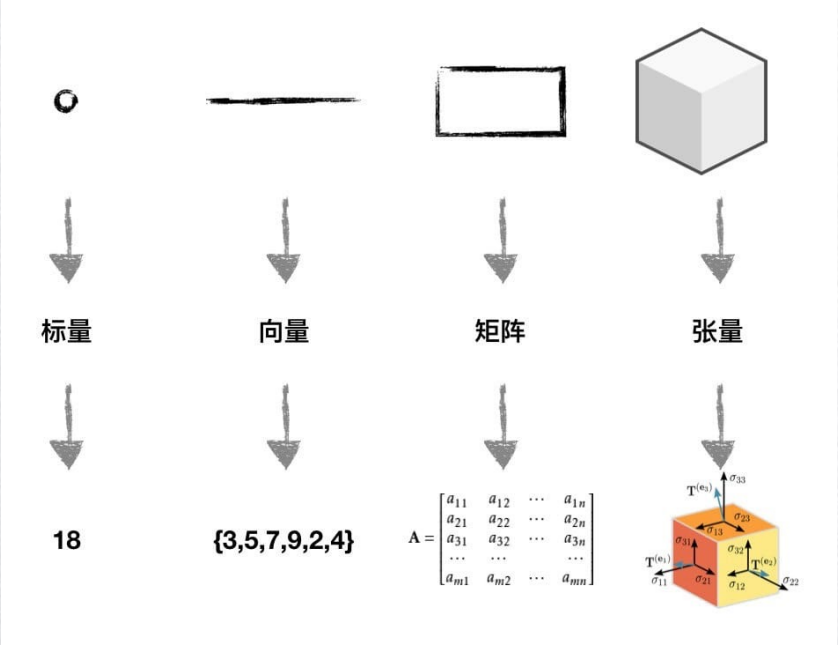
《人工智能与Python程序设计》——PyTorch基础II

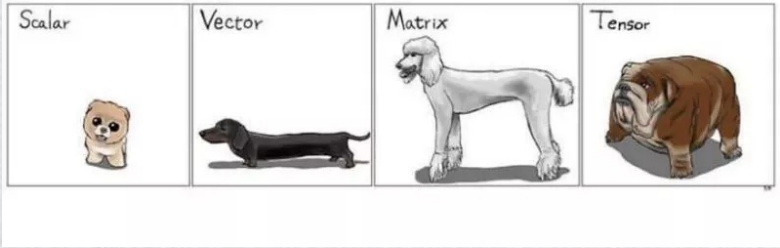
人工智能与Python程序设计 教研组

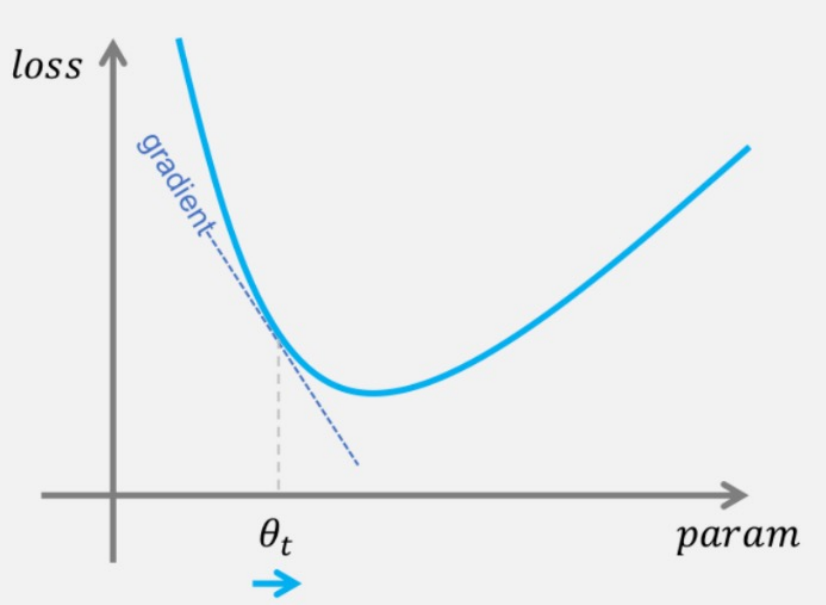
复习

- Pytorch是简单易学的人工智能平台
- 两个主要的组成部分
  - Tensor
  - AutoGrad

张量


标量
向量
矩阵
张量
18
{3,5,7,9,2,4}
$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ a_{31} & a_{32} & \cdots & a_{3n} \\ \cdots & \cdots & & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$


Scalar
Vector
Matrix
Tensor



自动求导

AutoGrad

- 如：函数 $y = x^2$，求$x = 3$处的导数
  - $\frac{dy}{dx} = 2x$
  - $x = 3 \Rightarrow \frac{dy}{dx} = 2 \times 3 = \mathbf{6}$
- AutoGrad是PyTorch中自动求导机制，能够求得一个给定的函数在某一给定点的导数值
  - 定义输入Tensor x，设置requires_grad=True
  - 定义原函数$y = x^2$
    - 用torch中定义的tensor运算符
  - 自动求导：y.backward()
    - 在y处求导，即dy
  - 在x.grad处获得对x的导数值dy/dx
    - 为x=3处的导数值

```
# 求函数 y= x^2 在x=3时的导数：y'(3) = dy/dx|x=3

x=t.tensor(3.0,requires_grad=True)
y=x.mul(x)

#判断x,y是否是可以求导的
print(x.requires_grad)
print(y.requires_grad)

#求导，通过backward函数来实现
y.backward()

#查看导数，也即所谓的梯度
print(x.grad)
```

```
True
True
tensor(6.)
```

多元函数自动求导

- 类似的调用方式
- 每一个输入参数的.grad变量保存了其对应的导数值

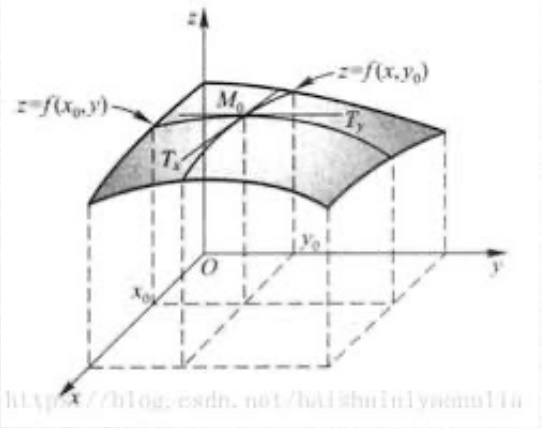
- $f = x^2 + 2y^2 + xy$
  - $\frac{df}{dx} = 2x + y$
  - $\frac{df}{dy} = 4y + x$
- 在(x=1, y=1)处的导数
  - 2 + 1 = 3
  - 4 * 1 + 1 = 5

```
# f(x,y) = x^2 + 2 * y^2 + xy
# 在 (1.0, 1.0)处的导数
x = t.tensor(1.0, requires_grad=True)
y = t.tensor(1.0, requires_grad=True)

f = x.pow(2) + t.tensor(2.0).mul(y.pow(2)) + x.mul(y)
f.backward()
print(x.grad)
print(y.grad)
```

```
tensor(3.)
tensor(5.)
```

以向量为输入的函数的导数

- 函数$f(x)$的输入x可以是向量、矩阵甚至张量(tensor)
  - 如果f的输入是一个标量，$\frac{df}{dx}$也是一个标量
  - 如果f的输入是一个N维向量，$\frac{df}{dx}$也是一个N维的向量
  - 如果f的输入是一个张量，$\frac{df}{dx}$也是一个同样大小的张量
  - 注意：在AutoGrad中我们要求f的返回值为标量



- 举例：$f(\mathbf{x}=[x_1,x_2])=x_1^2+2x_2^2+2x_1x_2+x_1=\mathbf{x}^{\mathrm{T}}\begin{bmatrix}1 & 1\\ 1 & 2\end{bmatrix}\mathbf{x}+\mathbf{x}^{\mathrm{T}}\begin{bmatrix}1\\ 0\end{bmatrix}$
  - $\frac{df}{d\mathbf{x}}=\left[\frac{df}{dx_1},\frac{df}{dx_2}\right]^T=[2x_1+2x_2+1,4x_2+2x_1]^T$
  - $\frac{df}{d\mathbf{x}}=2\begin{bmatrix}1 & 1\\ 1 & 2\end{bmatrix}\mathbf{x}+\begin{bmatrix}1\\ 0\end{bmatrix}$
  - 如果$\mathbf{x}=[1,2]$，导数为[7, 10]

概念回顾

- 数据：
  - 一般为空间中点的集合，每个点以 (x_i, y_i) 的形式出现
  - 一般称 x_i 为“特征”， y_i 为对应的“真实值”或“观测值”
- **任务（task）**：找到由 x 得到 y 的方法，即给定 x，预测 y
  - 回归 vs 分类
- **模型（model）**：
  - 猜想 x 到 y 的映射由 $f(x)$ 给出
  - 模型即 $f(x)$ 的具体形式，带有待定参数
- **损失函数（loss function）**：根据已有数据，评价参数质量的指标
- **训练（优化）（optimization）**：找到最好的 $f(x)$ 参数的过程，即最小化损失函数的过程

使用Tensor保存数据

继承nn.module实现模型

创建Parameter类的实例属性

nn工具箱

- torch.nn是专门为深度学习设计的工具箱
- 主要包括：
  - 参数类 Parameter
  - 模型基类 module
  - 具有不同功能的网络层：卷积、池化、全连接、回归等
  - 非线性激活
  - 损失函数
  - 辅助函数

torch.nn

- Containers
- Convolution Layers
- Pooling layers
- Padding Layers
- Non-linear Activations (weighted sum, nonlinearity)
- Non-linear Activations (other)
- Normalization Layers
- Recurrent Layers
- Transformer Layers
- Linear Layers
- Dropout Layers
- Sparse Layers
- Distance Functions
- Loss Functions
- Vision Layers
- DataParallel Layers (multi-GPU, distributed)
- Utilities
- Quantized Functions

线性回归模型

- 继承nn.Module，实现线性回归

```
class LinearRegression(nn.Module):
    def __init__(self, in_dim): #构造函数，需要调用nn.Mudule的构造函数
        super().__init__()        #等价于nn.Module.__init__()
        self.w=nn.Parameter(torch.randn(in_dim+1, 1))

    def forward(self, x):
        x = torch.cat([x, torch.ones((x.shape[0],1))], dim = 1)
        x = x.matmul(self.w)
        return x
```

Module

- Module是torch.nn 的核心数据结构，是一个抽象的概念，既可以表示神经网络的某个层(layer)，也可以表示一个包含很多层的神经网络。
- 是所有神经网络模块的基类
- Modules也可以包含其它modules。
- 成员变量：
  - training (bool)： 指示这个module是训练还是测试模式

- 线性回归模型可看作是一个简单的神经网络

参数类

`Parameter`	A kind of Tensor that is to be considered a module parameter.

