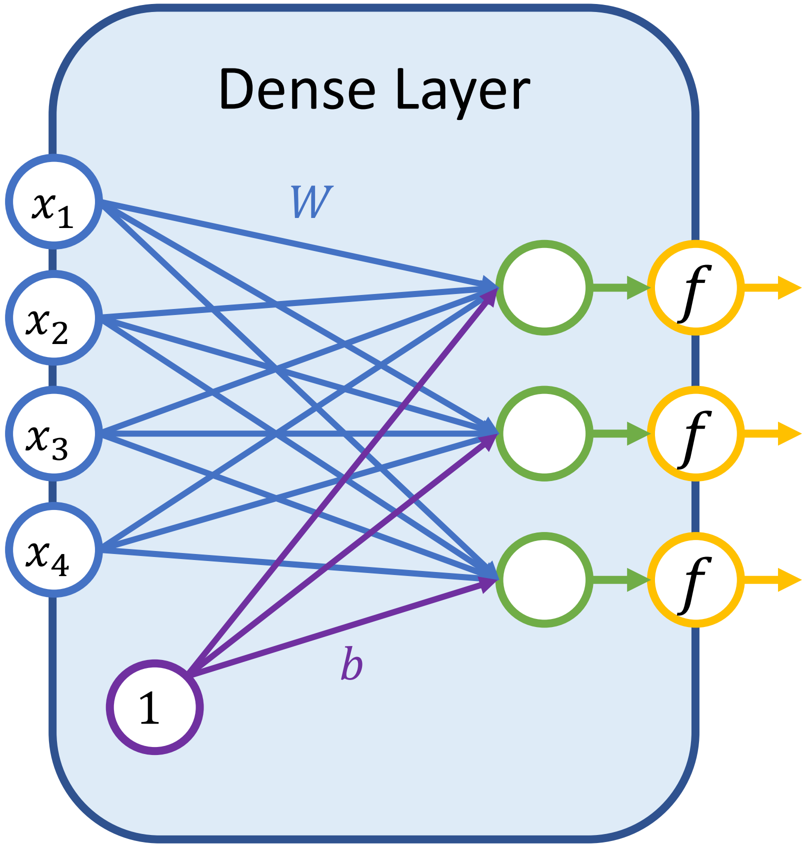
Deep Neural Networks
and Where to Find Them

Lecture 2

Artem Korenev, Nikita Gryaznov

Recap

Recap



x_1 Layer input

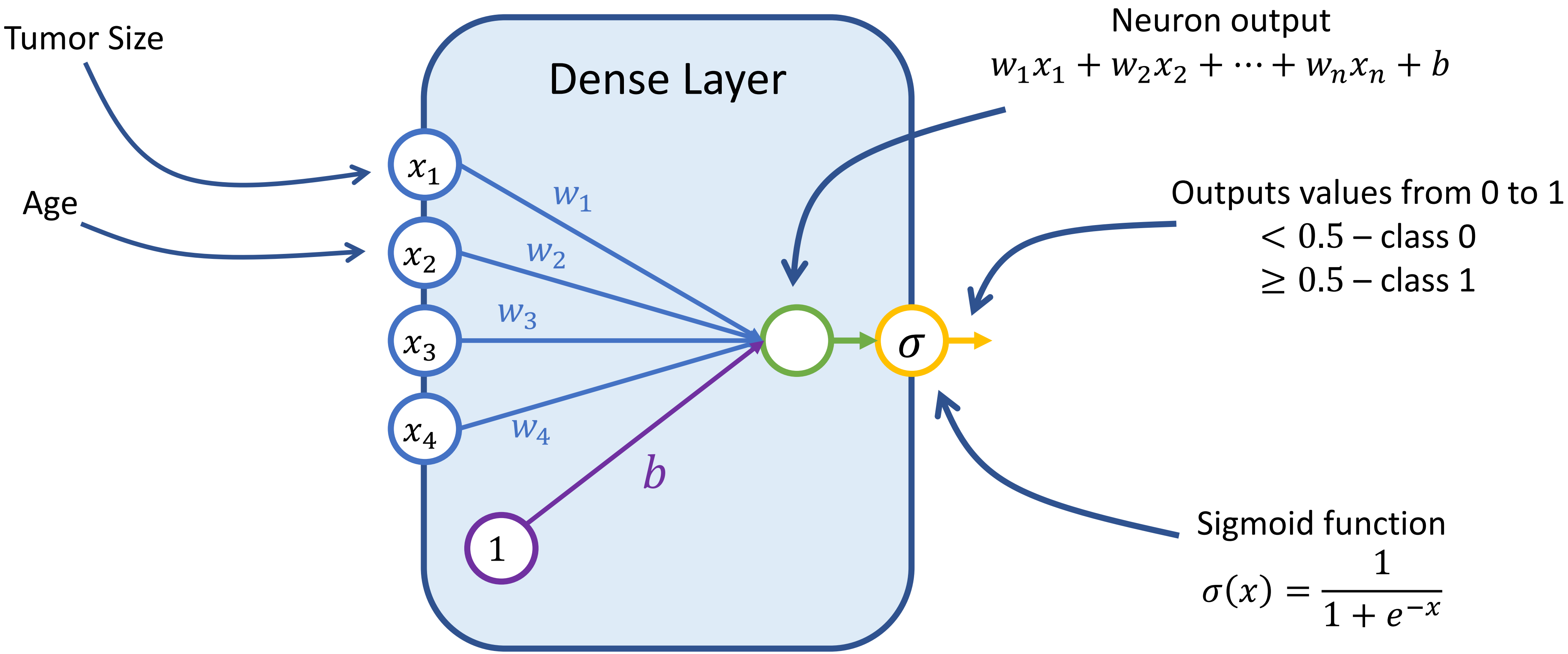
Neuron Output
$(W_{i1}x_1 + \cdots + W_{i2}x_2 + b_i)$

f Activation function
(Non-linearity)
(Neuron activation)

Layer weight
(trainable parameter)

Layer bias
(trainable parameter)

