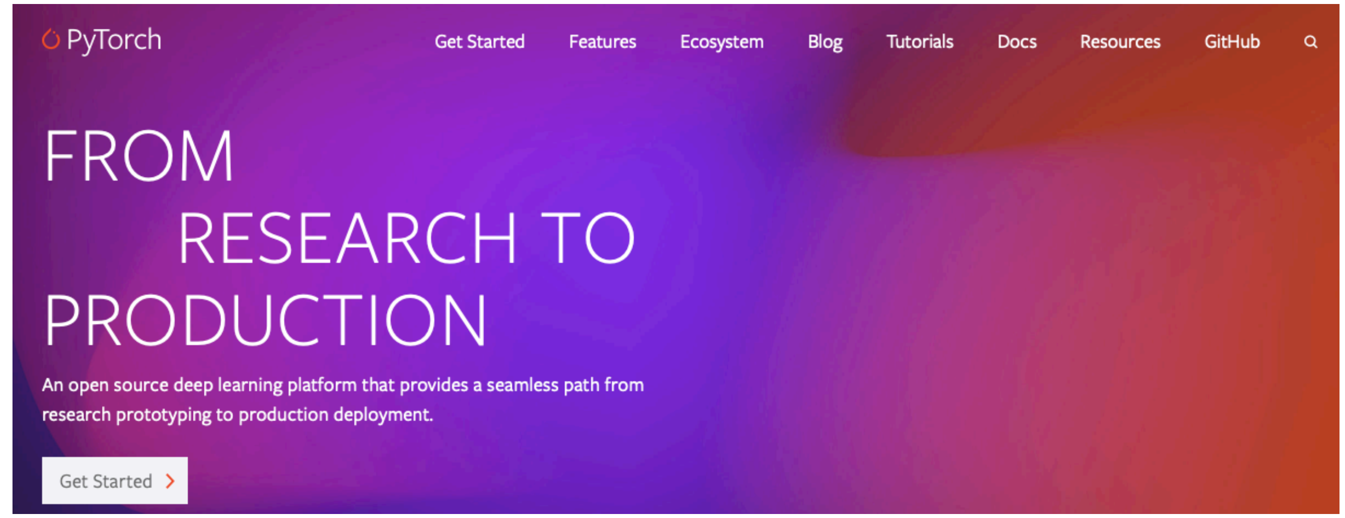
# Lecture Overview

1. PyTorch Resources
2. Computation Graphs
3. Automatic Differentiation in PyTorch
4. Training ADALINE Manually Vs Automatically in PyTorch
5. A Closer Look at the PyTorch API

# Learning More About PyTorch

1. **PyTorch Resources**
2. Computation Graphs
3. Automatic Differentiation in PyTorch
4. Training ADALINE Manually Vs Automatically in PyTorch
5. A Closer Look at the PyTorch API

PyTorch




https://pytorch.org/

**At a Glance:**

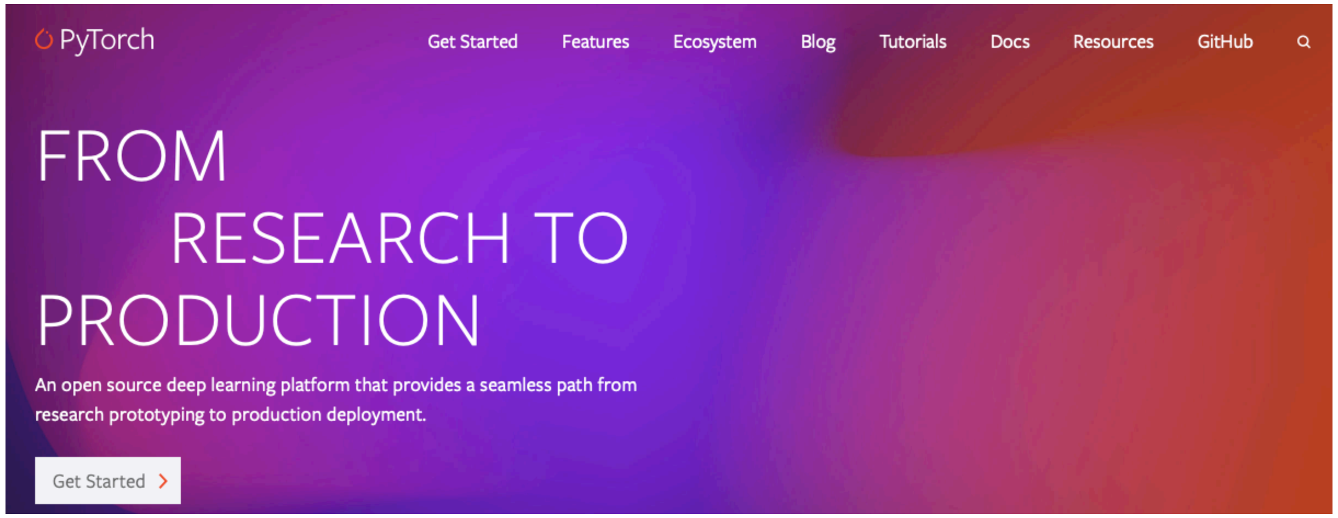
- Based on Torch 7, which was based on Lua and inspired by Lush
- PyTorch started in 2016
- Focuses on flexibility and minimizing cognitive overhead

- Dynamic nature of autograd API inspired by Chainer

- Core features
  - Automatic differentiation
  - Dynamic computation graphs
  - NumPy integration

- written in C++ and CUDA (CUDA is like C++ for the GPU)
- Python is the usability glue

# PyTorch




https://pytorch.org/

**At a Glance:**

- Based on Torch 7, which was based on Lua and inspired by Lush
- PyTorch started in 2016
- Focuses on flexibility and minimizing cognitive overhead

- Dynamic nature of autograd API inspired by Chainer

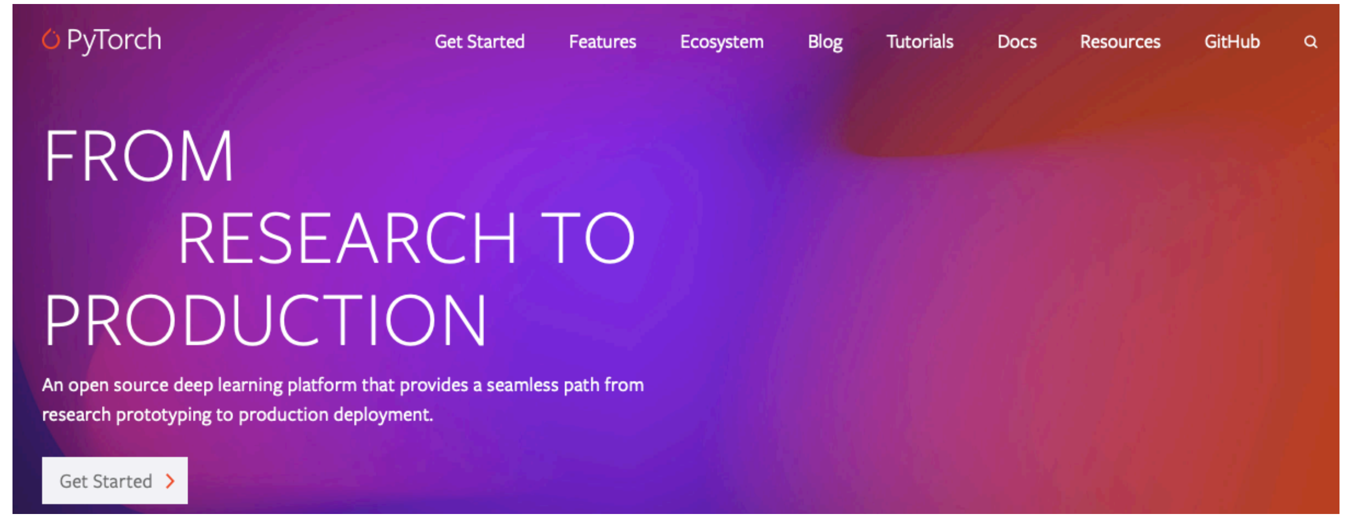
- Core features
  - Automatic differentiation
  - Dynamic computation graphs
  - NumPy integration

- written in C++ and CUDA (CUDA is
- Python is the usability glue

**PyTorch vs NumPy**

- Support GPU
- distribute ops across multiple devices
- keep track of computation graph and ops that created them

# PyTorch




[https://pytorch.org/](https://pytorch.org/)

**At a Glance:**

- Based on Torch 7, which was based on Lua and inspired by Lush
- PyTorch started in 2016
- Focuses on flexibility and minimizing cognitive overhead
- Dynamic nature of autograd API inspired by Chainer
- Core features
    - Automatic differentiation

**"the speedup gained by taking Python out of the computation is 10% or less"**
**-- Stevens *et al.*: Deep Learning with PyTorch**

- written in C++ and CUDA (CUDA is like C++ for the GPU)
- Python is the usability glue

# Installation

Recommendation for Laptop (e.g., MacBook)

| | | | | | |
|---|---|---|---|---|---|
| PyTorch Build | **Stable (1.7.1)** | Preview (Nightly) | | | |
| Your OS | Linux | **Mac** | Windows | | |
| Package | **Conda** | Pip | LibTorch | Source | |
| Language | **Python** | C++ / Java | | | |
| CUDA | 9.2 | 10.1 | 10.2 | 11.0 | **None** |
| Run this Command: | **NOTE:** Python 3.9 users will need to add '-c=conda-forge' for installation<br>conda install pytorch torchvision torchaudio -c pytorch | | | | |

Recommendation for Desktop (Linux) with GPU

| | | | | | |
|---|---|---|---|---|---|
| PyTorch Build | **Stable (1.7.1)** | Preview (Nightly) | | | |
| Your OS | **Linux** | Mac | Windows | | |
| Package | **Conda** | Pip | LibTorch | Source | |
| Language | **Python** | C++ / Java | | | |
| CUDA | 9.2 | 10.1 | 10.2 | **11.0** | None |
| Run this Command: | **NOTE:** Python 3.9 users will need to add '-c=conda-forge' for installation<br>conda install pytorch torchvision torchaudio cudatoolkit=11.0 -c pytorch | | | | |