CLASS `torch.nn.parameter.Parameter` [SOURCE]

A kind of Tensor that is to be considered a module parameter.

Parameters are `Tensor` subclasses, that have a very special property when used with `Module` s - when they're assigned as Module attributes they are automatically added to the list of its parameters, and will appear e.g. in `parameters()` iterator. Assigning a Tensor doesn't have such effect. This is because one might want to cache some temporary state, like last hidden state of the RNN, in the model. If there was no such class as `Parameter`, these temporaries would get registered too.

Parameters

- **data** (*Tensor*) – parameter tensor.
- **requires_grad** (*bool, optional*) – if the parameter requires gradient. See Excluding subgraphs from backward for more details. Default: *True*

参数类

- torch.nn.parameter.Parameter
- Tensor的子类，一种特殊的张量，module的参数
  - 当Parameter被指定为Module的属性时，自动注册在parameters() 迭代器
- 属性：
  - Data (Tensor)
  - requires_grad (bool, optional)：默认是True

参数类

迭代器

迭代是Python最强大的功能之一，是访问集合元素的一种方式。

迭代器是一个可以记住遍历的位置的对象。

迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

迭代器有两个基本的方法：**iter()** 和 **next()**。

字符串，列表或元组对象都可用于创建迭代器：

实例(Python 3.0+)

```
>>> list=[1,2,3,4]
>>> it = iter(list)    # 创建迭代器对象
>>> print (next(it))   # 输出迭代器的下一个元素
1
>>> print (next(it))
2
>>>
```

自定义线性回归Module

- Module中的可学习参数可以通过成员函数parameters(返回。

```
22    for parameter in layer.parameters():
23        print(parameter)
```

```
Parameter containing:
tensor([[1.2220],
        [1.0318],
        [0.3946],
        [0.3981]], requires_grad=True)
```

自定义线性回归Module

- 成员函数：forward()，实现前向传播过程，其输入可以是一个或多个Tensor, 对x 的任何操作也必须是Tensor 支持的操作。

  forward(self, x，[y, ...])
  - 定义每次调用Module实例时执行的计算，网络/层前向计算过程
  - 所有子类/继承类都应重写此函数

```
def forward(self, x):
    x = torch.cat([x, torch.ones((x.shape[0],1))], dim = 1)
    x = x.matmul(self.w)
    return x
```

- 无需写反向传播函数，nn.Module能够利用autograd自动实现反向传播。

自定义线性回归Module

- 使用时，首先实例化一个LinearRegression类

例如：

```
def testLRmodel(in_dim, data_size = 2):
    layer = LinearRegression(in_dim)
    input=torch.randn(data_size,in_dim)
    output=layer(input)  #前向传播 执行forward()
    print(output)
    for parameter in layer.parameters():
        print(parameter)
```

- 直观上可以将layer看成数学概念中的函数，调用model(input)即可得到input 对应的前向计算（forward）结果。

等价于layer.__call__(input).

pytroch在nn.Module中实现了__call__方法，并且在__call__方法中调用了forward函数

概念回顾

- 数据：
  - 一般为空间中点的集合，每个点以 (x_i, y_i) 的形式出现
  - 一般称 x_i 为“特征”，y_i 为对应的“真实值”或“观测值”
- **任务（task）**：找到由 x 得到 y 的方法，即给定 x，预测 y
  - 回归 vs 分类
- **模型（model）**：
  - 猜想 x 到 y 的映射由 $f(x)$ 给出
  - 模型即 $f(x)$ 的具体形式，带有待定参数
- **损失函数（loss function）**：根据已有数据，评价参数质量的指标
- **训练（优化）（optimization）**：找到最好的 $f(x)$ 参数的过程，即最小化损失函数的过程

使用Tensor保存数据

继承nn.module实现模型

创建Parameter类的实例属性

通过Tensor计算得到损失函数

使用AutoGrad计算损失函数关于参数的梯度

PyTorch训练

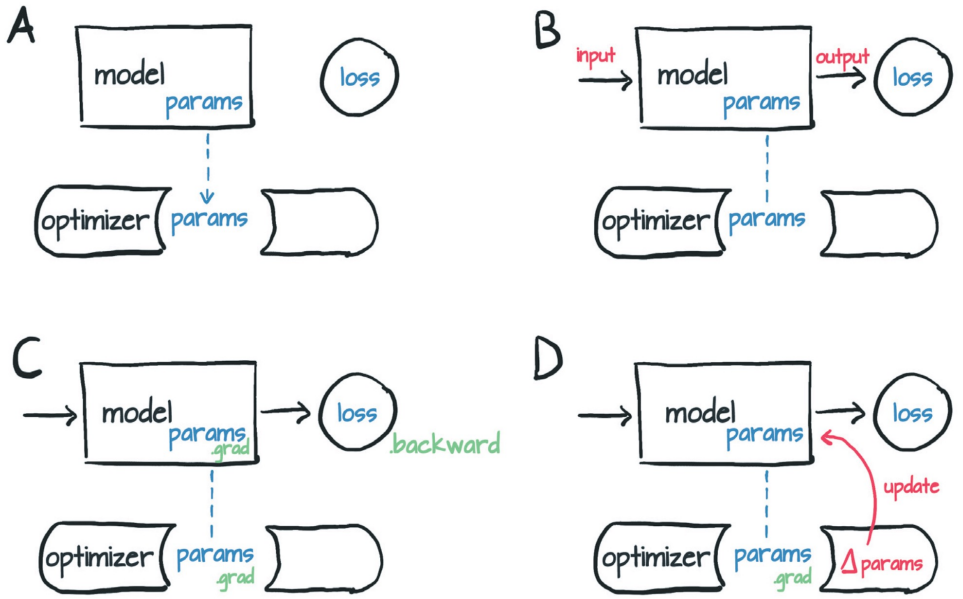


Figure 4.10 Conceptual representation of how an optimizer holds a reference to parameters (A), and after a loss is computed from inputs (B), a call to `.backward` leads to `.grad` being populated on parameter (C). At that point, the optimizer can access `.grad` and compute the parameter updates (D).

Pytorch模型训练

- 实现Linear_Model类，实现其成员函数对线性回归模型进行训练和测试
- 初始化：创建模型和优化器，初始化线性模型和优化器超参数

```
class Linear_Model():
    def __init__(self, in_dim):
        """
        创建模型和优化器，初始化线性模型和优化器超参数
        """
        self.learning_rate = 0.01
        self.epoches = 10000
        self.model = LinearRegression(in_dim) #torch.nn.Linear(in_dim,1)
        self.optimizer = torch.optim.SGD(self.model.parameters(), lr=self.learning_rate)
        self.loss_function = torch.nn.MSELoss()
```

- 线性回归模型参数已在实例化LinearRegression类时用正态分布随机初始化

```
class LinearRegression(nn.Module):
    def __init__(self, in_dim): #构造函数，需要调用nn.Mudule的构造函数
        super().__init__()        #等价于nn.Module.__init__()
        self.w=nn.Parameter(torch.randn(in_dim+1, 1))
```

Pytorch优化器

- Pytoch中用来优化模型权重的类：torch.optim.Optimizer
- 各种常用的优化器都是Optimizer这个基类的子类

- *CLASS torch.optim.Optimizer(params, defaults)*
- 参数：params的数据类型有两种:
  - 通常把model.parameters()传进去

Pytorch优化器

- 随机梯度下降优化器

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

- 参数：模型的parameters()方法返回模型参数iterator
  lr：学习率

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^{t} - lr\left(\frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}}\bigg|_{\boldsymbol{w}^{t}}\right)^{T}$$

Pytorch损失函数

$$L = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

- 均方误差损失（nn.MSELoss）实例化：

```
self.loss_function = torch.nn.MSELoss()
```

- forward：输入两个tensor（所有样本的预测值、所有样本的真值），返回均方误差

Pytorch模型训练

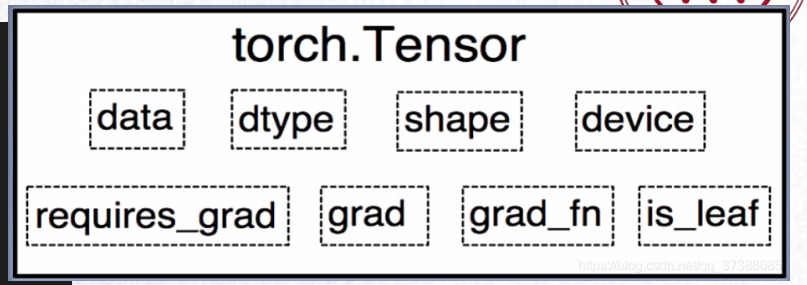
```
def train(self, x, y):
    """
    训练模型并保存参数
    输入:
        model_save_path: saved name of model
        x: 训练数据
        y: 回归真值
    返回:
        losses: 所有迭代中损失函数值
    """
    losses = []
    for epoch in range(self.epoches):
        prediction = self.model(x)
        loss = self.loss_function(prediction, y)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        losses.append(loss.item())

        if epoch % 500 == 0:
            print("epoch: {}, loss is: {}".format(epoch, loss.item()))
```

模型使用当前参数的对训练样本进行预测

计算预测值和真值的MSE损失

Pytorch模型训练

```python
def train(self, x, y):
    """
    训练模型并保存参数
    输入:
        model_save_path: saved name of model
        x: 训练数据
        y: 回归真值
    返回:
        losses: 所有迭代中损失函数值
    """
    losses = []
    for epoch in range(self.epoches):
        prediction = self.model(x)
        loss = self.loss_function(prediction, y)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        losses.append(loss.item())

        if epoch % 500 == 0:
            print("epoch: {}, loss is: {}".format(epoch, loss.item()))
```



pytorch中的backward()函数的计算，梯度是被积累的而不是被替换掉

所有参数的梯度置0

Pytorch模型训练

```
def train(self, x, y):
    """
    训练模型并保存参数
    输入:
        model_save_path: saved name of model
        x: 训练数据
        y: 回归真值
    返回:
        losses: 所有迭代中损失函数值
    """
    losses = []
    for epoch in range(self.epoches):
        prediction = self.model(x)
        loss = self.loss_function(prediction, y)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        losses.append(loss.item())

        if epoch % 500 == 0:
            print("epoch: {}, loss is: {}".format(epoch, loss.item()))
```