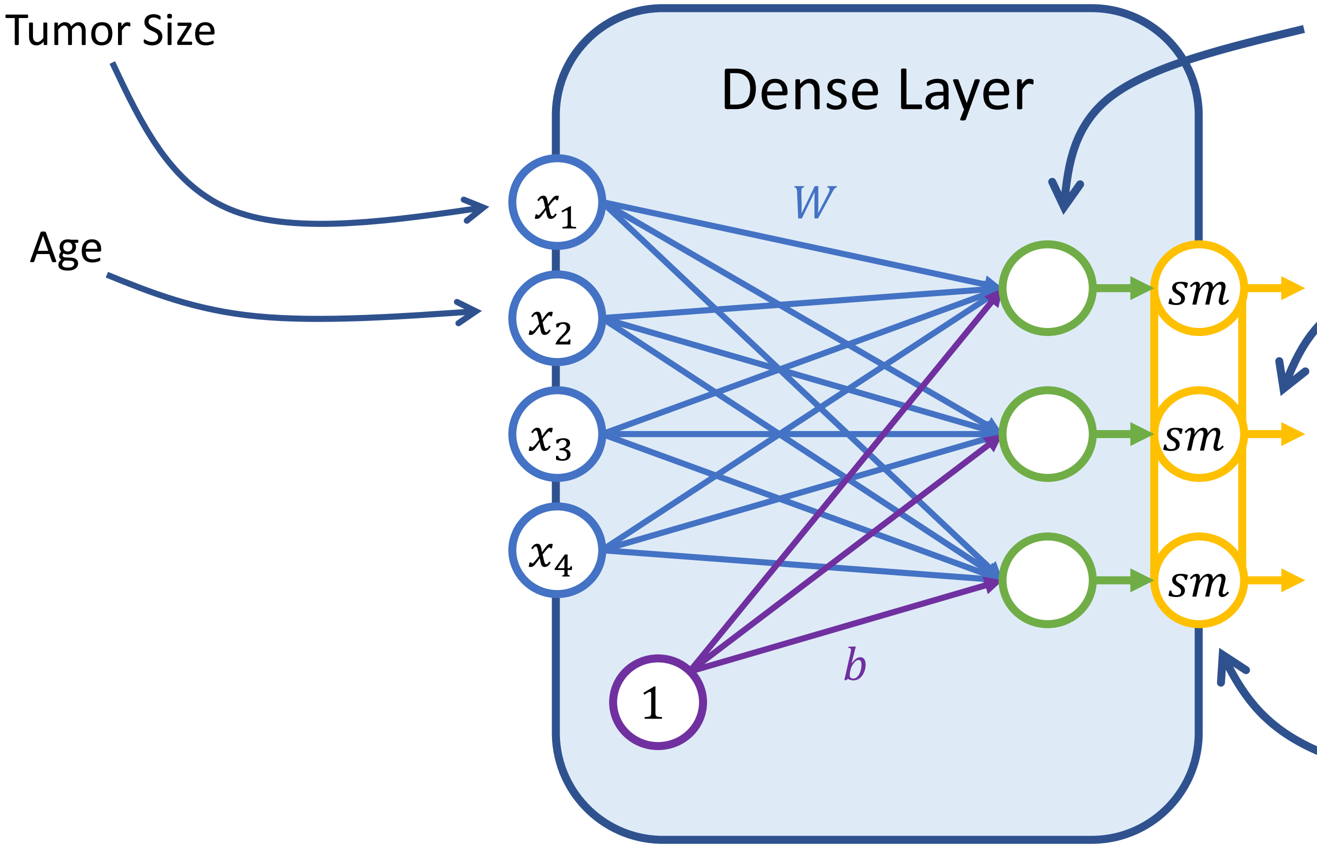
Binary Logistic Regression



Multiclass Logistic Regression

Neuron output

$W_{11}x_1 + W_{12}x_2 + \cdots + W_{1n}x_n + b_1$



Outputs values from 0 to 1

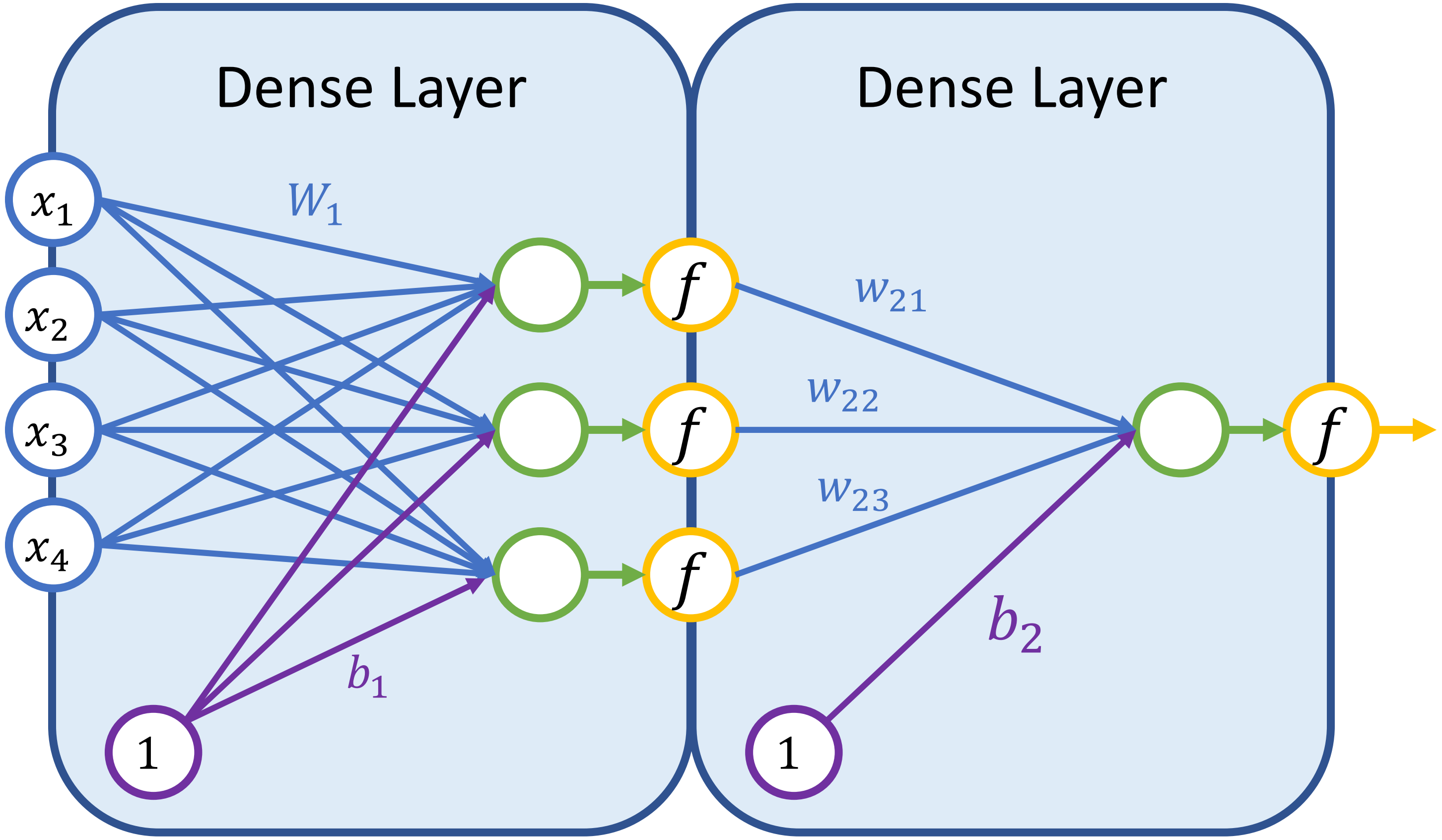
Decision made by

the biggest value in the output

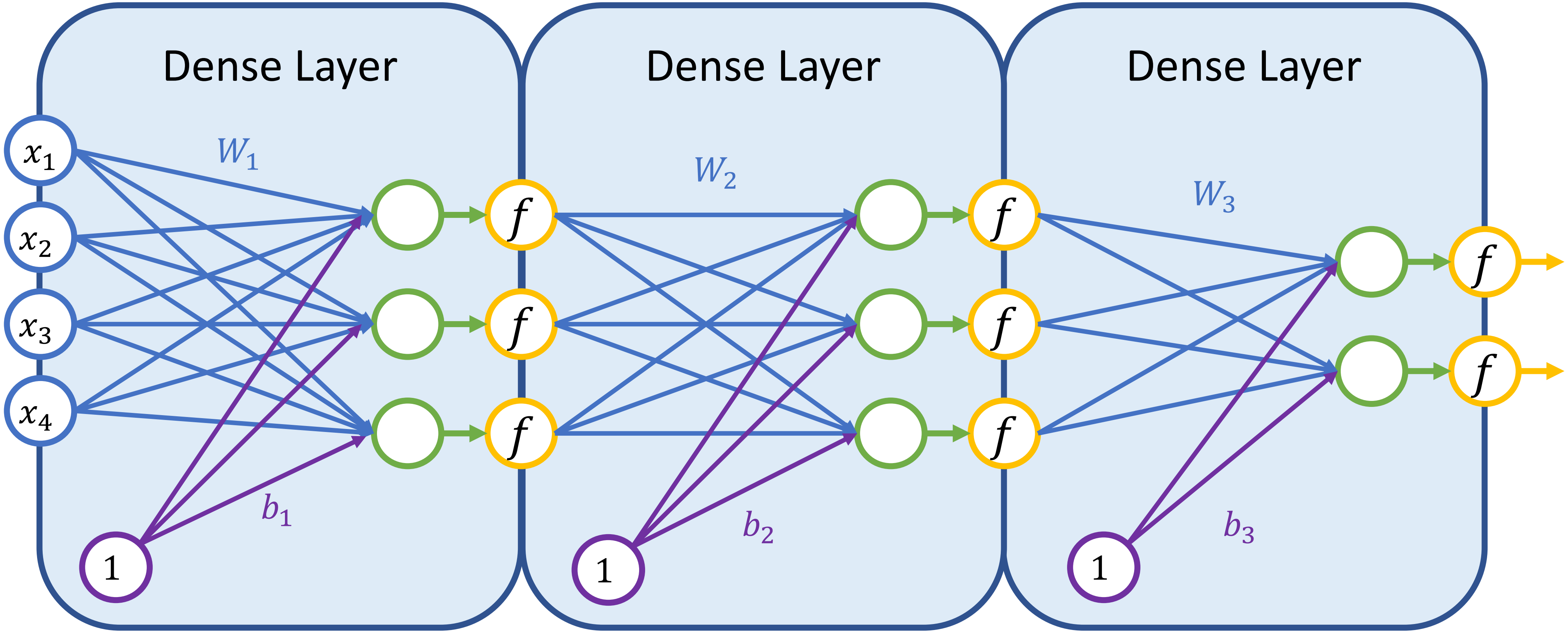
Softmax function

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}}$$

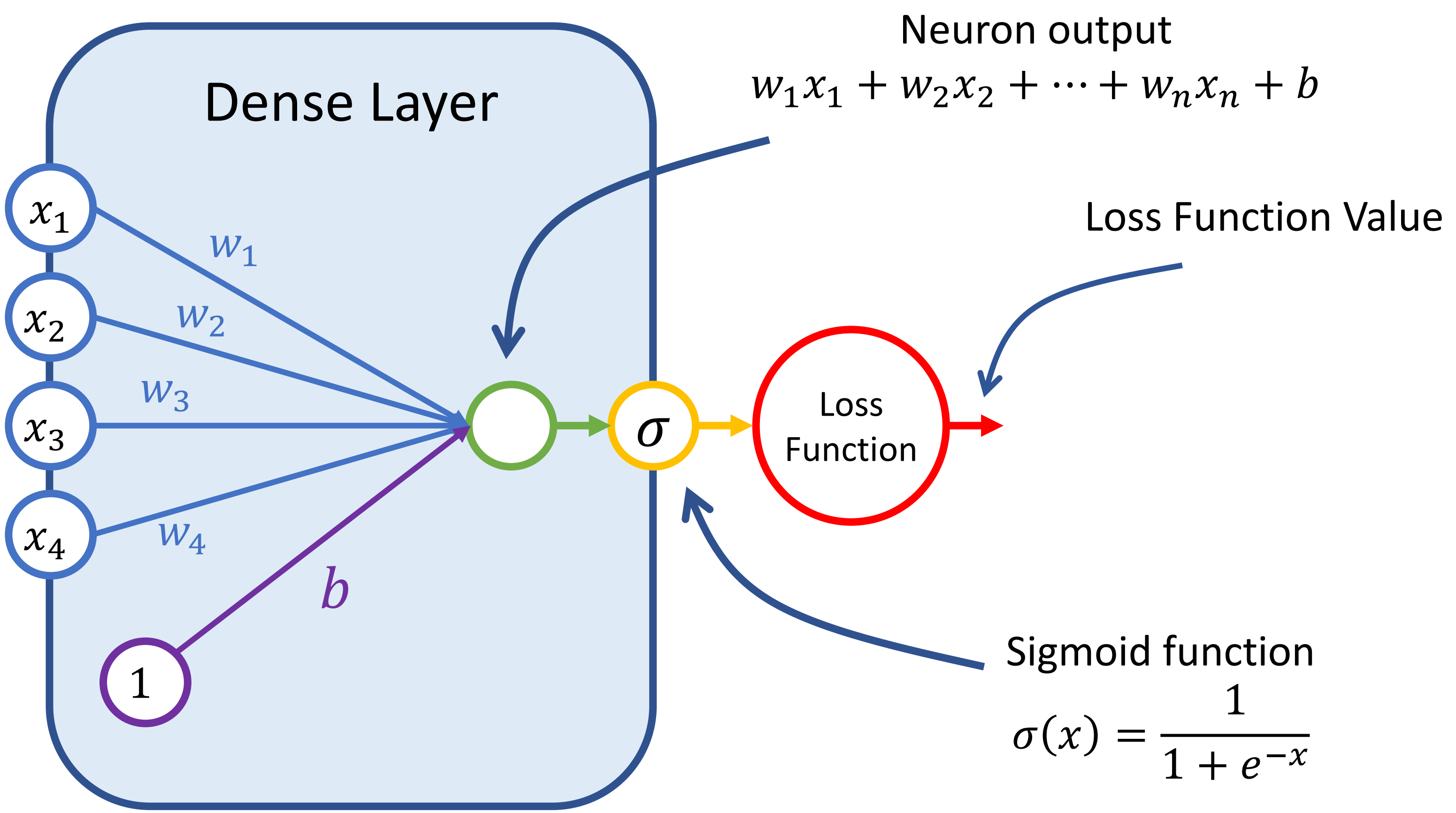
2 Layer NN with one output



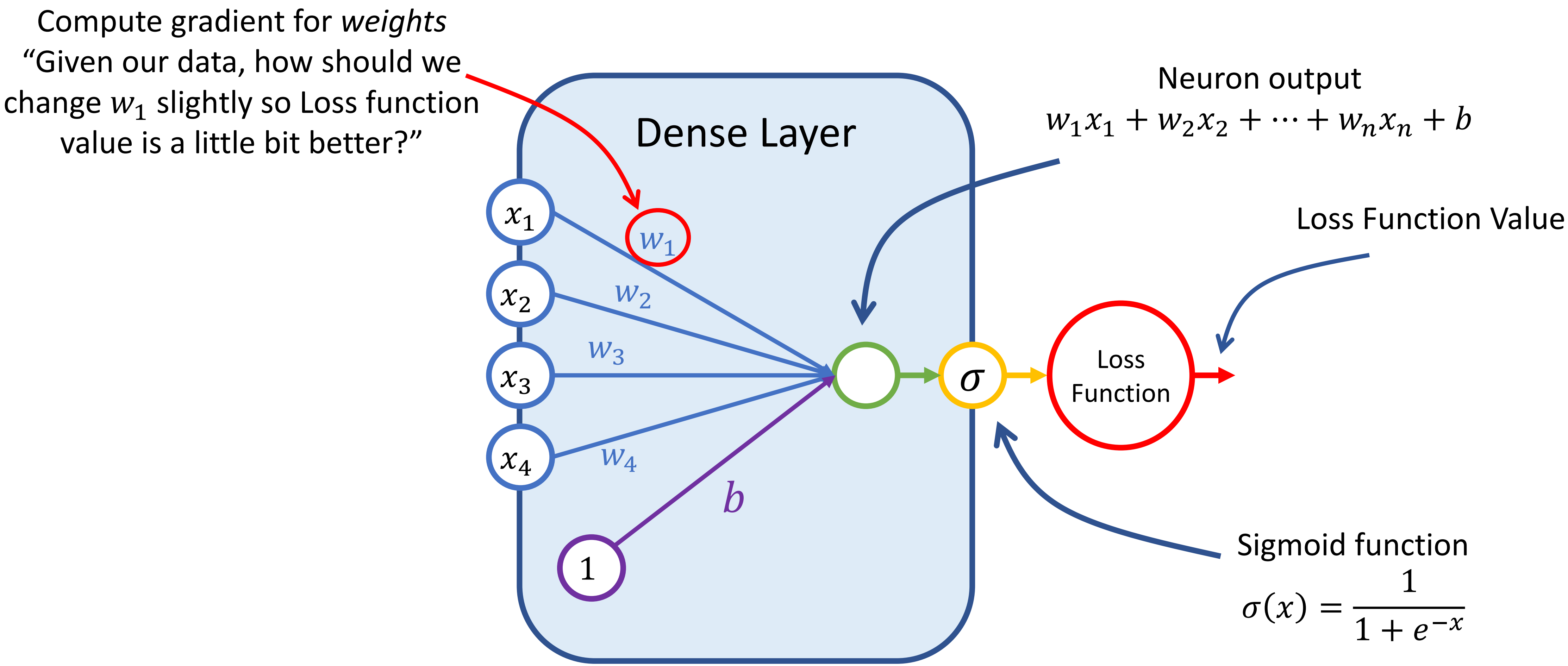
3 Layer NN with two outputs



Logistic Regression Optimization



Logistic Regression Optimization

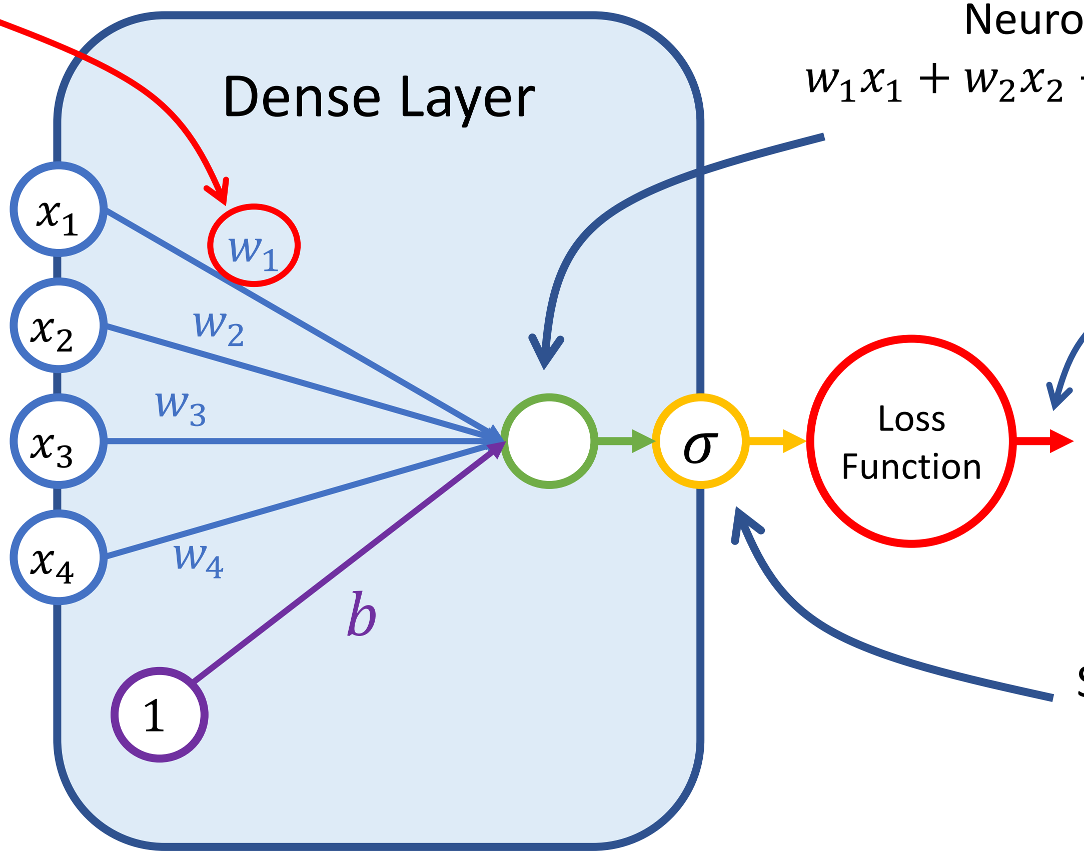


Logistic Regression Optimization