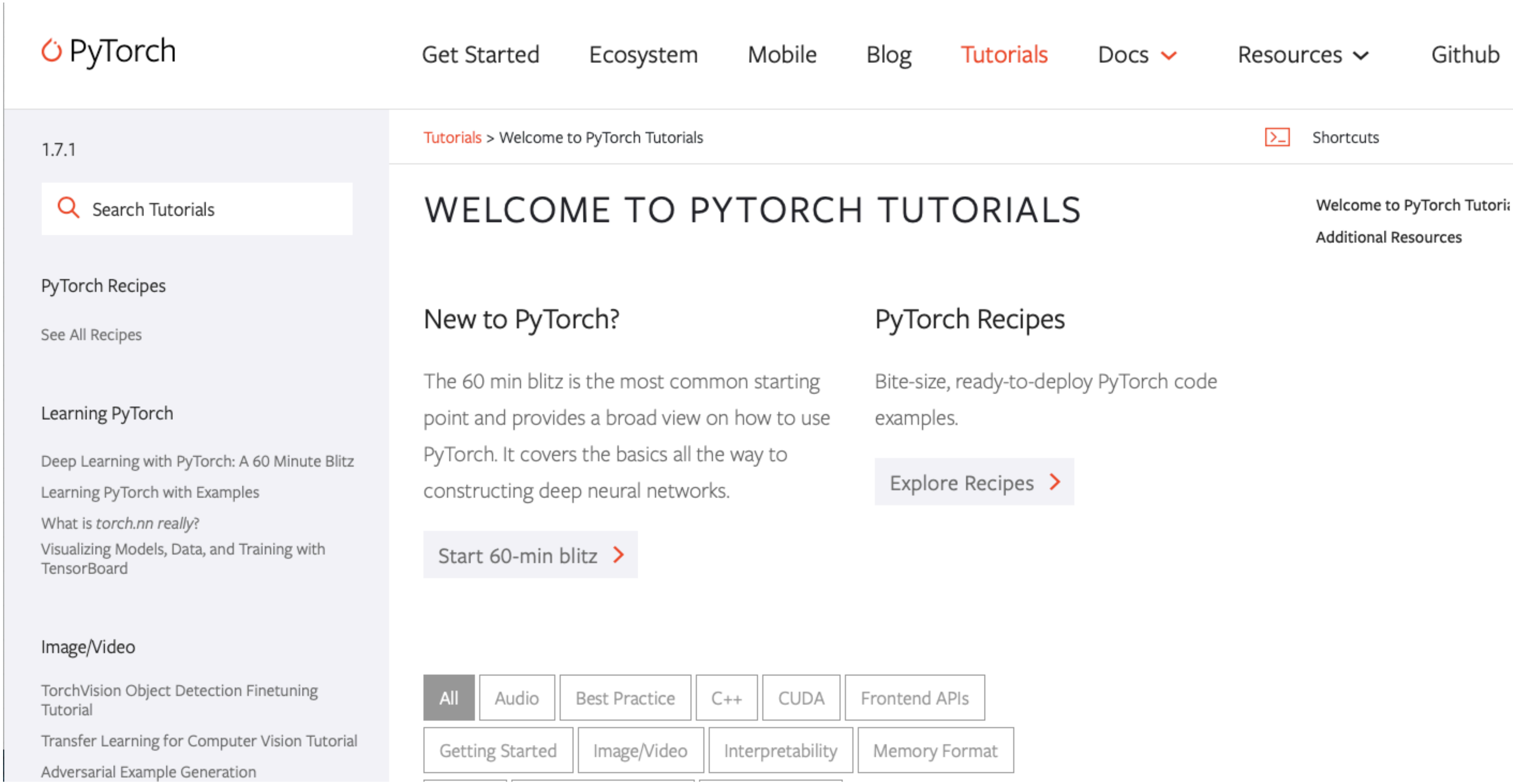
https://pytorch.org/

## As mention in the installation tips on Canvas

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

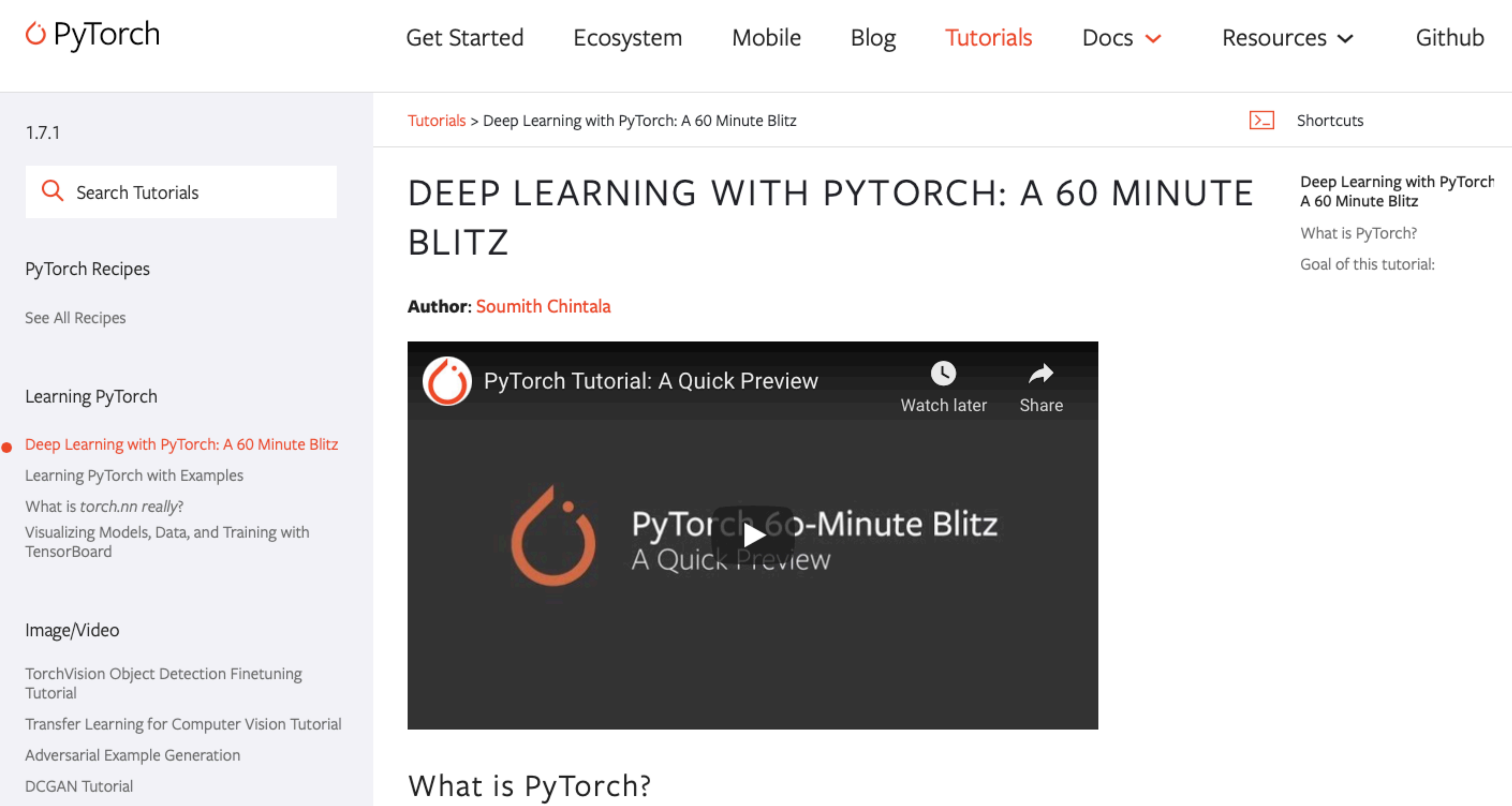
```
In [1]: import torch

In [2]: torch.__version__
Out[2]: '1.7.0'

In [3]:
```