计算梯度

Pytorch模型训练

```python
def train(self, x, y):
    """
    训练模型并保存参数
    输入:
        model_save_path: saved name of model
        x: 训练数据
        y: 回归真值
    返回:
        losses: 所有迭代中损失函数值
    """
    losses = []
    for epoch in range(self.epoches):
        prediction = self.model(x)
        loss = self.loss_function(prediction, y)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        losses.append(loss.item())

        if epoch % 500 == 0:
            print("epoch: {}, loss is: {}".format(epoch, loss.item()))
```

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^{t} - lr\left(\frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}}\bigg|_{\boldsymbol{w}^{t}}\right)^{T}$$

所有参数更新一次

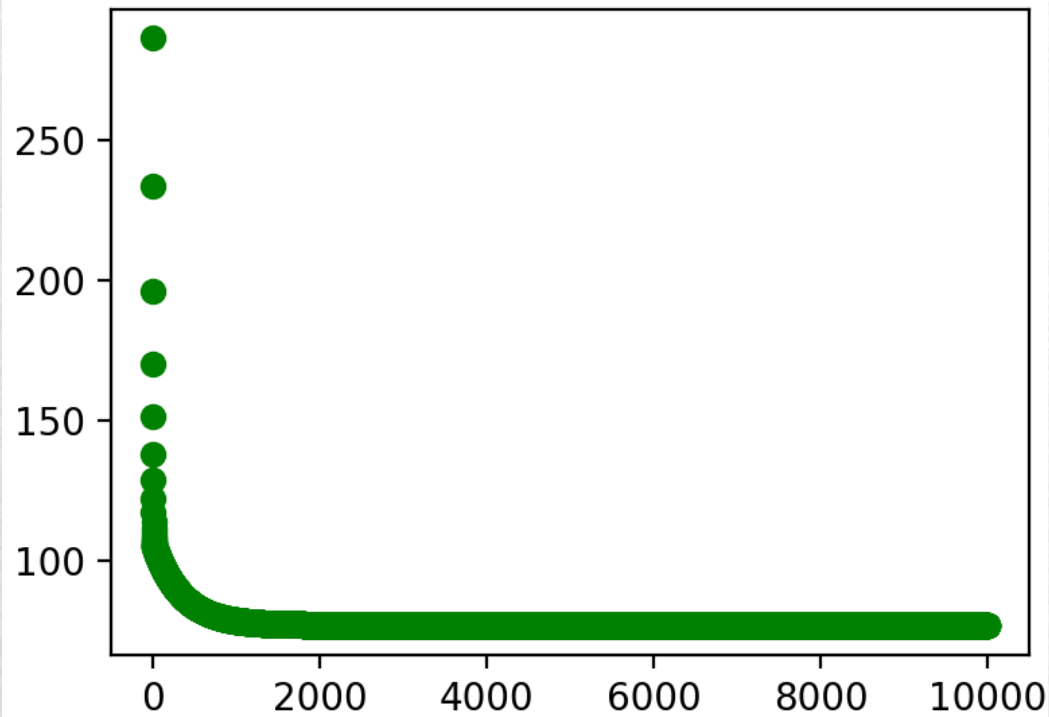
Pytorch模型训练

```python
def train(self, x, y):
    """
    训练模型并保存参数
    输入:
        model_save_path: saved name of model
        x: 训练数据
        y: 回归真值
    返回:
        losses: 所有迭代中损失函数值
    """
    losses = []
    for epoch in range(self.epoches):
        prediction = self.model(x)
        loss = self.loss_function(prediction, y)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        losses.append(loss.item())

        if epoch % 500 == 0:
            print("epoch: {}, loss is: {}".format(epoch, loss.item()))
```

每500轮打印一次损失值

Pytorch模型训练

- Loss随着迭代次数的变化



Pytorch模型测试

```python
def test(self, x, y, if_plot = True):
    """
    用保存或训练好的模型做测试
    输入:
        model_path: 训练好的模型的保存路径, e.g., "linear.pth"
        x: 测试数据
        y: 测试数据的回归真值
    返回:
        prediction: 测试数据的预测值
    """
    prediction = self.model(x)
    testMSE = self.loss_function(prediction, y)

    if if_plot and x.shape[1]==1:
        plt.figure()
        plt.scatter(x.numpy(), y.numpy())
        plt.plot(x.numpy(), prediction.numpy(), color="r")
        plt.show()

    return prediction, testMSE
```

用模型对测试数据进行预测

计算预测结果和测试数据真值的MSE损失

Pytorch模型测试

```
def test(self, x, y, if_plot = True):
    """
    用保存或训练好的模型做测试
    输入:
        model_path: 训练好的模型的保存路径, e.g., "linear.pth"
        x: 测试数据
        y: 测试数据的回归真值
    返回:
        prediction: 测试数据的预测值
    """
    prediction = self.model(x)
    testMSE = self.loss_function(prediction, y)

    if if_plot and x.shape[1]==1:
        plt.figure()
        plt.scatter(x.numpy(), y.numpy())
        plt.plot(x.numpy(), prediction.numpy(), color="r")
        plt.show()

    return prediction, testMSE
```

用模型对测试数据进行预测

计算预测结果和测试数据真值的MSE损失

Pytorch模型测试

- in_dim = 3

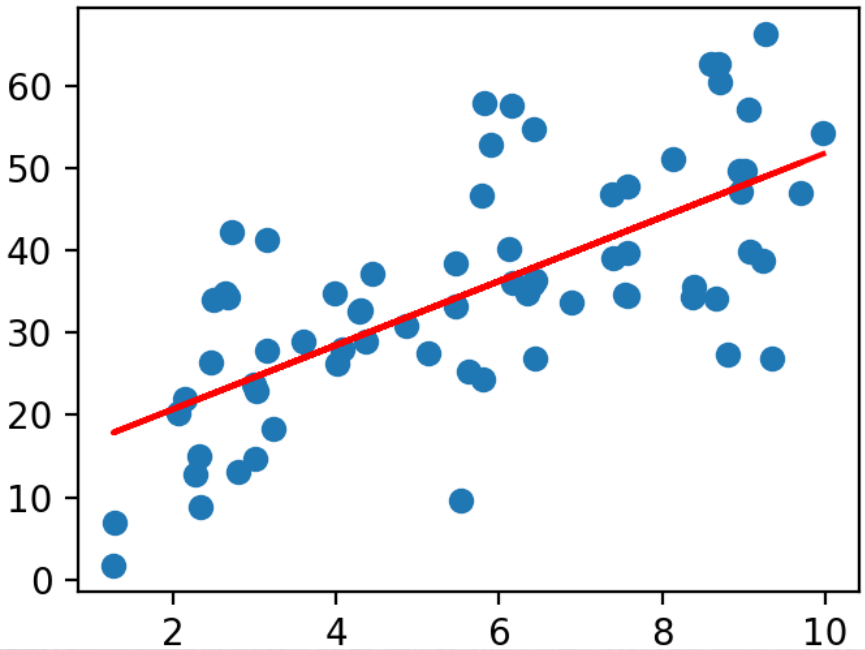
```
w真值:tensor([[-0.5206],
        [-3.0018],
        [ 1.4464]])
```

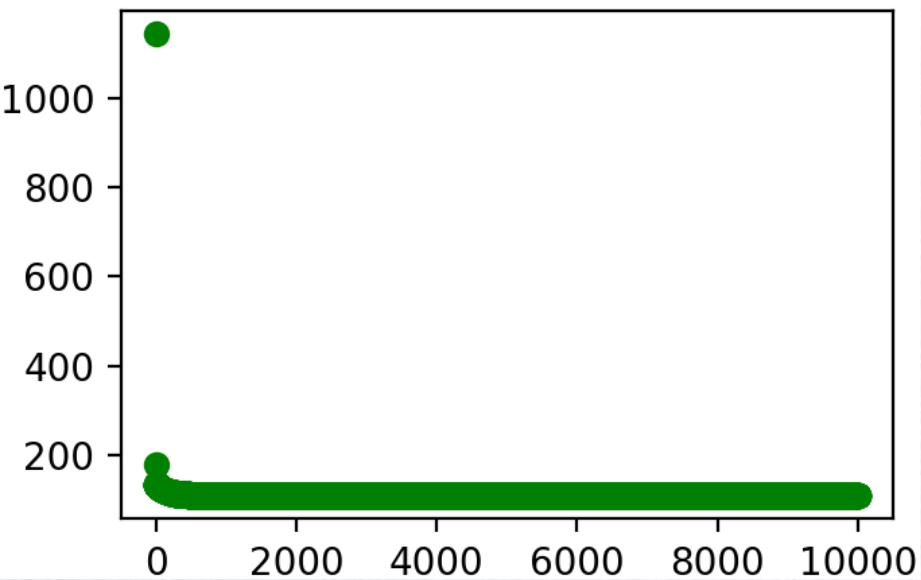
```
w Parameter containing:
tensor([[-0.6389],
        [-3.8588],
        [ 0.9694],
        [18.1736]], requires_grad=True)
```

```
测试集上MSE损失值:130.86734008789062
保存的模型在测试集上MSE损失值:130.86734008789062
```