Compute gradient for *weights*
"Given our data, how should we change w_1 slightly so Loss function value is a little bit better?"

Then we update all *weights* slightly towards the gradient hoping that loss function value will improve

$$w_i \coloneqq w_i - \alpha L'\big(y, \sigma(\ldots)\big)_{w_i}$$



Neuron output
$w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b$

Loss Function Value
$L(y, \sigma(\ldots))$

Sigmoid function
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

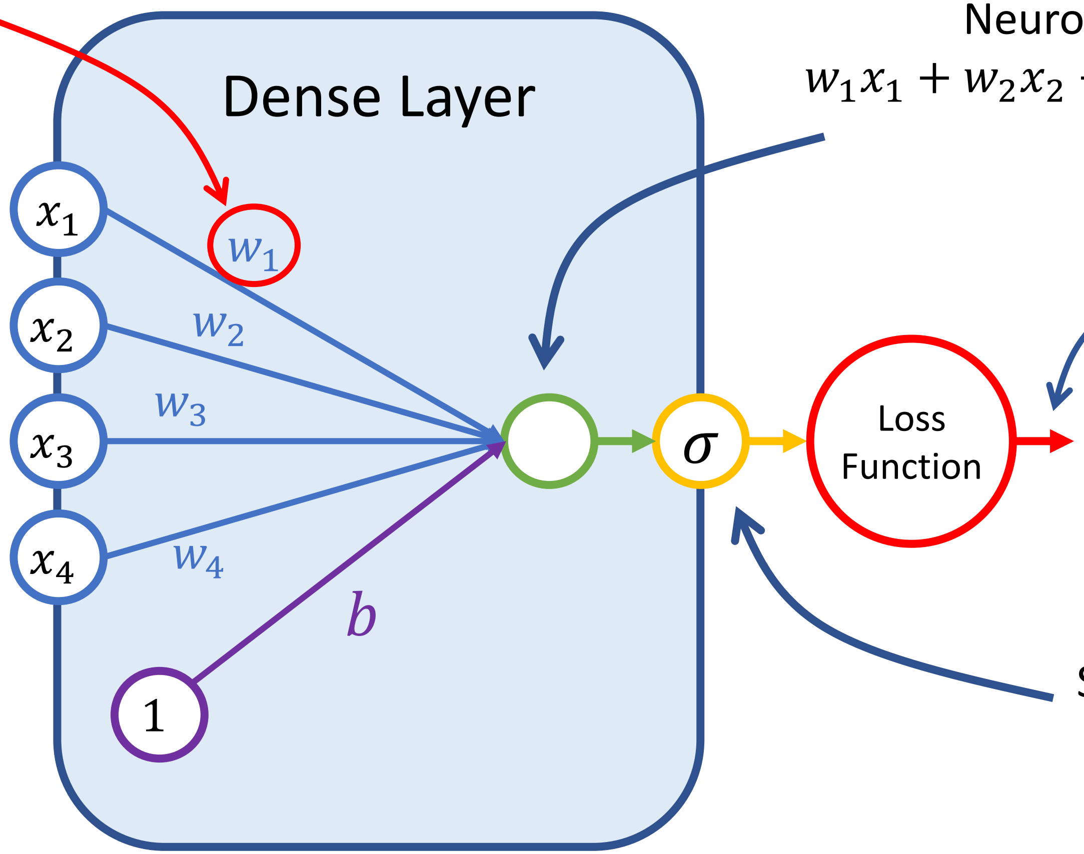
Logistic Regression Optimization

Compute gradient for *weights*
"Given our data, how should we change w_1 slightly so Loss function value is a little bit better?"

