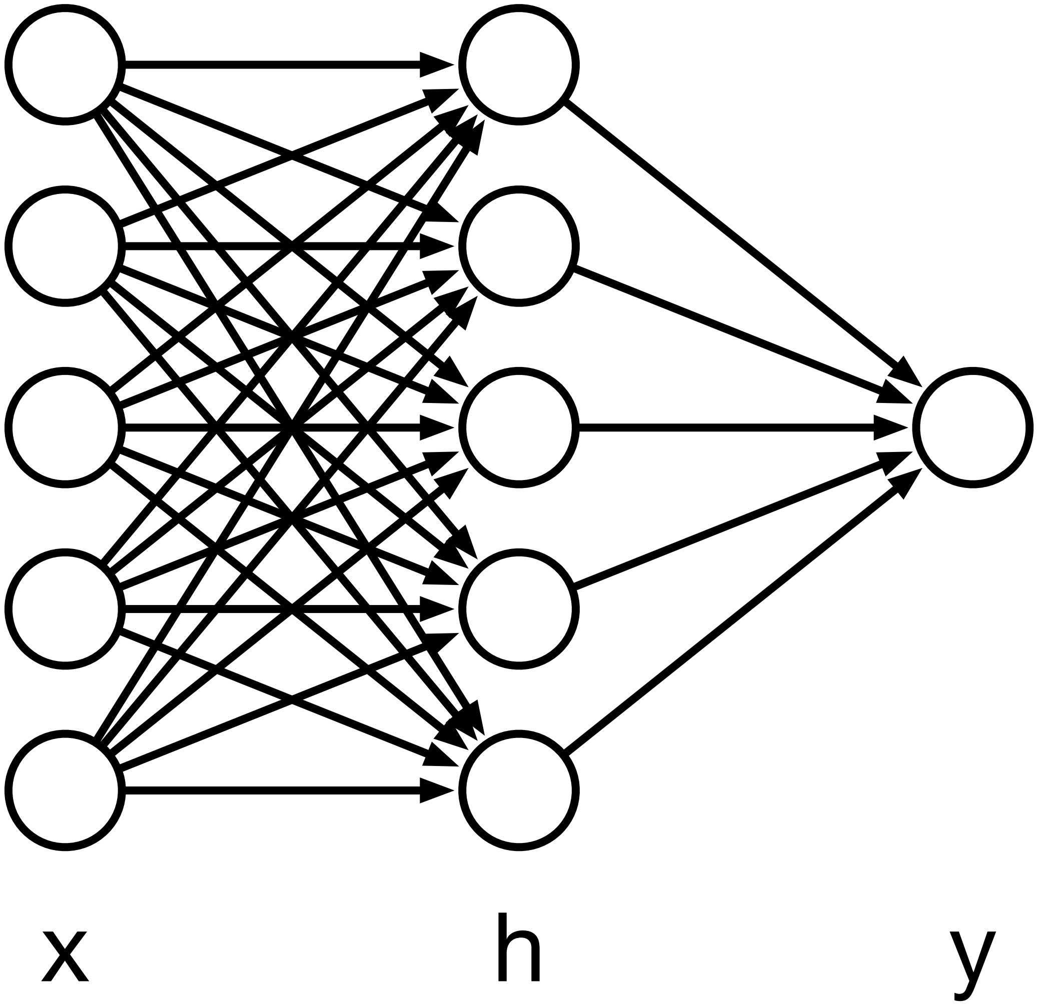
Computation Graphs

Philipp Koehn

29 September 2020

Neural Network Cartoon

- A common way to illustrate a neural network


x
h
y

Neural Network Math

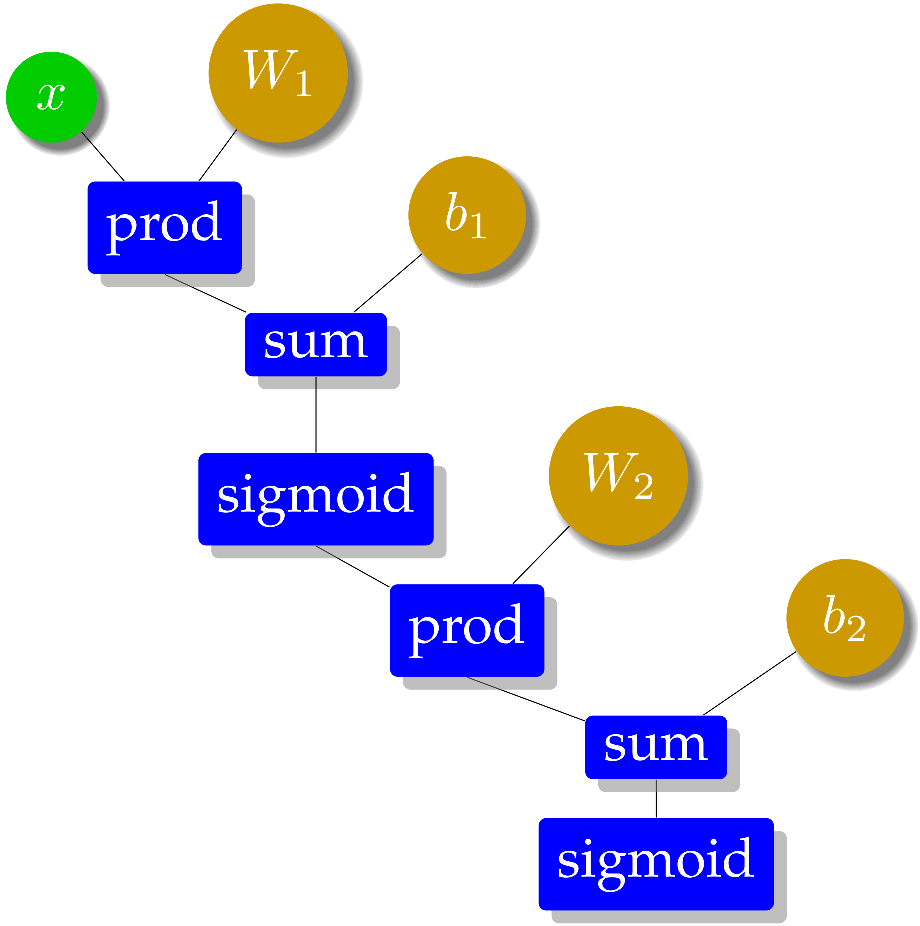
- Hidden layer

$$h = \text{sigmoid}(W_1 x + b_1)$$

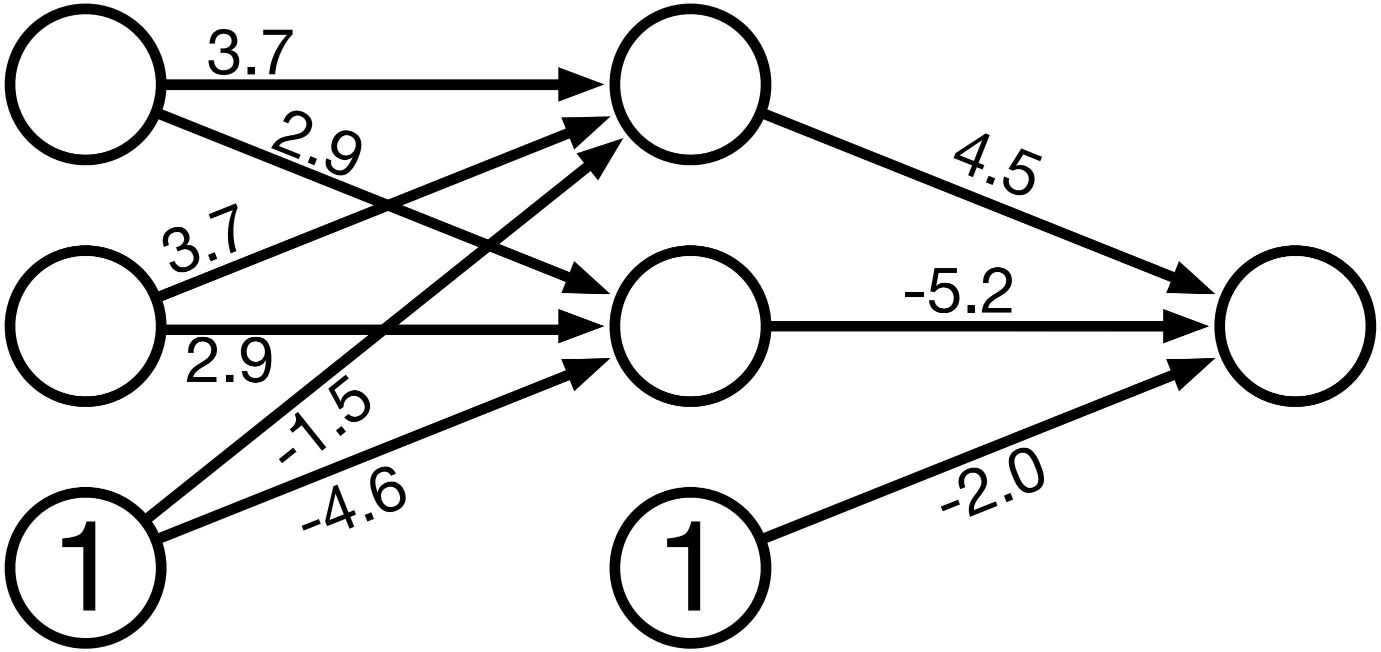
