Deep Learning

Philipp Koehn

11 April 2024

Supervised Learning

- Examples described by attribute values (Boolean, discrete, continuous, etc.)
- E.g., situations where I will/won't wait for a table:

Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*T*	*F*	*F*	*T*	*Some*	*$$$*	*F*	*T*	*French*	*0–10*	*T*
X_2	*T*	*F*	*F*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*30–60*	*F*
X_3	*F*	*T*	*F*	*F*	*Some*	*$*	*F*	*F*	*Burger*	*0–10*	*T*
X_4	*T*	*F*	*T*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*10–30*	*T*
X_5	*T*	*F*	*T*	*F*	*Full*	*$$$*	*F*	*T*	*French*	*>60*	*F*
X_6	*F*	*T*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Italian*	*0–10*	*T*
X_7	*F*	*T*	*F*	*F*	*None*	*$*	*T*	*F*	*Burger*	*0–10*	*F*
X_8	*F*	*F*	*F*	*T*	*Some*	*$$*	*T*	*T*	*Thai*	*0–10*	*T*
X_9	*F*	*T*	*T*	*F*	*Full*	*$*	*T*	*F*	*Burger*	*>60*	*F*
X_{10}	*T*	*T*	*T*	*T*	*Full*	*$$$*	*F*	*T*	*Italian*	*10–30*	*F*
X_{11}	*F*	*F*	*F*	*F*	*None*	*$*	*F*	*F*	*Thai*	*0–10*	*F*
X_{12}	*T*	*T*	*T*	*T*	*Full*	*$*	*F*	*F*	*Burger*	*30–60*	*T*

- Classification of examples is positive (T) or negative (F)

Naive Bayes Models

- Bayes rule

$$p(C|\mathbf{A}) = \frac{1}{Z}\, p(\mathbf{A}|C)\; p(C)$$

- Independence assumption

$$\begin{aligned} p(\mathbf{A}|C) &= p(a_1, a_2, a_3, ..., a_n|C) \\ &\simeq \prod_i p(a_i|C) \end{aligned}$$

- Weights

$$p(\mathbf{A}|C) = \prod_i p(a_i|C)^{\lambda_i}$$

Naive Bayes Models

- Linear model

$$
\begin{aligned}
p(\mathbf{A}|C) &= \prod_i p(a_i|C)^{\lambda_i} \\
&= \exp \sum_i \lambda_i \log p(a_i|C)
\end{aligned}
$$

- Probability distribution as features

$$
\begin{aligned}
h_i(\mathbf{A}, C) &= \log p(a_i|C) \\
h_0(\mathbf{A}, C) &= \log p(C)
\end{aligned}
$$

- Linear model with features

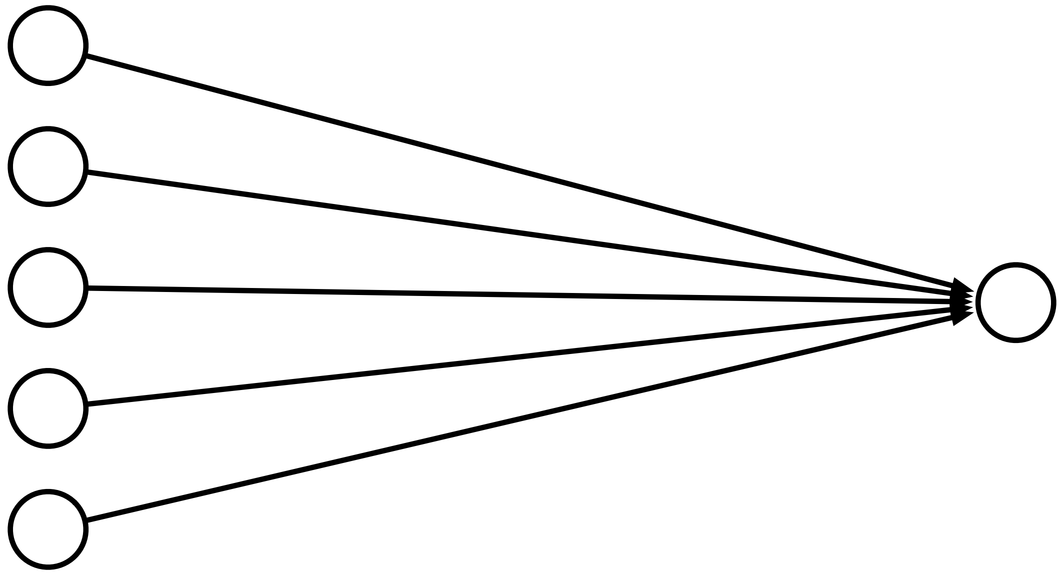
$$
p(C|\mathbf{A}) \propto \sum_i \lambda_i \, h_i(\mathbf{A}, C)
$$

Linear Model

- Weighted linear combination of feature values h_j and weights λ_j for example $\mathbf{d}_i$

$$\text{score}(\lambda, \mathbf{d}_i) = \sum_j \lambda_j \, h_j(\mathbf{d}_i)$$

- Such models can be illustrated as a ”network”



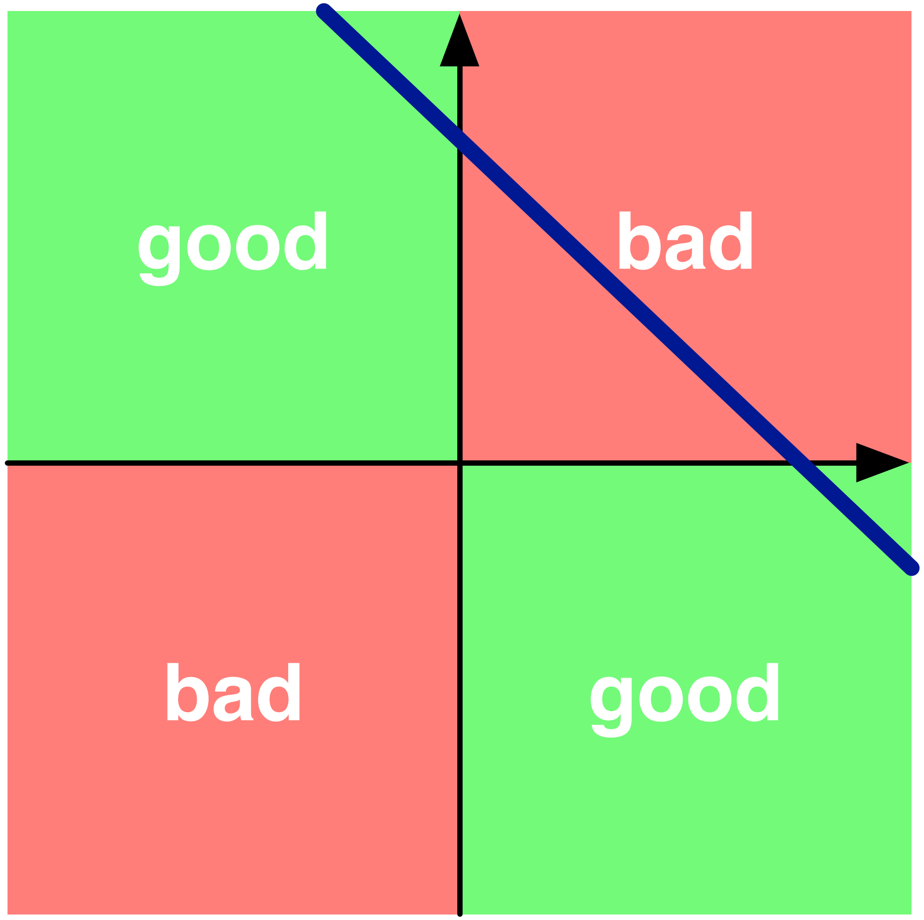
Limits of Linearity

- We can give each feature a weight

- But not more complex value relationships, e.g,
  - there is one one critical range for a value (non-linear impact)
  - interactions between multiple features

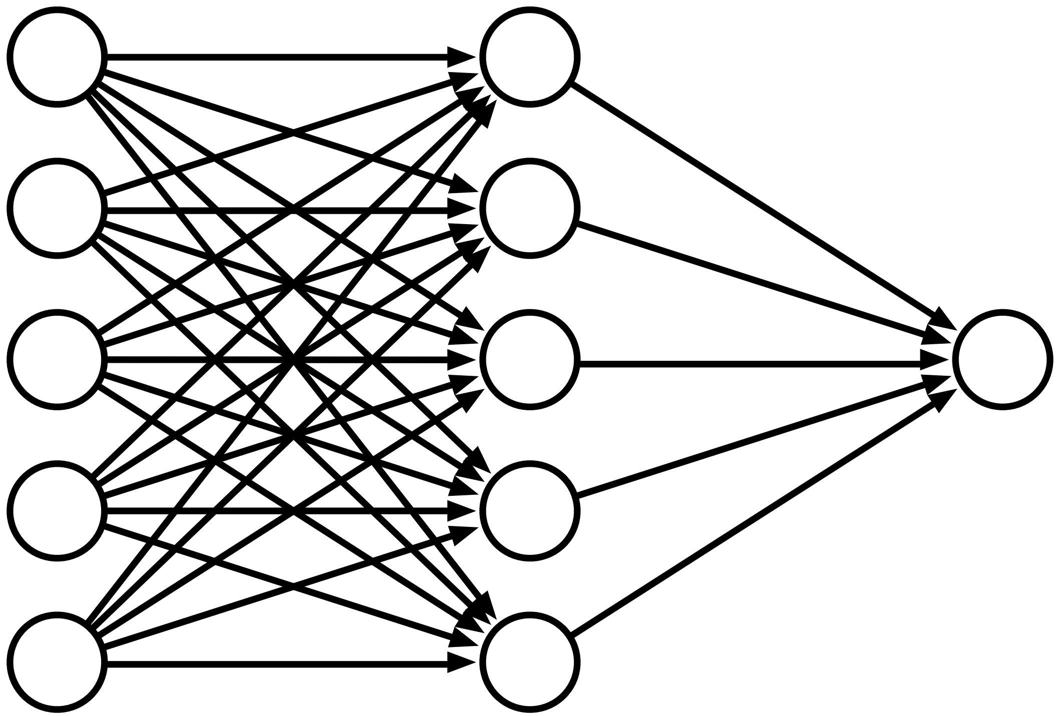
XOR

- Linear models cannot model XOR



Multiple Layers

- Add an intermediate ("hidden") layer of processing
  (each arrow is a weight)



- Have we gained anything so far?

Non-Linearity

- Instead of computing a linear combination

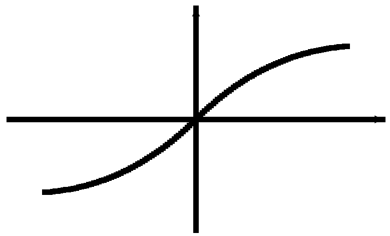
$$\text{score}(\lambda, \mathbf{d}_i) = \sum_j \lambda_j \, h_j(\mathbf{d}_i)$$

- Add a non-linear function

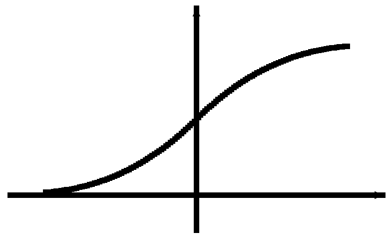
$$\text{score}(\lambda, \mathbf{d}_i) = f\Big(\sum_j \lambda_j \, h_j(\mathbf{d}_i) \Big)$$

- Popular choices

tanh(x)



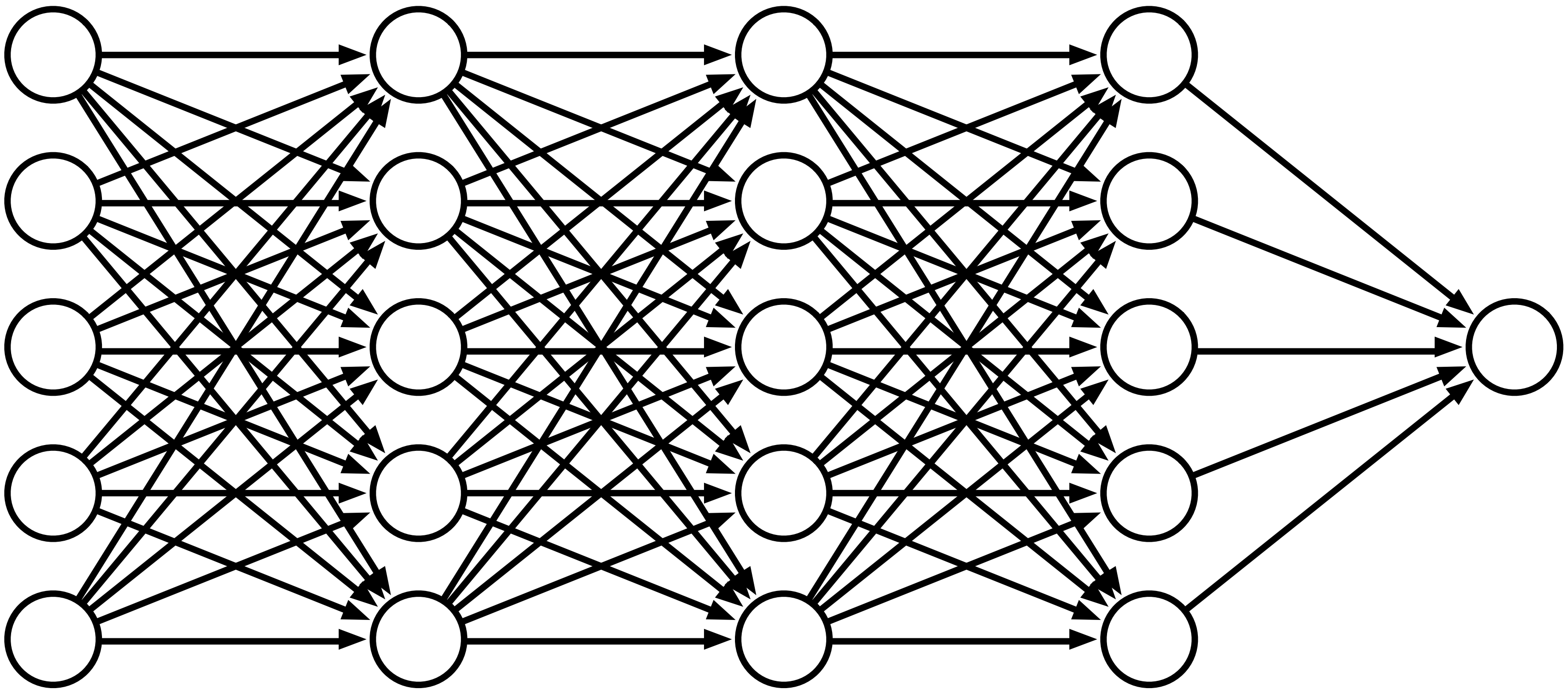
sigmoid(x) = $\frac{1}{1+e^{-x}}$



(sigmoid is also called the "logistic function")