Then we update all *weights* slightly towards the gradient hoping that loss function value will improve

$$w_i \coloneqq w_i - \alpha L'\big(y, \sigma(\ldots)\big)_{w_i}$$

Repeat calculating loss, computing gradients and updating the *weights* *(gradient descent)*



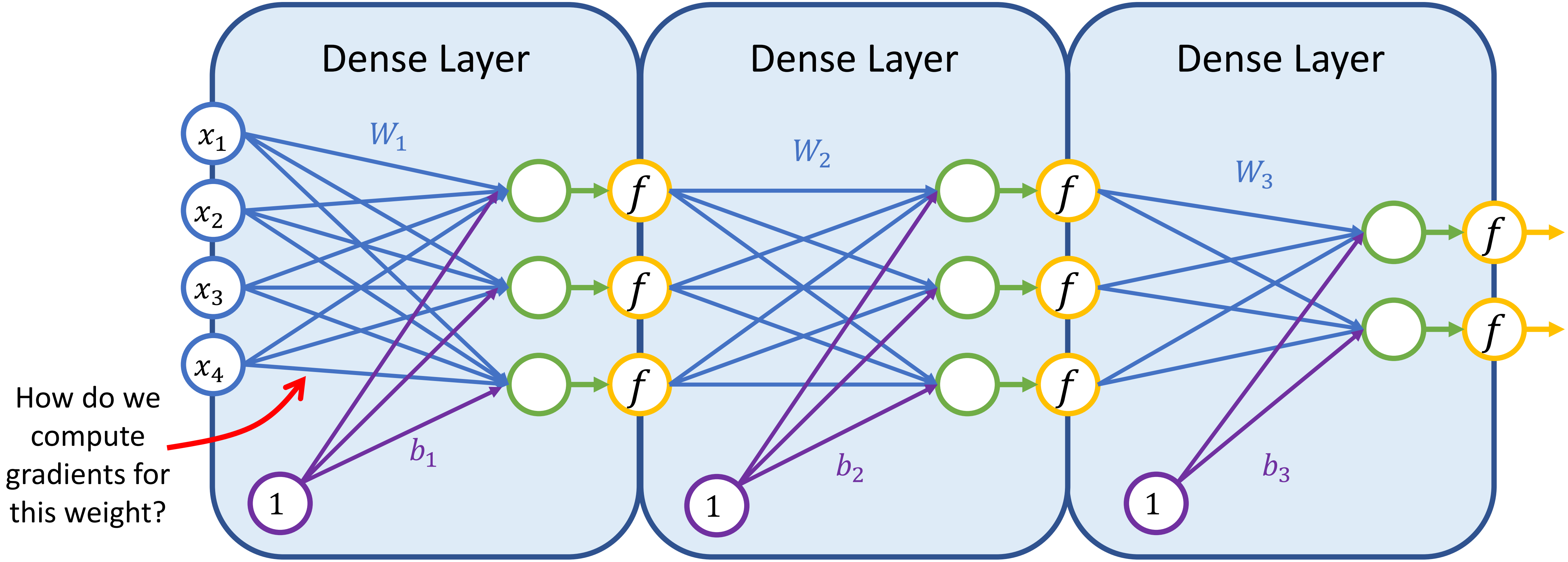
Neuron output
$w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$

Loss Function Value
$L(y, \sigma(\ldots))$

Sigmoid function
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

NN Optimization. Chain Rule

Optimizing Complex Networks



How to update weights?

- We need to compute gradients for deeper layers
- We can do it viewing our neural network as a *computational graph* and using *the chain rule*
- Computing gradients and optimizing the neural network called *backpropagation*

Chain rule

$$L = f\big(g(x)\big)$$

$$\frac{dL}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

$$L = \log(x^2)$$

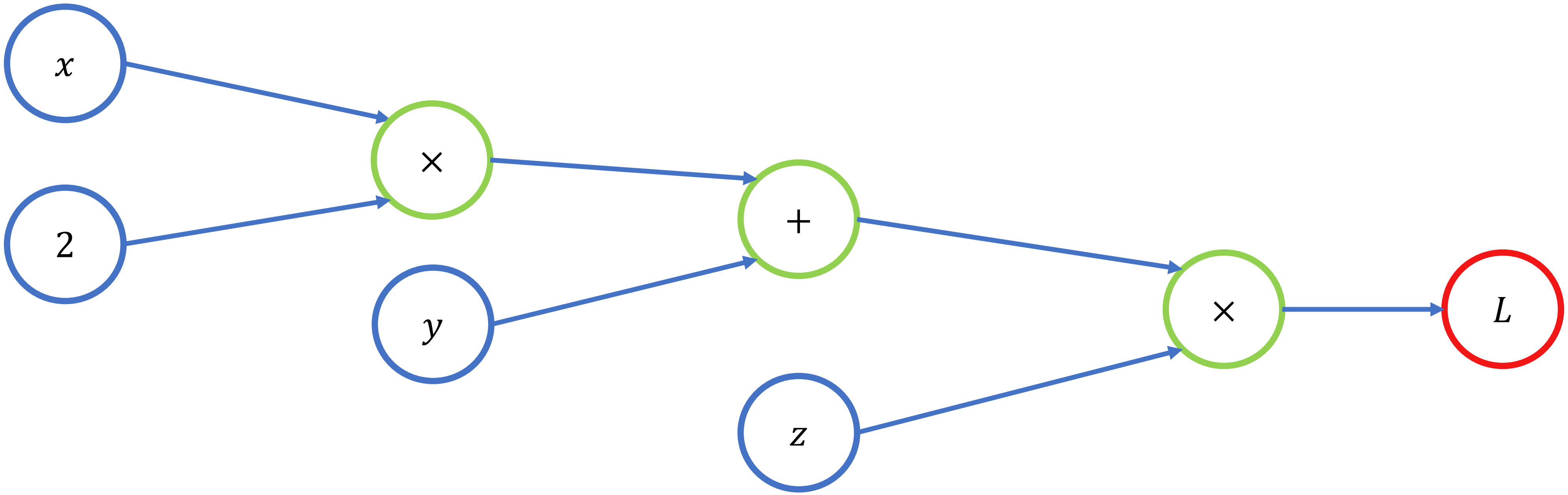
$$f(x) = \log(x) \qquad g(x) = x^2$$

$$\frac{dL}{dx} = \frac{1}{x^2} \times 2x = \frac{2}{x}$$

Chain rule

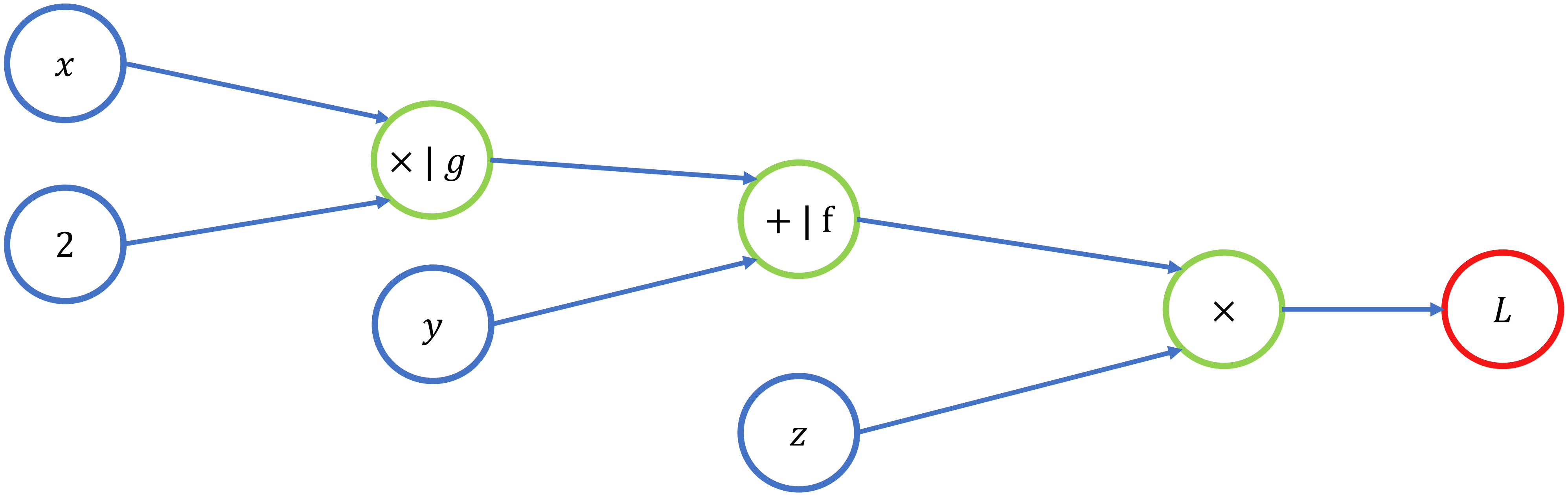
Assume that we did forward pass and evaluated values at each node