- Final layer

$$y = \text{sigmoid}(W_2 h + b_2)$$

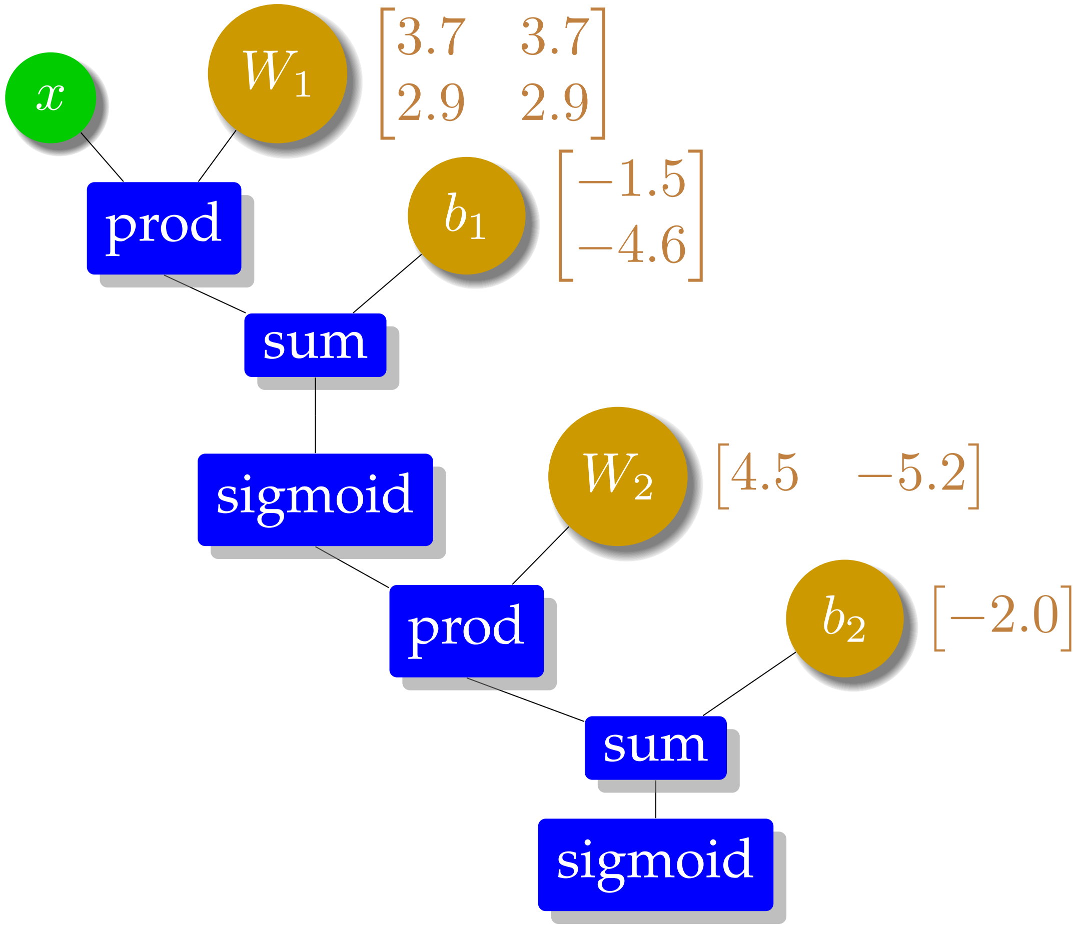
Computation Graph



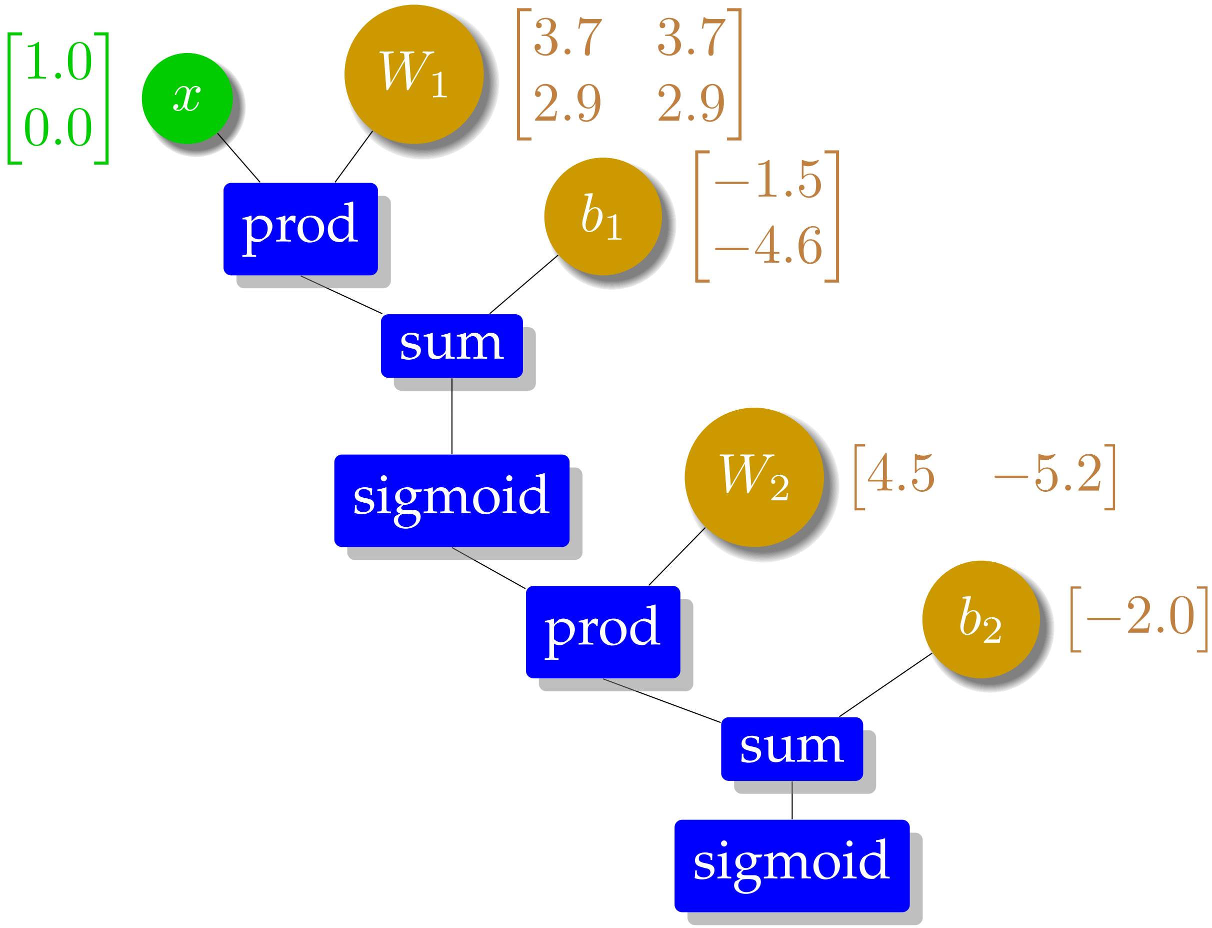
Simple Neural Network



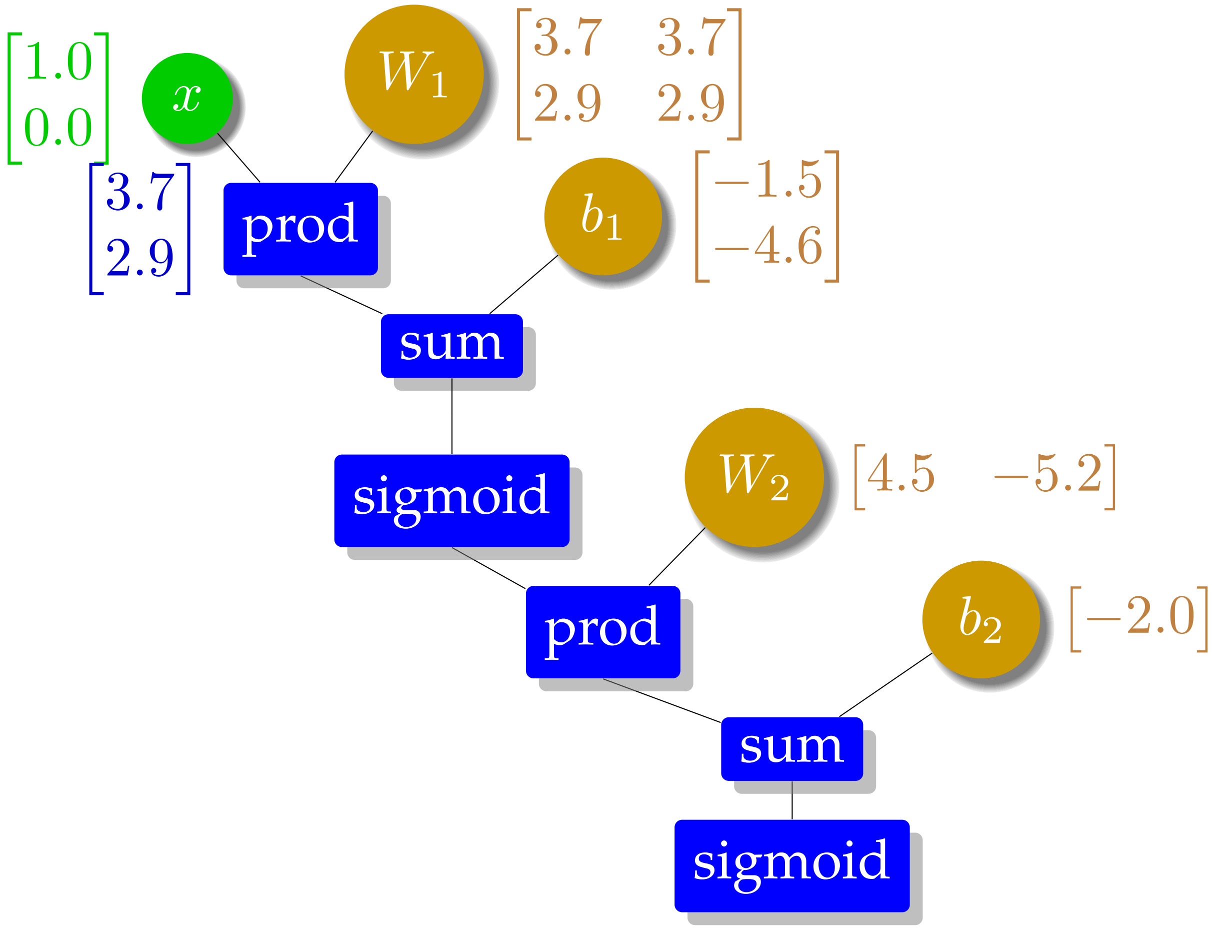
Computation Graph



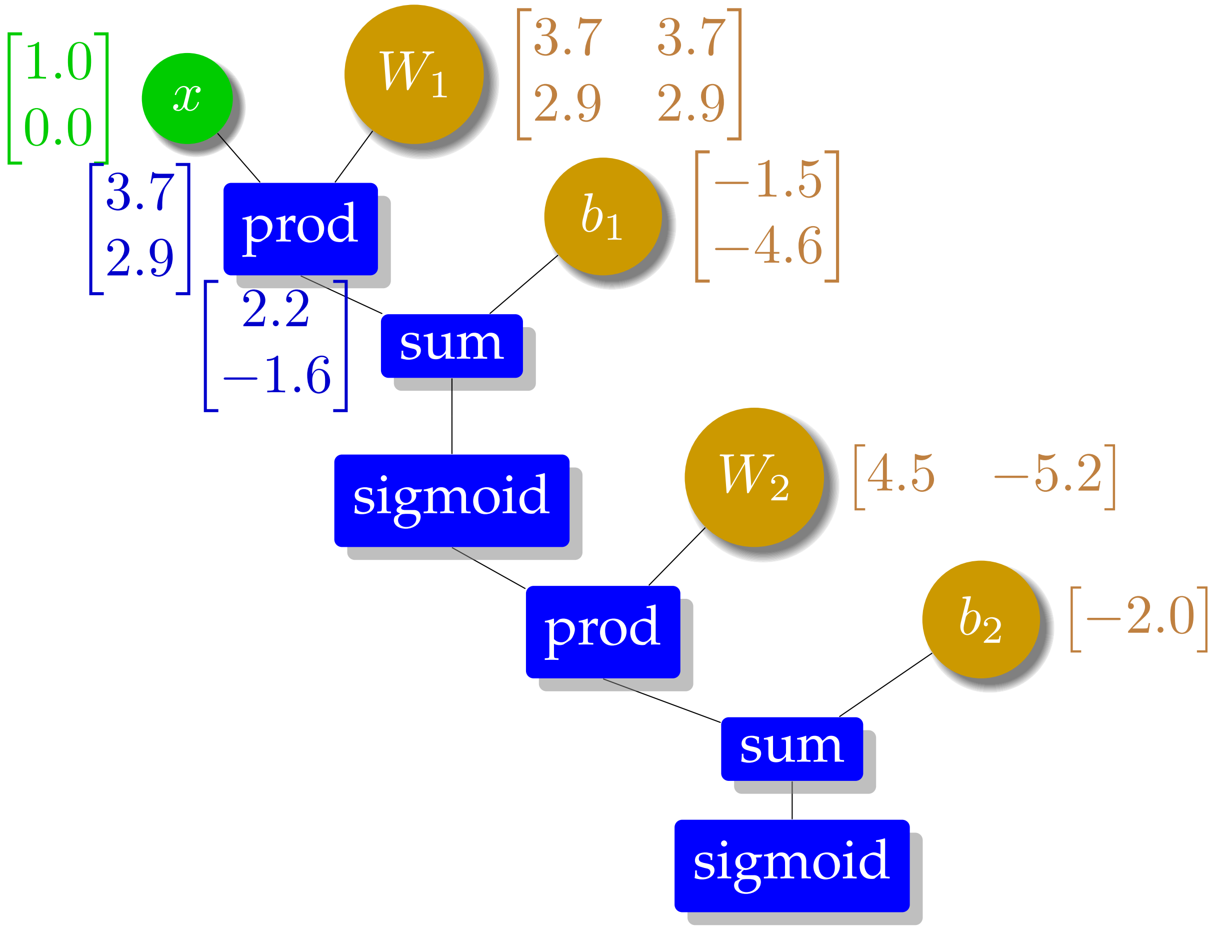
Processing Input