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

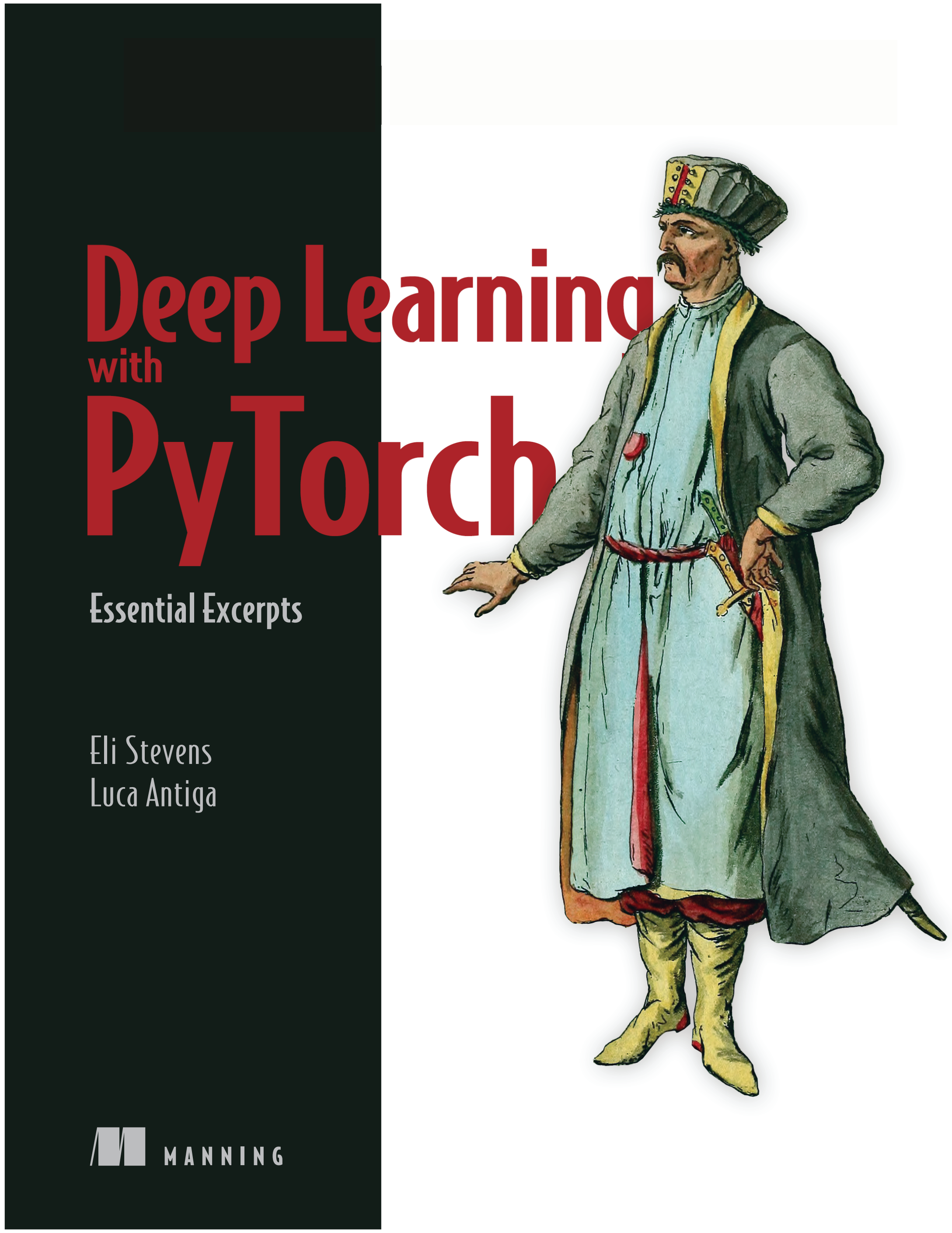



https://pytorch.org/tutorials/

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




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




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

# Very Active & Friendly Community and Help/Discussion Forum

PyTorch

Do you want live notifications when people reply to your posts? Enable Notifications

all categories | all | Latest | New (47) | Unread (104) | Top | Categories | + New Topic

| Topic | Replies | Views | Activity |
|---|---|---|---|
| Using MSELoss instead of CrossEntropy for Ordinal Regression/Classification<br>vision | 2 | 83 | 1h |
| Optimizer.load_state_dict() weird behaviour with Adam optimizer<br>vision | 7 | 2.0k | 1h |
| Is there a way to train 3 dataloaders using multiprocessing? | 0 | 11 | 2h |
| Getting different feature vectors from frozen layers after training<br>vision | 5 | 86 | 2h |
| Libtorch_cuda.so is too large (>2GB)<br>deployment | 22 | 346 | 2h |
| Undo pruning - How to ‘unmask’ pruned weights<br>vision | 0 | 8 | 2h |
| If input.dim() == 2 and bias is not None: AttributeError: ‘tuple’ object has no attribute ‘dim | 3 | 41 | 2h |
| Export unsupported/compound ops to ONNX<br>deployment | 0 | 9 | 2h |

https://discuss.pytorch.org

# Understanding Automatic Differentiation via Computation Graphs

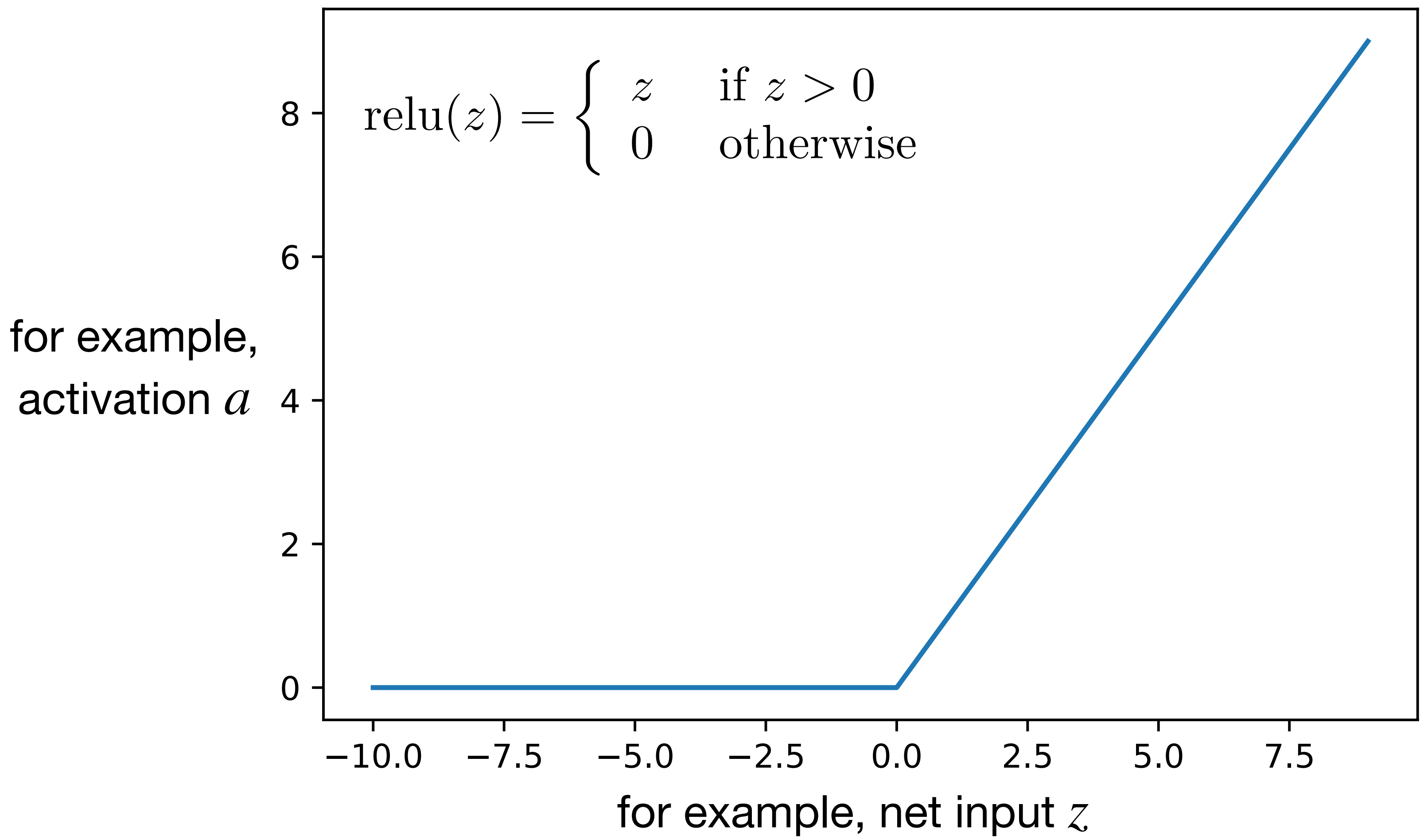
1. PyTorch Resources
2. **Computation Graphs**
3. Automatic Differentiation in PyTorch
4. Training ADALINE Manually Vs Automatically in PyTorch
5. A Closer Look at the PyTorch API

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

$$\sigma'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \text{DNE} & \text{if } z = 0 \end{cases}$$

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

$$\sigma'(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$

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:

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

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

$$\sigma'(0) = \lim_{z \rightarrow 0^{-}} \frac{0 - 0}{\Delta z} = 0$$

# Computation Graphs

Suppose we have the following activation function:

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

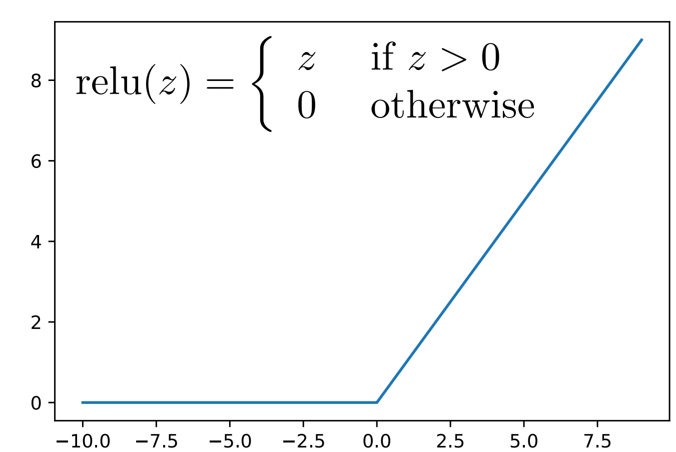
multivariable function

activation function

weight parameter
(assume only 1 input feature)

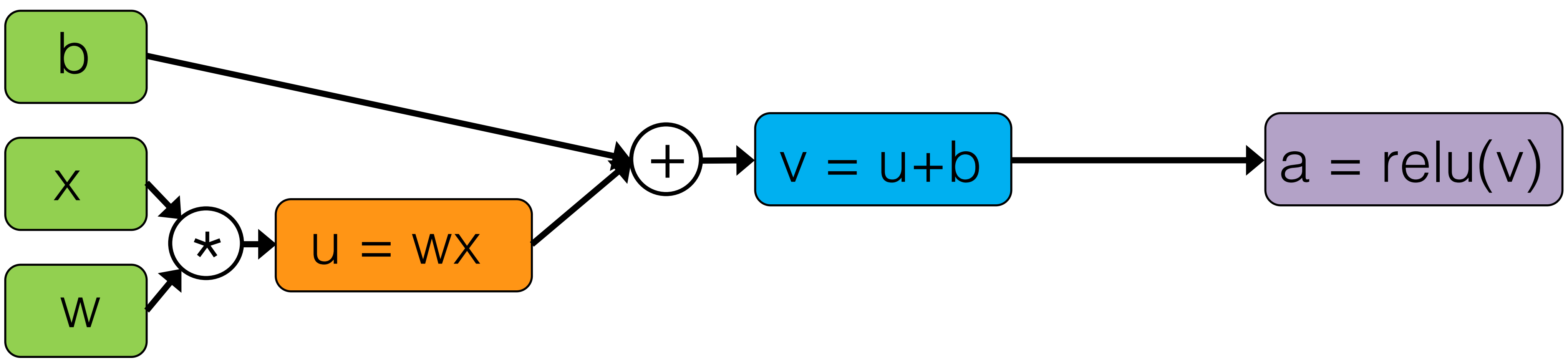
feature
(suppose only 1 training example)