$$L = z(2x + y)$$



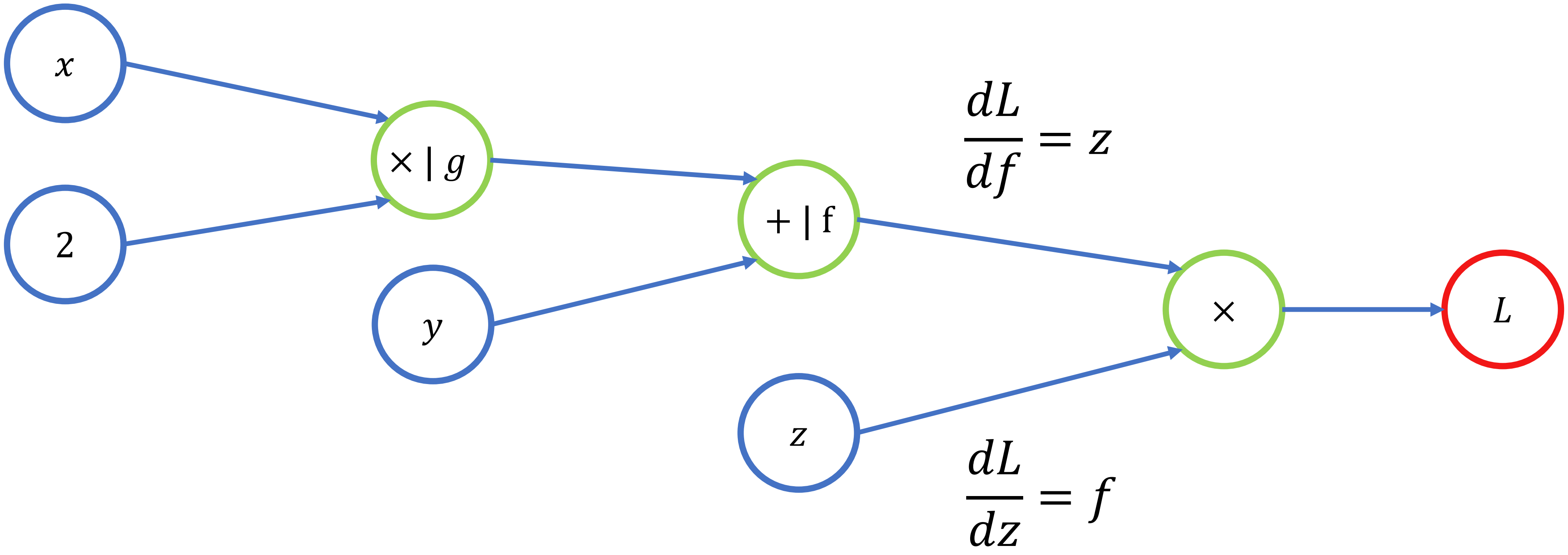
Chain rule

$$g(x) = 2x$$
$$f(g, y) = g + y$$
$$L(z, f) = zf$$



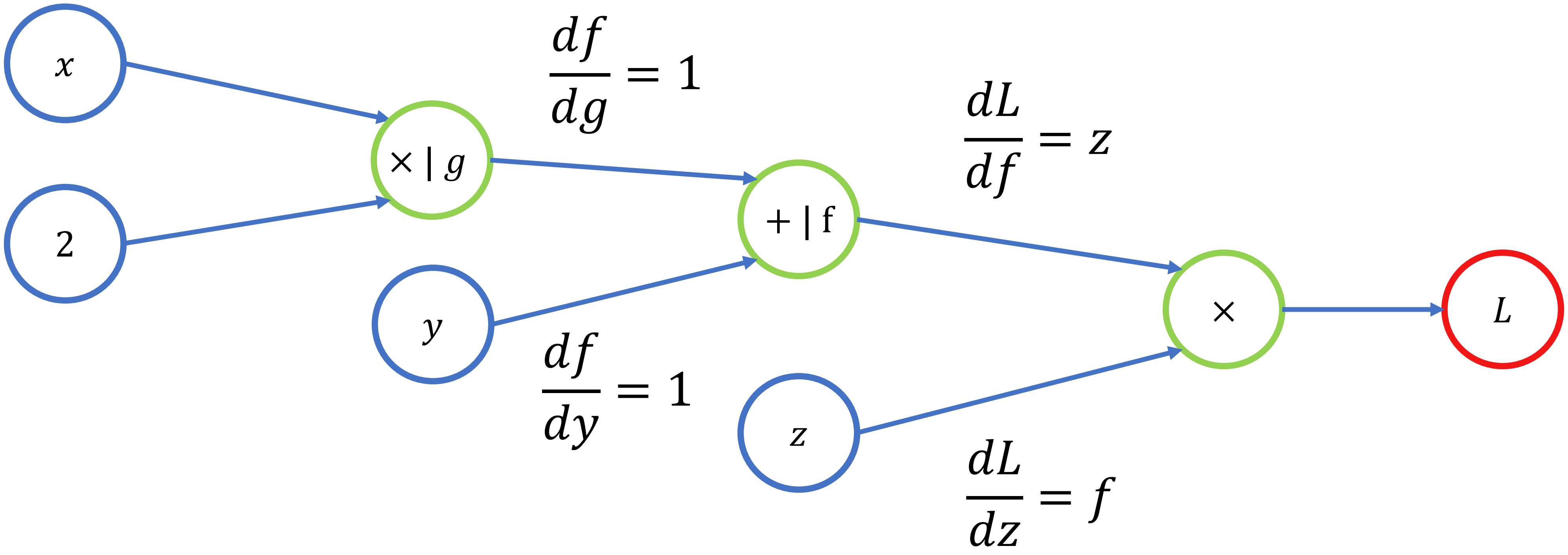
Chain rule

$$g(x) = 2x$$
$$f(g, y) = g + y$$
$$L(z, f) = zf$$



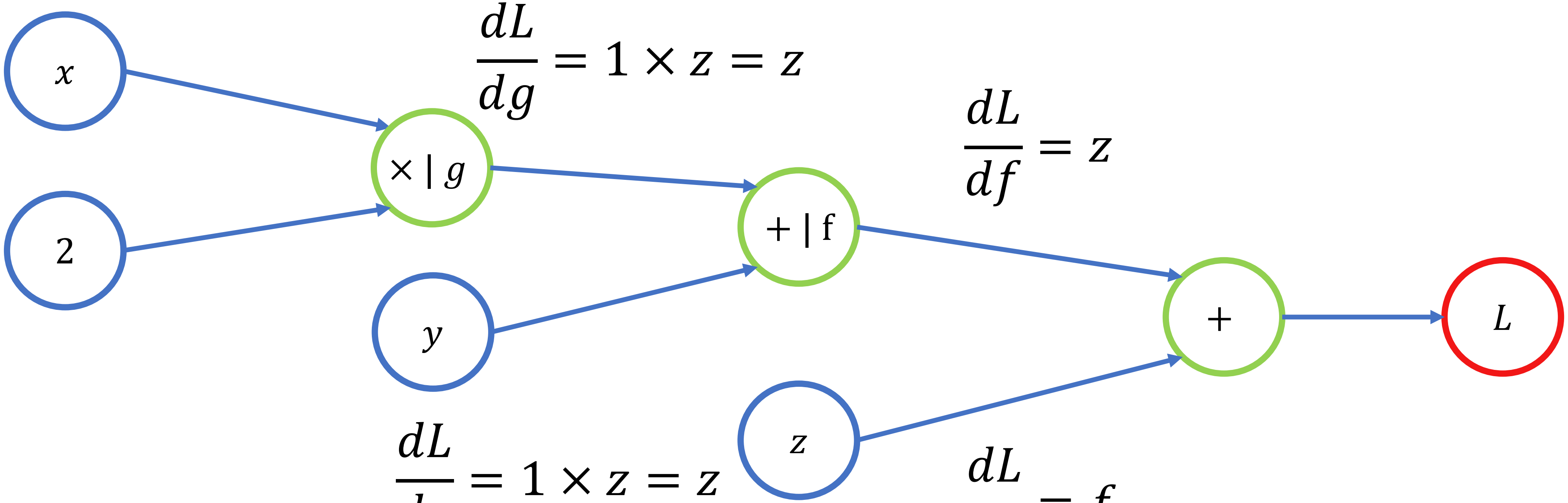
Chain rule

$$g = 2x$$
$$f = g + y$$
$$L = zf$$



Chain rule

$$g = 2x$$
$$f = g + y$$
$$L = zf$$



$$\frac{dL}{dg} = 1 \times z = z$$

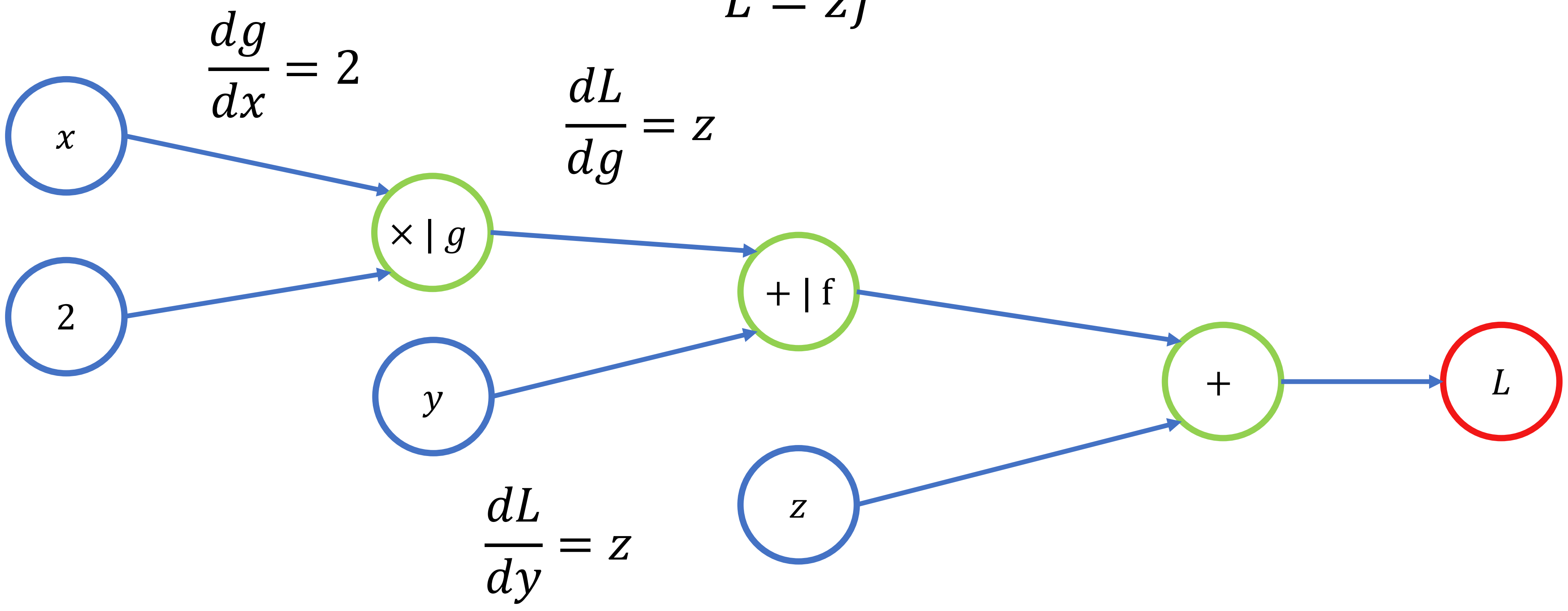
$$\frac{dL}{df} = z$$

$$\frac{dL}{dy} = 1 \times z = z$$

$$\frac{dL}{dz} = f$$

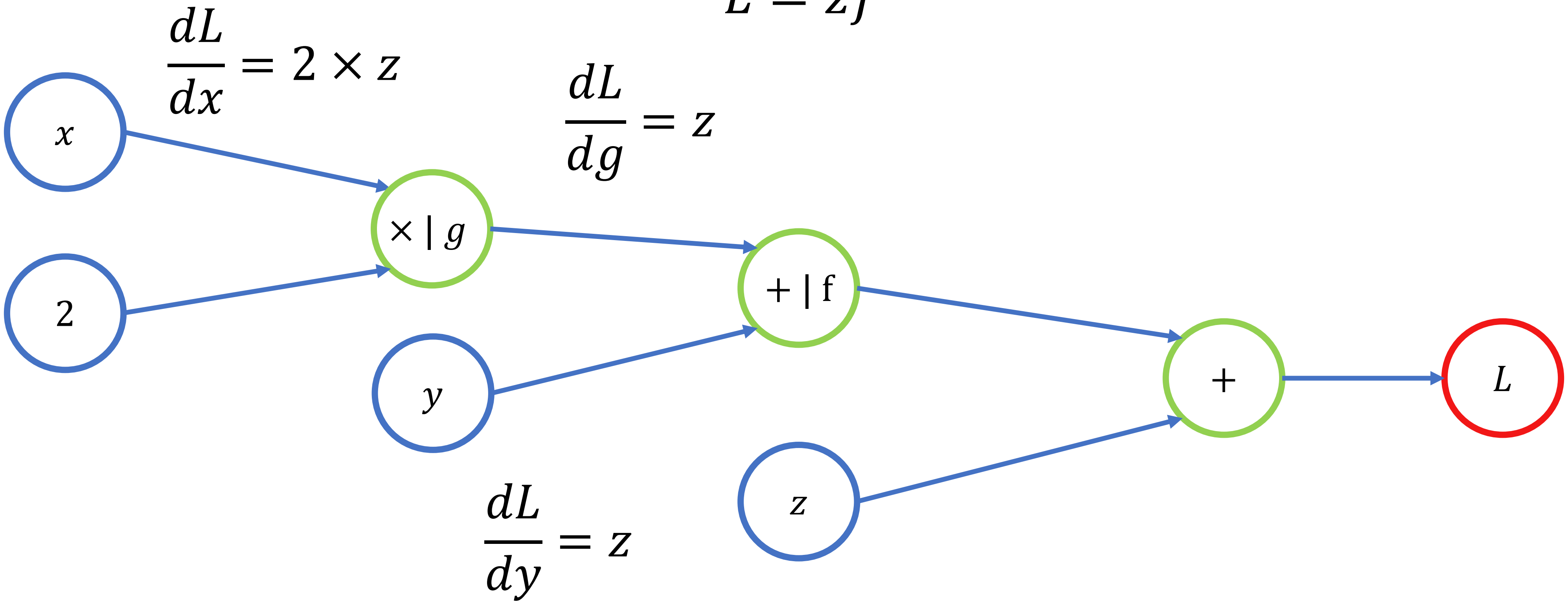
Chain rule

$$g = 2x$$
$$f = g + y$$
$$L = zf$$



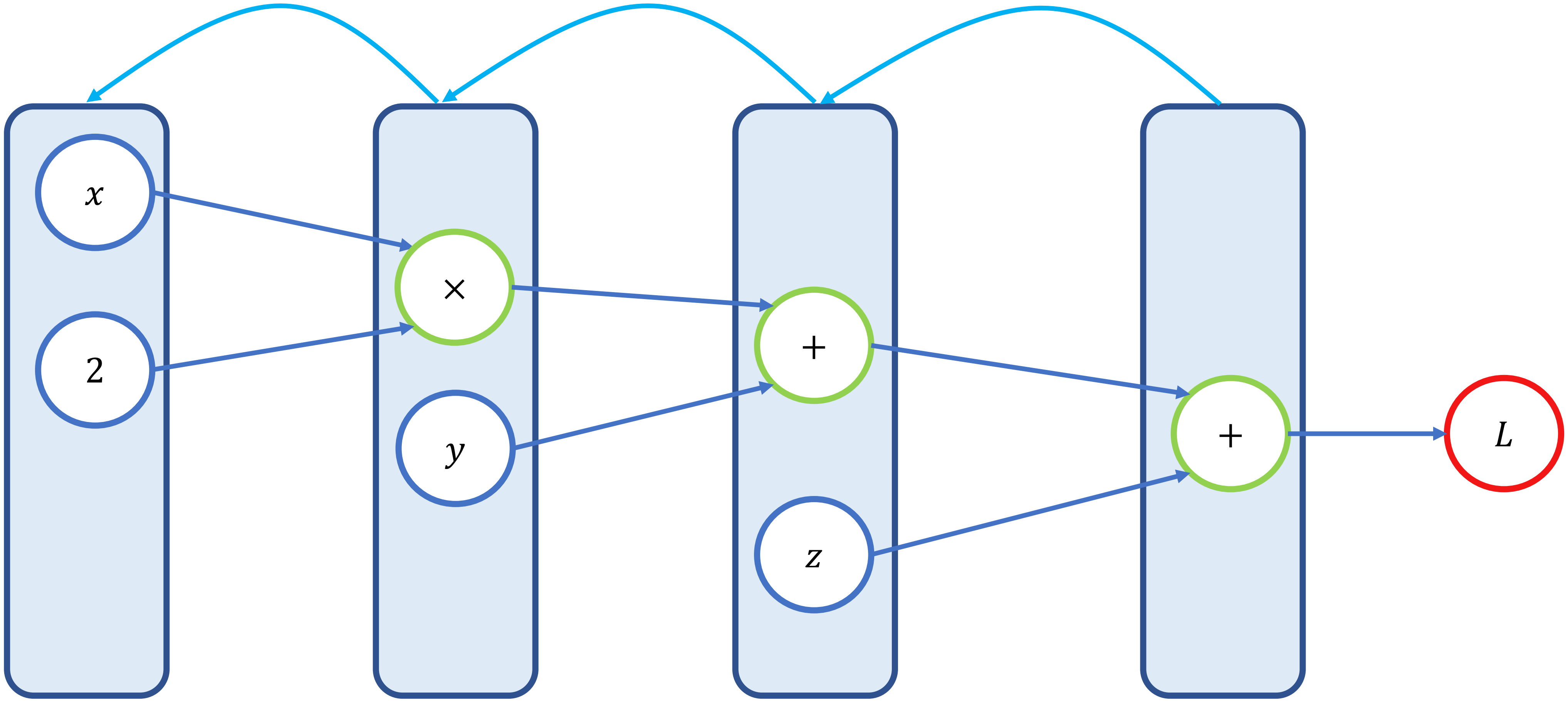
Chain rule

$$g = 2x$$
$$f = g + y$$
$$L = zf$$

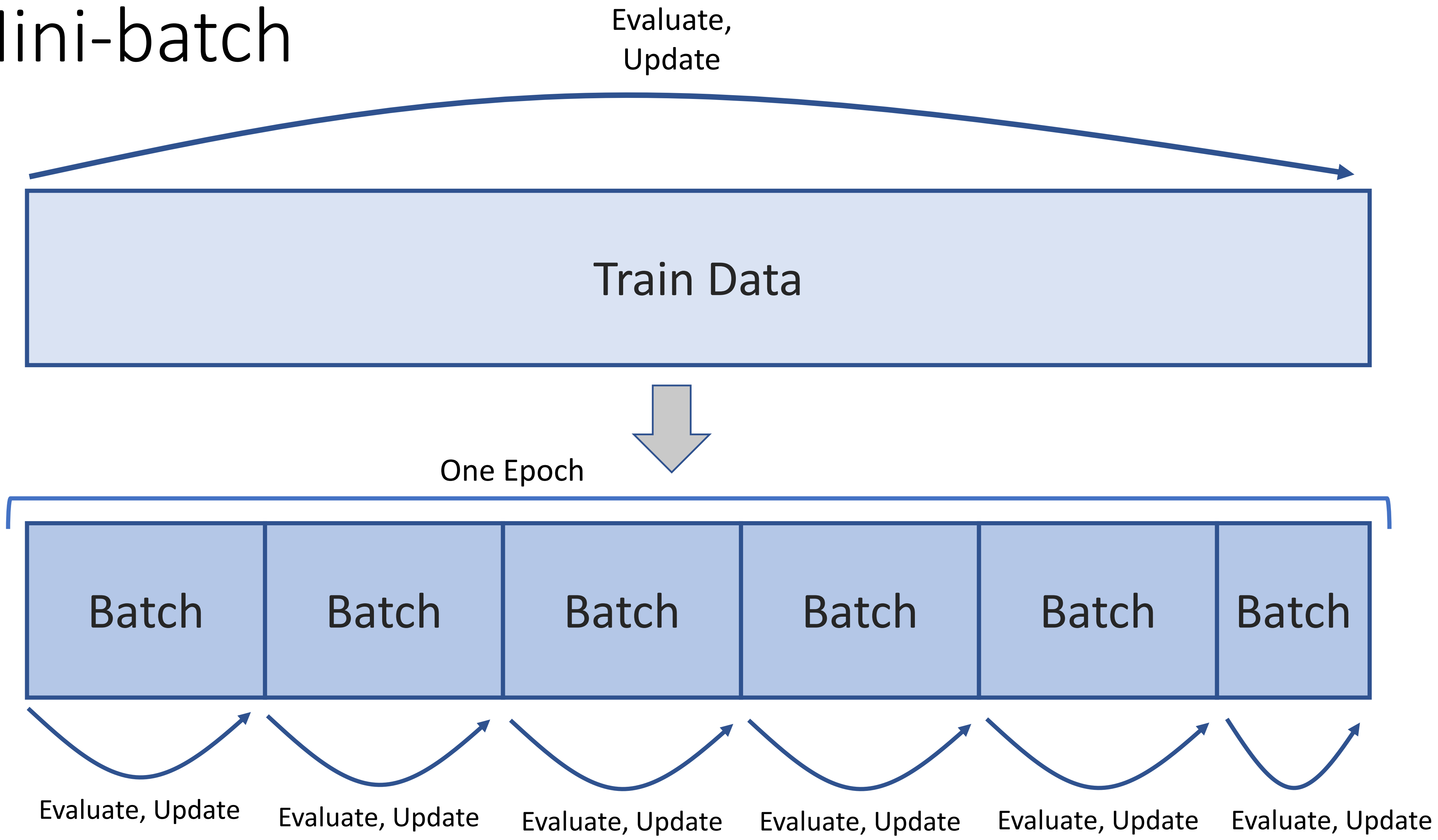


Backpropagation

At each step we only need to store gradient of the next layer and values of its inputs



Mini-batch


Evaluate,
Update
Train Data