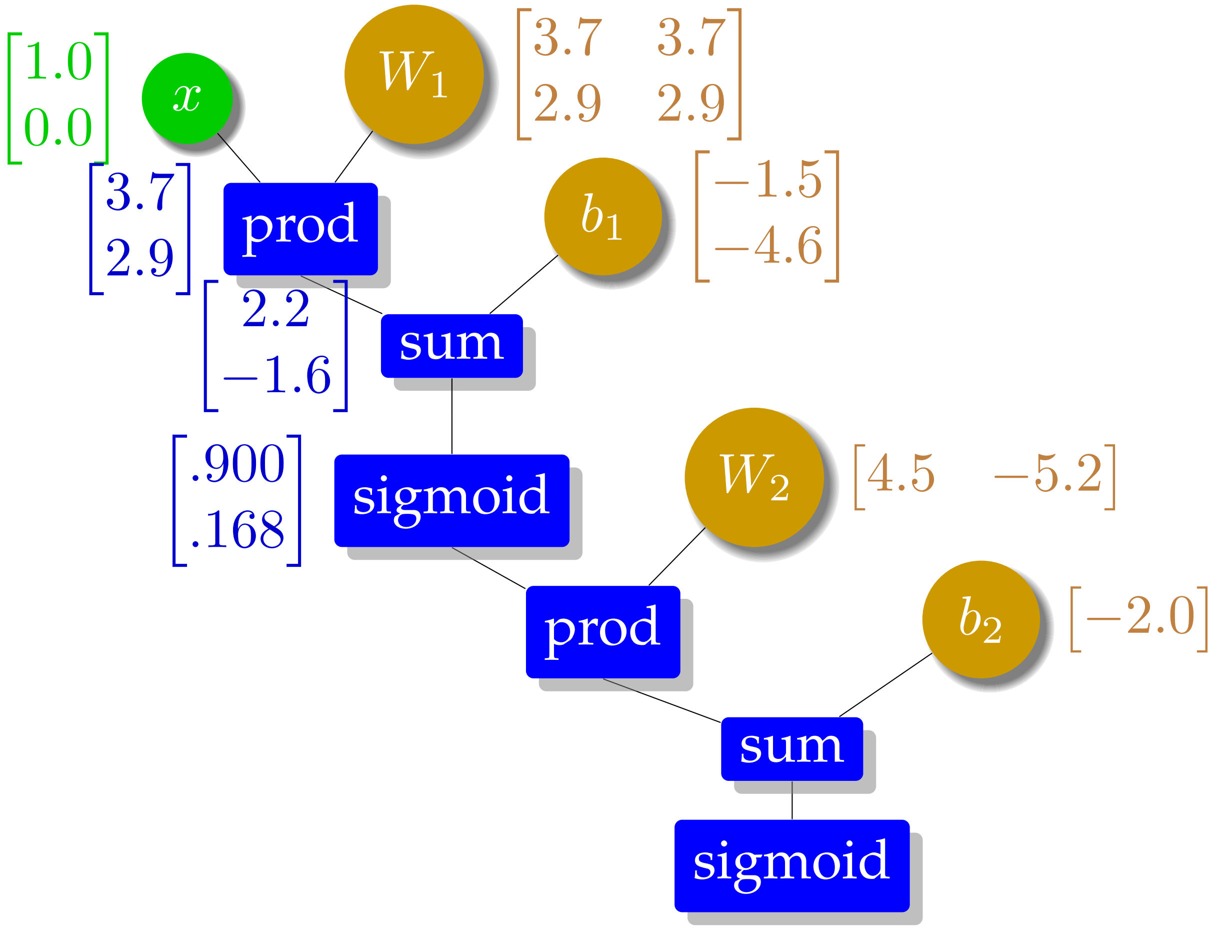
Processing Input



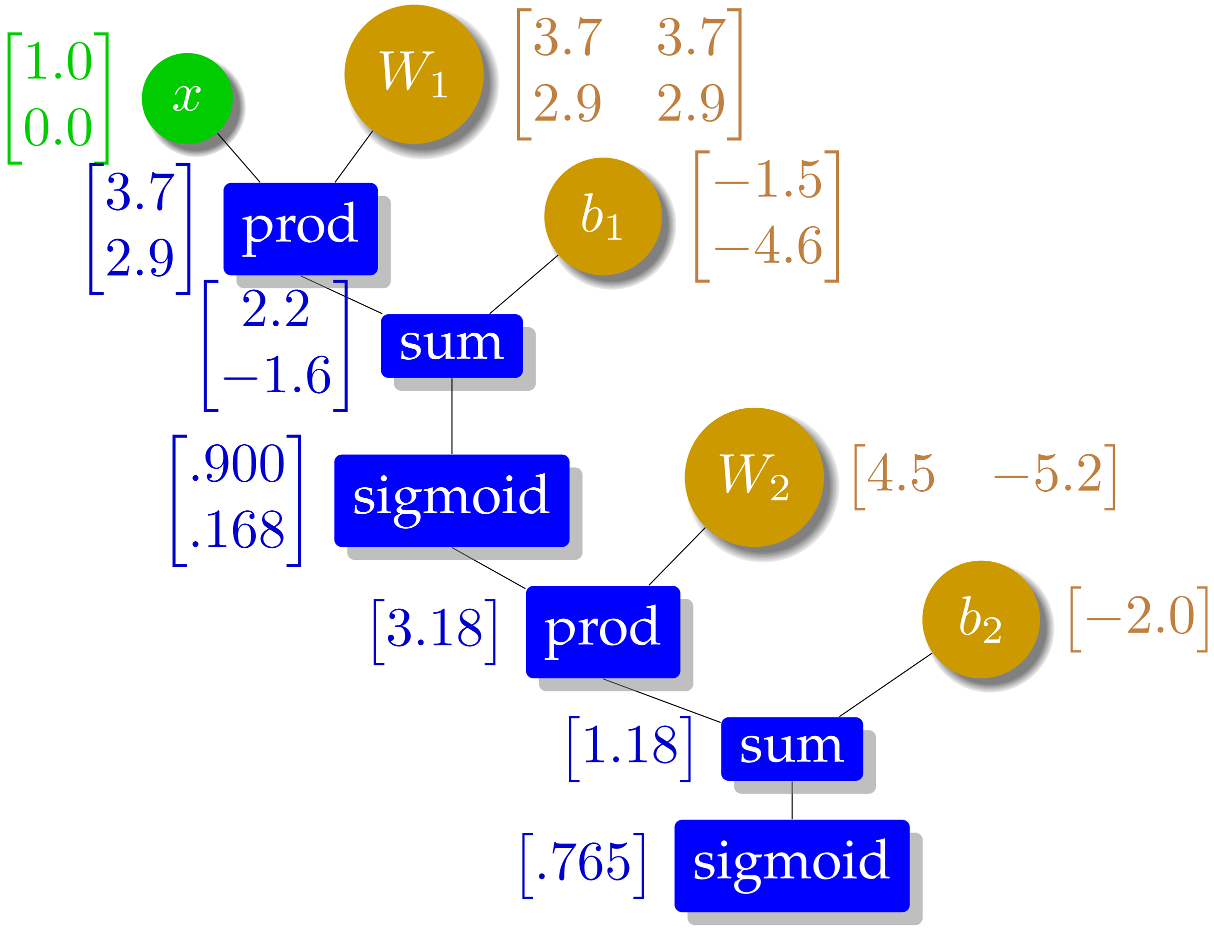
Processing Input



Processing Input



Processing Input



Error Function

- For training, we need a measure how well we do

$\Rightarrow$ Cost function

also known as objective function, loss, gain, cost, ...