Pytorch模型训练测试

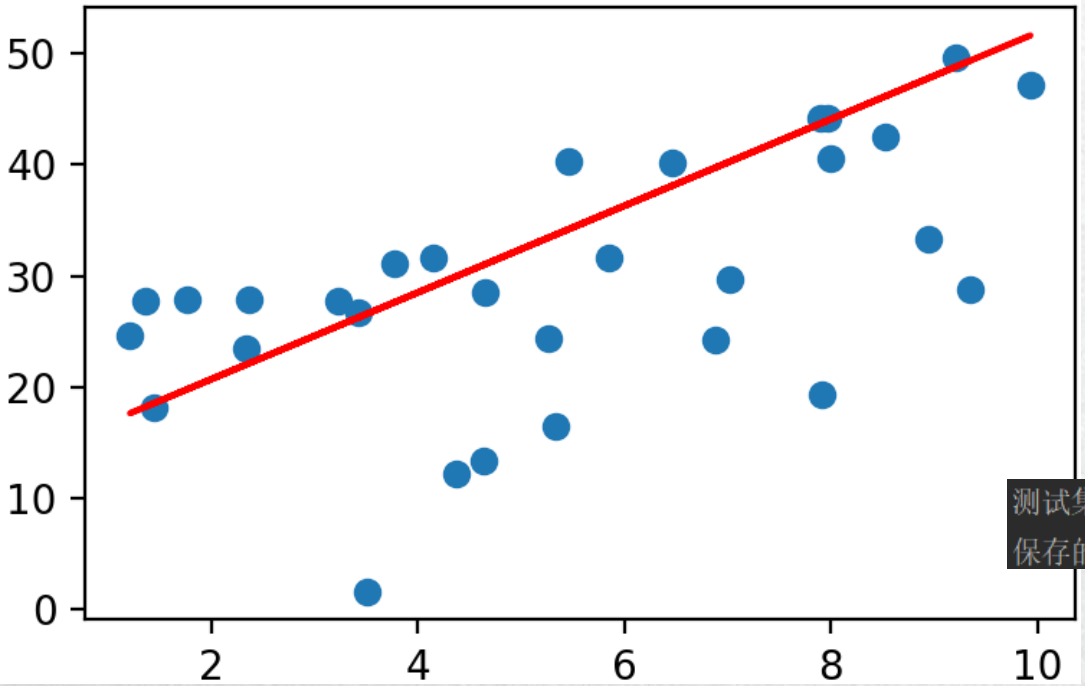
- in_dim=1, 训练:





Pytorch模型训练测试

- in_dim=1, 测试：




```
w真值:tensor([[4.1975]])
```

```
w Parameter containing:
tensor([[ 3.9004],
        [12.9037]], requires_grad=True)
```

```
测试集上MSE损失值:119.6384048461914
保存的模型在测试集上MSE损失值:119.6384048461914
```

Pytorch和numpy实现线性回归对比

-

```python
def fit(self, x, y):
    '''
    类的方法：训练函数
    参数 自变量：x
    参数 因变量：y
    返回每一次迭代后的损失函数
    '''
    for i in range(self.max_iter):
        self.__train_step(x,y)
        y_pred = self.predict(x)
        self.loss_arr.append(self.loss(y, y_pred))
```

```python
def __train_step(self,x,y):
    '''
    类的方法：单步迭代，即一次迭代中对梯度进行更新
    '''
    d_w, d_b = self.__calc_gradient(x,y)
    self.w = self.w - self.lr * d_w
    self.b = self.b - self.lr * d_b
    return self.w, self.b
```

```python
def __calc_gradient(self, x, y):
    '''
    类的方法：分别计算对w和b的梯度
    '''
    d_w = np.mean(2* (x * self.w + self.b - y) * x)
    d_b = np.mean(2*(x * self.w + self.b - y))
    return d_w, d_b
```

PyTorch

```python
def train(self, x, y):
    """
    训练模型并保存参数
    输入:
        model_save_path: saved name of model
        x: 训练数据
        y: 回归真值
    返回:
        losses: 所有迭代中损失函数值
    """
    losses = []
    for epoch in range(self.epoches):
        prediction = self.model(x)
        loss = self.loss_function(prediction, y)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        losses.append(loss.item())

        if epoch % 500 == 0:
            print("epoch: {}, loss is: {}".format(epoch, loss.item()))
```

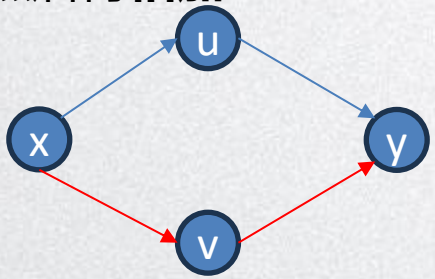
*PyTorch是如何完成自动求导的?

- 数学原理:
  - 多元函数的链式法则
  - $y = f\big(u(x), v(x)\big)$
  - $\frac{dy}{dx} =?$

*PyTorch是如何完成自动求导的?

- 数学原理：
  - 多元函数的链式法则
  - $y = f\big(u(x), v(x)\big)$
  - $y = f(\boldsymbol{z})$: f是一个以2维向量$\boldsymbol{z}$为输入，标量为输出的函数
  - $\boldsymbol{z} = g(x) = \big(u(x), v(x)\big)$: g是一个以标量x为输入， 2维向量$\boldsymbol{z}$为输出的向量值函数
  - 将每一条x->y路径上的偏导数**相乘**，然后再**相加**

$$\frac{dy}{dx} = \left(\frac{\partial y}{\partial z_1}, \frac{\partial y}{\partial z_2}\right) \cdot \begin{pmatrix} \frac{\partial z_1}{\partial x} \\ \frac{\partial z_2}{\partial x} \end{pmatrix} = \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial x} + \frac{\partial y}{\partial v} \cdot \frac{\partial v}{\partial x}$$



PyTorch是如何完成自动求导的?

- 数学原理：
  - 以向量为输入，向量为输出的函数: $\boldsymbol{y} = f(\boldsymbol{x}), \boldsymbol{y} \in R^m, \boldsymbol{x} \in R^n$
  - 雅克比矩阵（Jacobian Matrix）：

$$\boldsymbol{J}_f = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \frac{\partial y_i}{\partial x_j} & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

  - $l = g(\boldsymbol{y}), l \in R$

$$\boldsymbol{v} = \frac{\partial l}{\partial \boldsymbol{y}} = \begin{pmatrix} \frac{\partial l}{\partial y_1} & \cdots & \frac{\partial l}{\partial y_m} \end{pmatrix}$$

$$\boldsymbol{v} \cdot \boldsymbol{J}_f = ?$$

PyTorch是如何完成自动求导的?

- 数学原理:
  - 复合函数: $\boldsymbol{y} = f(\boldsymbol{u}), \boldsymbol{u} = g(\boldsymbol{x}), \boldsymbol{y} \in R^m, \boldsymbol{u} \in R^n, \boldsymbol{x} \in R^p$
  - $l = g(\boldsymbol{y}), l \in R$

$$\boldsymbol{v} = \frac{\partial l}{\partial \boldsymbol{y}} = \begin{pmatrix} \frac{\partial l}{\partial y_1} & \cdots & \frac{\partial l}{\partial y_m} \end{pmatrix}$$

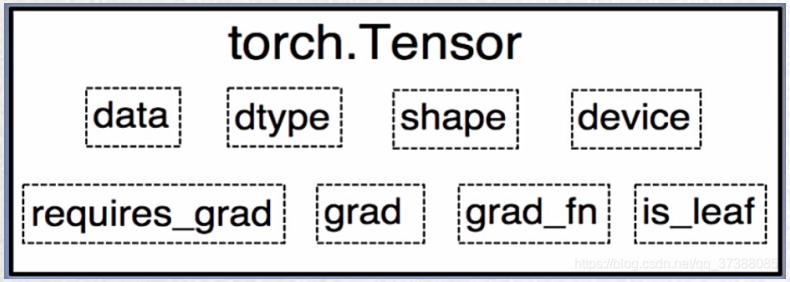
$$\boldsymbol{v} \cdot \boldsymbol{J}_f = \begin{pmatrix} \frac{\partial l}{\partial u_1} & \cdots & \frac{\partial l}{\partial u_n} \end{pmatrix} = \frac{\partial l}{\partial \boldsymbol{u}}$$

$$\left(\boldsymbol{v} \cdot \boldsymbol{J}_f\right) \cdot \boldsymbol{J}_g = \begin{pmatrix} \frac{\partial l}{\partial x_1} & \cdots & \frac{\partial l}{\partial x_p} \end{pmatrix} = \frac{\partial l}{\partial \boldsymbol{x}}$$

实现方法：计算图

- Tensor的属性中，4个与数据相关，4个与梯度求导相关

- Pytorch是如何实现自动求导的?
- 深度学习和PyTorch中，将深度神经网络看作计算图
- 节点表示数据
  - Tensors；Parameters
- 边表示计算
  - 加减乘除
  - 卷积、池化

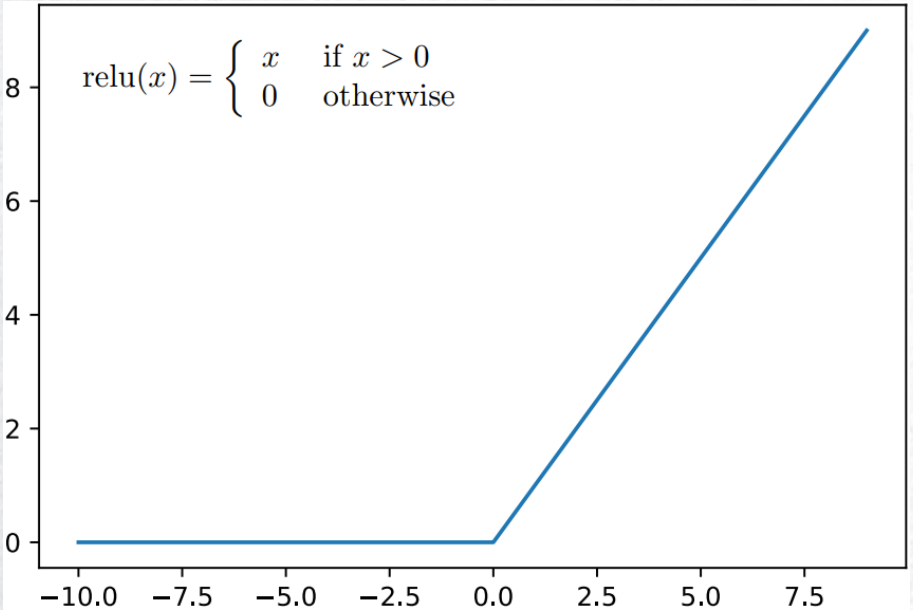


计算图

- 例如：对激活函数

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

- ReLU = Rectified Linear Unit
  - 可能是现在神经网络中最常用的激活函数
  - 原因（之一）：数值过大或者过小，sigmoid，tanh的导数接近于0
  - 练习：计算Relu函数的梯度