bias

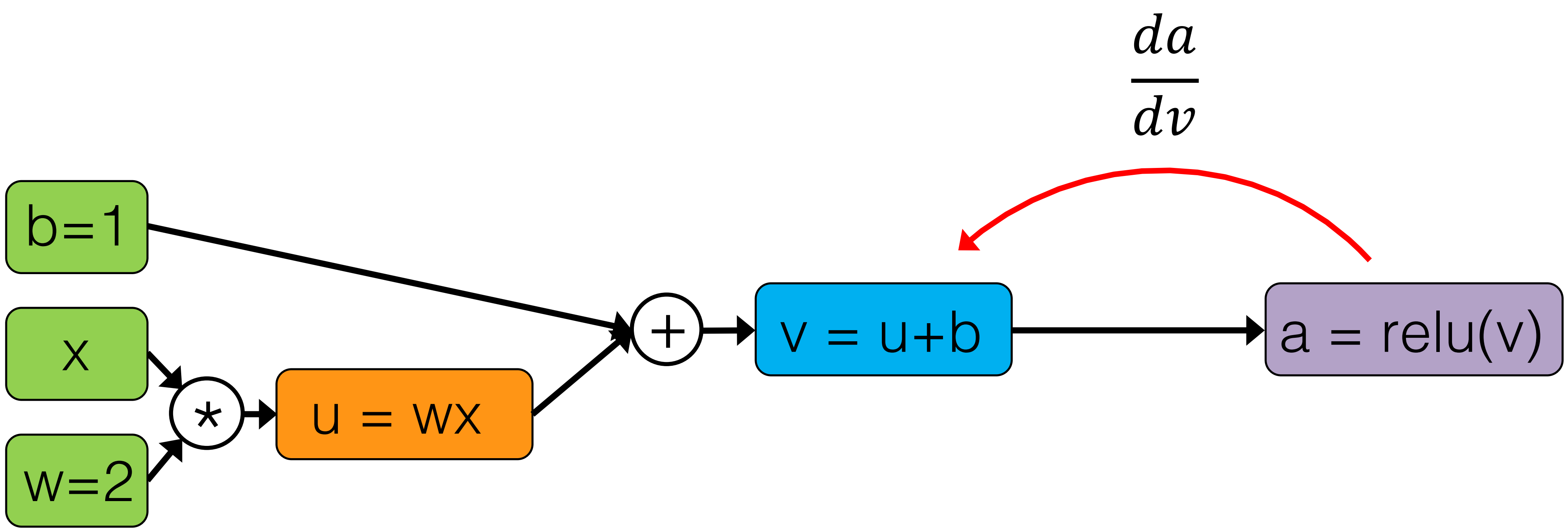


# Computation Graphs

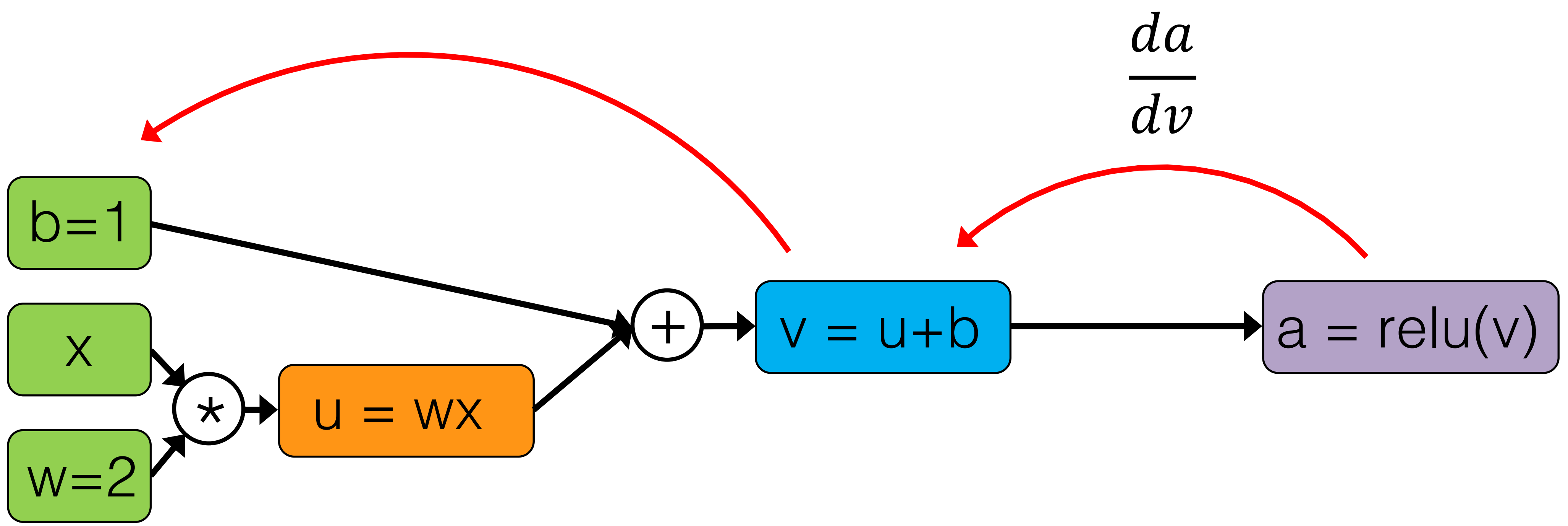
$$a(x,\ w,\ b) = relu(\underbrace{\underbrace{w \cdot x}_{u} + \ b}_{v})$$