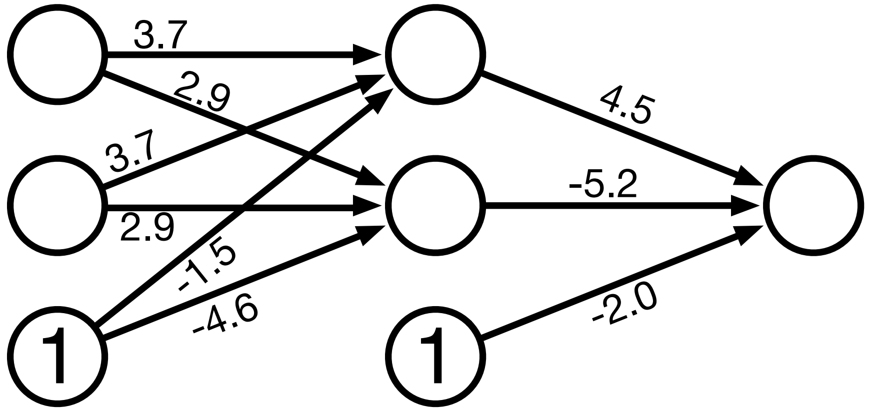
Deep Learning

- More layers = deep learning



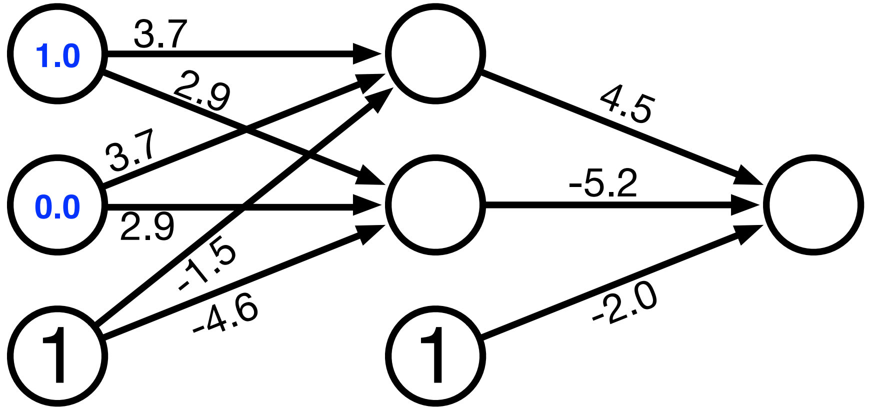
example

Simple Neural Network



- One innovation: bias units (no inputs, always value 1)

Sample Input

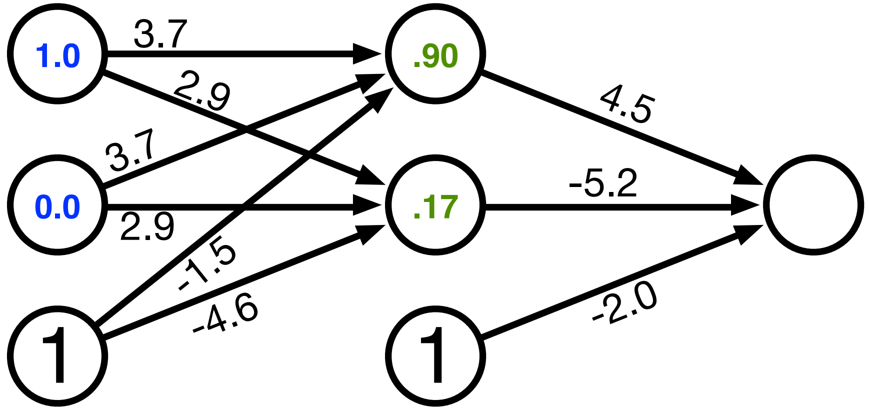


- Try out two input values
- Hidden unit computation

$$\text{sigmoid}(1.0 \times 3.7 + 0.0 \times 3.7 + 1 \times -1.5) = \text{sigmoid}(2.2) = \frac{1}{1 + e^{-2.2}} = 0.90$$

$$\text{sigmoid}(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times -4.5) = \text{sigmoid}(-1.6) = \frac{1}{1 + e^{1.6}} = 0.17$$

Computed Hidden

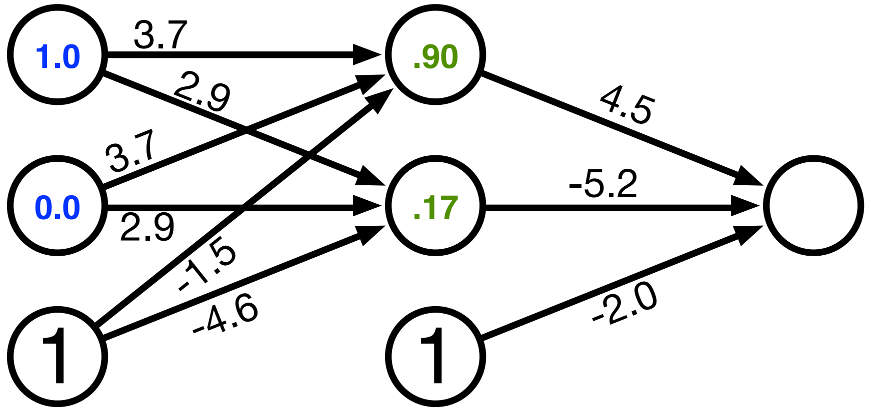


- Try out two input values
- Hidden unit computation

$$\text{sigmoid}(1.0 \times 3.7 + 0.0 \times 3.7 + 1 \times -1.5) = \text{sigmoid}(2.2) = \frac{1}{1+e^{-2.2}} = 0.90$$

$$\text{sigmoid}(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times -4.5) = \text{sigmoid}(-1.6) = \frac{1}{1+e^{1.6}} = 0.17$$

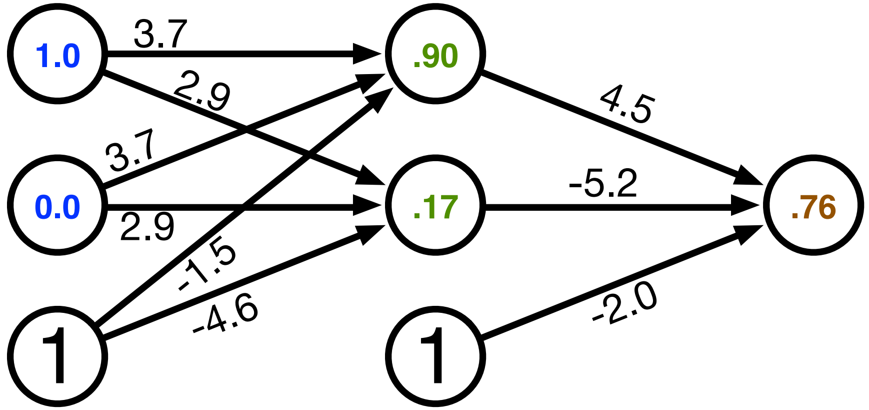
Compute Output



- Output unit computation

$$\text{sigmoid}(.90 \times 4.5 + .17 \times -5.2 + 1 \times -2.0) = \text{sigmoid}(1.17) = \frac{1}{1+e^{-1.17}} = 0.76$$

Computed Output



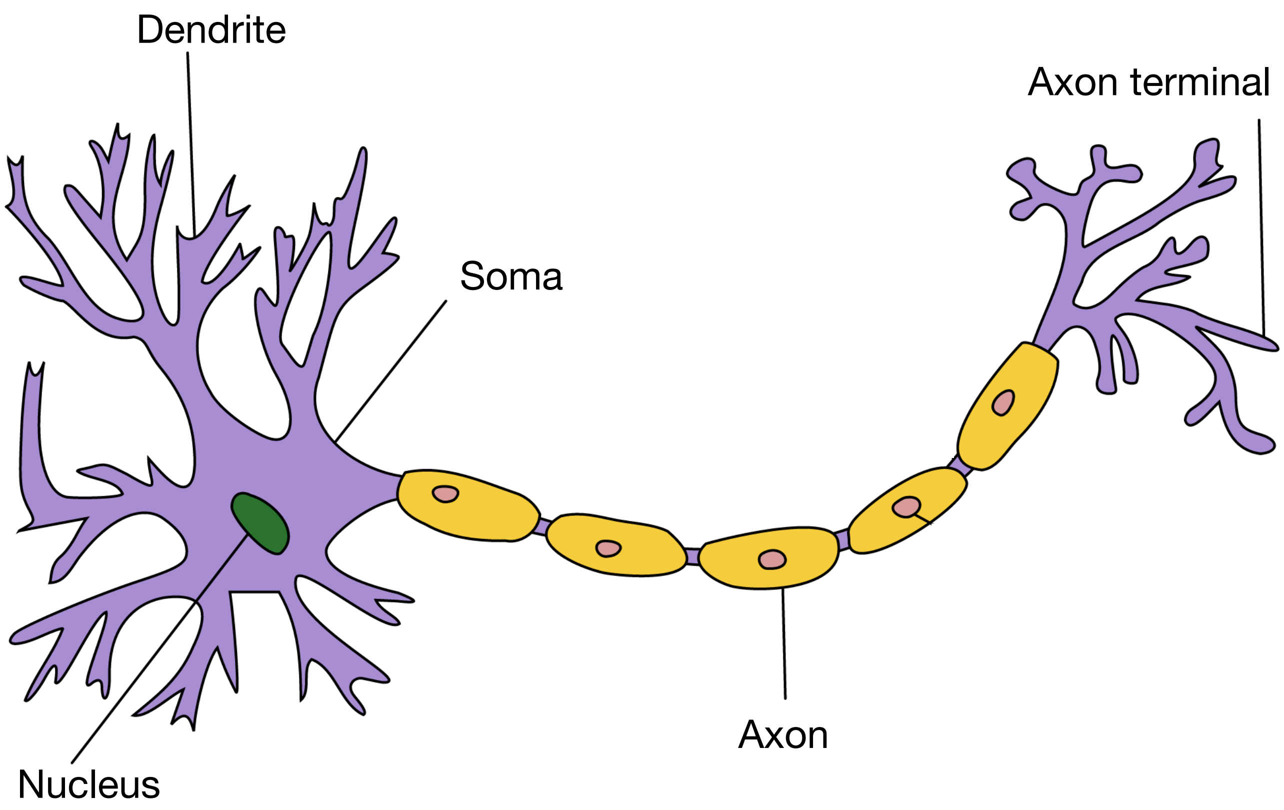
- Output unit computation

$$\text{sigmoid}(.90 \times 4.5 + .17 \times -5.2 + 1 \times -2.0) = \text{sigmoid}(1.17) = \frac{1}{1 + e^{-1.17}} = 0.76$$

”neural” networks

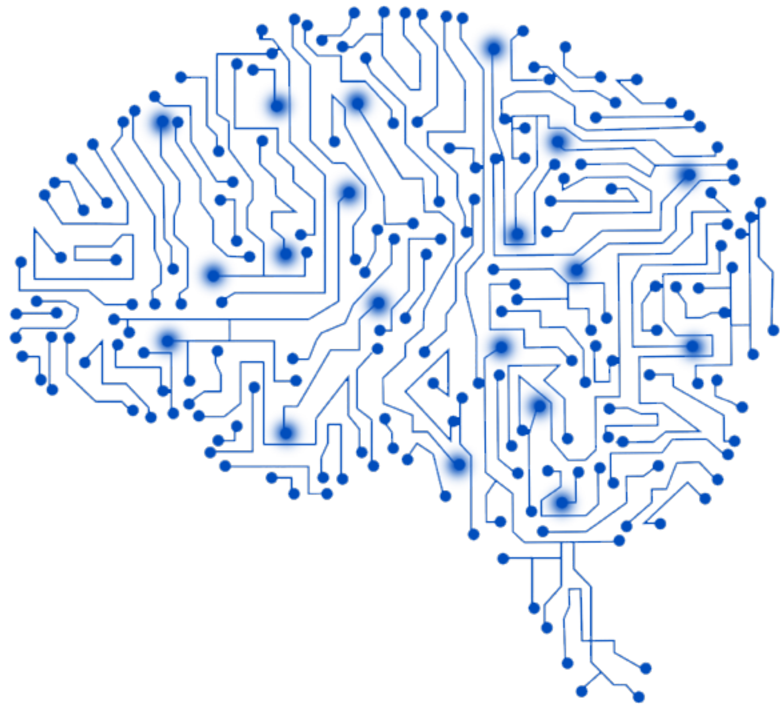
Neuron in the Brain

- The human brain is made up of about 100 billion neurons



- Neurons receive electric signals at the dendrites and send them to the axon

The Brain vs. Artificial Neural Networks

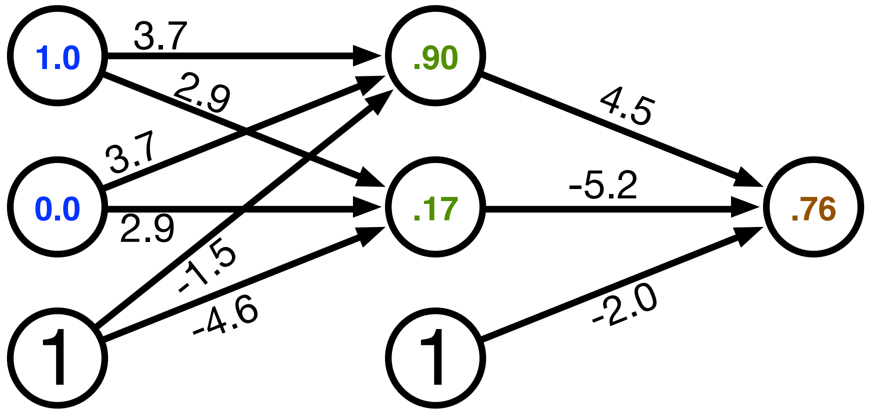


- Similarities
  - Neurons, connections between neurons
  - Learning = change of connections, not change of neurons
  - Massive parallel processing