- For instance L2 norm

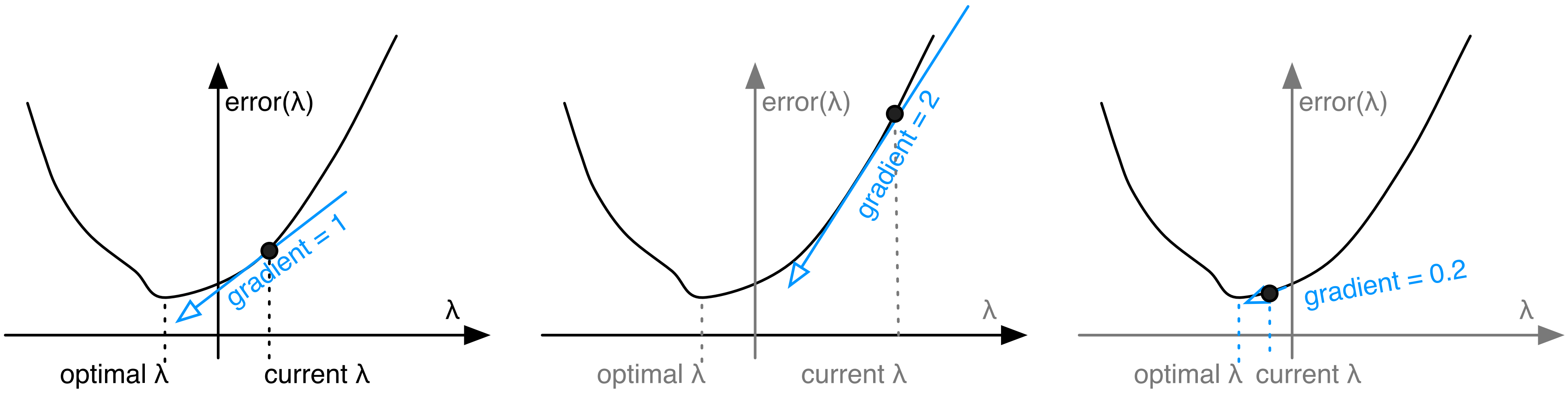
$$\text{error} = \frac{1}{2}(t - y)^2$$

Gradient Descent

- We view the error as a function of the trainable parameters

$$\text{error}(\lambda)$$

- We want to optimize $\text{error}(\lambda)$ by moving it towards its optimum



- Why not just set it to its optimum?
  - we are updating based on one training example, do not want to overfit to it
  - we are also changing all the other parameters, the curve will look different

Calculus Refresher: Chain Rule

- Formula for computing derivative of composition of two or more functions
  - functions f and g
  - composition $f \circ g$ maps x to $f(g(x))$

- Chain rule

$$(f \circ g)' = (f' \circ g) \cdot g'$$

  or

$$F'(x) = f'(g(x))g'(x)$$

- Leibniz's notation

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

  if $z = f(y)$ and $y = g(x)$, then

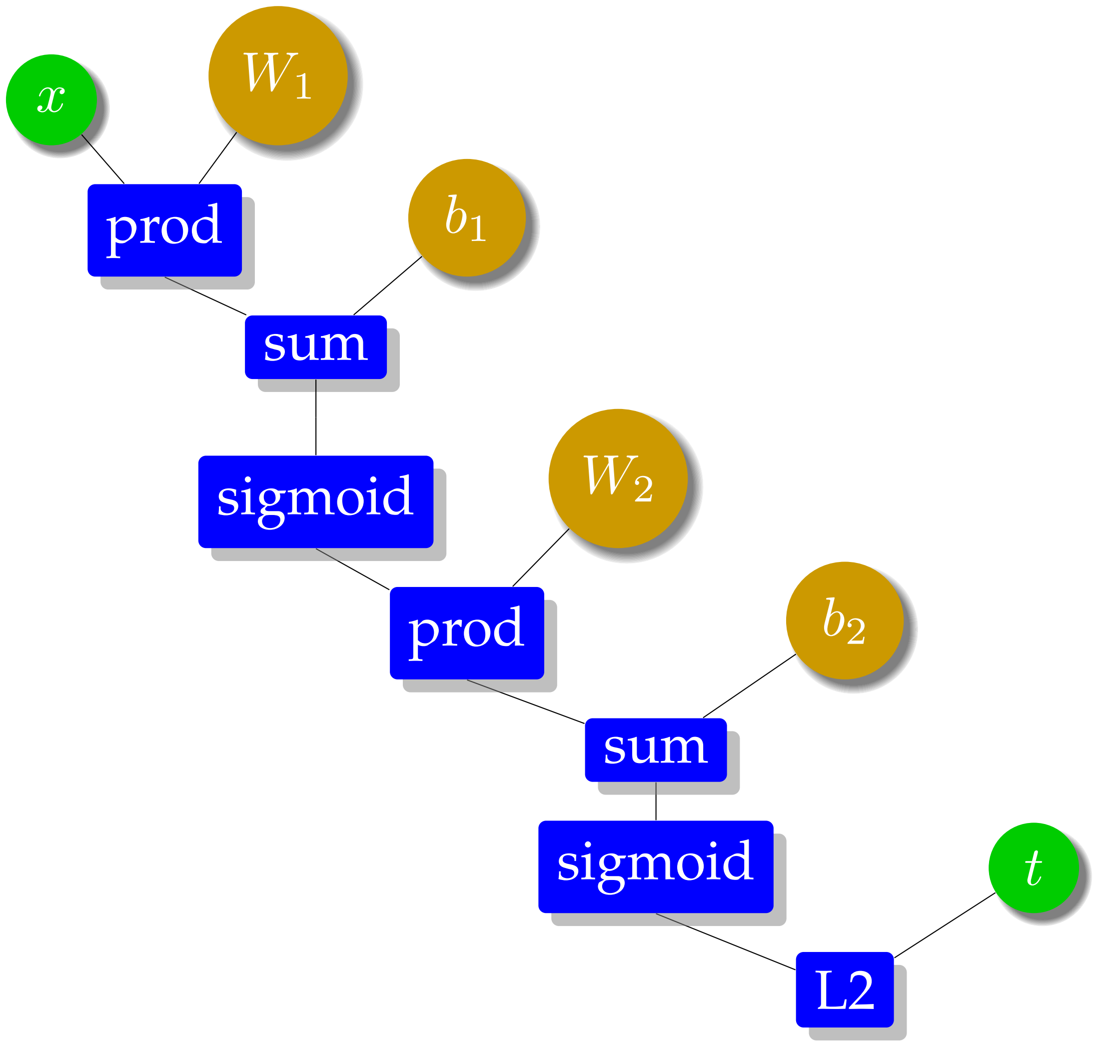
$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} = f'(y)g'(x) = f'(g(x))g'(x)$$

Final Layer Update

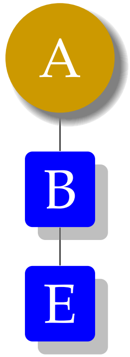
- Linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \text{sigmoid}(s)$
- Error (L2 norm) $E = \frac{1}{2}(t - y)^2$
- Derivative of error with regard to one weight w_k