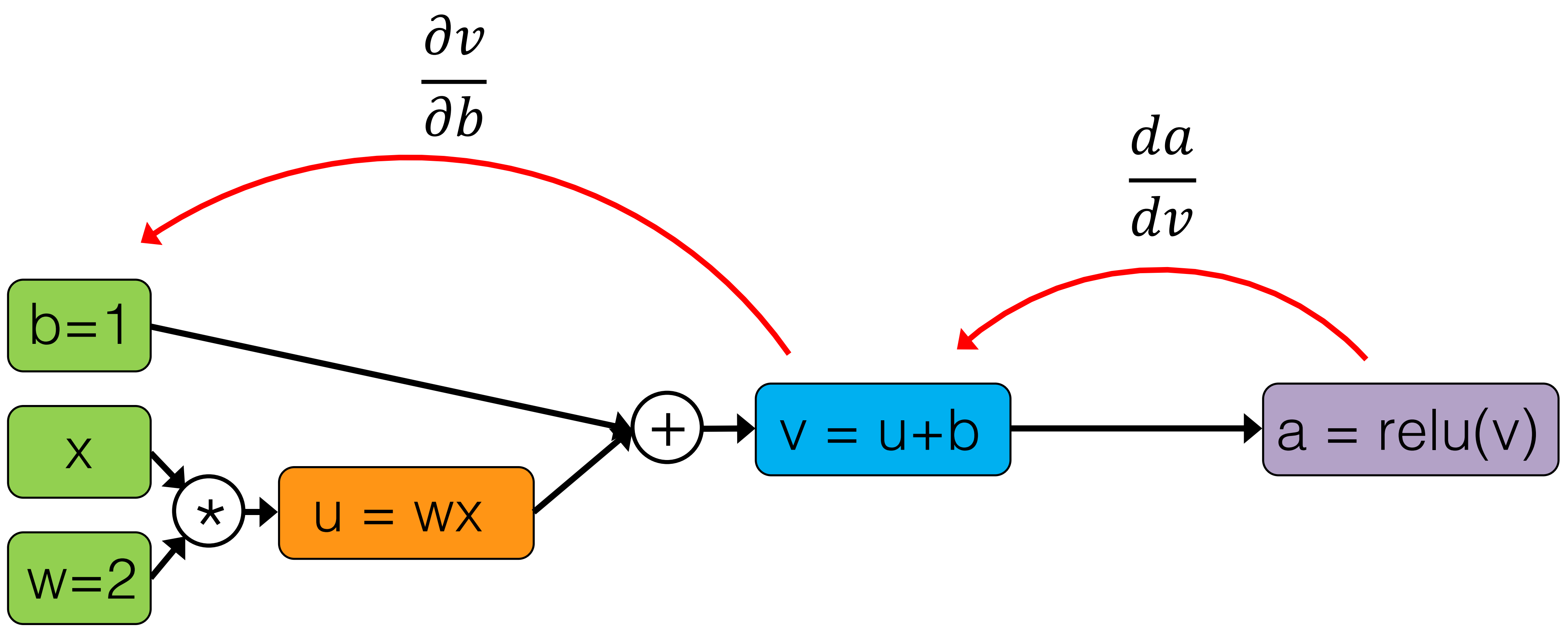
# Computation Graphs



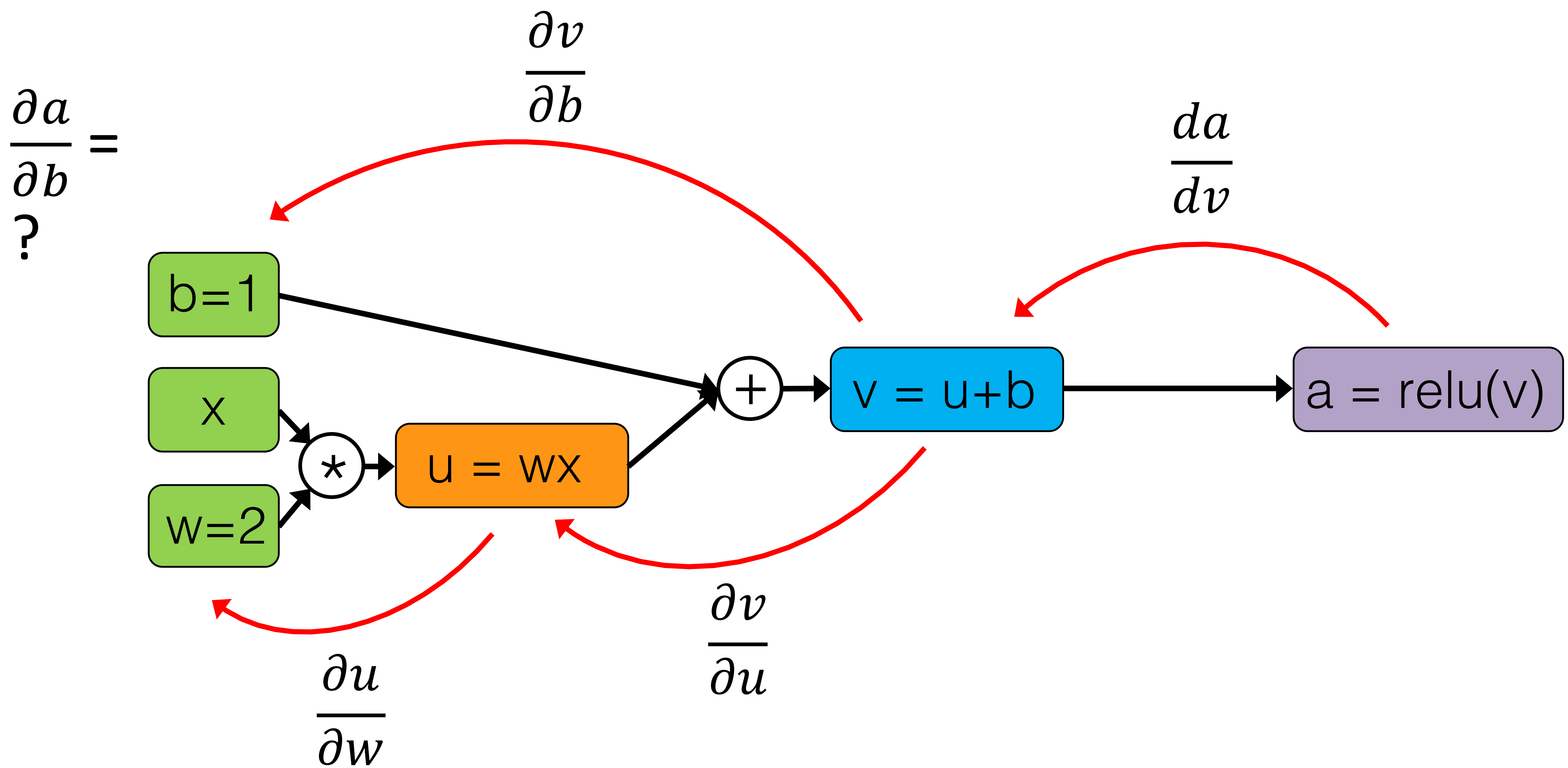
# Computation Graphs



# Computation Graphs

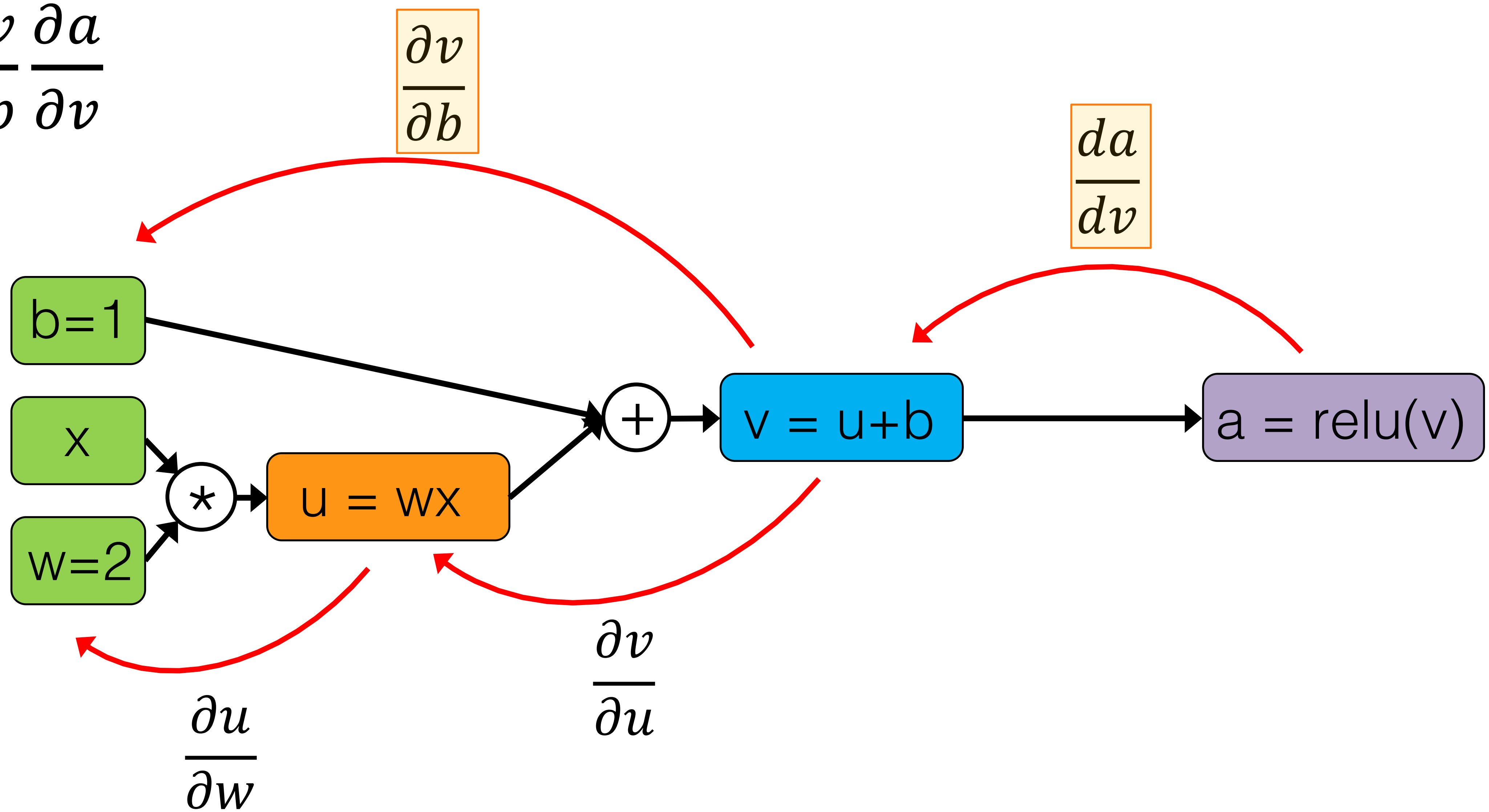


# Computation Graphs



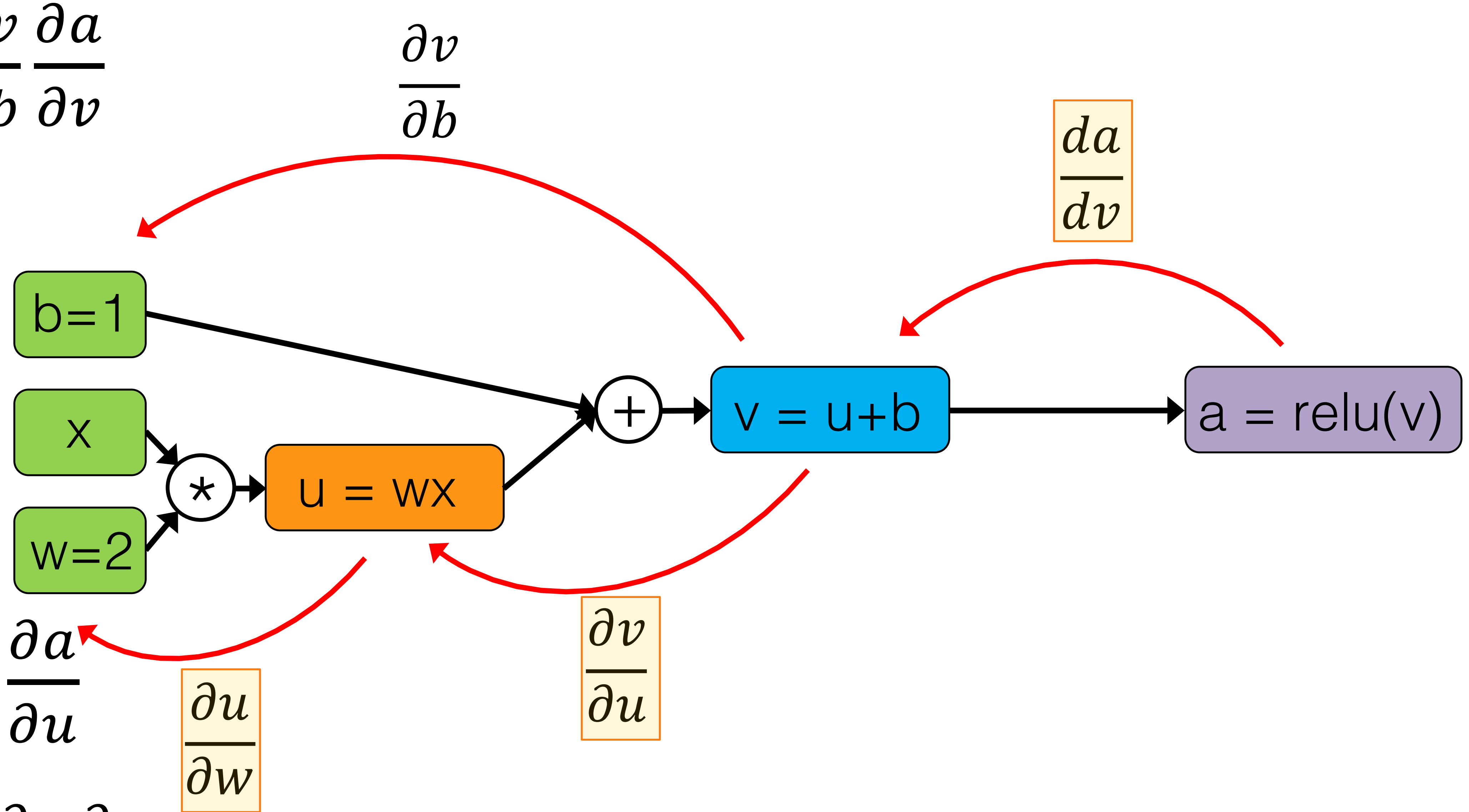
# Computation Graphs

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



# Computation Graphs

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

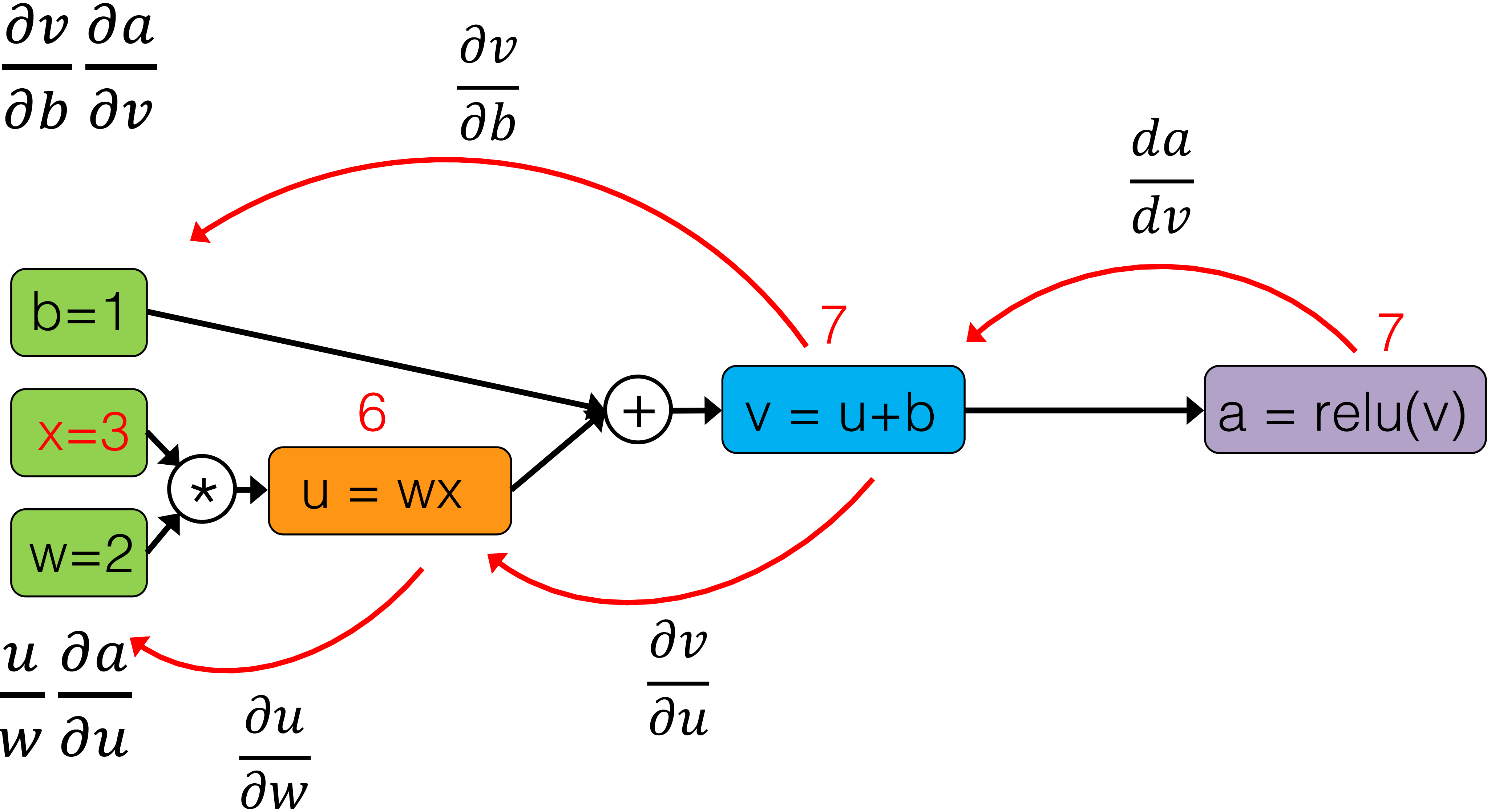



$$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w}\frac{\partial a}{\partial u}$$

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