- But artificial neural networks are much simpler
  - computation within neuron vastly simplified
  - discrete time steps
  - typically some form of supervised learning with massive number of stimuli

back-propagation training

Error



- Computed output: $y = .76$
- Correct output: $t = 1.0$

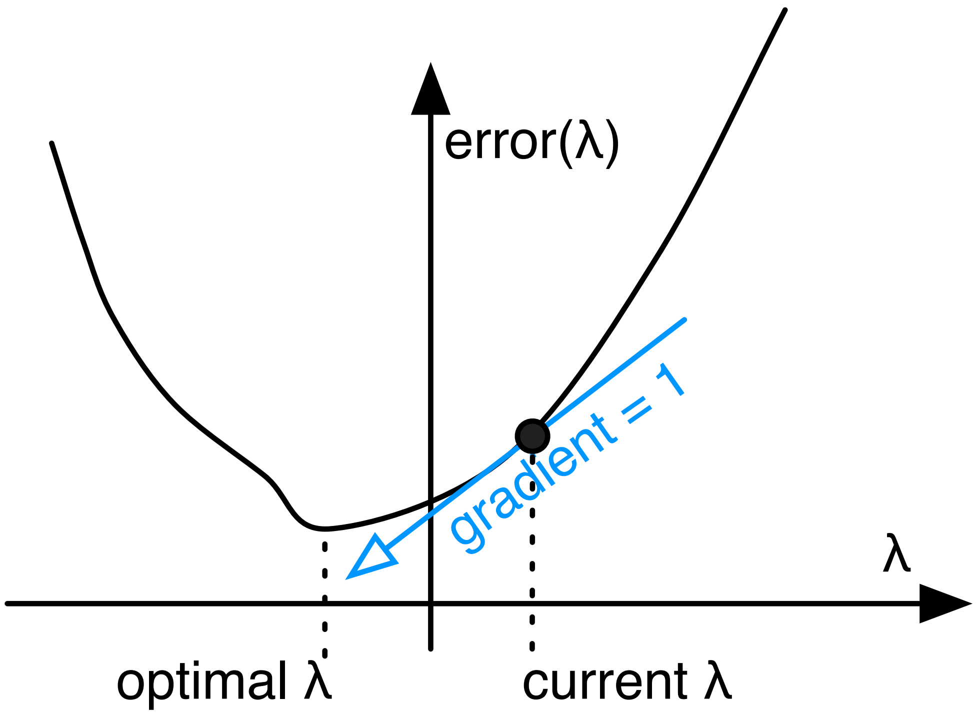
⇒ How do we adjust the weights?

Key Concepts

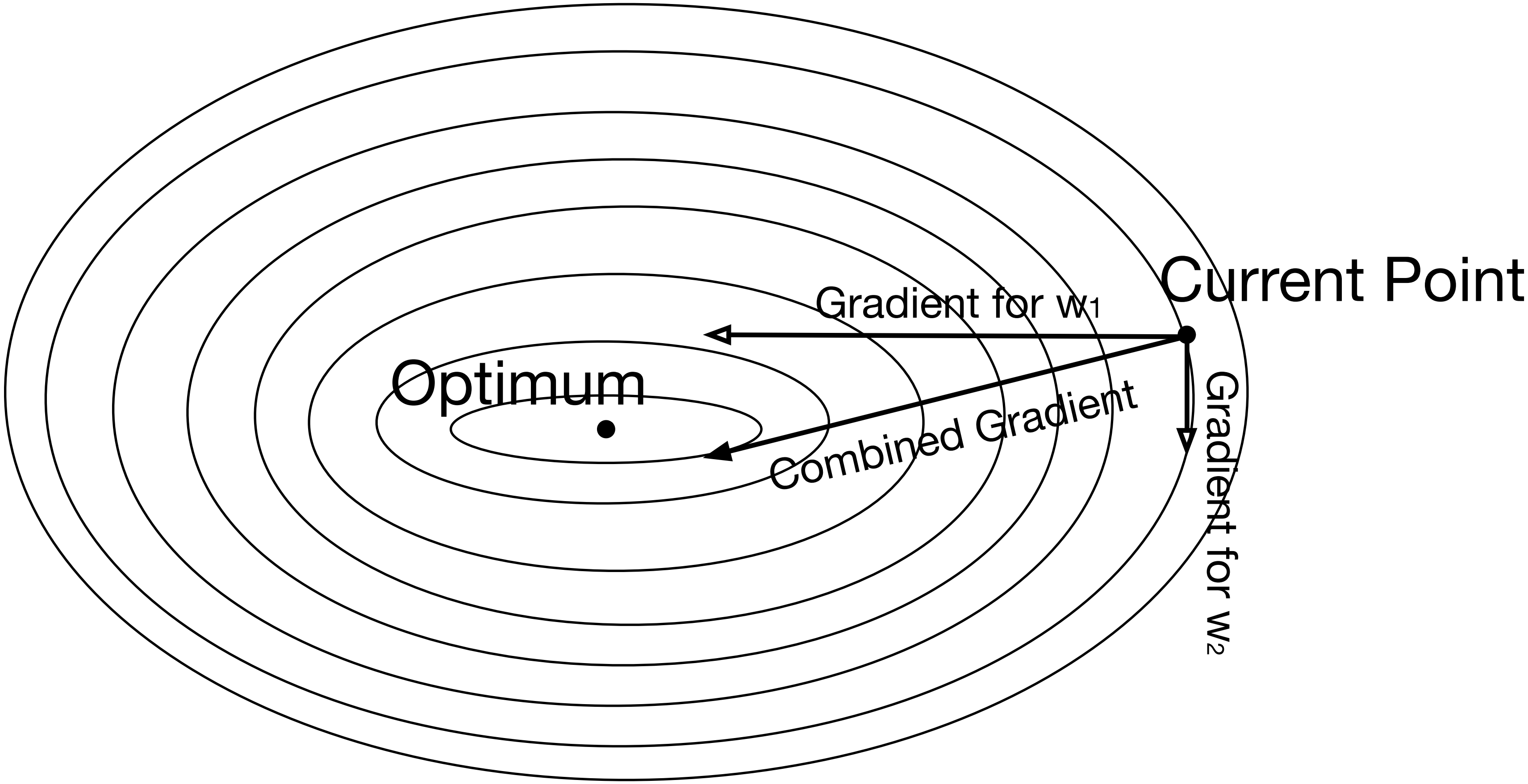
- Gradient descent
  - error is a function of the weights
  - we want to reduce the error
  - gradient descent: move towards the error minimum
  - compute gradient → get direction to the error minimum
  - adjust weights towards direction of lower error

- Back-propagation
  - first adjust last set of weights
  - propagate error back to each previous layer
  - adjust their weights

Gradient Descent



Gradient Descent

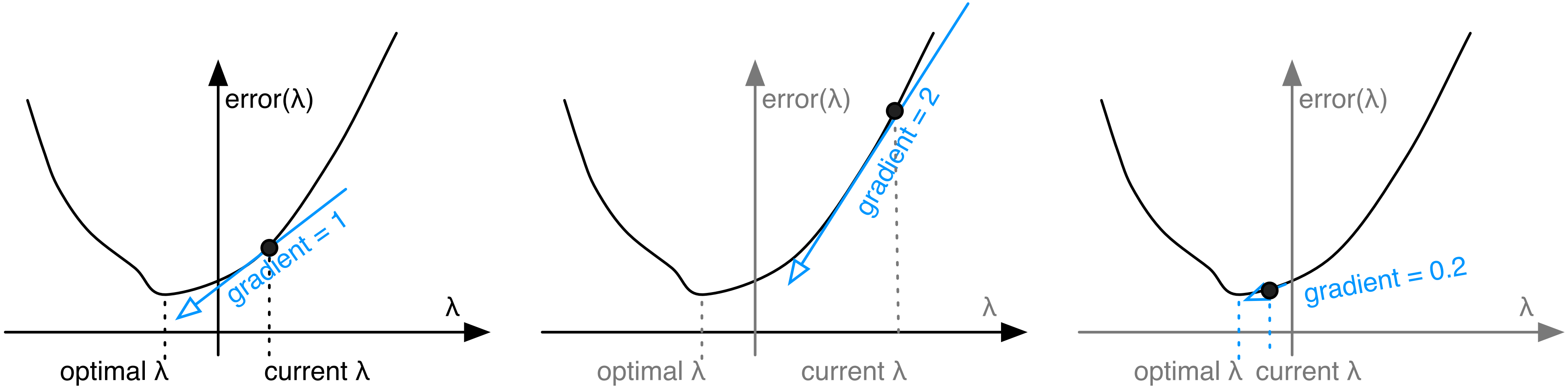


Gradient Descent

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

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

- We want to optimize error(λ) 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

Derivative of Sigmoid

- Sigmoid

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

- Reminder: quotient rule

$$\left(\frac{f(x)}{g(x)}\right)' = \frac{g(x)f'(x) - f(x)g'(x)}{g(x)^2}$$

- Derivative

$$\begin{aligned}
\frac{d\,\text{sigmoid}(x)}{dx} &= \frac{d}{dx}\frac{1}{1+e^{-x}} \\
&= \frac{0 \times (1-e^{-x}) - (-e^{-x})}{(1+e^{-x})^2} \\
&= \frac{1}{1+e^{-x}}\left(\frac{e^{-x}}{1+e^{-x}}\right) \\
&= \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right) \\
&= \text{sigmoid}(x)(1-\text{sigmoid}(x))
\end{aligned}$$

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

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

Final Layer Update (1)

- 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 E is defined with respect to y

$$\frac{dE}{dy} = \frac{d}{dy} \frac{1}{2}(t - y)^2 = -(t - y)$$

Final Layer Update (2)

- 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

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

- y with respect to x is $\text{sigmoid}(s)$

$$\frac{dy}{ds} = \frac{d\,\text{sigmoid}(s)}{ds} = \text{sigmoid}(s)(1 - \text{sigmoid}(s)) = y(1 - y)$$

Final Layer Update (3)

- 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

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

- x is weighted linear combination of hidden node values h_k

$$\frac{ds}{dw_k} = \frac{d}{dw_k}\sum_k w_k h_k = h_k$$

Putting it All Together

- Derivative of error with regard to one weight w_k

$$\frac{dE}{dw_k} = \frac{dE}{dy}\frac{dy}{ds}\frac{ds}{dw_k}$$
$$= -(t - y) \quad y(1 - y) \quad h_k$$

  - error
  - derivative of sigmoid: y'

- Weight adjustment will be scaled by a fixed learning rate μ

$$\Delta w_k = \mu \, (t - y) \, y' \, h_k$$

Hidden Layer Update

- In a hidden layer, we do not have a target output value
- But we can compute how much each node contributed to downstream error
- Definition of error term of each node

$$\delta_j = (t_j - y_j)\ y'_j$$

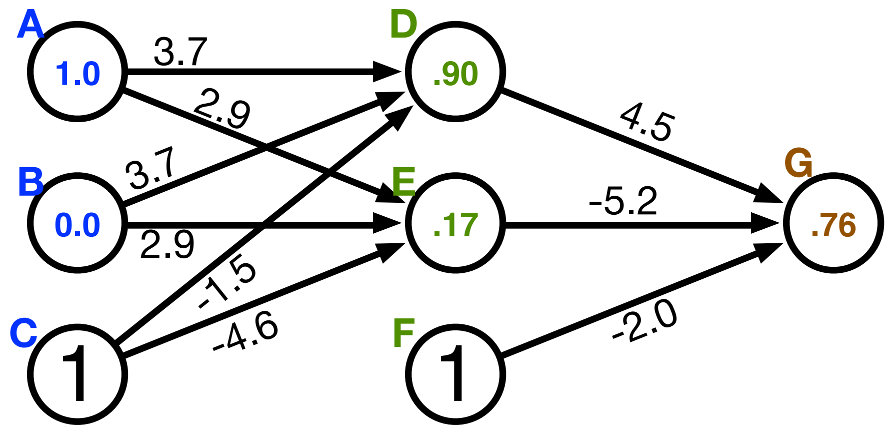
- Back-propagate the error term
  (why this way? there is math to back it up...)