$$\frac{dE}{dw_k} = \frac{dE}{dy} \frac{dy}{ds} \frac{ds}{dw_k}$$

Error Computation in Computation Graph

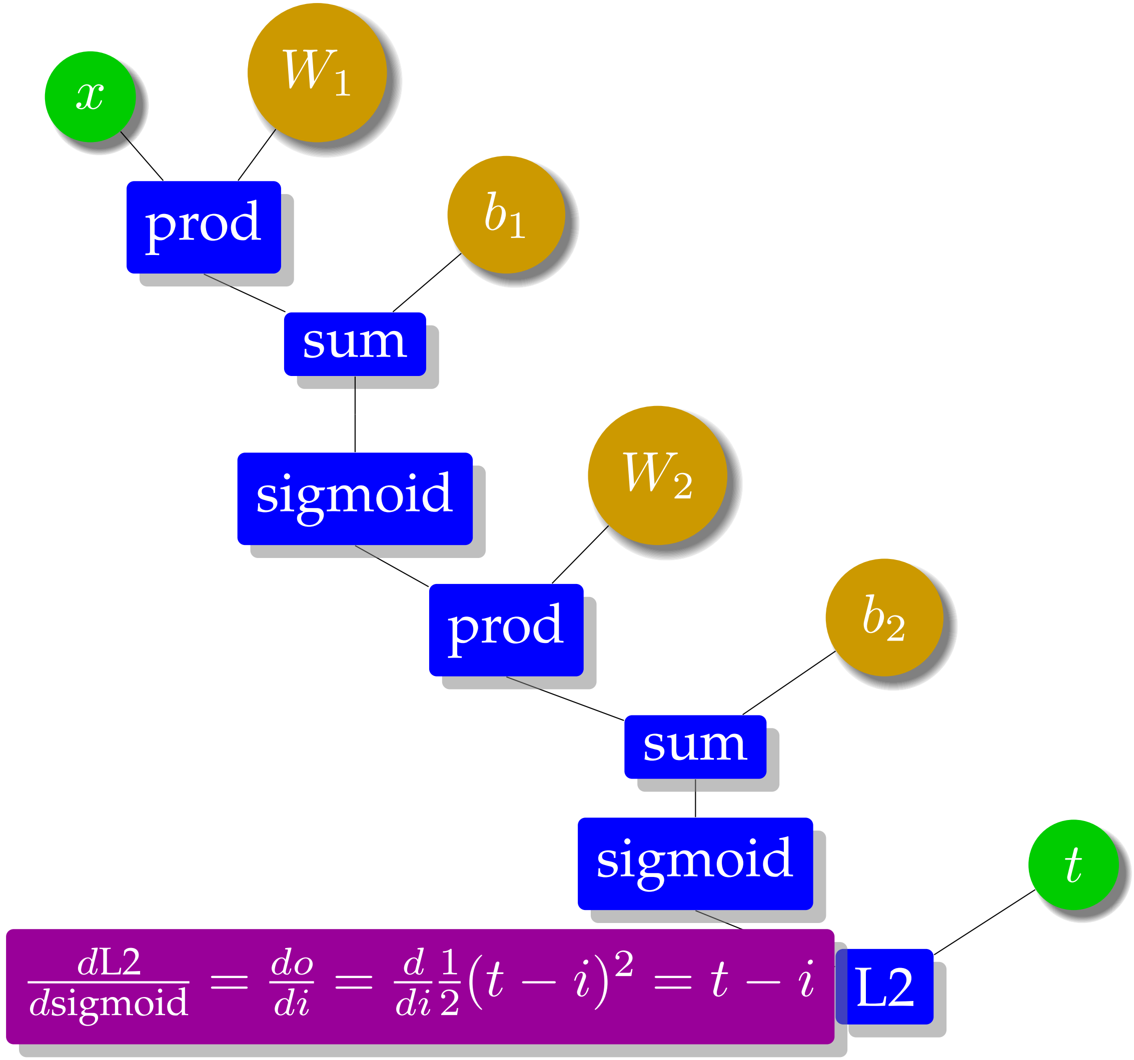


Error Propagation in Computation Graph

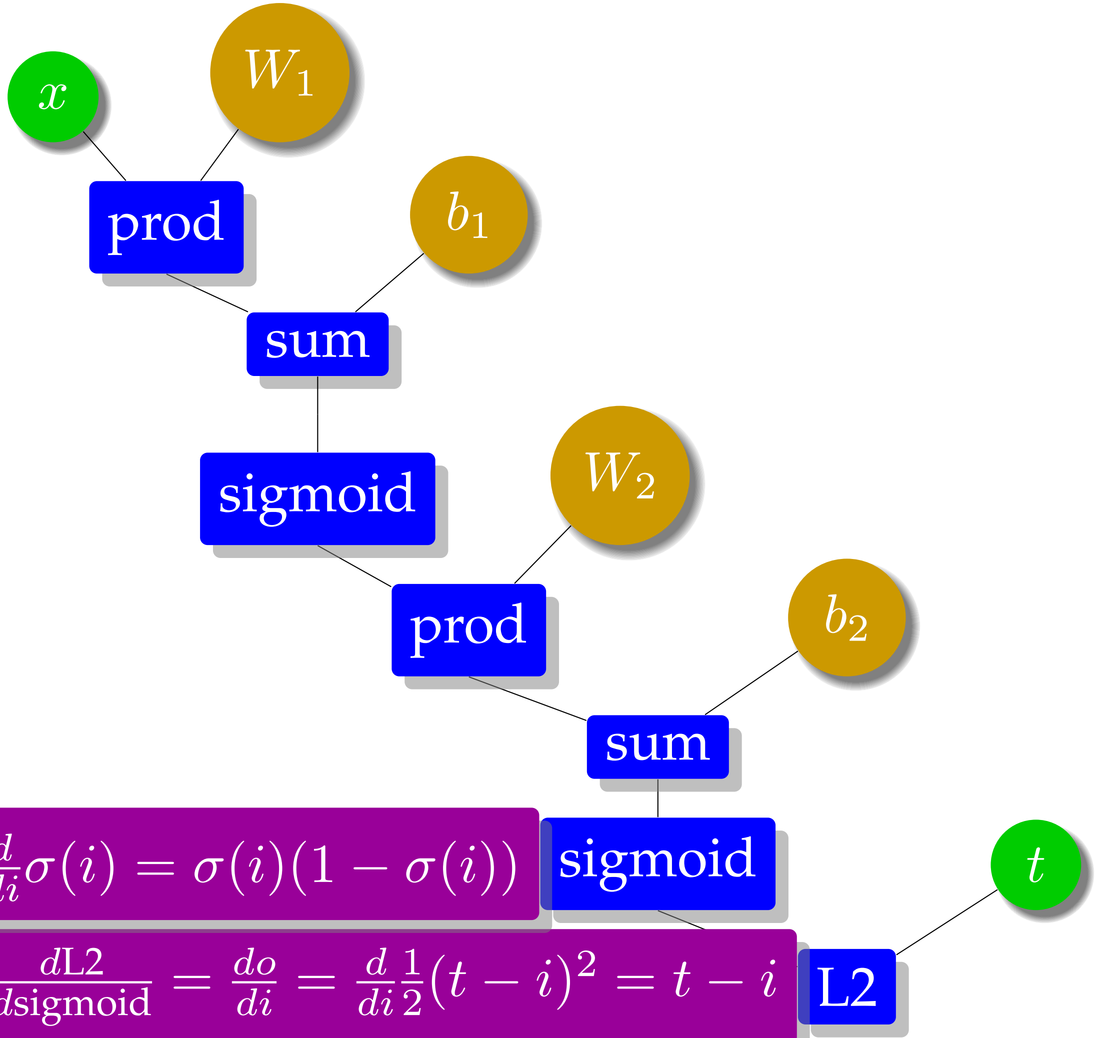


- Compute derivative at node A: $\frac{dE}{dA} = \frac{dE}{dB}\frac{dB}{dA}$
- Assume that we already computed $\frac{dE}{dB}$ (backward pass through graph)
- So now we only have to get the formula for $\frac{dB}{dA}$
- For instance B is a square node
  - forward computation: $B = A^2$
  - backward computation: $\frac{dB}{dA} = \frac{dA^2}{dA} = 2A$