# Computation Graphs

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

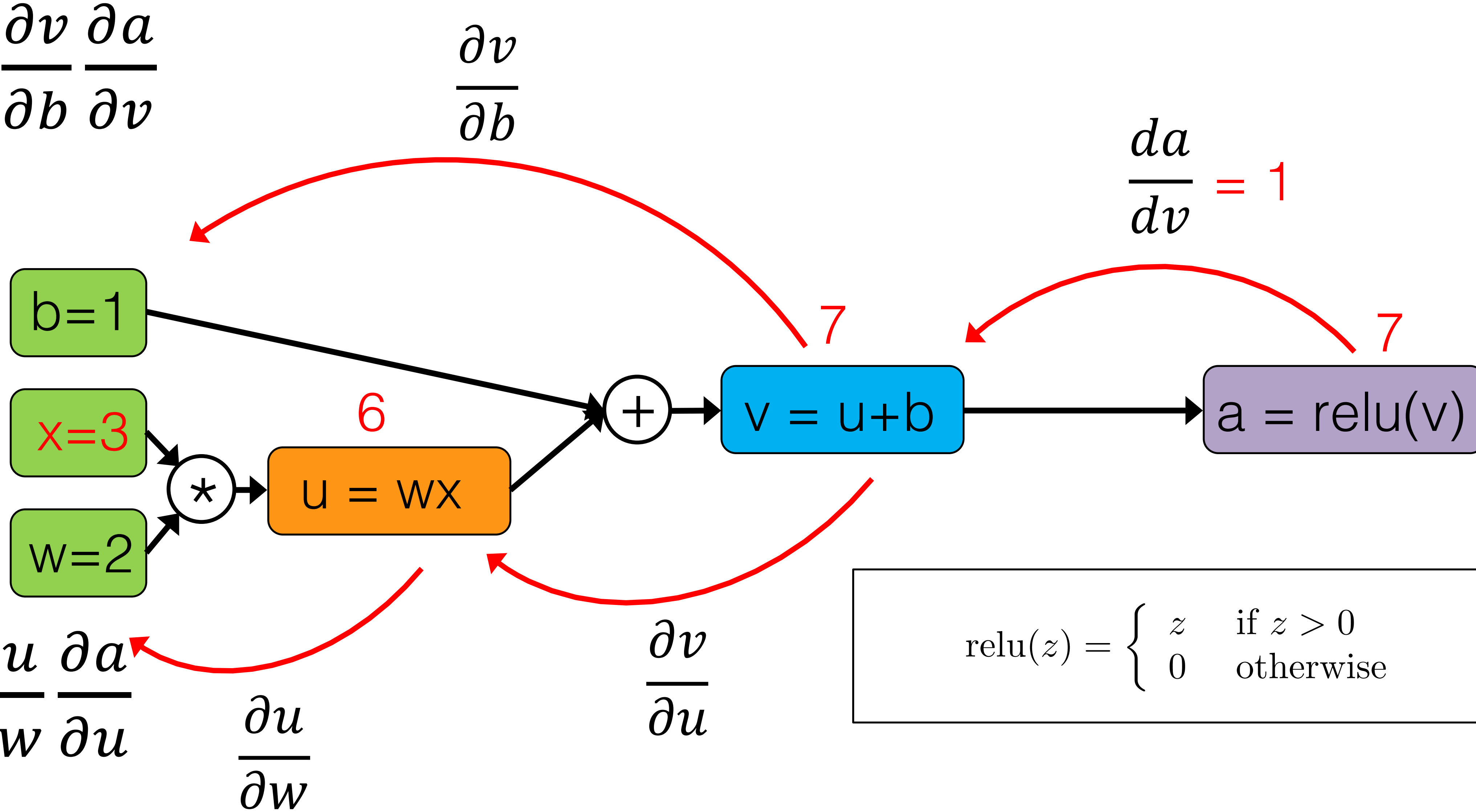



$$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w}\frac{\partial a}{\partial u}$$

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

# Computation Graphs

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




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

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

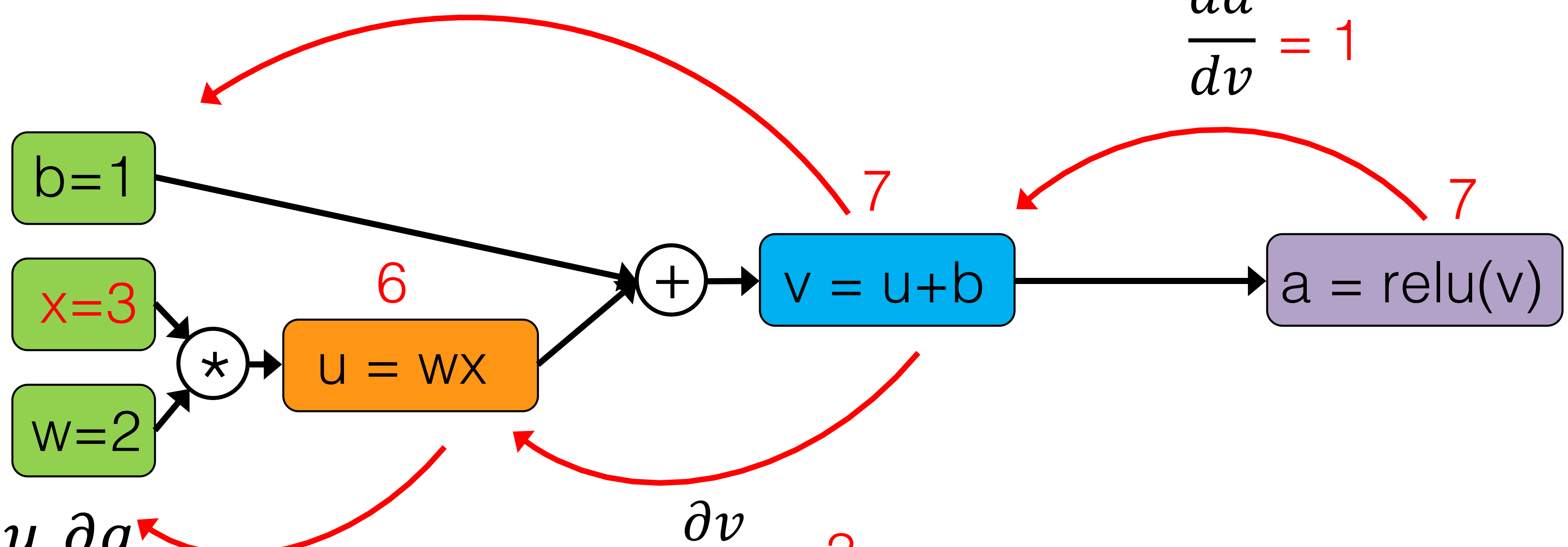
$$\mathrm{relu}(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