$$\delta_i = \Big(\sum_j w_{j \leftarrow i} \delta_j \Big)\ y'_i$$

- Universal update formula

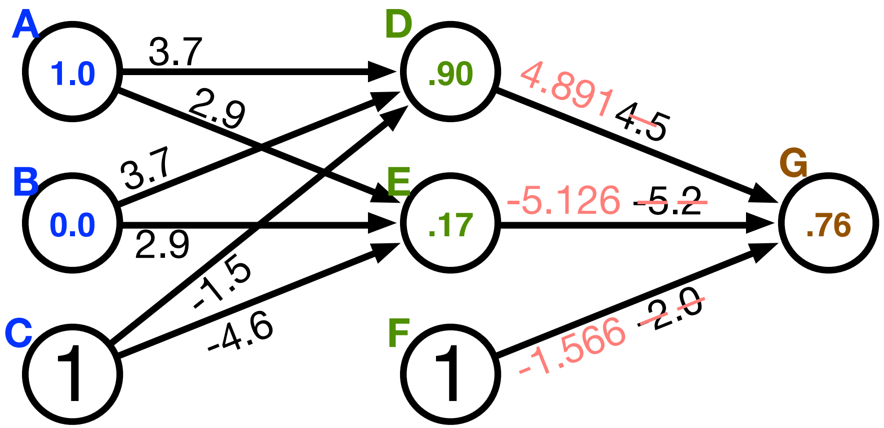
$$\Delta w_{j \leftarrow k} = \mu\ \delta_j\ h_k$$

Our Example



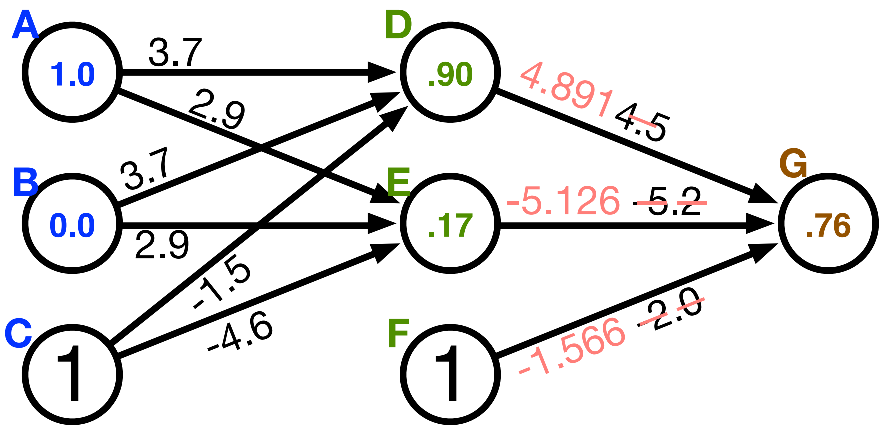
- Computed output: $y = .76$
- Correct output: $t = 1.0$
- Final layer weight updates (learning rate $\mu = 10$)
  - $\delta_\text{G} = (t - y)\ y' = (1 - .76)\ 0.181 = .0434$
  - $\Delta w_\text{GD} = \mu\ \delta_\text{G}\ h_\text{D} = 10 \times .0434 \times .90 = .391$
  - $\Delta w_\text{GE} = \mu\ \delta_\text{G}\ h_\text{E} = 10 \times .0434 \times .17 = .074$
  - $\Delta w_\text{GF} = \mu\ \delta_\text{G}\ h_\text{F} = 10 \times .0434 \times 1 = .434$

Our Example



- Computed output: $y = .76$
- Correct output: $t = 1.0$
- Final layer weight updates (learning rate $\mu = 10$)
  - $\delta_{\mathsf{G}} = (t - y)\, y' = (1 - .76)\, 0.181 = .0434$
  - $\Delta w_{\mathsf{GD}} = \mu\, \delta_{\mathsf{G}}\, h_{\mathsf{D}} = 10 \times .0434 \times .90 = .391$
  - $\Delta w_{\mathsf{GE}} = \mu\, \delta_{\mathsf{G}}\, h_{\mathsf{E}} = 10 \times .0434 \times .17 = .074$
  - $\Delta w_{\mathsf{GF}} = \mu\, \delta_{\mathsf{G}}\, h_{\mathsf{F}} = 10 \times .0434 \times 1 = .434$

Hidden Layer Updates



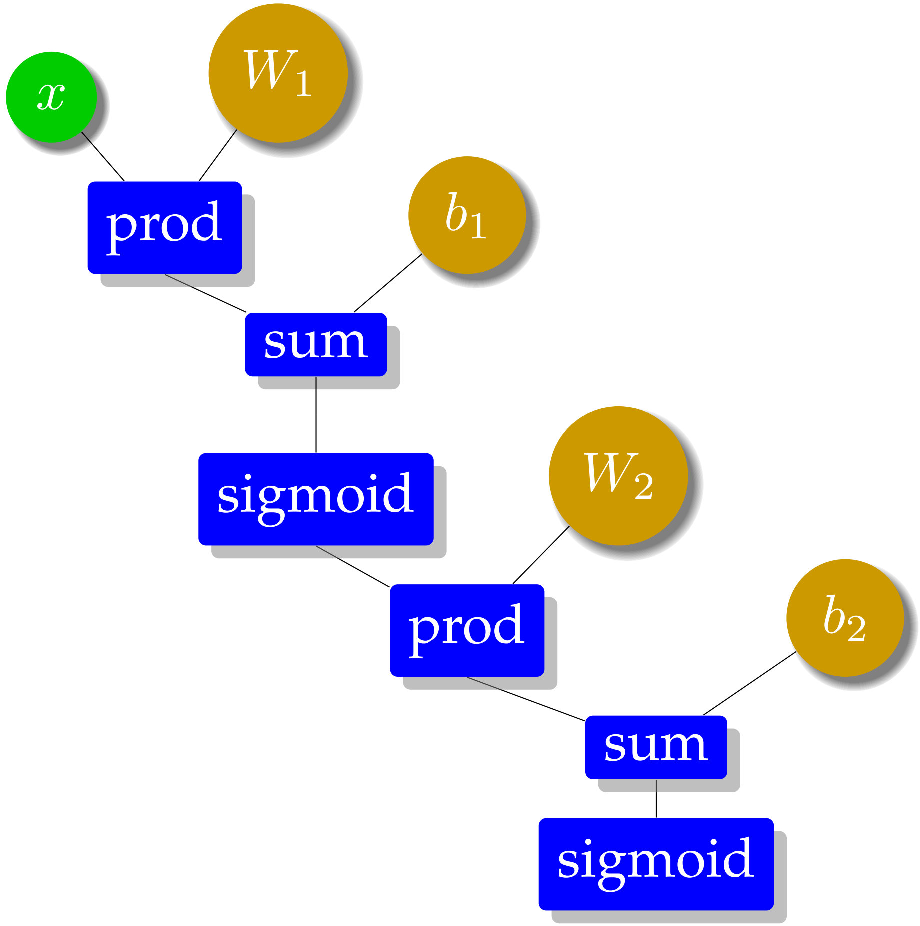
- Hidden node **D**
  - $\delta_{\text{D}} = \left(\sum_j w_{j \leftarrow i} \delta_j \right) y'_{\text{D}} = w_{\text{GD}}\ \delta_{\text{G}}\ y'_{\text{D}} = 4.5 \times .0434 \times .0898 = .0175$
  - $\Delta w_{\text{DA}} = \mu\ \delta_{\text{D}}\ h_{\text{A}} = 10 \times .0175 \times 1.0 = .175$
  - $\Delta w_{\text{DB}} = \mu\ \delta_{\text{D}}\ h_{\text{B}} = 10 \times .0175 \times 0.0 = 0$
  - $\Delta w_{\text{DC}} = \mu\ \delta_{\text{D}}\ h_{\text{C}} = 10 \times .0175 \times 1 = .175$
- Hidden node **E**
  - $\delta_{\text{E}} = \left(\sum_j w_{j \leftarrow i} \delta_j \right) y'_{\text{E}} = w_{\text{GE}}\ \delta_{\text{G}}\ y'_{\text{E}} = -5.2 \times .0434 \times 0.1411 = -.0318$
  - $\Delta w_{\text{EA}} = \mu\ \delta_{\text{E}}\ h_{\text{A}} = 10 \times -.0318 \times 1.0 = -.318$
  - etc.

computation graphs

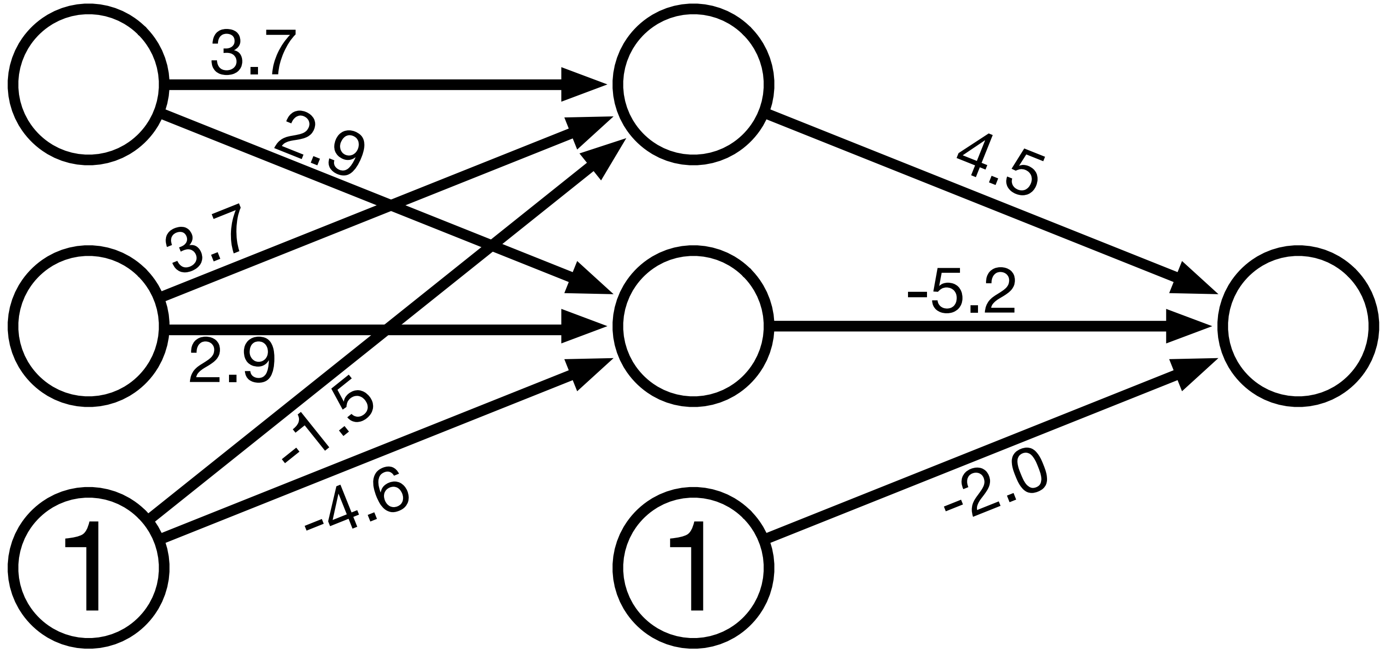
Vector and Matrix Multiplications

- Forward computation: $\vec{s} = W\vec{h}$
- Activation function: $\vec{y} = \text{sigmoid}(\vec{h})$
- Error term: $\vec{\delta} = (\vec{t} - \vec{y}) \ \text{sigmoid}'(\vec{s})$
- Propagation of error term: $\vec{\delta}_i = W\vec{\delta}_{i+1} \cdot \text{sigmoid}'(\vec{s})$
- Weight updates: $\Delta W = \mu\vec{\delta}\vec{h}^T$