关于Relu函数

- Relu函数求导
  - DNE：does not exist
  - 在0处不可导
  - 为了方便，令其导数为：

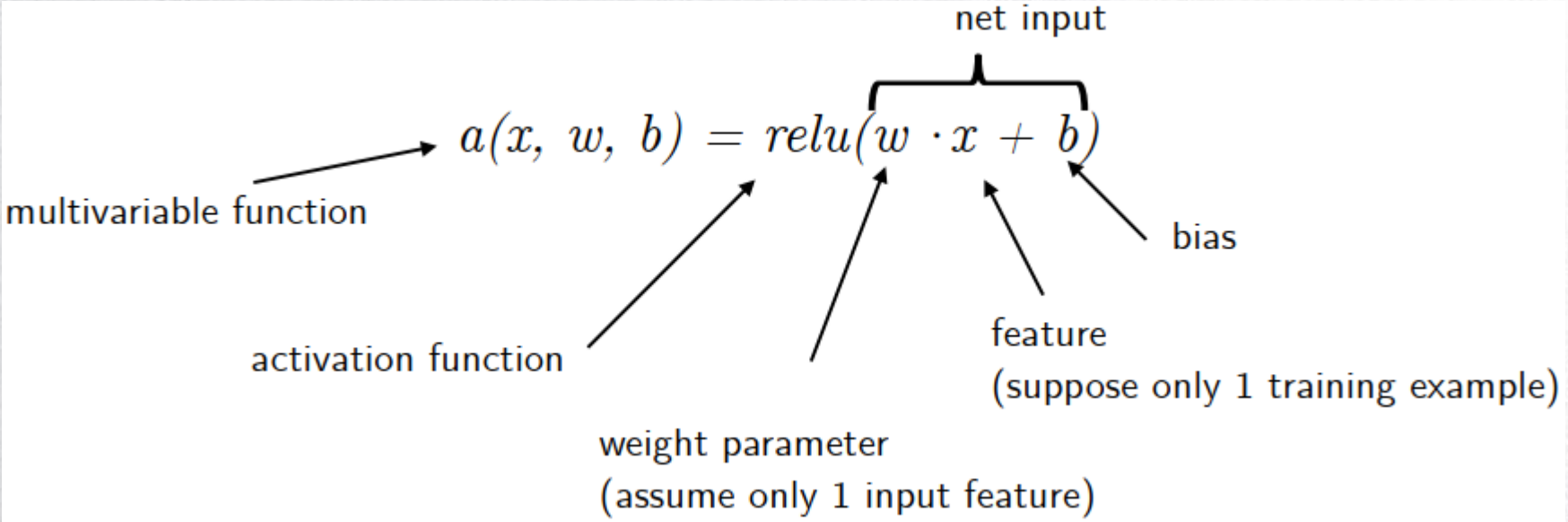
$$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x > 0 \\ \text{DNE} & \text{if } x = 0 \end{cases}$$