# Computation Graphs

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

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

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




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

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

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

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

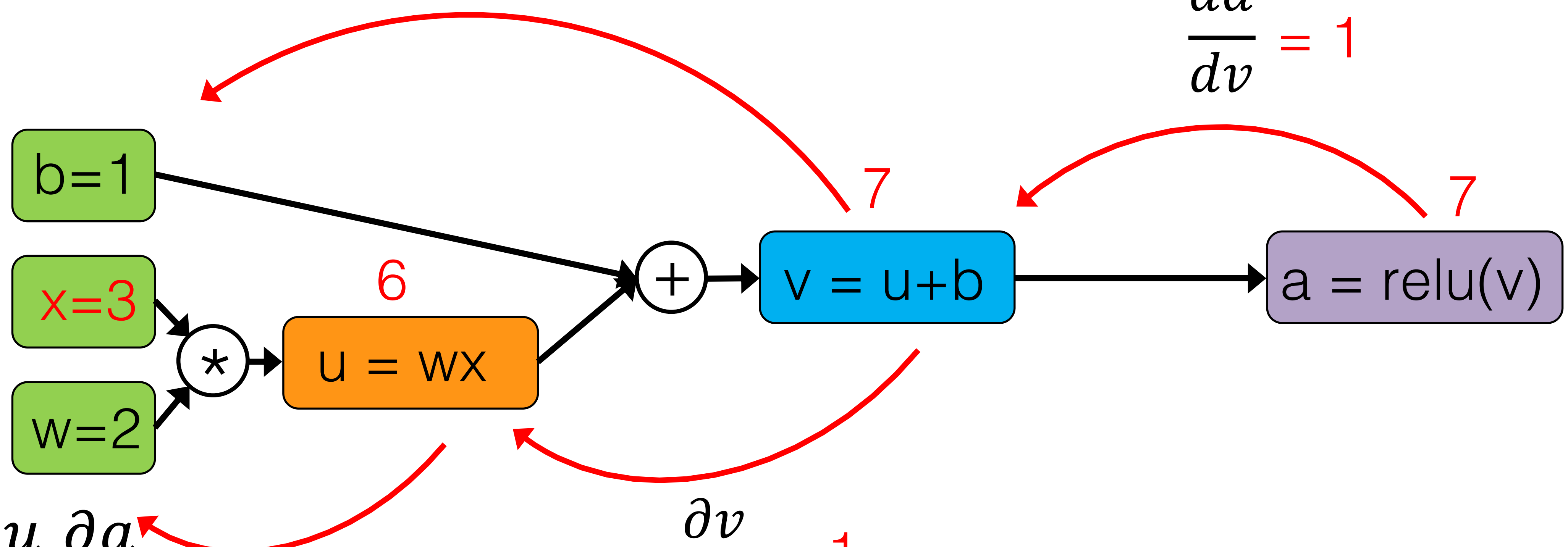
| Function | Derivative |
|---|---|
| $f(x) + g(x)$ | $f'(x) + g'(x)$ |

# Computation Graphs

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

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

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




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

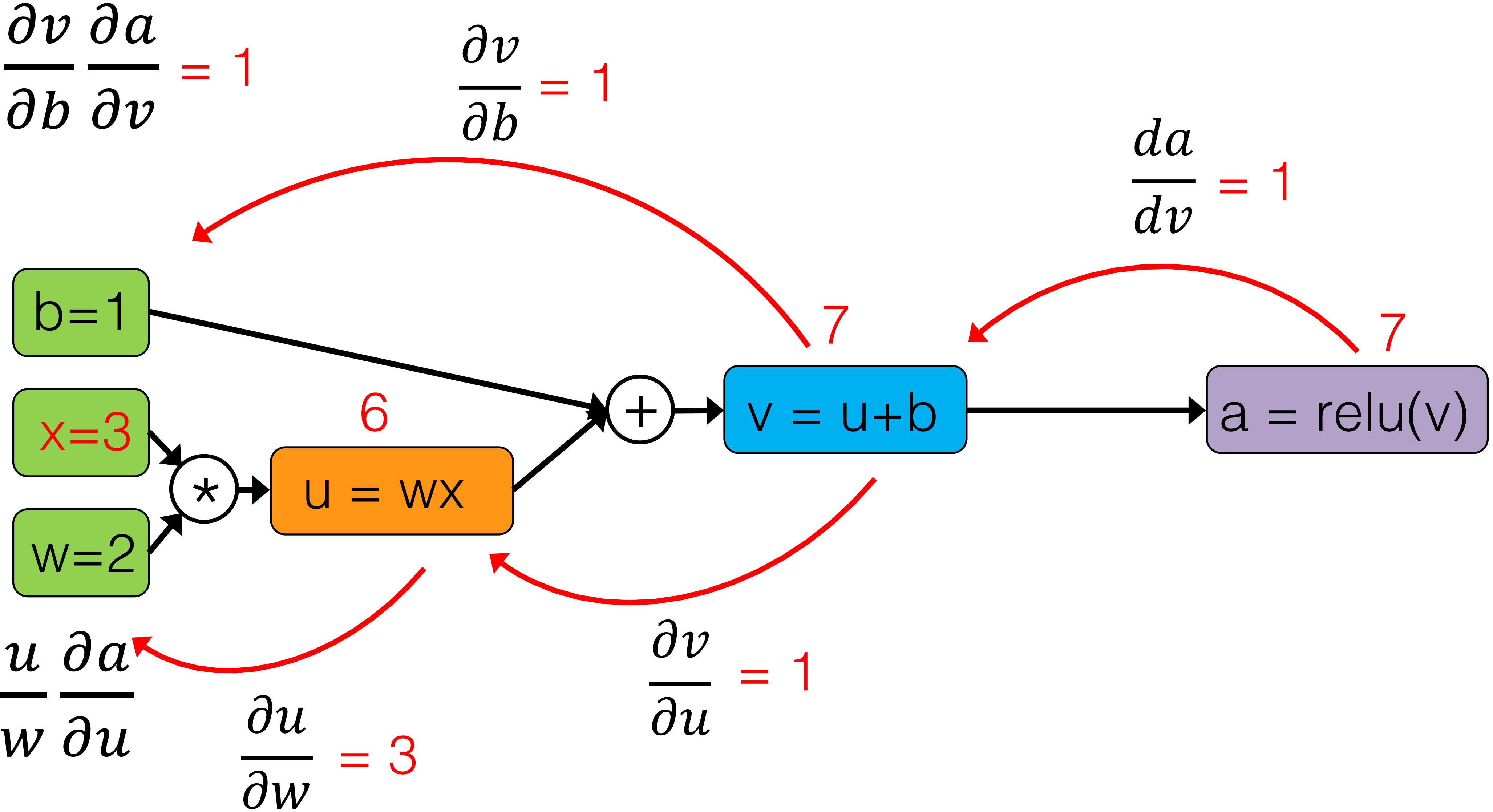
$$= \frac{\partial u}{\partial w}\frac{\partial v}{\partial u}\frac{\partial a}{\partial v}$$

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

$$\frac{\partial v}{\partial u} = 1$$

# Computation Graphs

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



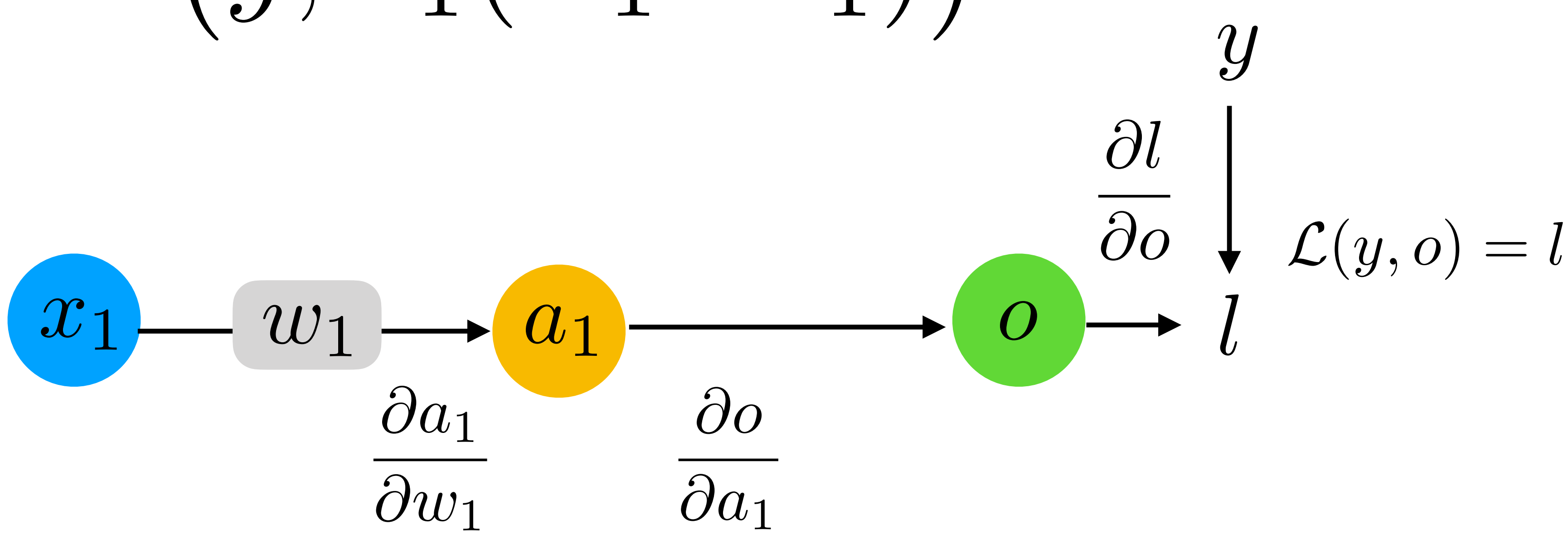

$$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w}\frac{\partial a}{\partial u}$$