Derivatives for Each Node



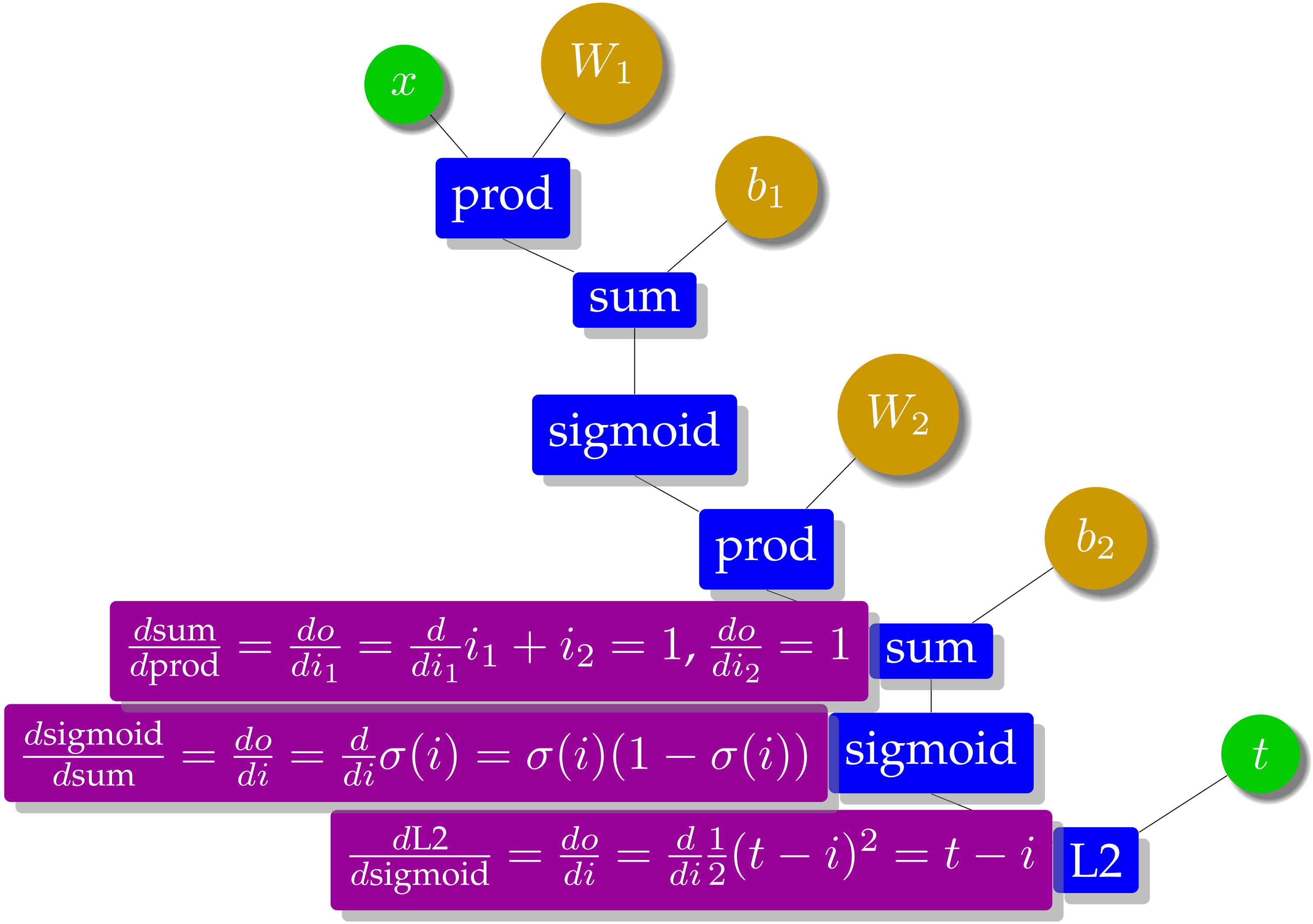
Derivatives for Each Node



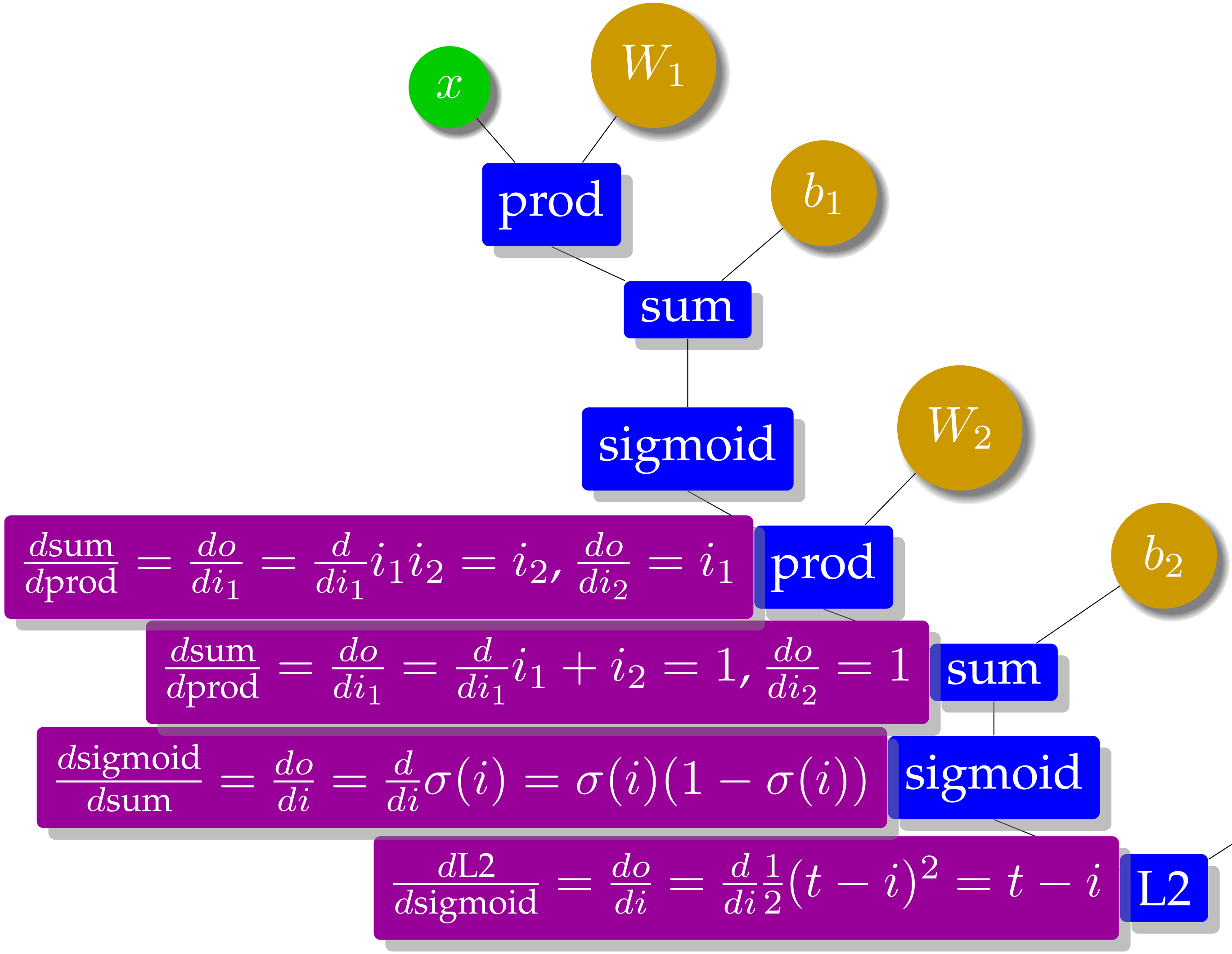
$$\frac{d\text{sigmoid}}{d\text{sum}} = \frac{do}{di} = \frac{d}{di}\sigma(i) = \sigma(i)(1-\sigma(i))$$

$$\frac{d\text{L2}}{d\text{sigmoid}} = \frac{do}{di} = \frac{d}{di}\frac{1}{2}(t-i)^2 = t-i$$