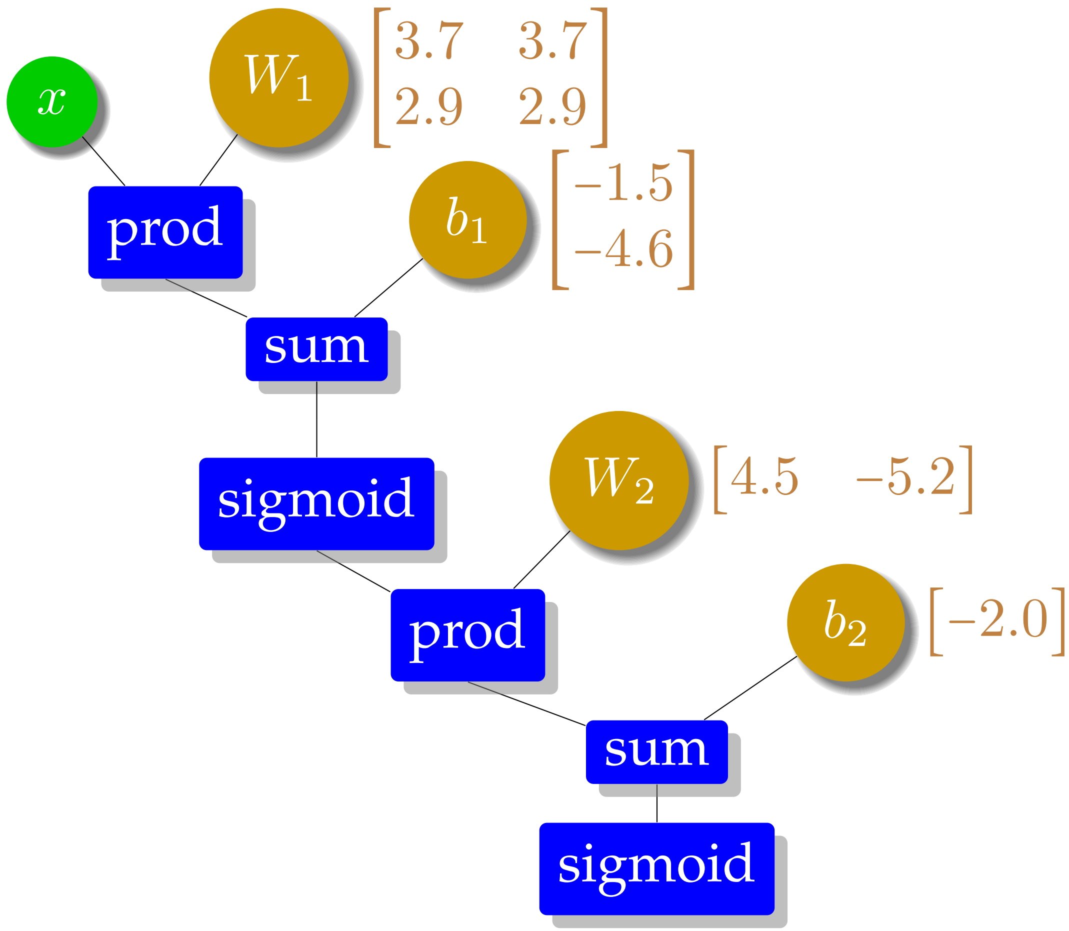
Computation Graph



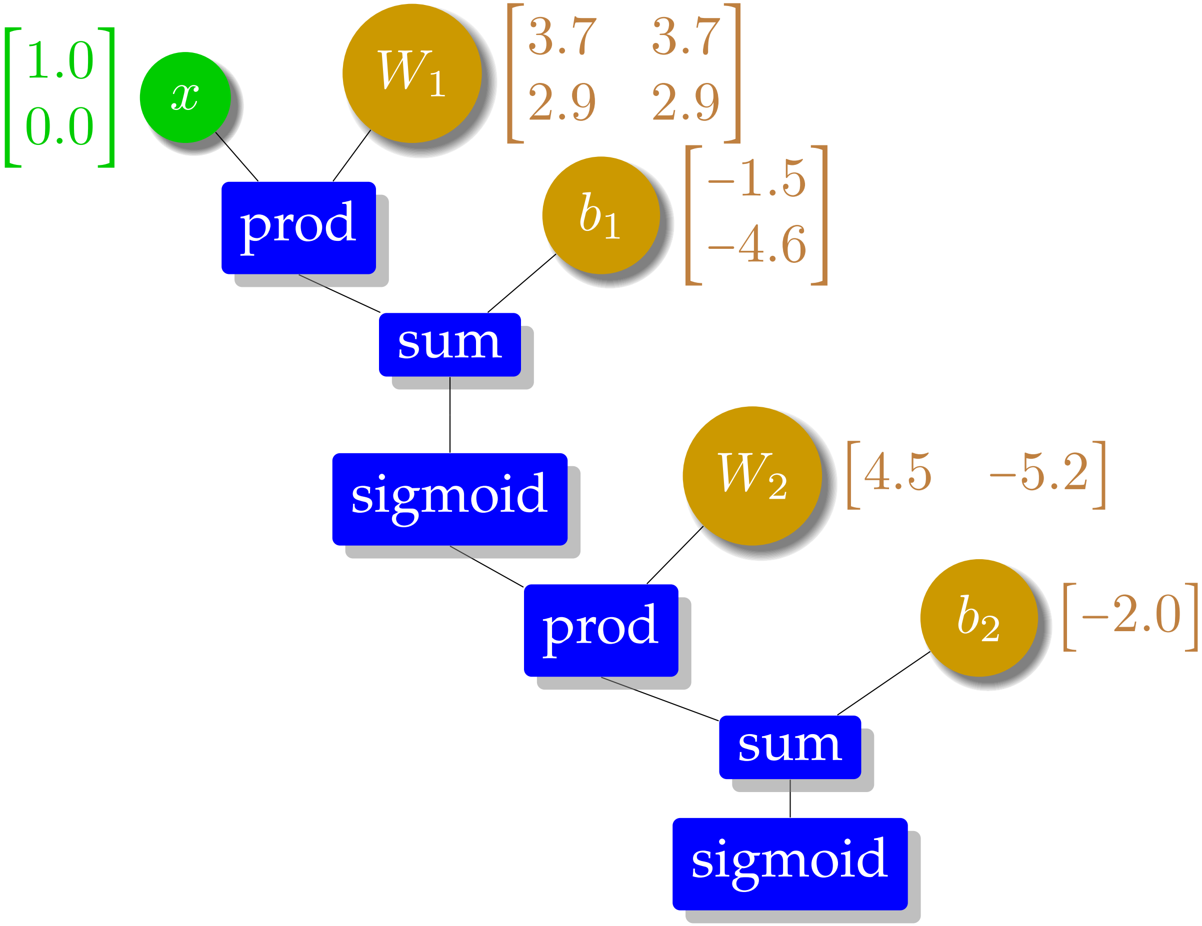
Simple Neural Network



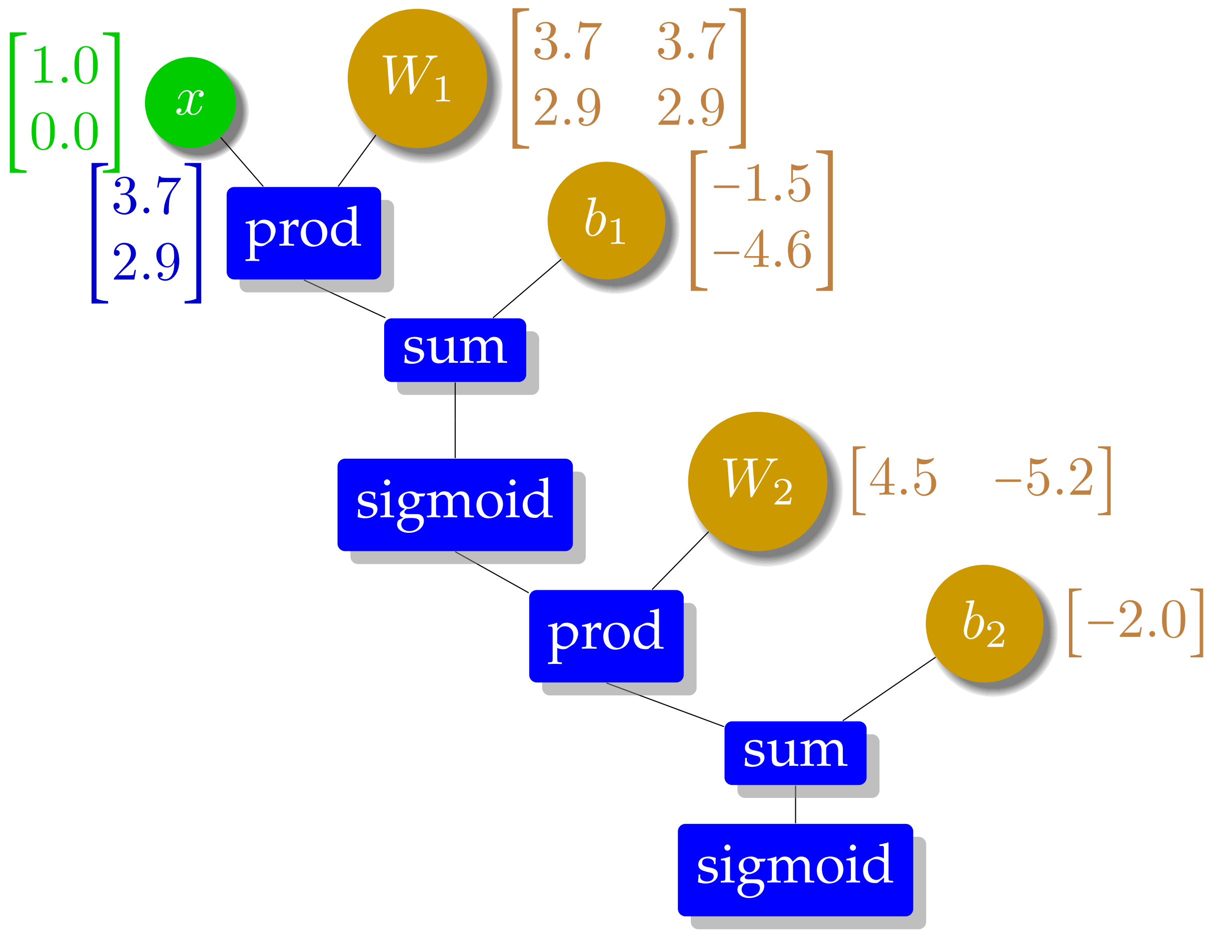
Computation Graph



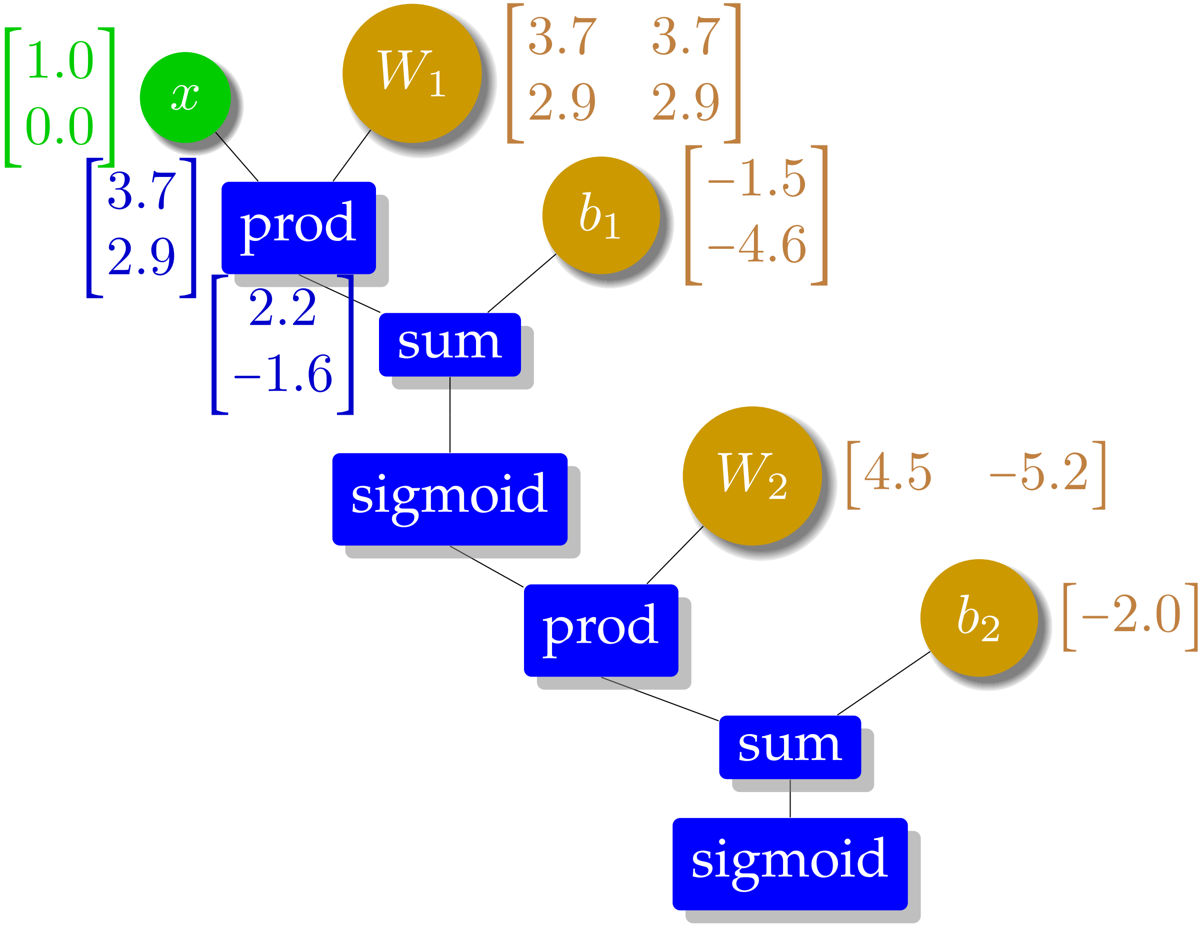
Processing Input



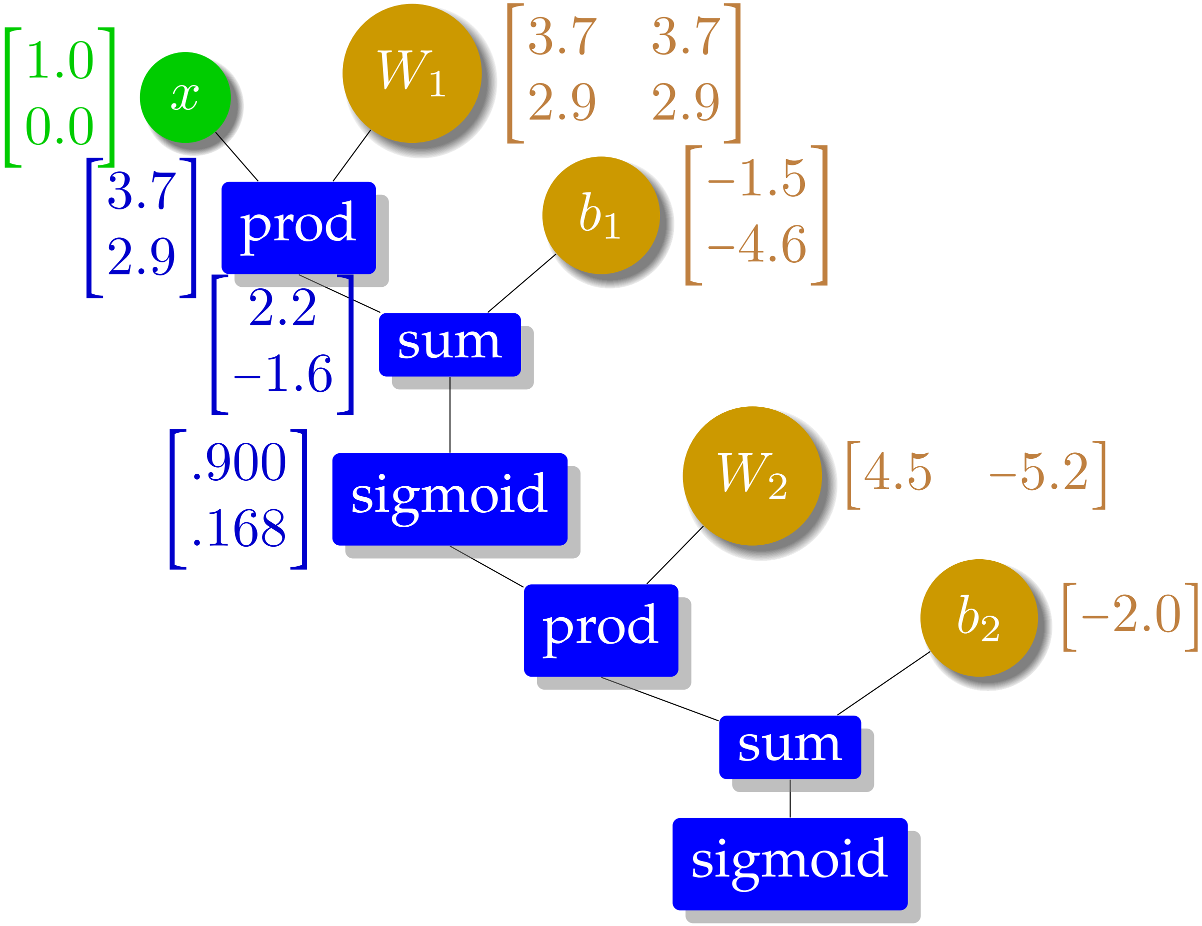
Processing Input



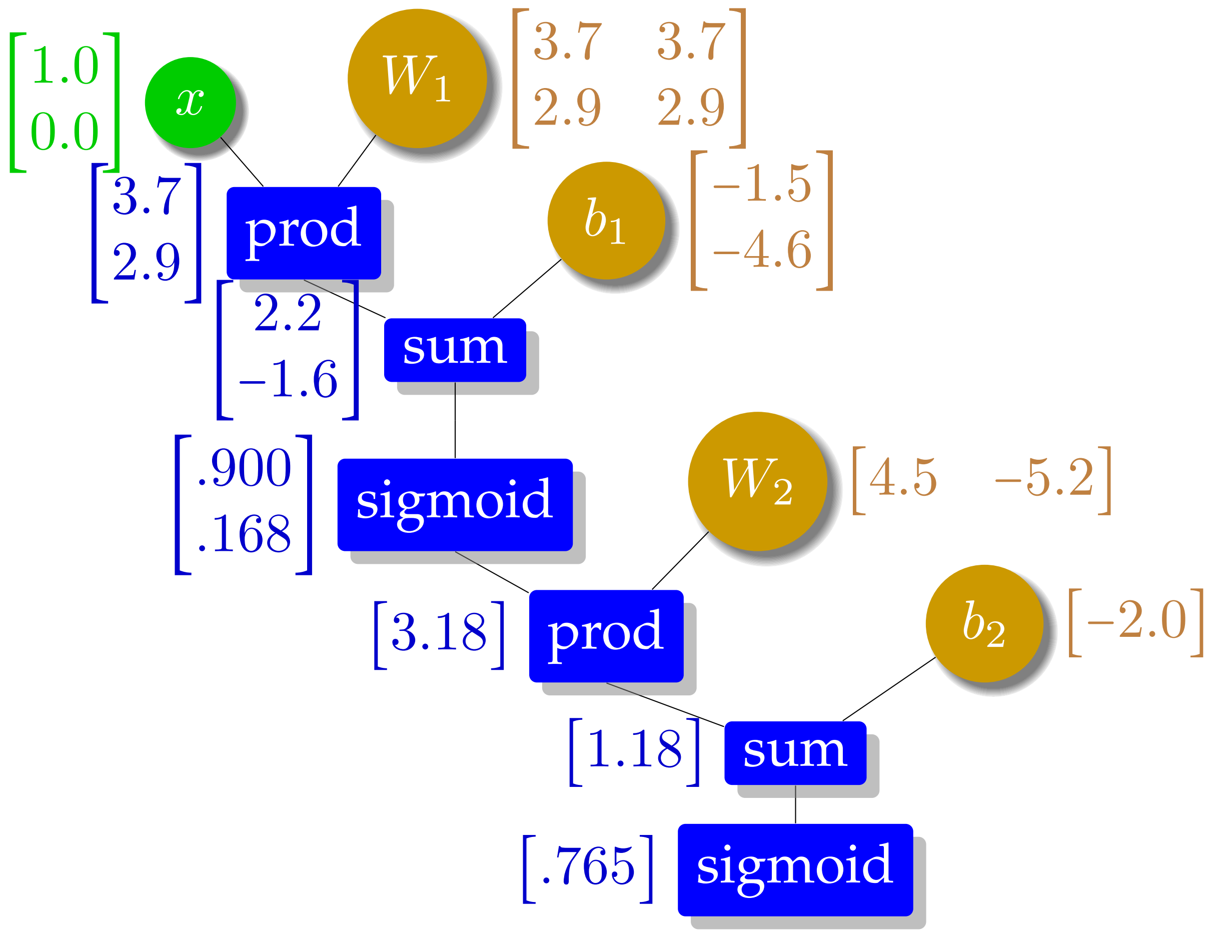
Processing Input



Processing Input



Processing Input



Error Function

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

⇒ Cost function

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

- For instance L2 norm

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