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

# Some More Computation Graphs

# Graph with Single Path

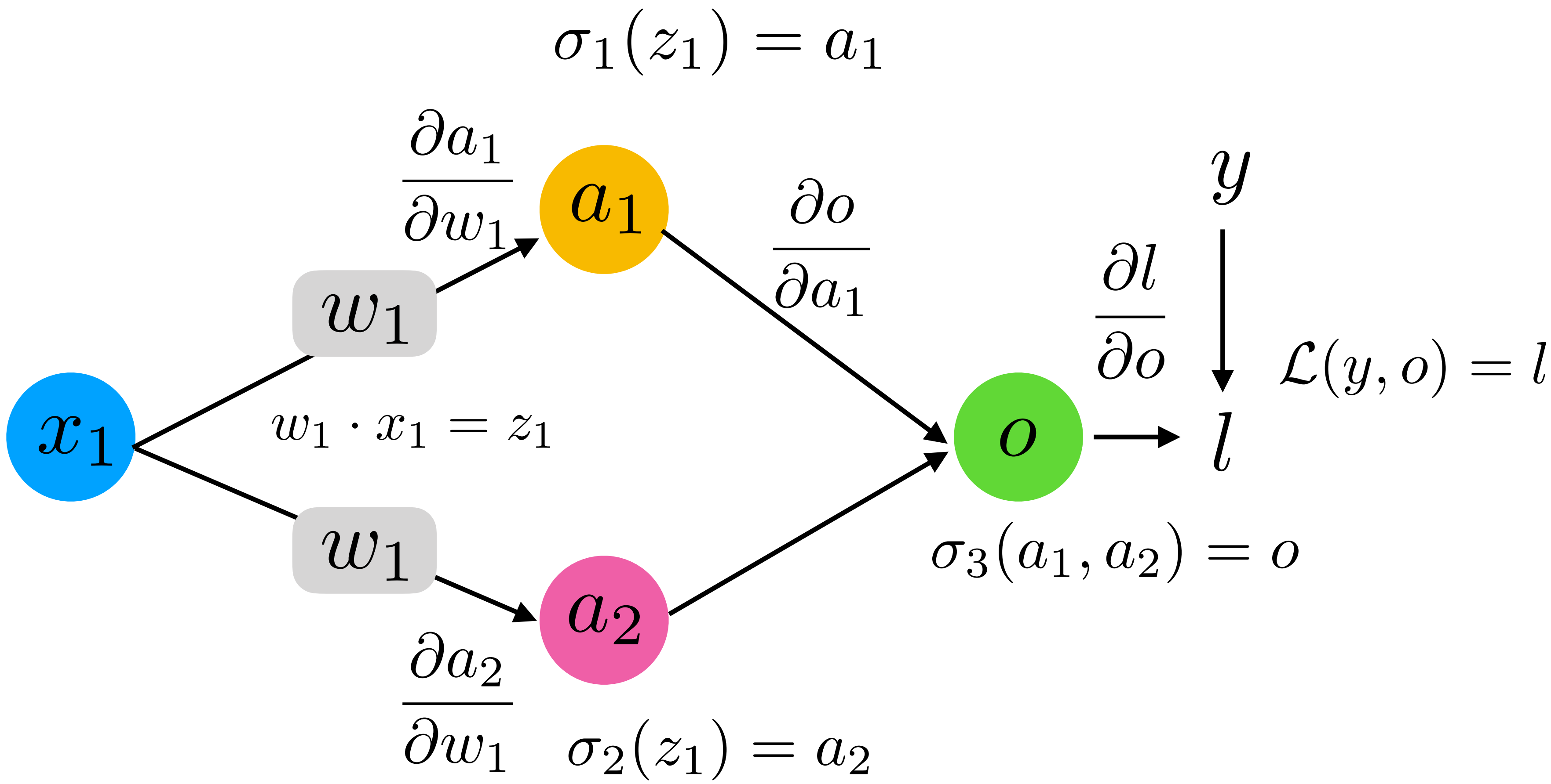
$$\mathcal{L}\Big(y, \sigma_1\big(w_1 \cdot x_1\big)\Big)$$




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

# Graph with Weight Sharing

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

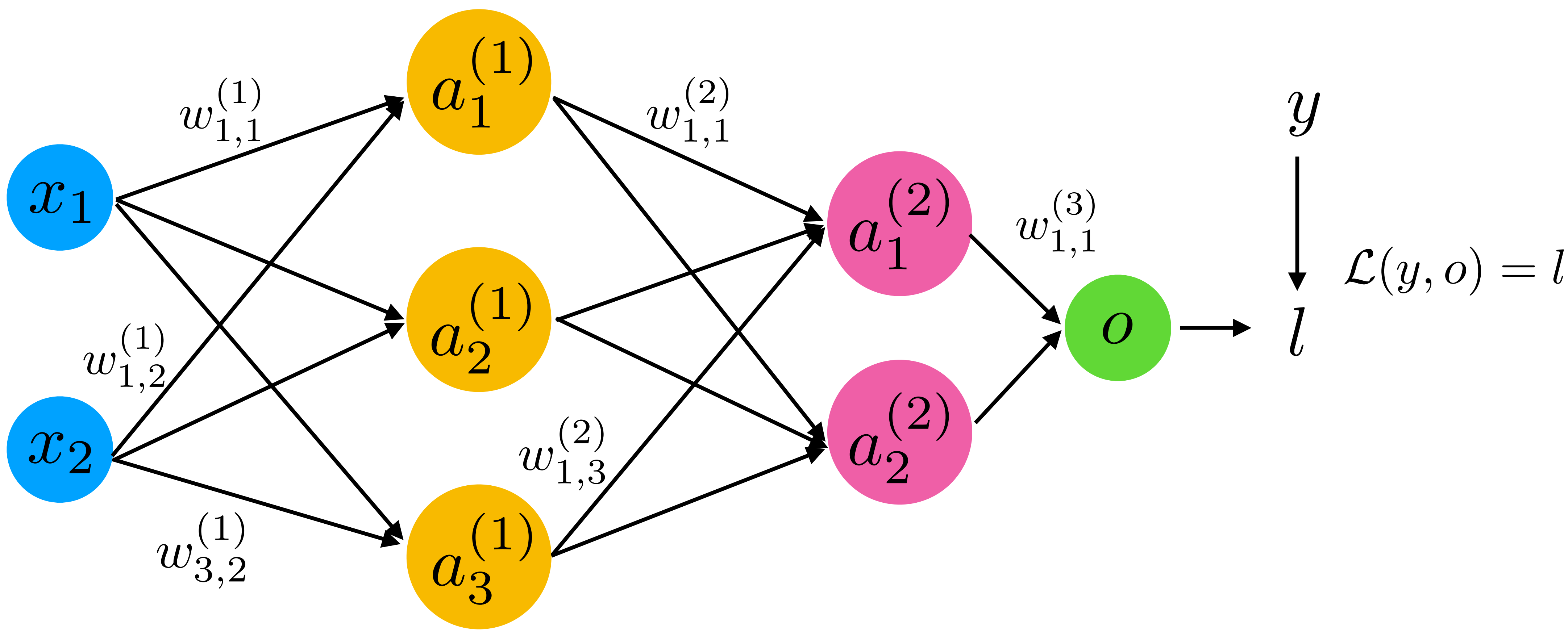



Upper path

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

Lower path

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




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

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

# Automatic Differentiation with PyTorch -- An Autograd Example

1. PyTorch Resources
2. Computation Graphs
3. **Automatic Differentiation in PyTorch**
4. Training ADALINE Manually Vs Automatically in PyTorch
5. A Closer Look at the PyTorch API

# PyTorch Autograd Example

https://github.com/rasbt/stat453-deep-learning-ss21/tree/master/L06/code/pytorch-autograd.ipynb

# Training an Adaptive Linear Neuron in PyTorch

1. PyTorch Resources
2. Computation Graphs
3. Automatic Differentiation in PyTorch
4. **Training ADALINE Manually Vs Automatically in PyTorch**
5. A Closer Look at the PyTorch API

simplify to super().__init__() in constructor

# PyTorch ADALINE (neuron model) Example

https://github.com/rasbt/stat453-deep-learning-ss21/tree/master/L06/code/adaline-with-autograd.ipynb

# Using PyTorch: A Closer Look at the Object-Oriented and Functional APIs

1. PyTorch Resources
2. Computation Graphs
3. Automatic Differentiation in PyTorch
4. Training ADALINE Manually Vs Automatically in PyTorch
5. **A Closer Look at the PyTorch API**

# PyTorch Usage: Step 1 (Definition)

```python
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

```
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
```

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

y = model(x) calls .__call__ and then .forward(), where some extra stuff is done in __call__;

don't run y = model.forward(x) directly

Gradients at each leaf node are accumulated under the .grad attribute, not just stored. This is why we have to zero them before each backward pass

# 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
```

$\text{logits}$ because of computational efficiency. Basically, it internally uses a $\log\_\text{softmax}(\text{logits})$ function that is more stable than $\log(\text{softmax}(\text{logits}))$. More on logits ("net inputs" of the last layer) in the next lecture. Please also see

# Objected-Oriented vs Functional* API

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

torch.nn.functional = api without internal state

```
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"

```python
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
```

```python
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)

```python
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>)]
```

https://github.com/IgorSusmelj/pytorch-styleguide

## Jupyter Notebook vs Python Scripts

In general, we recommend to use jupyter notebooks for initial exploration/ playing around with new models and code. Python scripts should be used as soon as you want to train the model on a bigger dataset where also reproducibility is more important.

**Our recommended workflow:**

1. Start with a jupyter notebook
2. Explore the data and models
3. Build your classes/ methods inside cells of the notebook
4. Move your code to python scripts
5. Train/ deploy on server

| Jupyter Notebook | Python Scripts |
|---|---|
| + Exploration | + Running longer jobs without interruption |
| + Debugging | + Easy to track changes with git |
| - Can become a huge file | - Debugging mostly means rerunning the whole script |
| - Can be interrupted (don't use for long training) | |
| - Prone to errors and become a mess | |

| Type | Convention | Example |
|---|---|---|
| Packages & Modules | lower_with_under | from **prefetch_generator** import BackgroundGenerator |
| Classes | CapWords | class **DataLoader** |
| Constants | CAPS_WITH_UNDER | **BATCH_SIZE=16** |
| Instances | lower_with_under | **dataset** = Dataset |
| Methods & Functions | lower_with_under() | def **visualize_tensor()** |
| Variables | lower_with_under | **background_color='Blue'** |

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