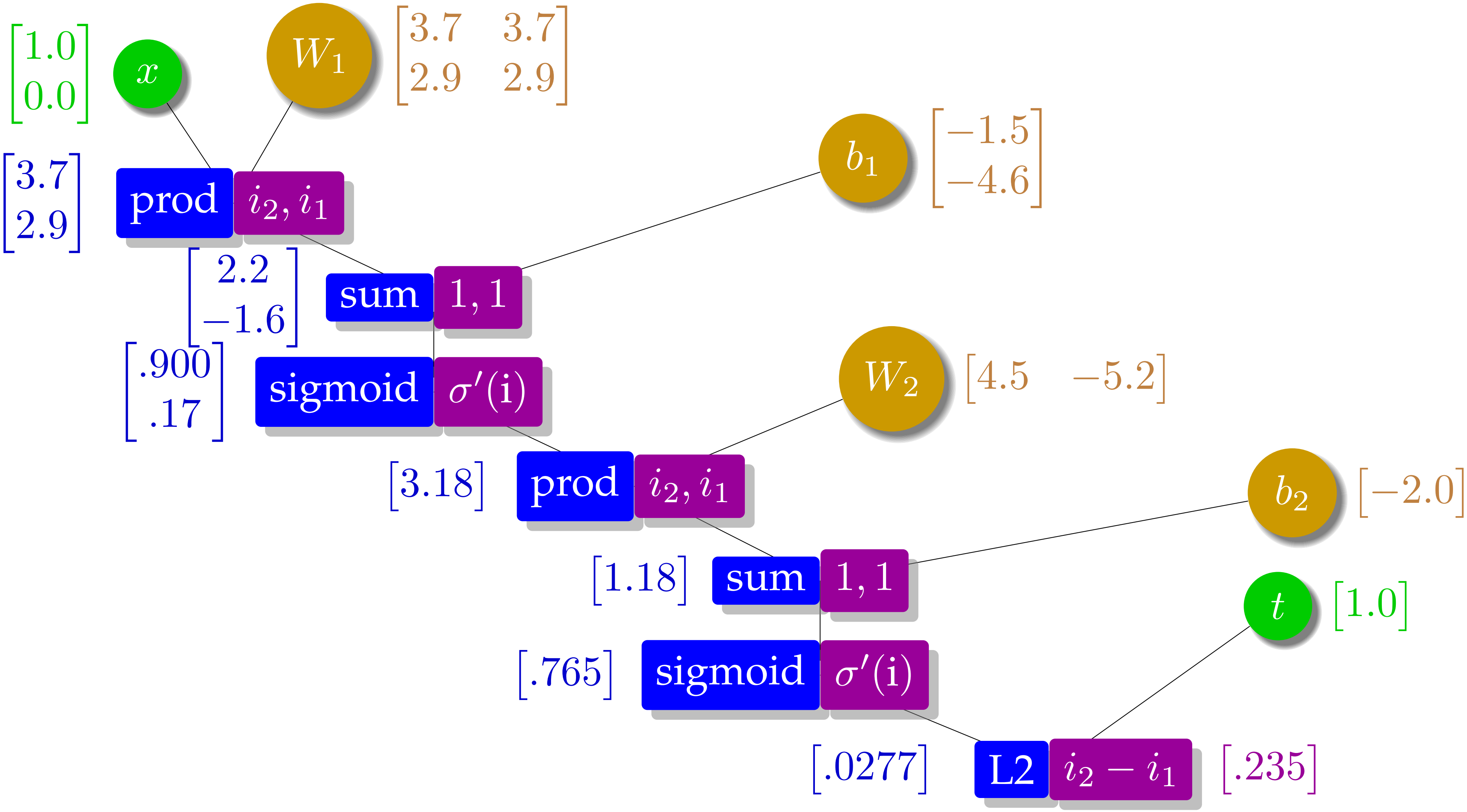
Derivatives for Each Node



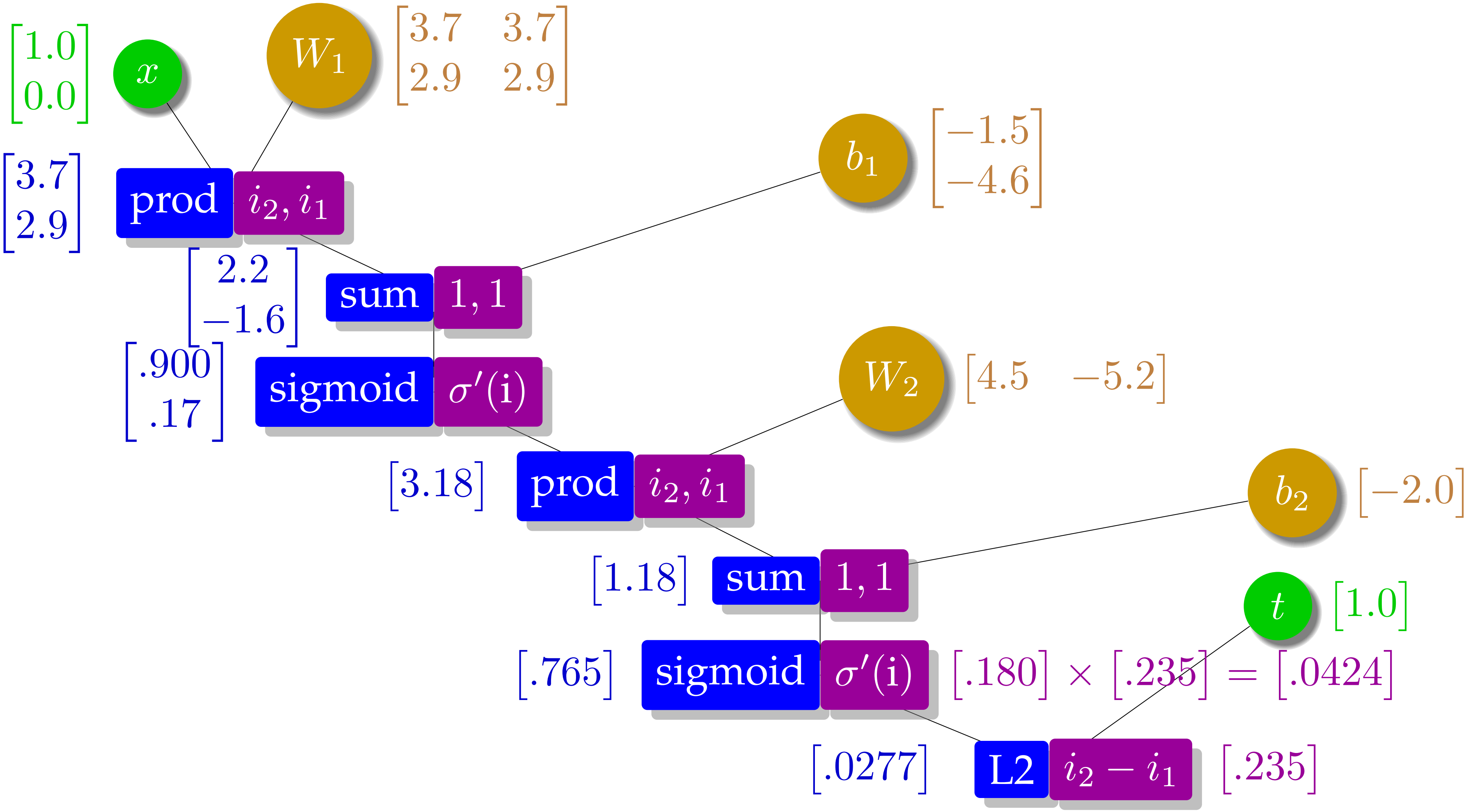
Derivatives for Each Node



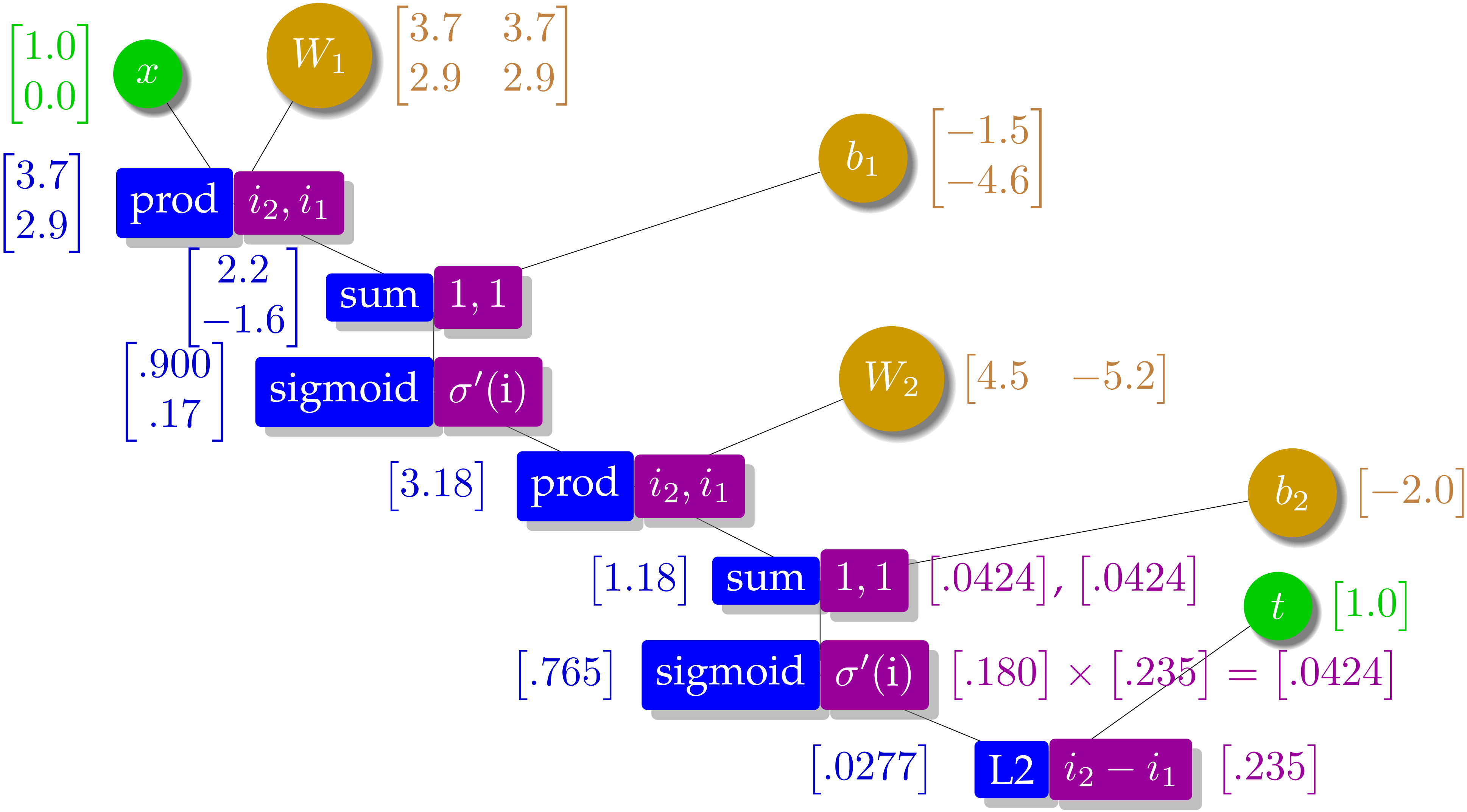
Backward Pass: Derivative Computation



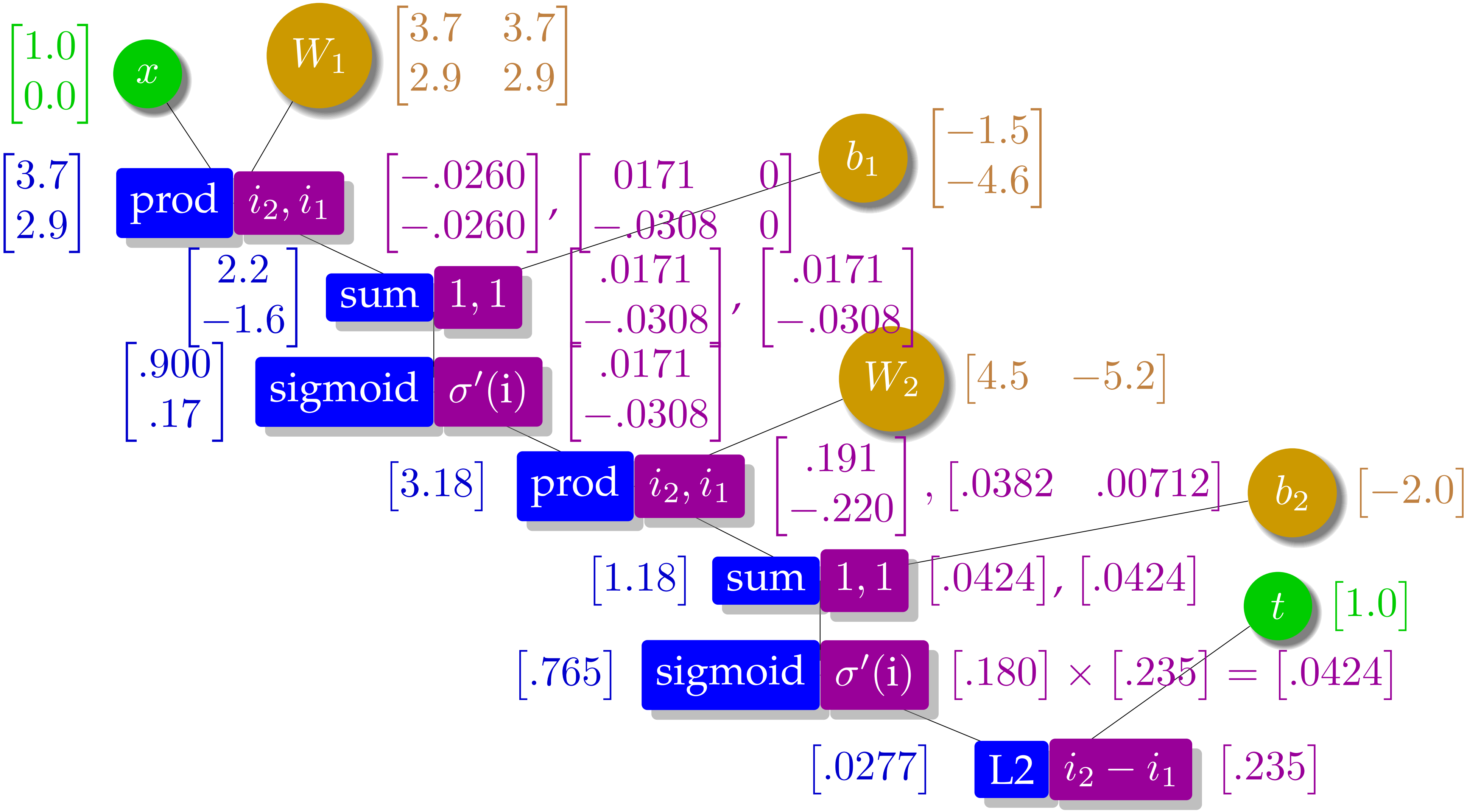
Backward Pass: Derivative Computation



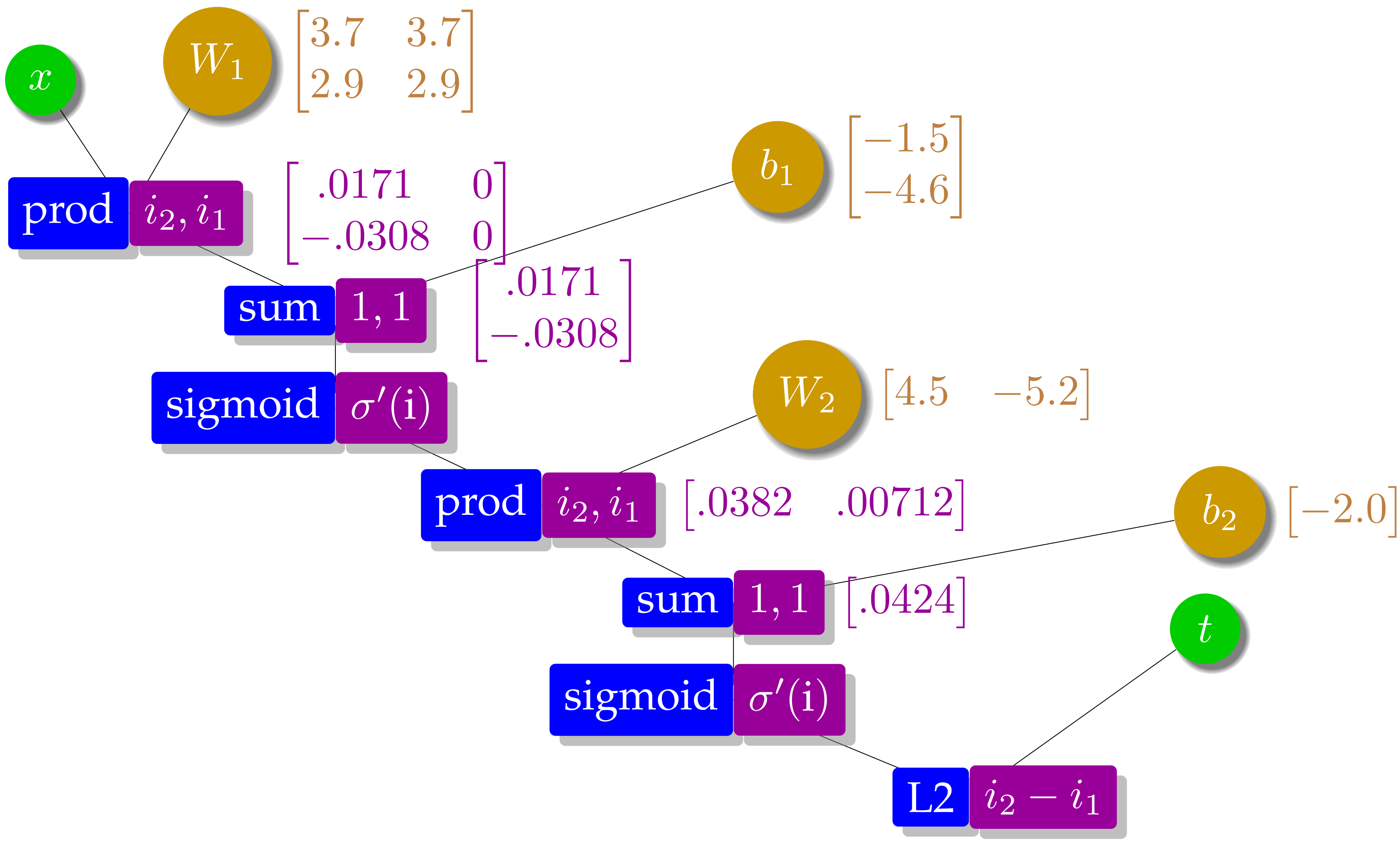
Backward Pass: Derivative Computation



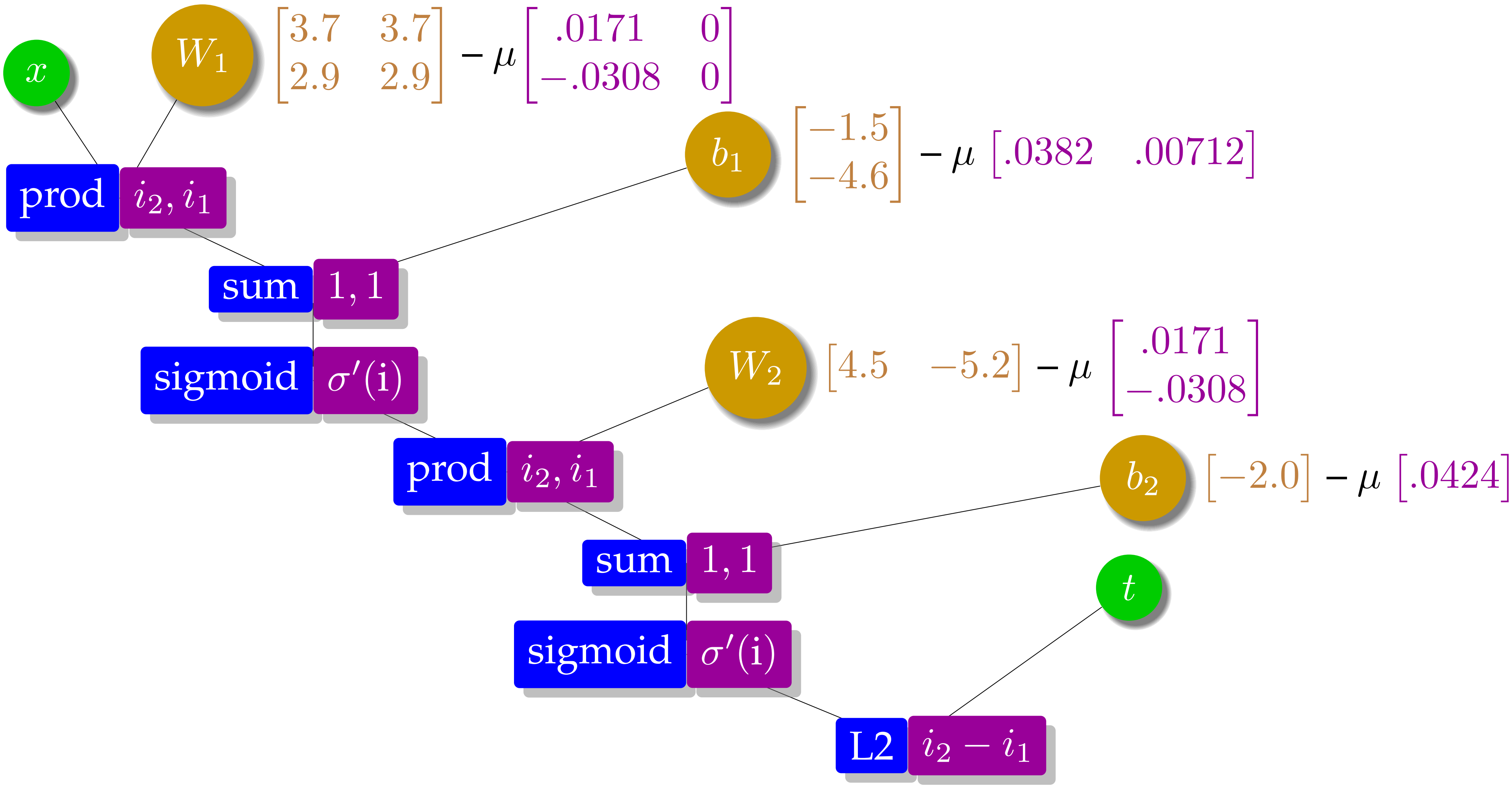
Backward Pass: Derivative Computation



Gradients for Parameter Update



Parameter Update



toolkits

Explosion of Deep Learning Toolkits

- University of Montreal: Theano (early, now defunct)
- Google: Tensorflow
- Facebook: Torch, pyTorch
- Microsoft: CNTK
- Amazon: MX-Net
- CMU: Dynet
- AMU/Edinburgh/Microsoft: Marian
- ... and many more

Toolkits

- Machine learning architectures around computations graphs very powerful
  - define a computation graph
  - provide data and a training strategy (e.g., batching)
  - toolkit does the rest
  - seamless support of GPUs

Example: PyTorch

- Installation

```
pip install torch
```

- Usage

```
import torch
```

Some Data Types

- PyTorch data type for parameter vectors, matrices etc., called `torch.tensor`

```
W = torch.tensor([[3,4],[2,3]], requires_grad=True, dtype=torch.float)
b = torch.tensor([-2,-4], requires_grad=True, dtype=torch.float)
W2 = torch.tensor([5,-5], requires_grad=True, dtype=torch.float)
b2 = torch.tensor([-2], requires_grad=True, dtype=torch.float)
```

- Definition of variables includes
  - specification of their basic data type (`float`)
  - indication to compute gradients (`requires_grad=True`)

- Input and output

```
x = torch.tensor([1,0], dtype=torch.float)
t = torch.tensor([1], dtype=torch.float)
```

Computation Graph

- Computation graph

```
s = W.mv(x) + b
h = torch.nn.Sigmoid()(s)

z = torch.dot(W2, h) + b2
y = torch.nn.Sigmoid()(z)

error = 1/2 * (t - z) ** 2
```

- Note
  - PyTorch sigmoid function `torch.nn.Sigmoid()`
  - multiplication between matrix `W` and vector `x` is `mv`
  - multiplication between two vectors `W2` and `h` is `torch.dot`.

Backward Computation

- Here it is:

```
error.backward()
```

- No need to derive gradients — all is done automatically

- We can look up computed gradients

```
>>> W2.grad
tensor([-0.0360, -0.0059])
```

- Note
  - when you run this code multiple times, then gradients accumulate
  - reset them with, e.g., `W2.grad.data.zero_()`

Training Data

- Our training set consists of the four examples of binary XOR operations.

x	**y**	$\mathbf{x} \oplus \mathbf{y}$
0	0	0
0	1	1
1	0	1
1	1	0

- Placed into array

```
training_data =
    [ [ torch.tensor([0.,0.]), torch.tensor([0.])  ],
      [ torch.tensor([1.,0.]), torch.tensor([1.])  ],
      [ torch.tensor([0.,1.]), torch.tensor([1.])  ],
      [ torch.tensor([1.,1.]), torch.tensor([0.])  ]  ]
```

Training Loop: Forward


```
mu = 0.1

for epoch in range(1000):
  total_error = 0