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

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

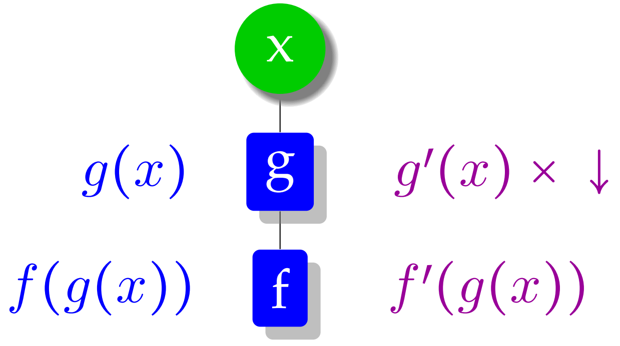
- Leibniz's notation

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

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

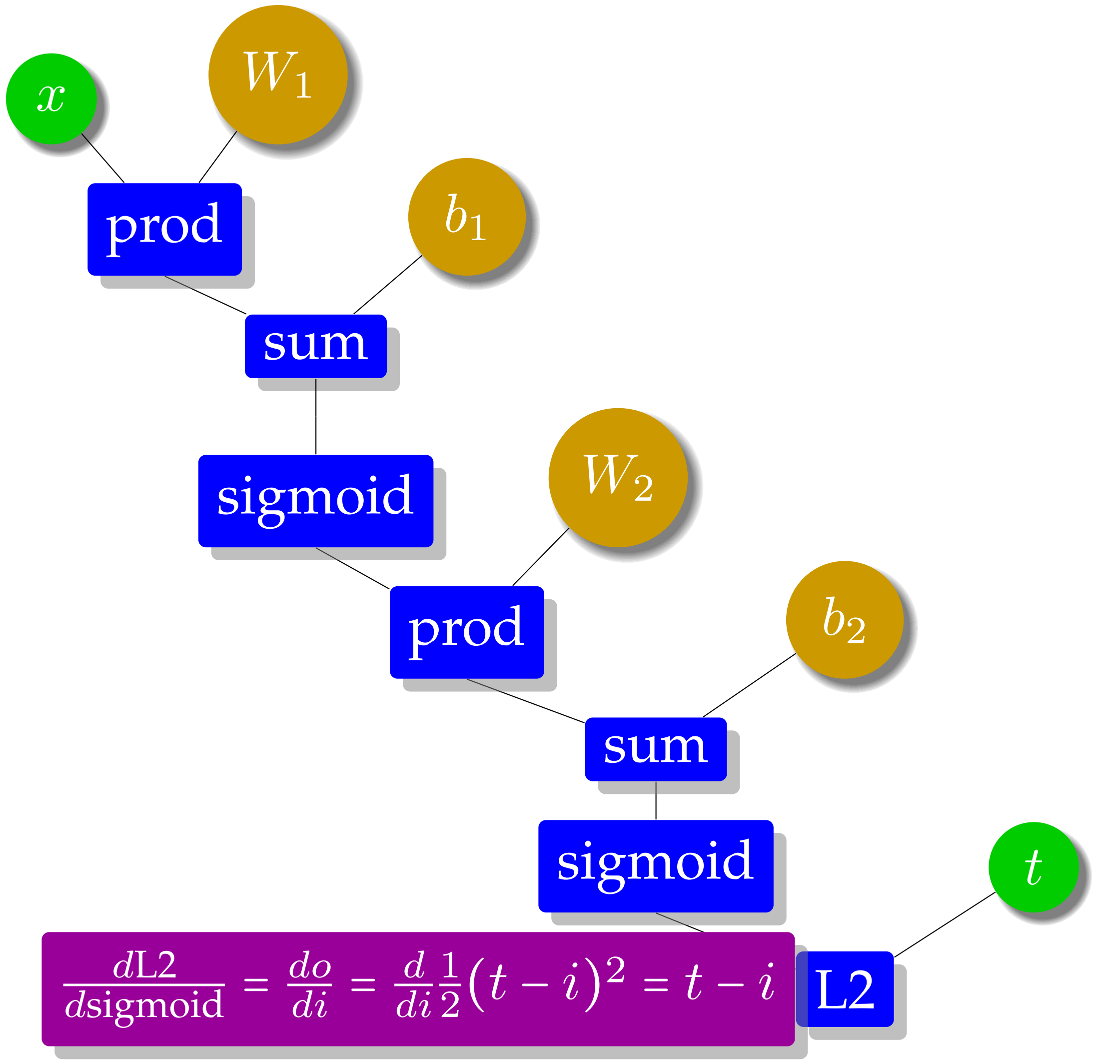
$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} = f'(y)g'(x) = f'(g(x))g'(x)$$

Chain Rule in the Computation Graph

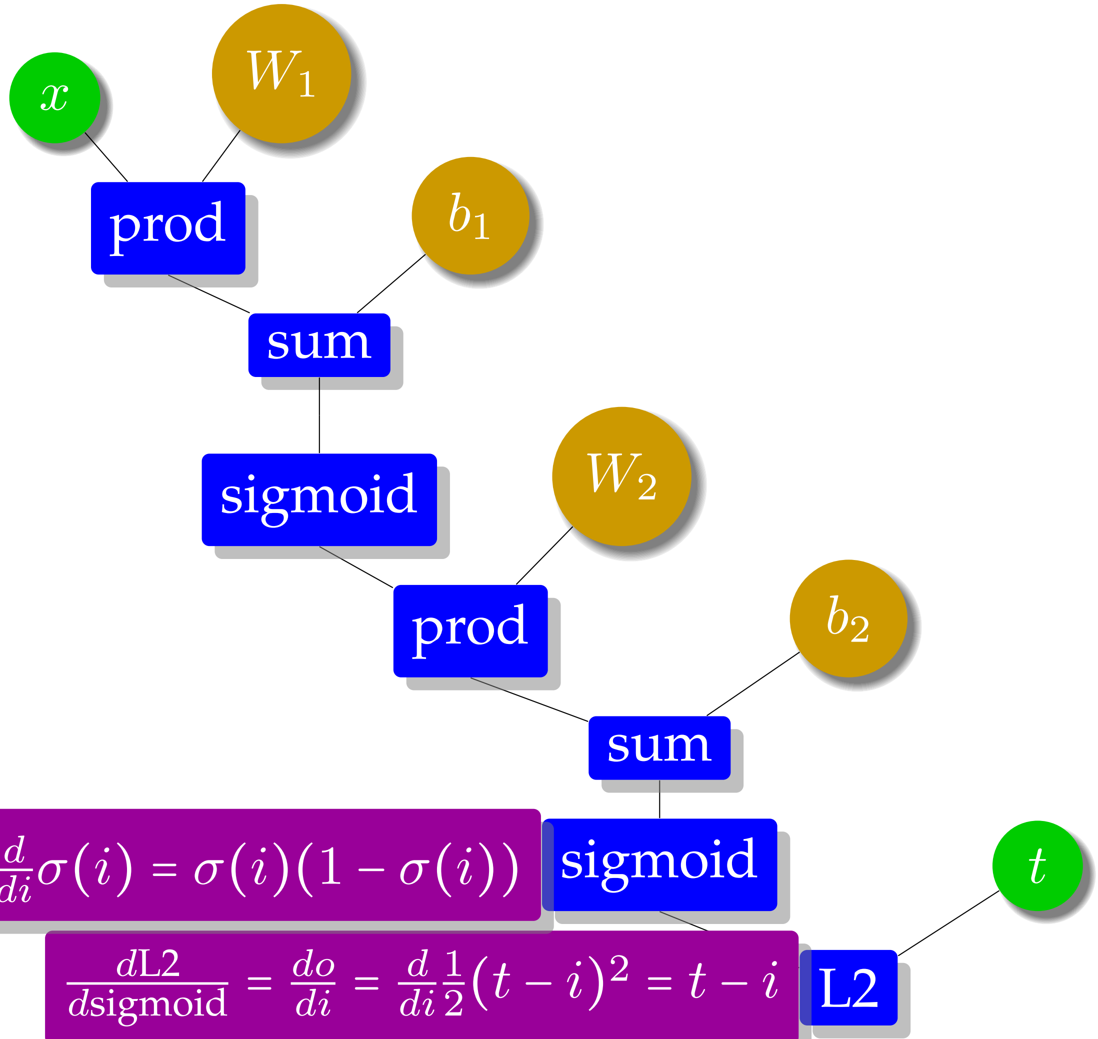


$$\underbrace{f'(g(x))}_{\text{recurse down the graph}} \quad \underbrace{\times}_{\text{multiply values}} \quad \underbrace{g'(x)}_{\text{apply derivative of function to forward value}}$$