One Epoch
Batch
Batch
Batch
Batch
Batch
Batch
Evaluate, Update
Evaluate, Update
Evaluate, Update
Evaluate, Update
Evaluate, Update
Evaluate, Update

NN Matrix Representation

Linear Algebra Recap

- Vector – a list of numbers

$$b = \begin{pmatrix} 1 \\ 7 \\ 5 \end{pmatrix}$$

- Matrix – a 2-dimensional list of numbers

$$A = \begin{pmatrix} 6 & 2 & 3 \\ 1 & 5 & 7 \end{pmatrix}$$

- Matrix A have sizes (shape) $A.shape = (2, 3)$
- Vector b is a matrix of shape $b.shape = (1, 3)$
- Tensor is just another name for multi-dimensional matrix

Vector and Matrix Operations

- Transposition (flipping)

$$b = \begin{pmatrix} 1 \\ 7 \\ 5 \end{pmatrix}, b^T = (1, 7, 5)$$

$$A = \begin{pmatrix} 6 & 2 & 3 \\ 1 & 5 & 7 \end{pmatrix}, A^T = \begin{pmatrix} 6 & 1 \\ 2 & 5 \\ 3 & 7 \end{pmatrix}$$

- Vector scalar multiplication

$$x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, z = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}, x^T z = 1 * 4 + 2 * 5 + 3 * 6 = 32$$

Vector and Matrix Operations

- Matrix-vector multiplication

$$A = \begin{pmatrix} 6 & 2 & 3 \\ 1 & 5 & 7 \end{pmatrix}, x = \begin{pmatrix} 1 \\ 7 \\ 5 \end{pmatrix}$$

$$Ax = \begin{pmatrix} 6 * 1 + 2 * 7 + 3 * 5 \\ 1 * 1 + 5 * 7 + 7 * 5 \end{pmatrix} = \begin{pmatrix} 35 \\ 71 \end{pmatrix}$$

- Matrix-matrix multiplication

$$A = \begin{pmatrix} 6 & 1 \\ 2 & 5 \\ 3 & 7 \end{pmatrix}, B = \begin{pmatrix} 0 & 2 \\ 1 & 3 \end{pmatrix}$$

$$AB = \begin{pmatrix} 0 * 6 + 1 * 1 & 6 * 2 + 1 * 3 \\ 0 * 2 + 1 * 5 & 2 * 2 + 5 * 3 \\ 3 * 0 + 7 * 1 & 2 * 3 + 7 * 3 \end{pmatrix} = \begin{pmatrix} 1 & 15 \\ 5 & 19 \\ 7 & 27 \end{pmatrix}$$

Vector and Matrix Operations

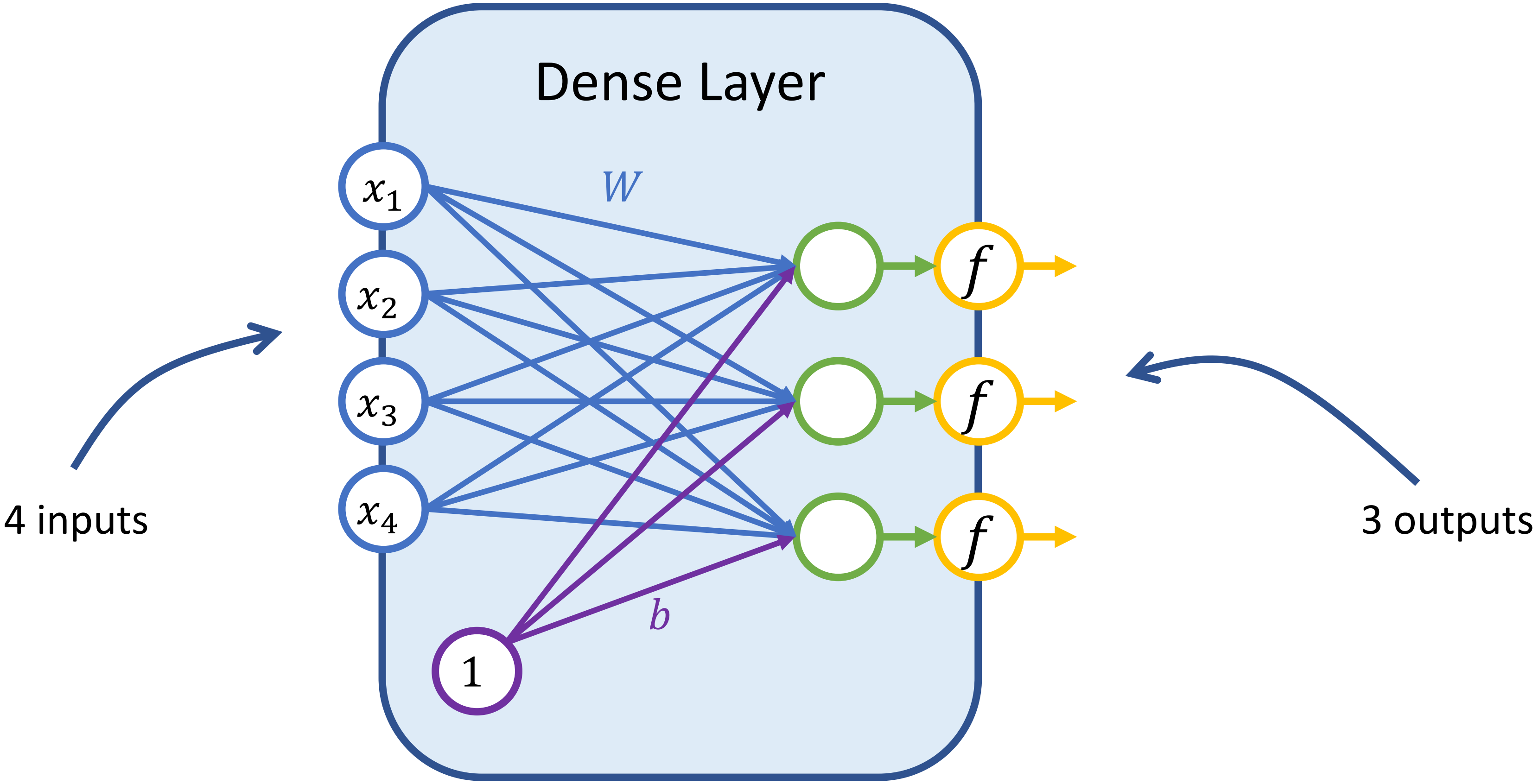
- Matrix-matrix multiplication
- To multiply matrix A and B, last dimension of A and first dimension of B must match