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

- Relu函数其它优点
  - 防止梯度消失
  - 梯度简单易计算，使网络训练更快
  - 增加网络的非线性
  - 使网络具有稀疏性，减少过拟合

计算图

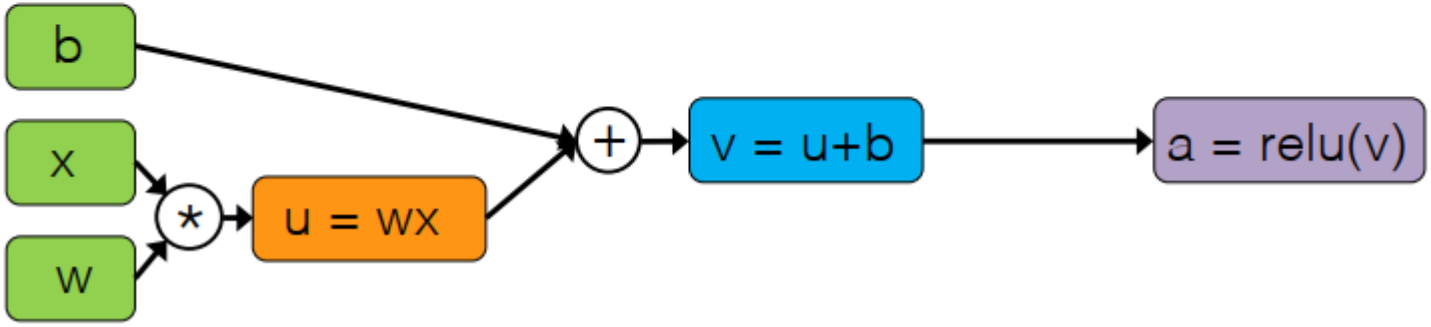
- 单层神经网络输出后接Relu激活函数



计算图

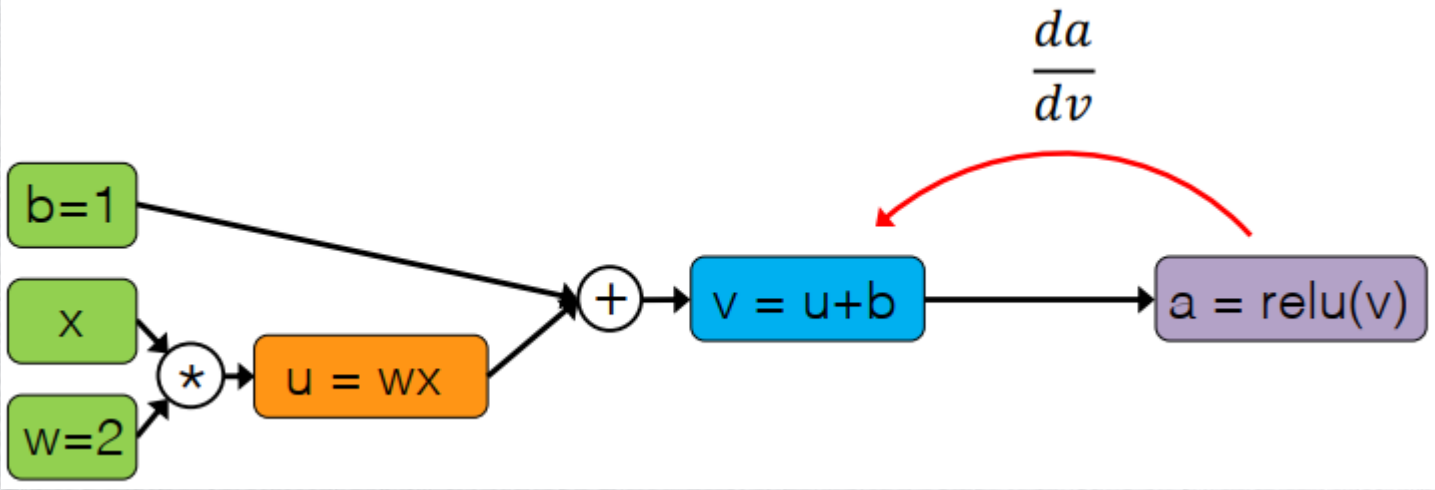
- 定义了如下计算图
- 前向计算：x数据；w、b参数

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



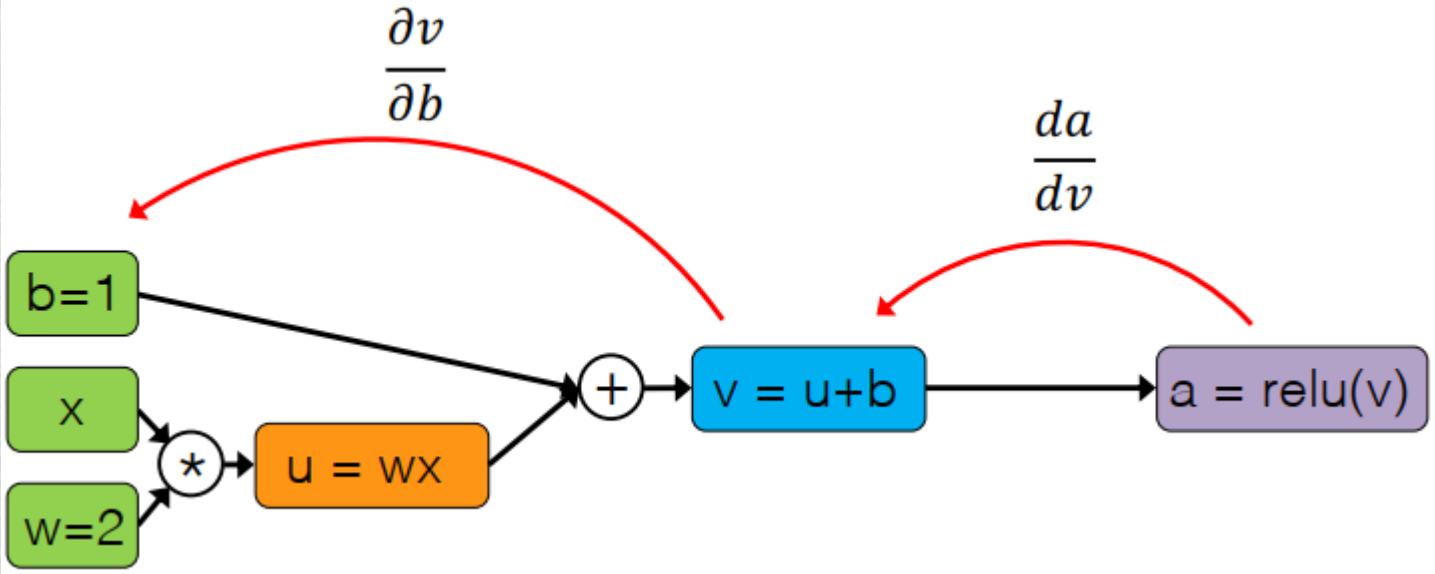
计算图

- 深度学习中，我们关心损失函数对参数的导数
- 用链式法则计算激活值对参数的损失
- 反向传播



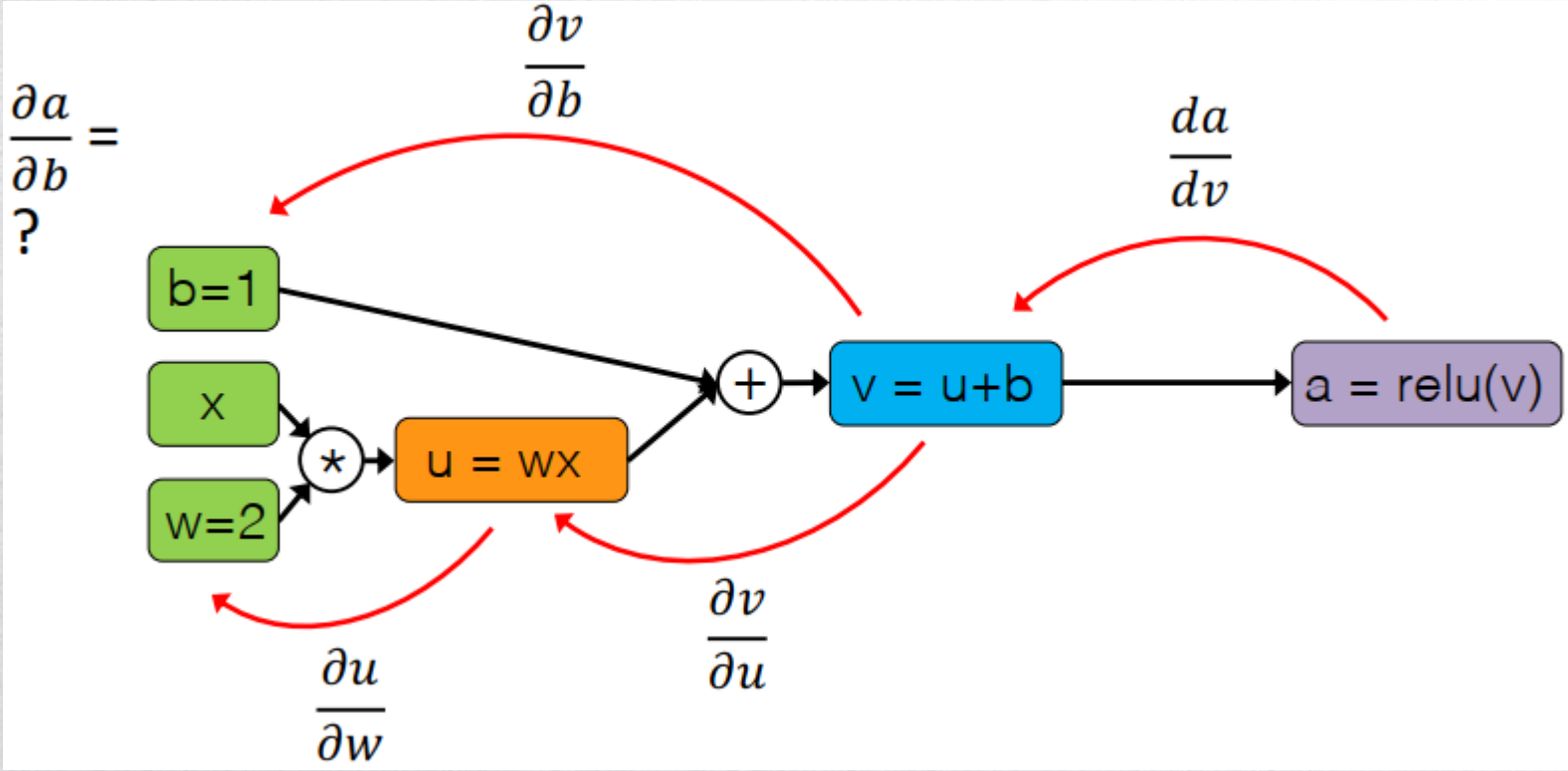
计算图

- 反向传播
  - 一般将用户自己创建的变量是叶子节点，而由叶子节点计算得到的变量是非叶子节点。调用非叶子节点的backward方法，就会沿着非叶子节点一直回溯到叶子节点结束。



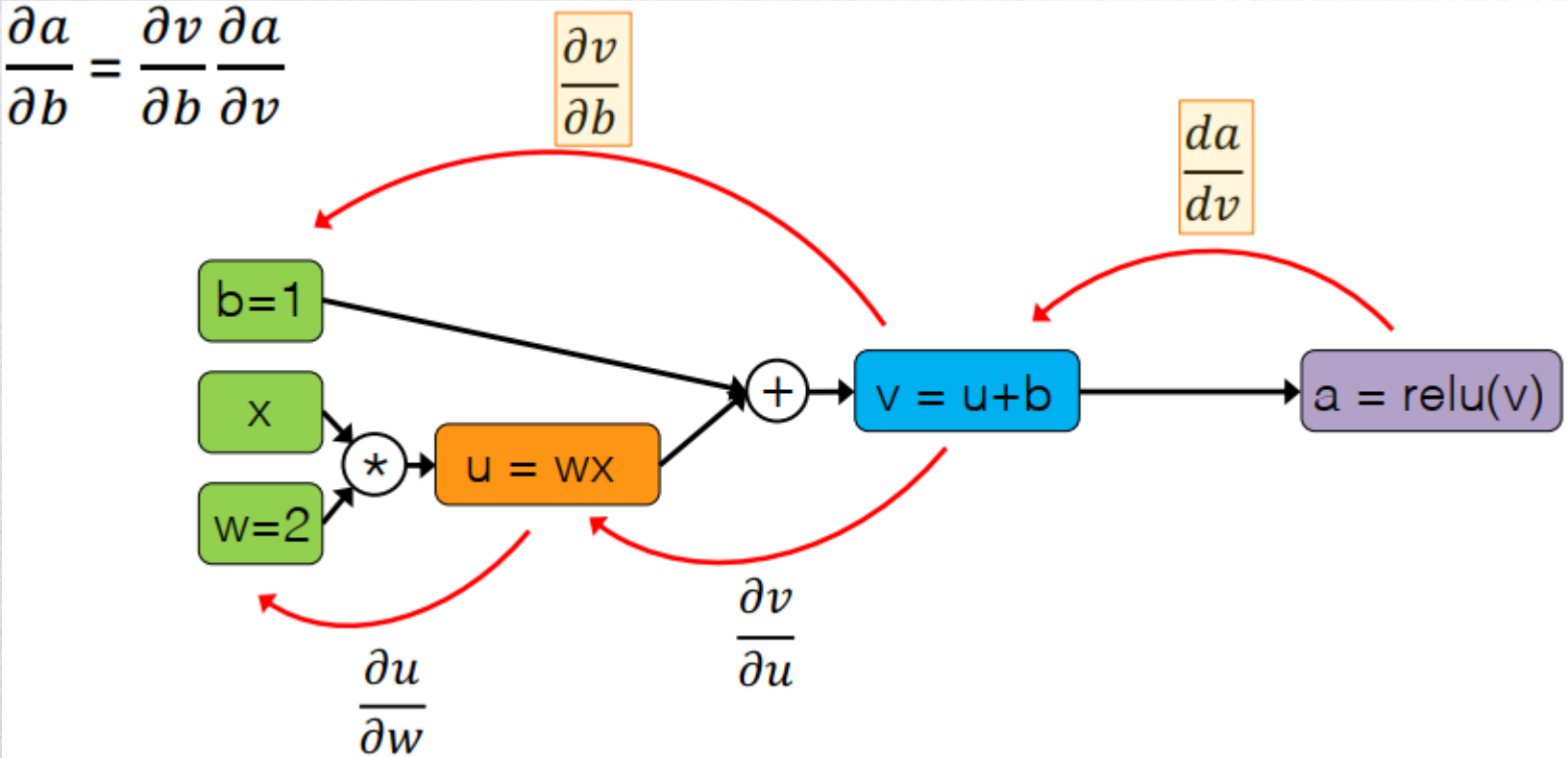
计算图

- 反向传播


$\frac{\partial a}{\partial b}$ =
?
$\frac{\partial v}{\partial b}$
$\frac{da}{dv}$
b=1
x
w=2
*
u = wx
+
v = u+b
a = relu(v)
$\frac{\partial u}{\partial w}$
$\frac{\partial v}{\partial u}$