  for item in training_data:
    x = item[0]
    t = item[1]

    # forward computation
    s = W.mv(x) + b
    h = torch.nn.Sigmoid()(s)
    z = torch.dot(W2, h) + b2
    y = torch.nn.Sigmoid()(z)
    error = 1/2 * (t - y) ** 2
    total_error = total_error + error
```

Training Loop: Backward and Updates


```
    # backward computation
    error.backward()

    # weight updates
    W.data  = W  - mu * W.grad.data
    b.data  = b  - mu * b.grad.data
    W2.data = W2 - mu * W2.grad.data
    b2.data = b2 - mu * b2.grad.data

    W.grad.data.zero_()
    b.grad.data.zero_()
    W2.grad.data.zero_()
    b2.grad.data.zero_()

print("error: ", total_error/4)
```

Batch Training

- We computed gradients for each training example, update model immediately
- More common: process examples in batches, update after batch processed
- Instead

```
error.backward()
```

- Run back-propagation on accumulated error

```
total_error.backward()
```

Training Data Batch


```
x = torch.tensor([ [0.,0.], [1.,0.], [0.,1.], [1.,1.]  ])
t = torch.tensor([ 0., 1., 1., 0.  ])
```

- Change to computation graph (input now a matrix, output a vector)

```
s = x.mm(W) + b
h = torch.nn.Sigmoid()(s)
z = h.mv(W2) + b2
y = torch.nn.Sigmoid()(z)
```

- Convert error vector into single number

```
error = 1/2 * (t - y) ** 2
mean_error = error.mean()
mean_error.backward()
```

Parameter Updates (Optimizer)

- Our code has explicit parameter update computations

```
# weight updates
W.data  = W  - mu * W.grad.data
b.data  = b  - mu * b.grad.data
W2.data = W2 - mu * W2.grad.data
b2.data = b2 - mu * b2.grad.data
```

- But fancier optimizers are typically used (Adam, etc.)

- This requires more complex implementation

torch.nn.Module

- Neural network model is defined as class derived from `torch.nn.Module`

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

  def __init__(self):
    super(ExampleNet, self).__init__()
    self.layer1 = torch.nn.Linear(2,2)
    self.layer2 = torch.nn.Linear(2,1)
    self.layer1.weight = torch.nn.Parameter(torch.tensor([[3.,2.],[4.,3.]]))
    self.layer1.bias = torch.nn.Parameter(torch.tensor([-2.,-4.]))
    self.layer2.weight = torch.nn.Parameter(torch.tensor([[5.,-5.]]))
    self.layer2.bias = torch.nn.Parameter(torch.tensor([-2.]))

  def forward(self, x):
    s = self.layer1(x)
    h = torch.nn.Sigmoid()(s)
    z = self.layer2(h)
    y = torch.nn.Sigmoid()(z)
    return y
```

Optimizer Definition

- Instantiation of neural network object

```
net = ExampleNet()
```

- Optimizer definition

```
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
```

Training Loop

```
for iteration in range(1000):
  optimizer.zero_grad()
  out = net.forward( x )
  error = 1/2 * (t - out) ** 2
  mean_error = error.mean()
  print("error:  ",mean_error.data)
  mean_error.backward()
  optimizer.step()
```

code available on web page for textbook

http://www.statmt.org/nmt-book/