$$A.shape = (6,7), \qquad B.shape(7,10),$$
$$(AB).shape = (6,10)$$

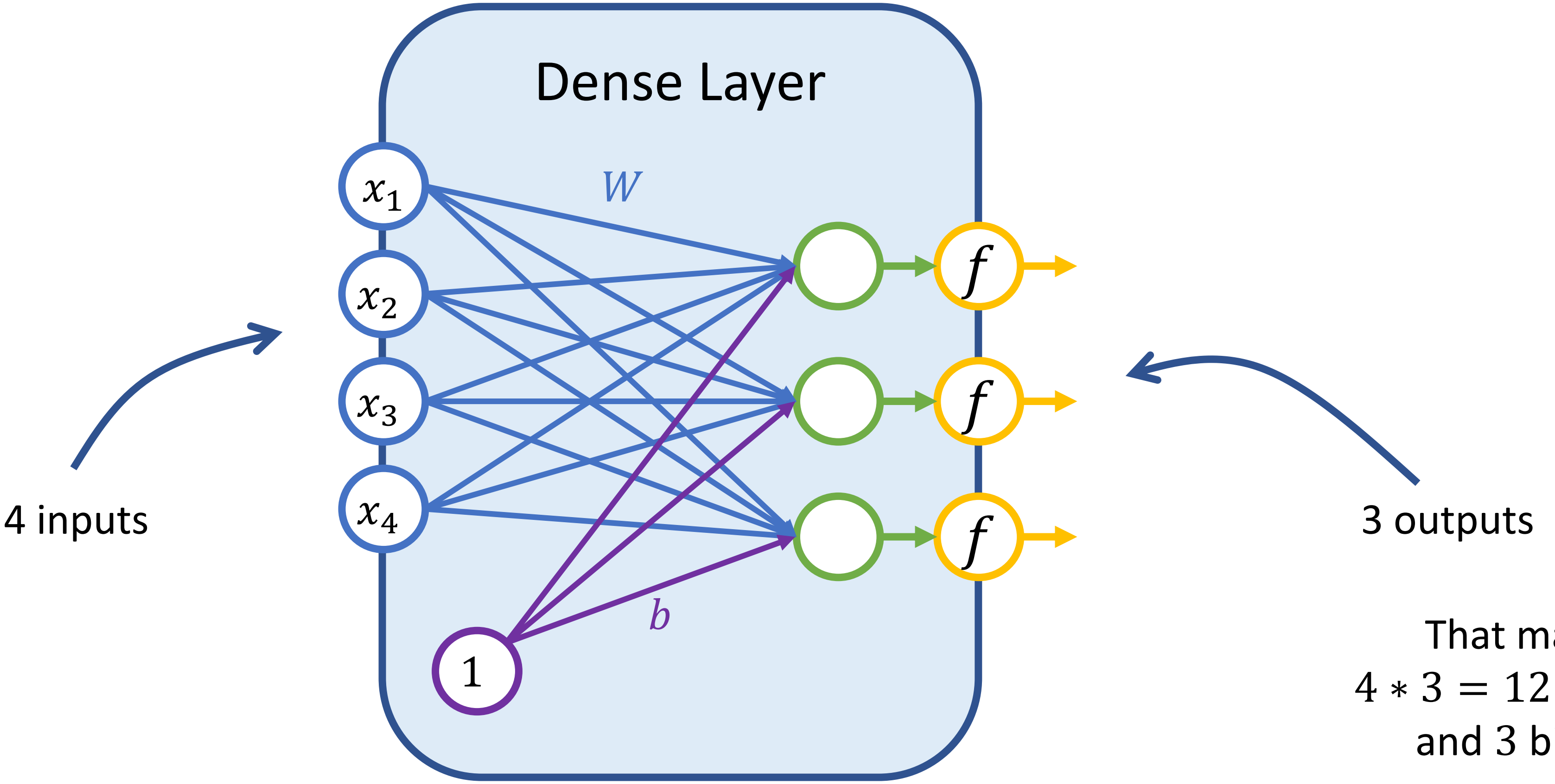
Input as a Matrix

- Our NN has input of size N (tumor size, age, …)
- We pass B (batch size) data points at once
- We can represent our input batch as a matrix of shape (B, N)
- Let’s think of Dense layer matrix representation

Dense Layer via Matrices

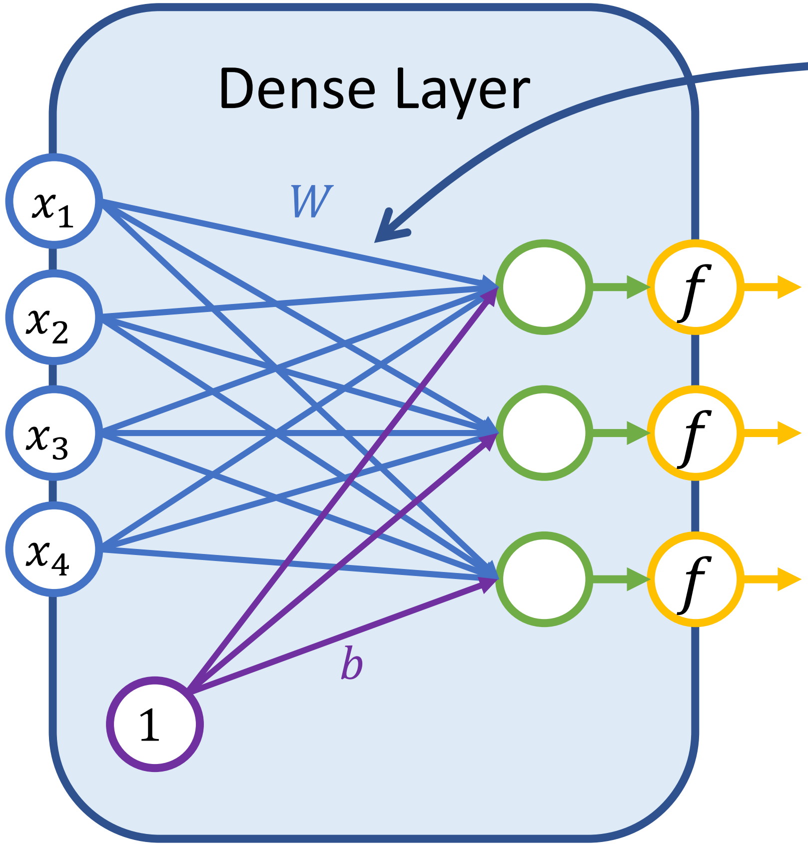


Dense Layer via Matrices



That makes
$4 * 3 = 12$ weights
and 3 biases

Dense Layer via Matrices



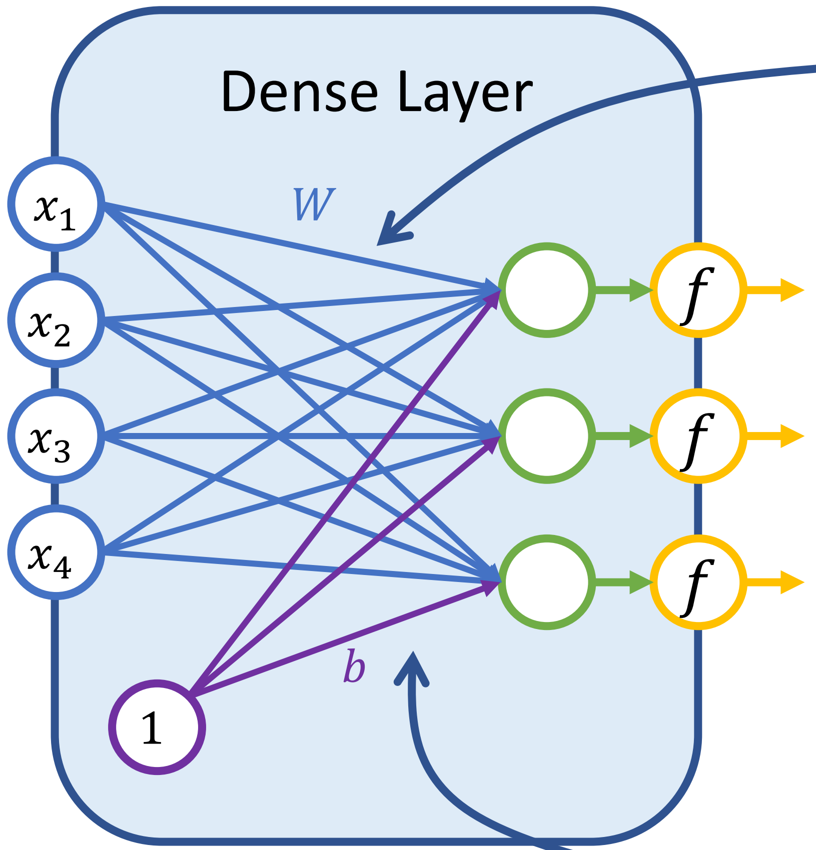
4 inputs

We can represent *weights* as a matrix of shape $(3, 4)$

3 outputs

That makes
$4 * 3 = 12$ weights
and 3 biases

Dense Layer via Matrices



We can represent *weights* as a matrix of shape $(3, 4)$

4 inputs

3 outputs

That makes
$4 * 3 = 12$ weights
and 3 biases

We can represent *biases* as a vector of shape (3, 1)

Dense Layer via Matrices

• Number of inputs:	N	
• Batch Size:	B	
• Input batch:	X	$X.shape = (B, N)$
• Layer input size:	N	
• Layer output size:	M	
• Layer weights:	W	$W.shape = (N, M)$
• Layer bias:	b	$b.shape = (M, 1)$
• Layer output:	$WX + b$	$(WX + b).shape = (B, M)$

Applying Activation Functions

- We have output from the layer $(WX + b)$
- Most of activation functions apply function to the every entry in the matrix individually
- For instance, sigmoid:

$$A = \begin{pmatrix} 6 & 2 & 3 \\ 1 & 5 & 7 \end{pmatrix}$$

$$\sigma(A) = \begin{pmatrix} \sigma(6) & \sigma(2) & \sigma(3) \\ \sigma(1) & \sigma(5) & \sigma(7) \end{pmatrix}$$

- An exception – *softmax*. But is still quite straightforward

Why Do We Care About These Matrices

- It is faster
- People wrote a lot of code for efficient matrix operations
- Graphical Processing Units can process this even more efficiently

Matrices Recap

- We can represent our batch as a matrix X of shape (B, N)
- We can represent our dense layer *weights* with a matrix W of shape $(Inputs, Outputs)$
- We can represent our dense layer bias with a vector b of shape $(Outputs, 1)$
- We can compute dense layer output via $WX + b$
- And we apply non-linearity just like that $f(WX + b)$

Lecture Recap

- Logistic regression is a 1-layer Neural Network
- We optimize it with gradient descent
- To compute gradients for deeper models we use *backpropagation*
- We also train models in a mini-batch setting
- We can represent our neural networks via matrix multiplications

This is It For the Second Lecture