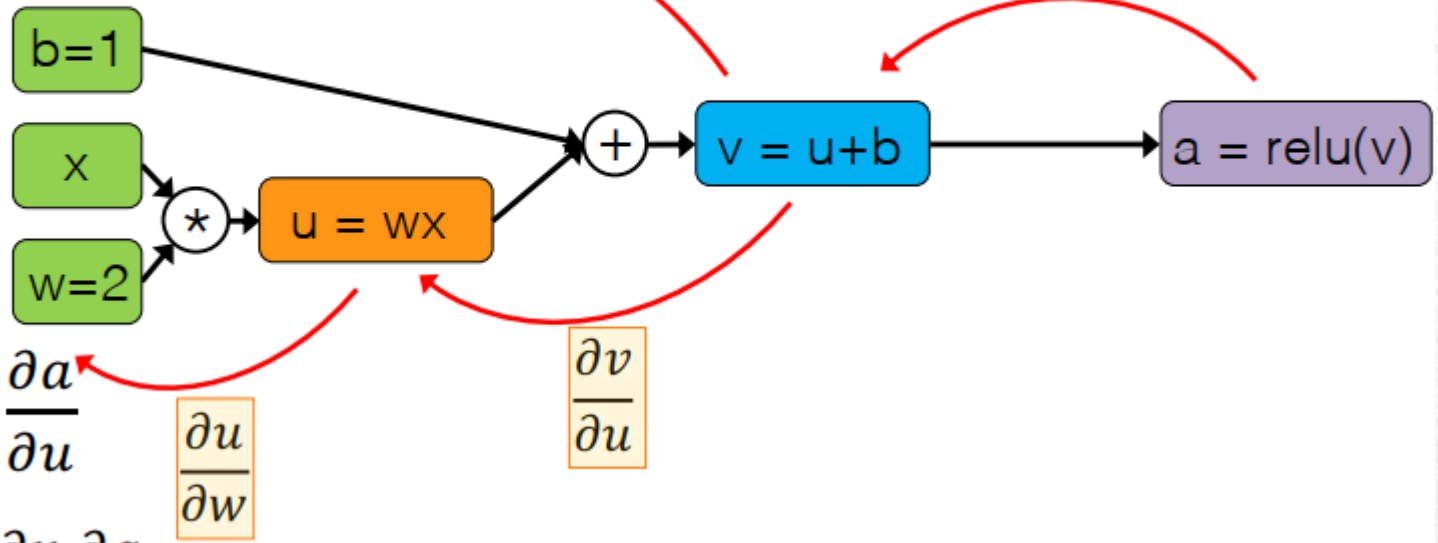
计算图

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



计算图

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



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

$\frac{da}{dv}$

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

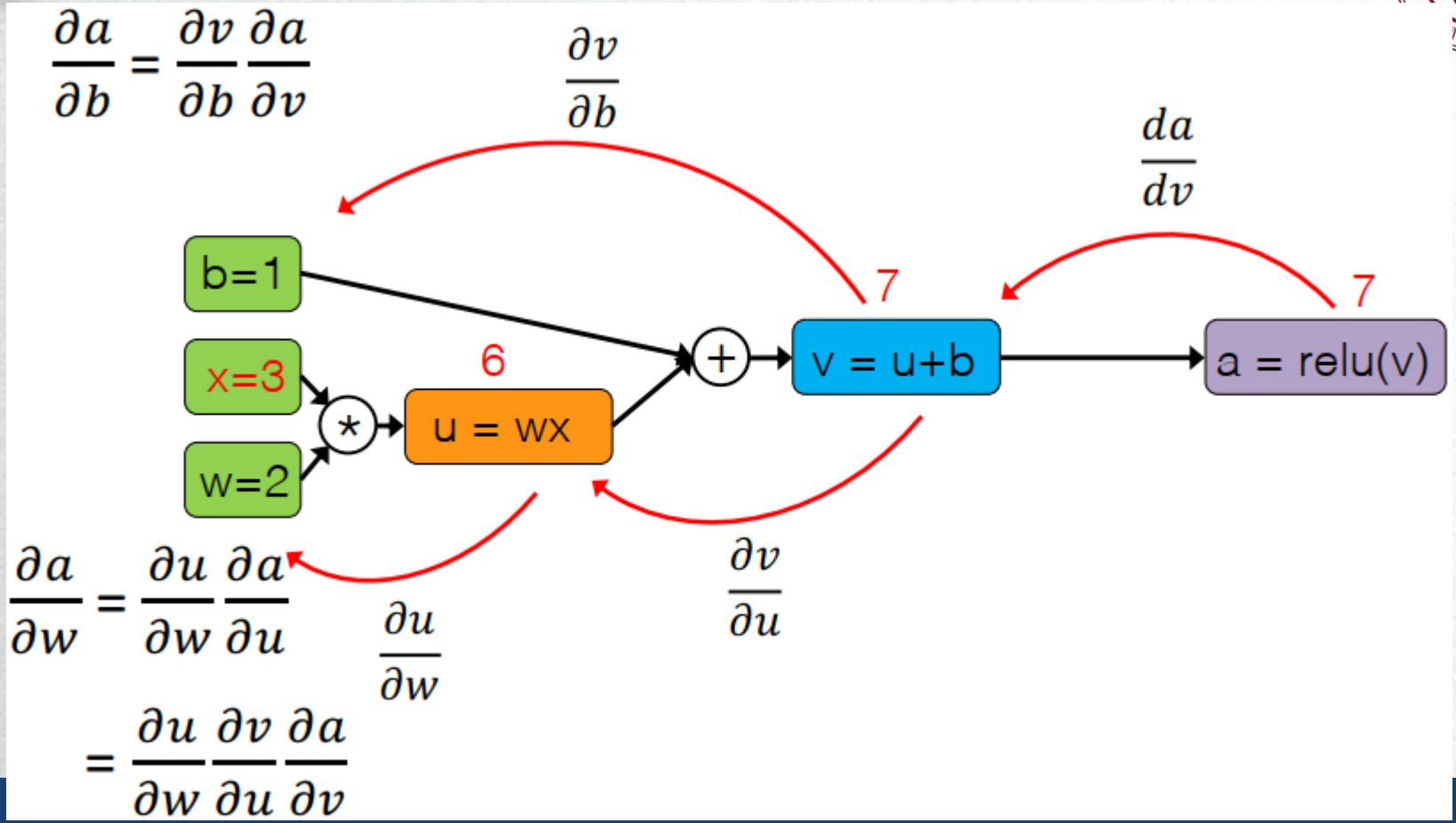
$\frac{\partial u}{\partial w}$

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

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

计算图

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

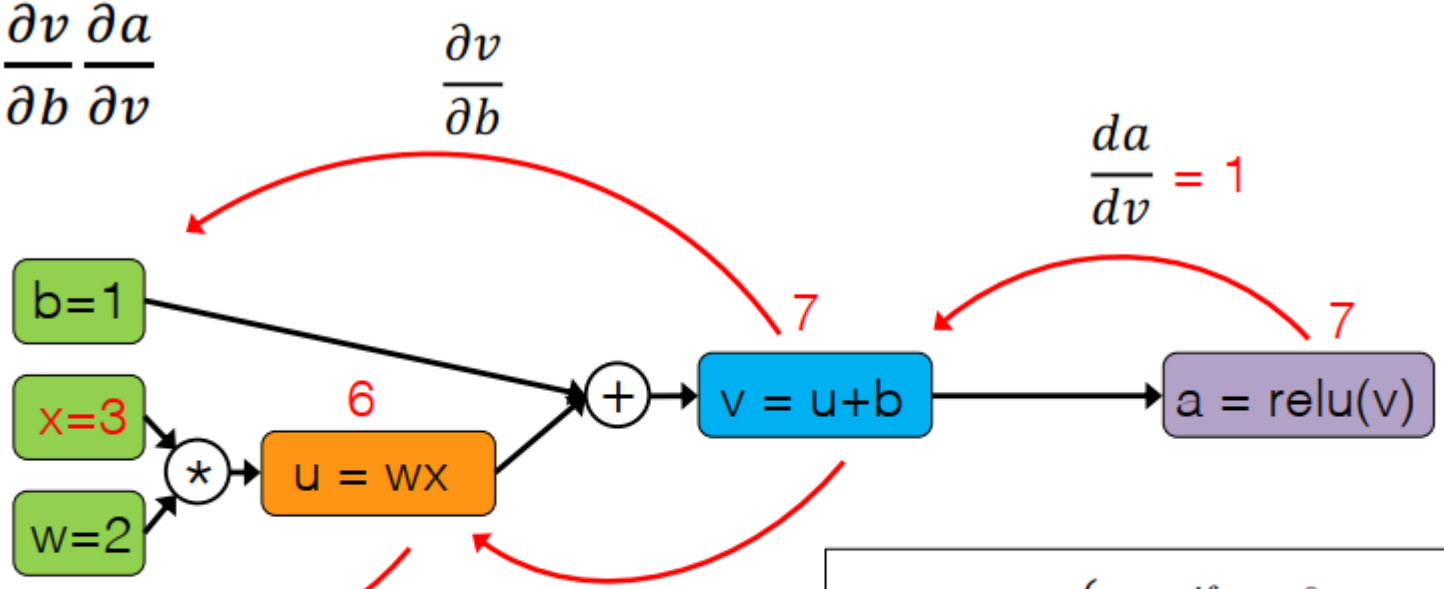


$$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w}\frac{\partial a}{\partial u}$$

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

计算图

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



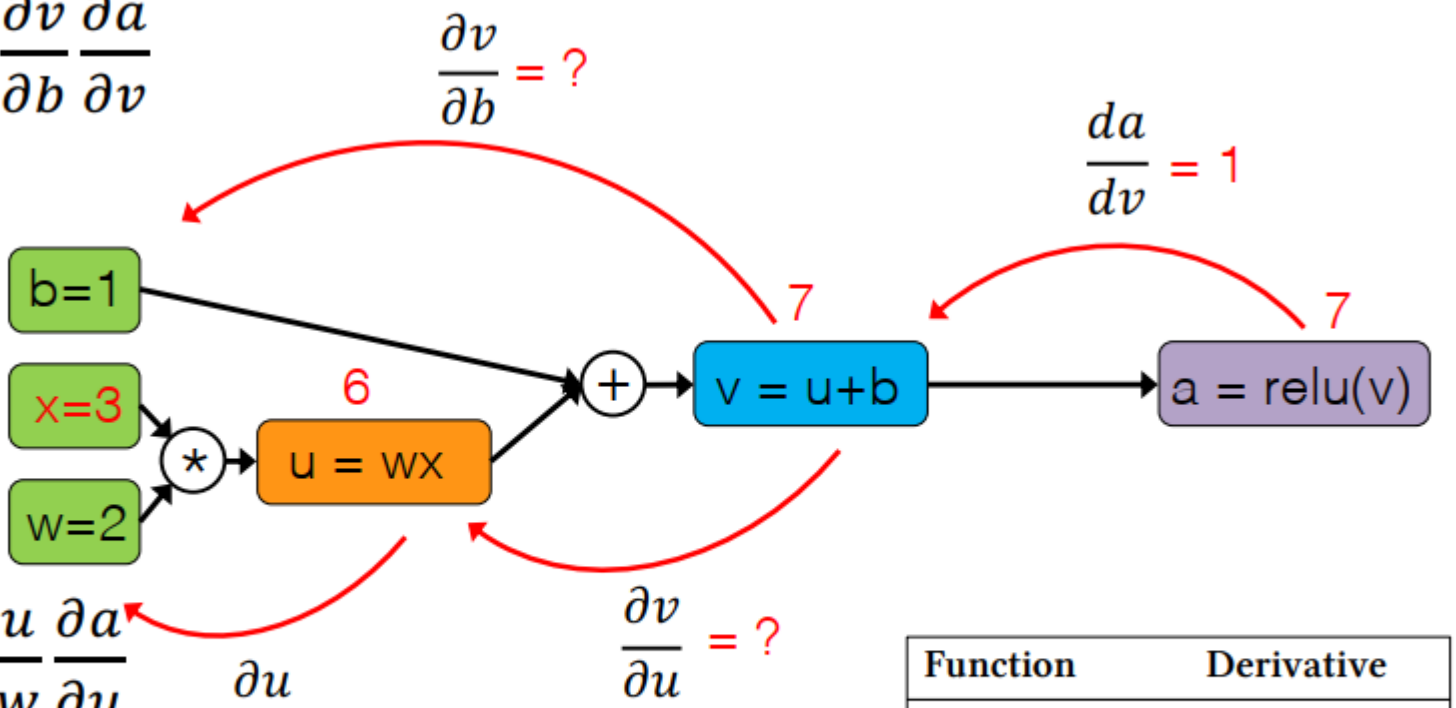
$$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w}\frac{\partial a}{\partial u}$$