Derivatives for Each Node



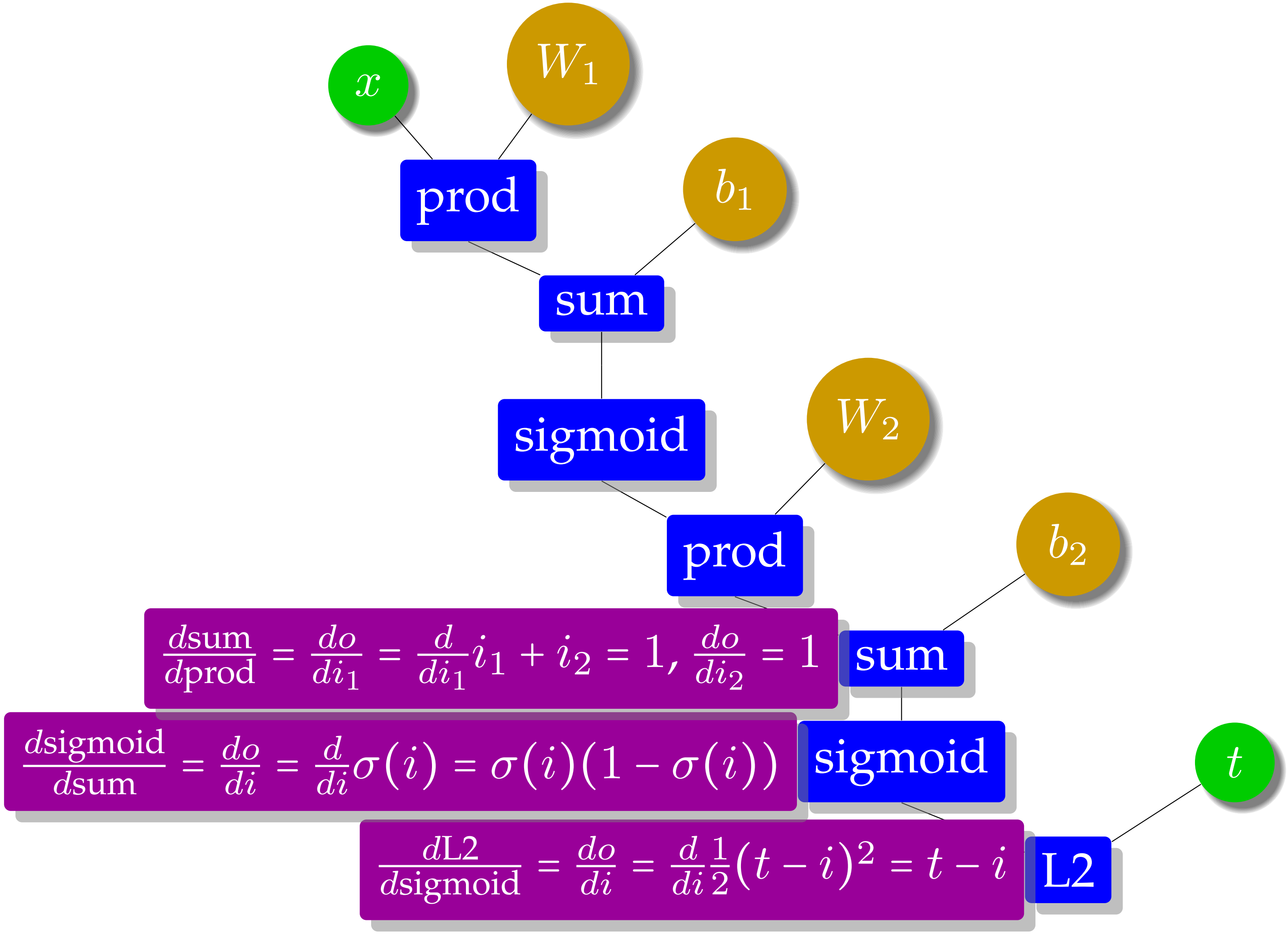
Derivatives for Each Node



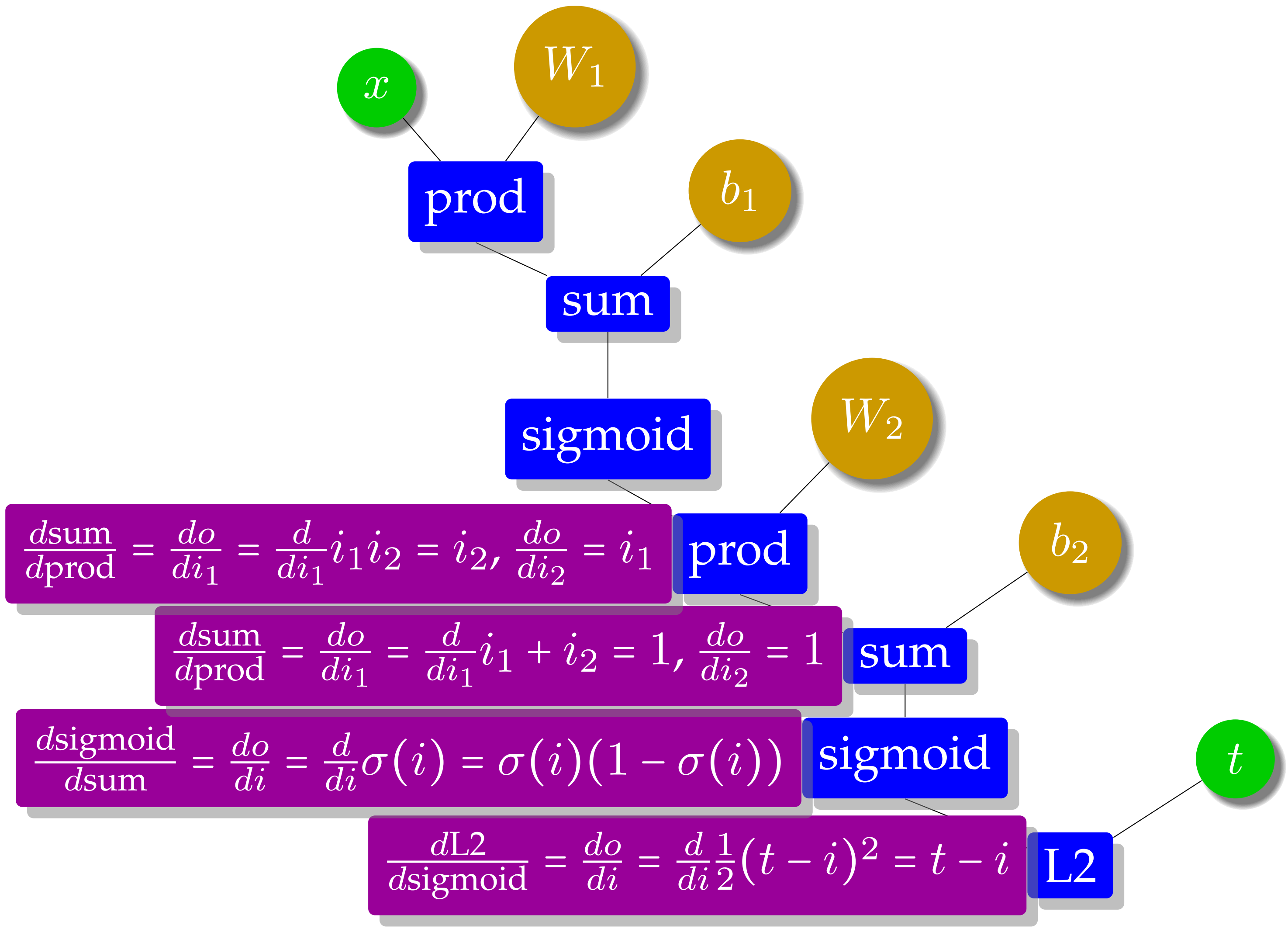
$$\frac{d\text{sigmoid}}{d\text{sum}} = \frac{do}{di} = \frac{d}{di}\sigma(i) = \sigma(i)(1-\sigma(i))$$