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

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

计算图

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

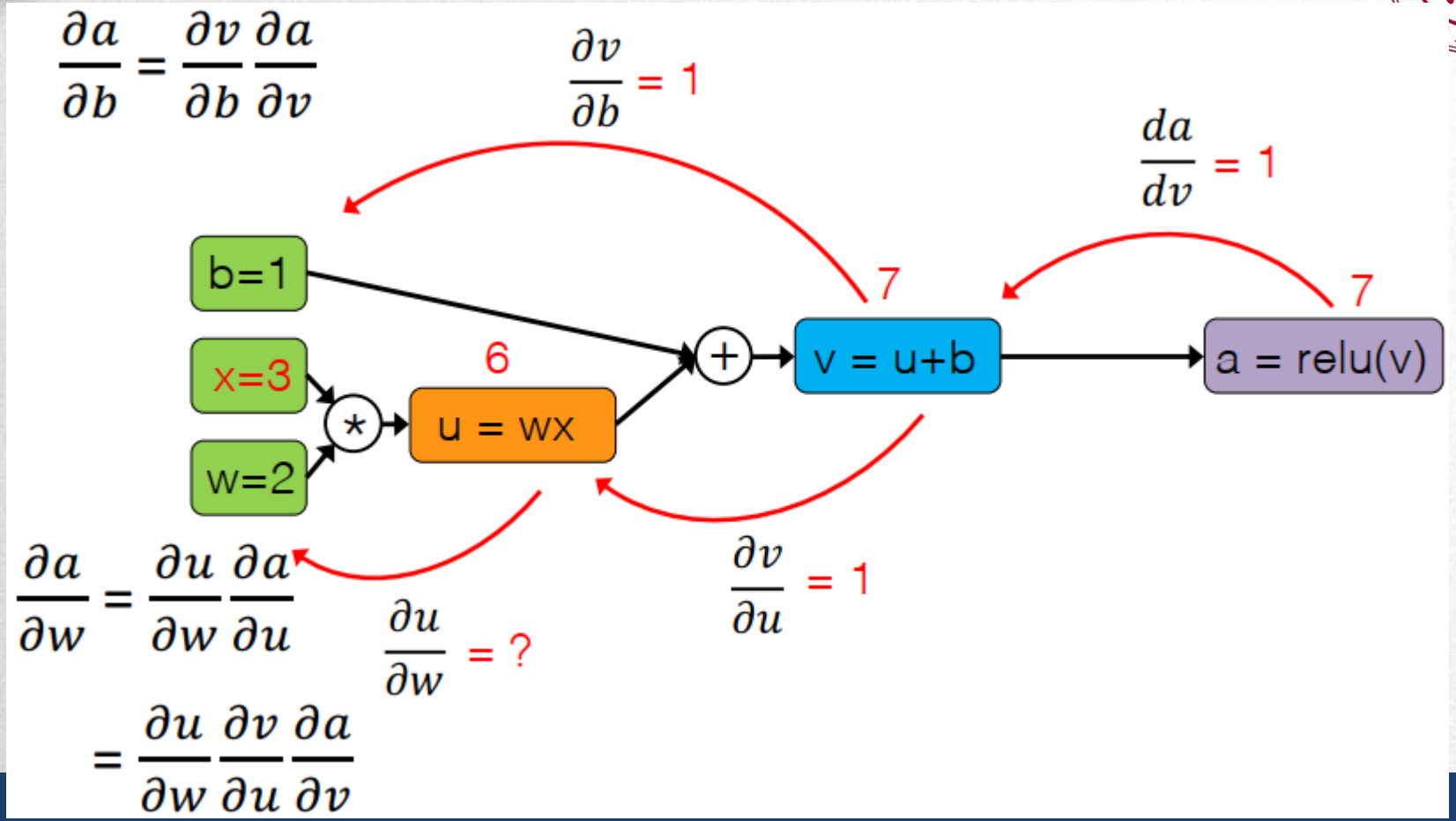


$$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w}\frac{\partial a}{\partial u}$$

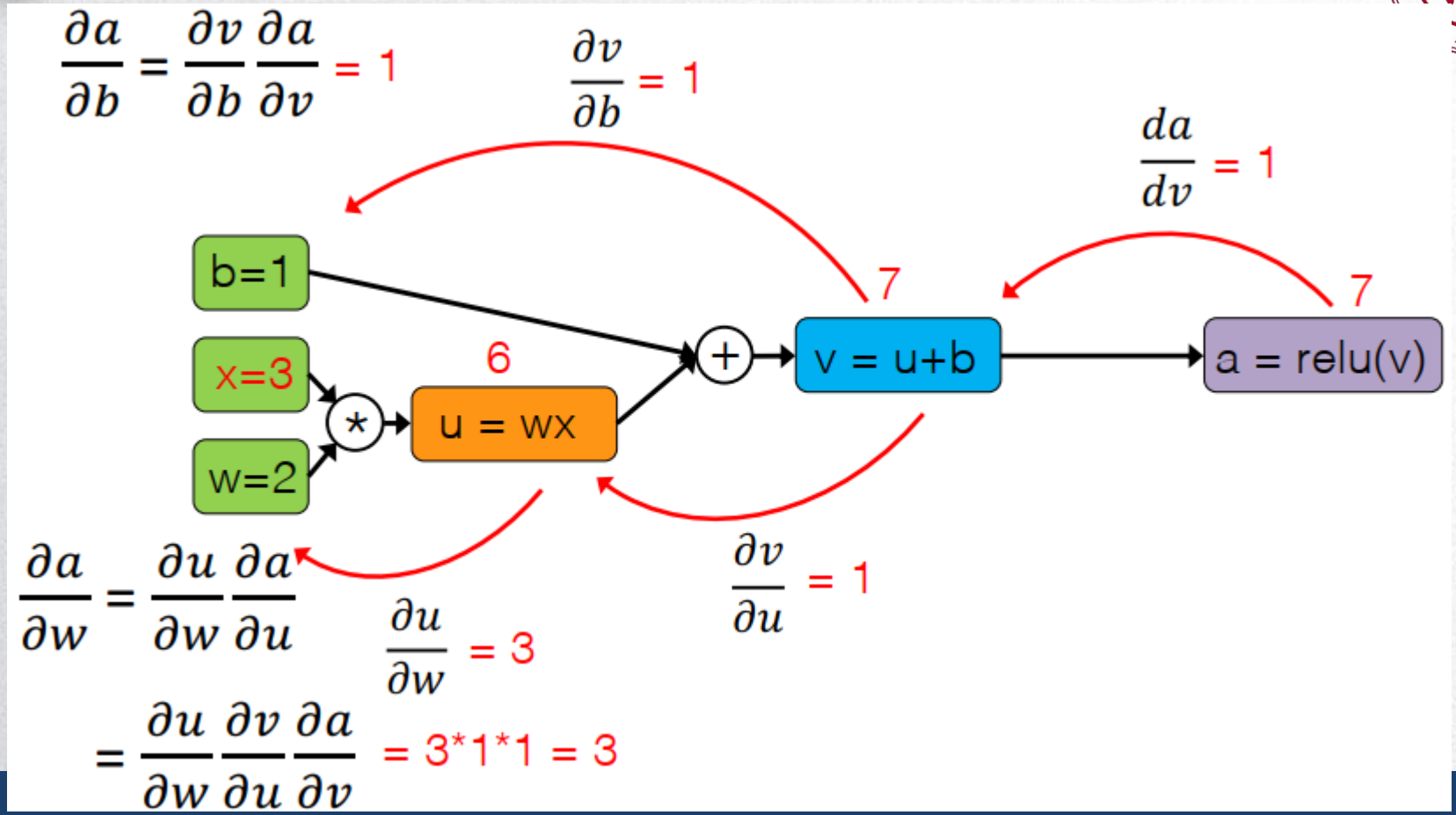
$$= \frac{\partial u}{\partial w}\frac{\partial v}{\partial u}\frac{\partial a}{\partial v}$$

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

计算图


$\frac{\partial a}{\partial b} = \frac{\partial v}{\partial b}\frac{\partial a}{\partial v}$
$\frac{\partial v}{\partial b} = 1$
$\frac{da}{dv} = 1$
b=1
x=3
w=2
*
6
u = wx
+
7
v = u+b
7
a = relu(v)
$\frac{\partial v}{\partial u} = 1$
$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w}\frac{\partial a}{\partial u}$
$\frac{\partial u}{\partial w} = ?$
$= \frac{\partial u}{\partial w}\frac{\partial v}{\partial u}\frac{\partial a}{\partial v}$

计算图


$\frac{\partial a}{\partial b} = \frac{\partial v}{\partial b}\frac{\partial a}{\partial v} = 1$
$\frac{\partial v}{\partial b} = 1$
$\frac{da}{dv} = 1$
b=1
x=3
w=2
*
6
u = wx
+
7
v = u+b
7
a = relu(v)
$\frac{\partial v}{\partial u} = 1$
$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w}\frac{\partial a}{\partial u}$
$\frac{\partial u}{\partial w} = 3$
$= \frac{\partial u}{\partial w}\frac{\partial v}{\partial u}\frac{\partial a}{\partial v} = 3*1*1 = 3$

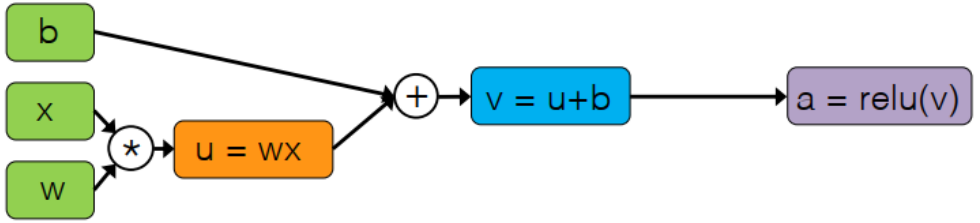
计算图

- y对w求导就是在计算图中找到所有y到w的路径，把路径上的导数进行求和
- 采用计算图来描述运算的好处不仅仅是让运算更加简洁，更重要的作用是使梯度求导更加方便
  - 叶子节点是整个计算图的根基，例如前面求导的计算图，在前向计算中的a、o和l都要依据创建的叶子节点x和w进行计算的。同样，在反向传播过程中，所有梯度的计算都要依赖叶子节点。
  - 设置叶子节点主要是为了节省内存，在梯度反向传播结束之后，非叶子节点的梯度都会被释放掉。可以根据代码分析一下非叶子节点a、b和y的梯度情况。
  - 如果想使用非叶子结点梯度，可以使用pytorch中的retain_grad()。

计算图

- Tensor中属性grad_fn的作用是记录创建该张量时所用的方法（函数）
- u.grad_fn = MulBackward0 object
  - u在反向传播的时候会记录u是用乘法得到的，所用在求解w和x的梯度的时候就会用到乘法的求导法则去求解w和x的梯度。
- 对于v有v.grad_fn= AddBackword0 object，v是通过加法得到的
- 可以通过代码查看tensor的属性
- v.grad_fn

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



计算图

```python
import torch
import torch.nn as nn

x = torch.tensor(3.0, requires_grad=False)
w = torch.tensor(2.0, requires_grad=True)
b = torch.tensor(1.0, requires_grad=True)

u = x*w
v = u + b
F = nn.ReLU(True)
a = F(v)
a.backward()

print(w.grad)
print(b.grad)
```

```
tensor(3.)
tensor(1.)
```

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