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

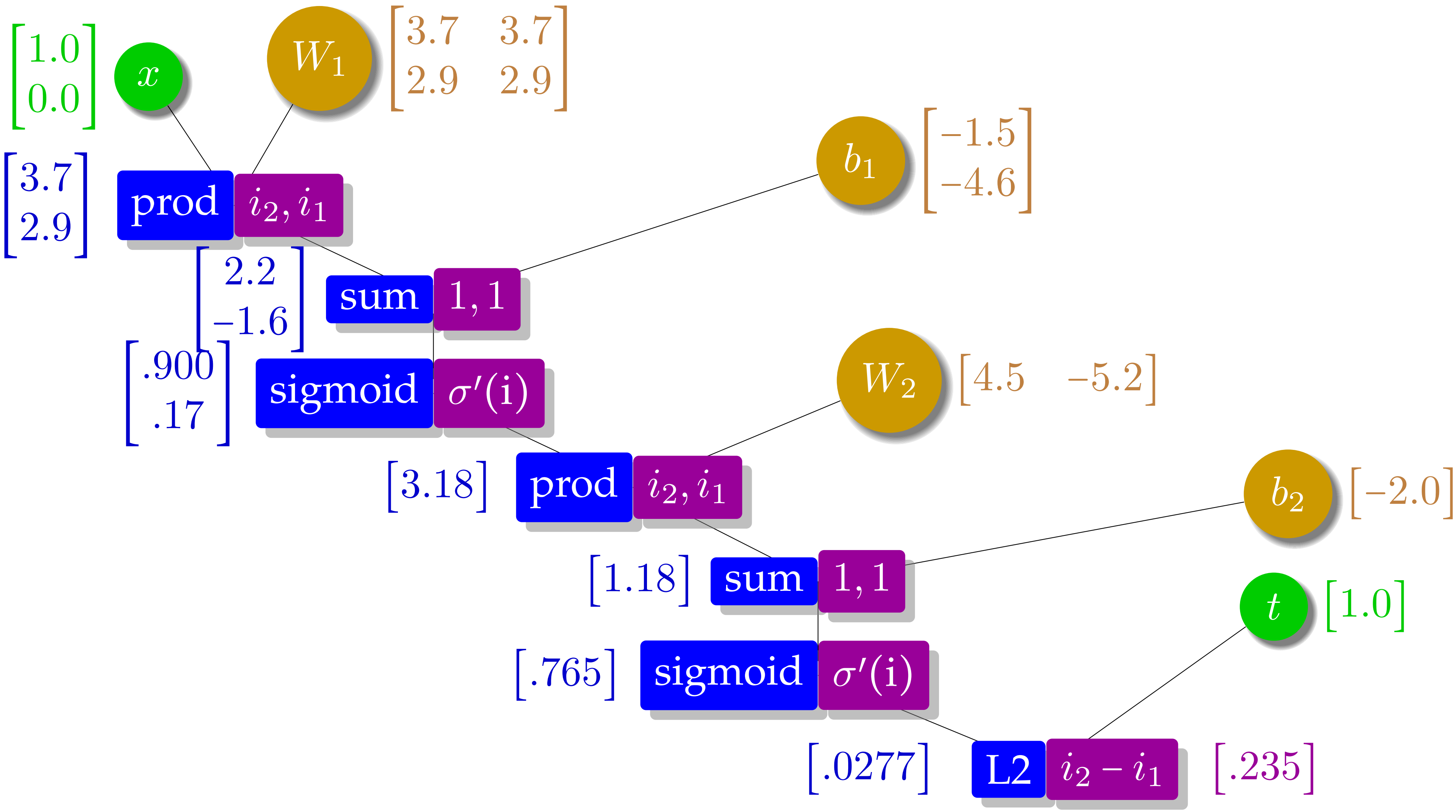
Derivatives for Each Node



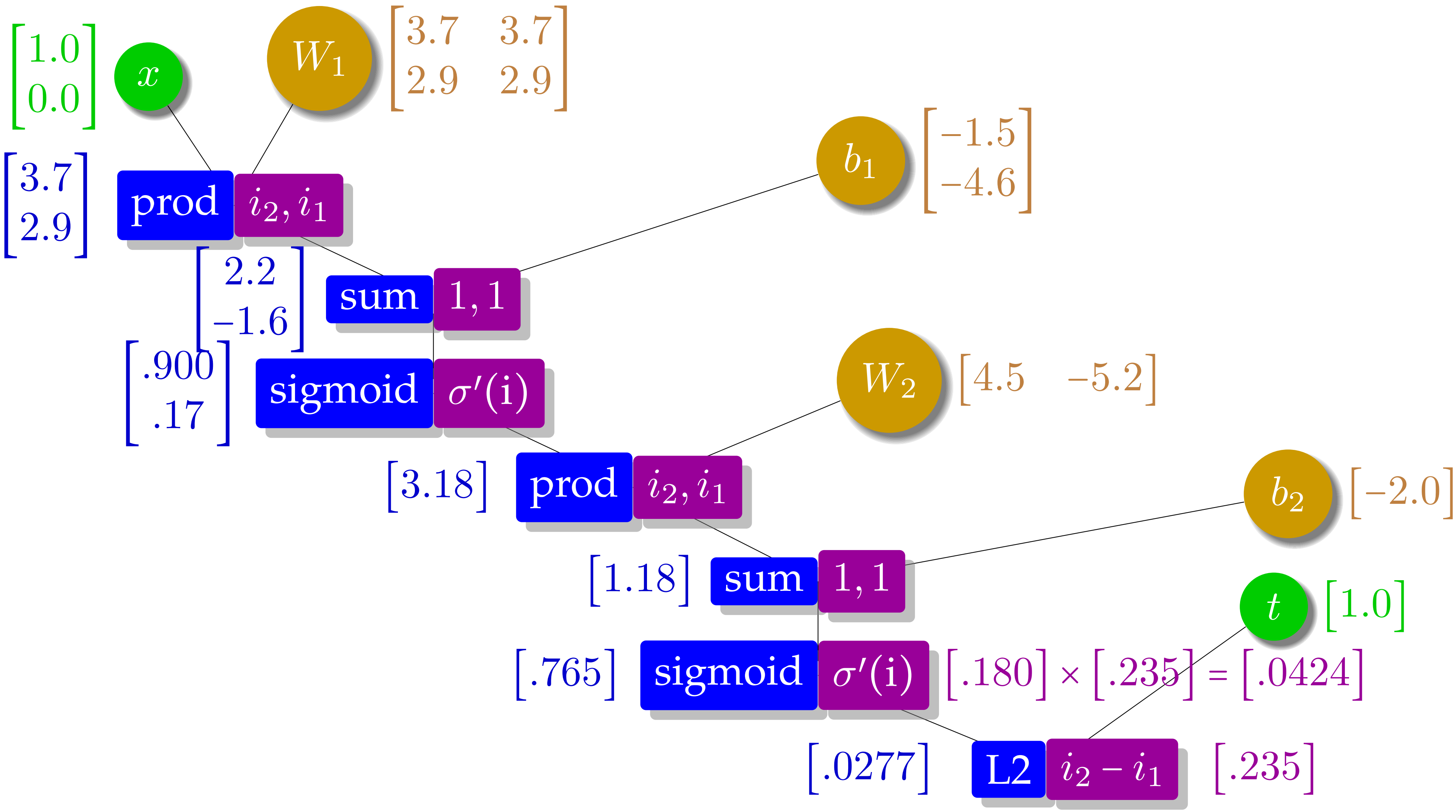
Derivatives for Each Node



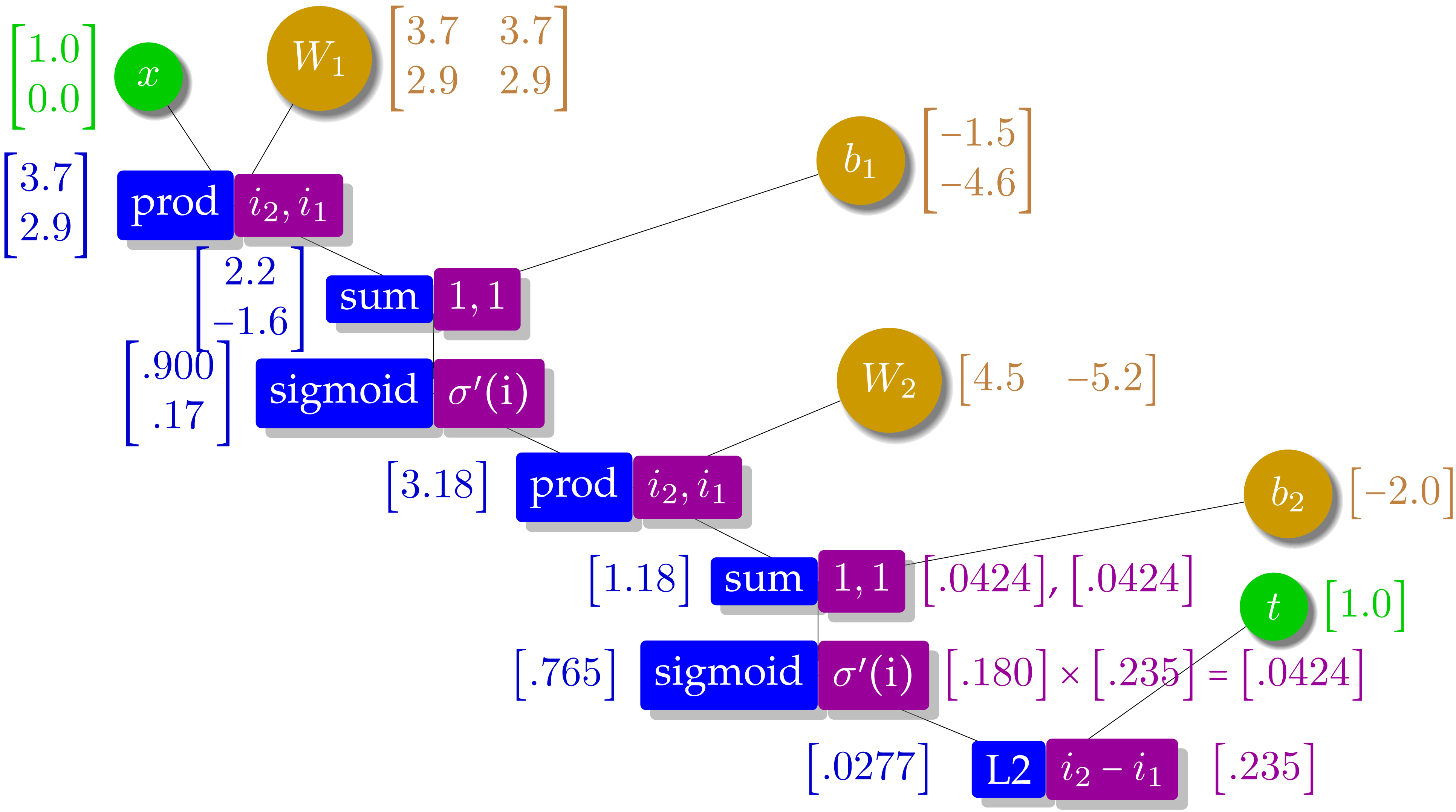
Backward Pass: Derivative Computation



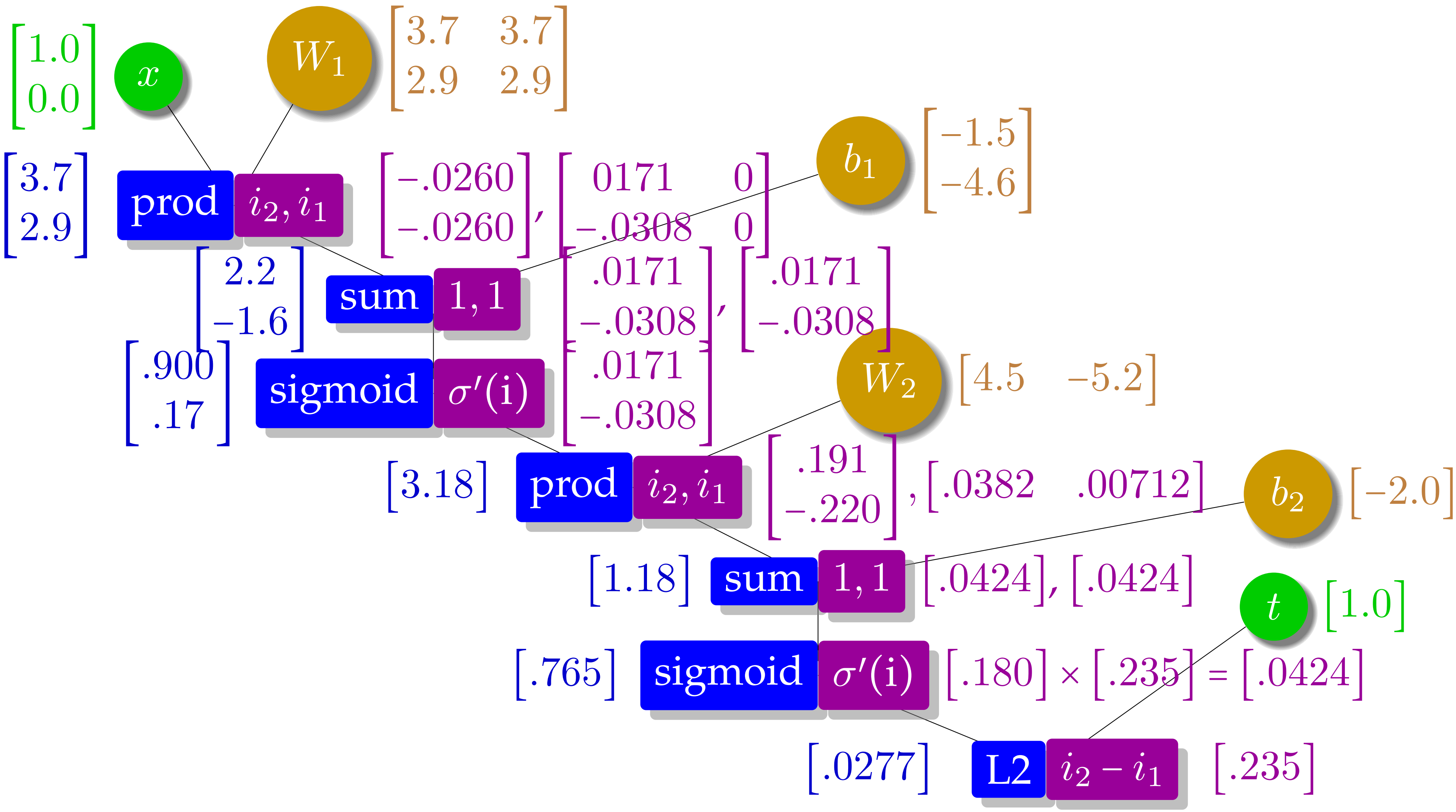
Backward Pass: Derivative Computation



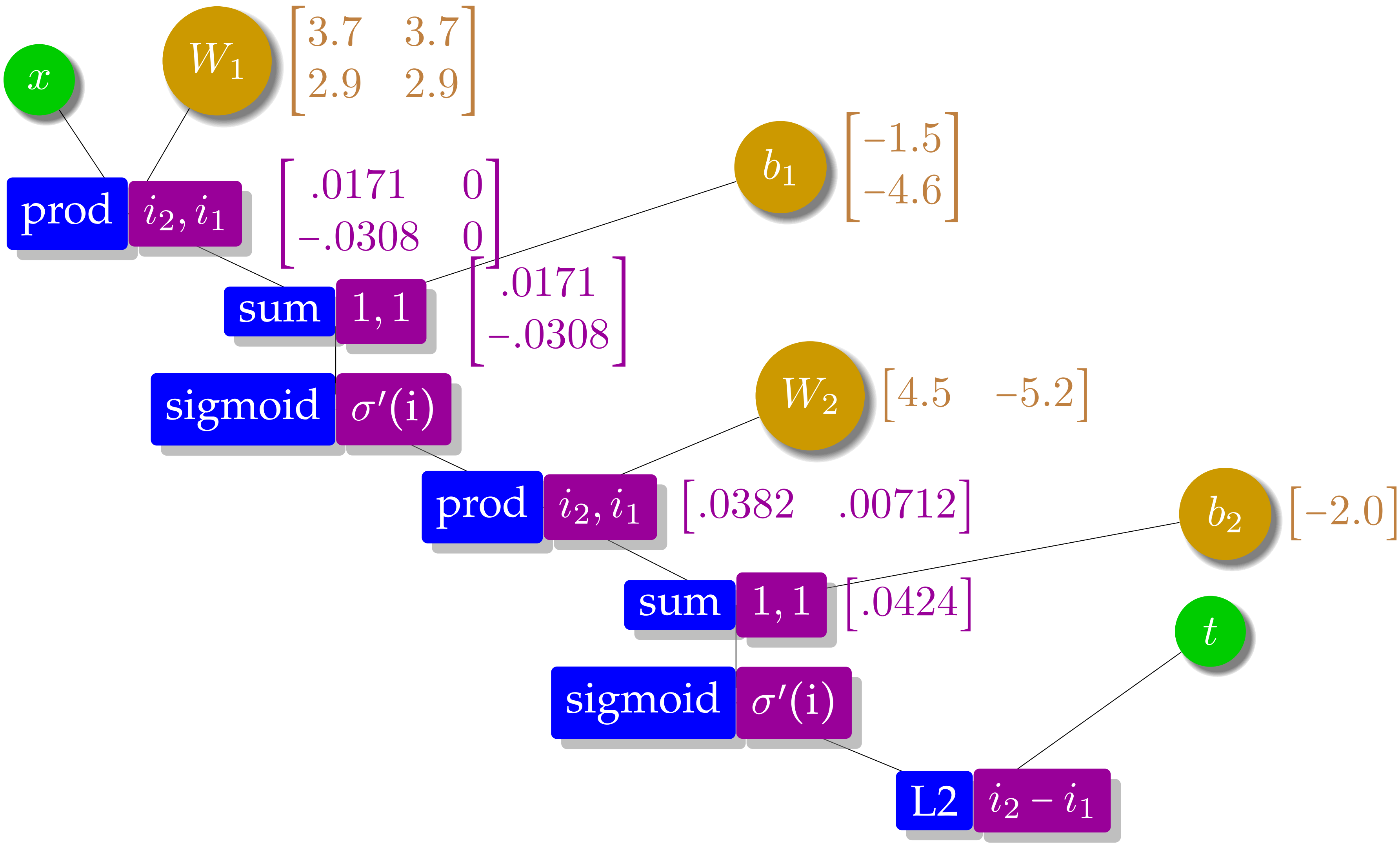
Backward Pass: Derivative Computation



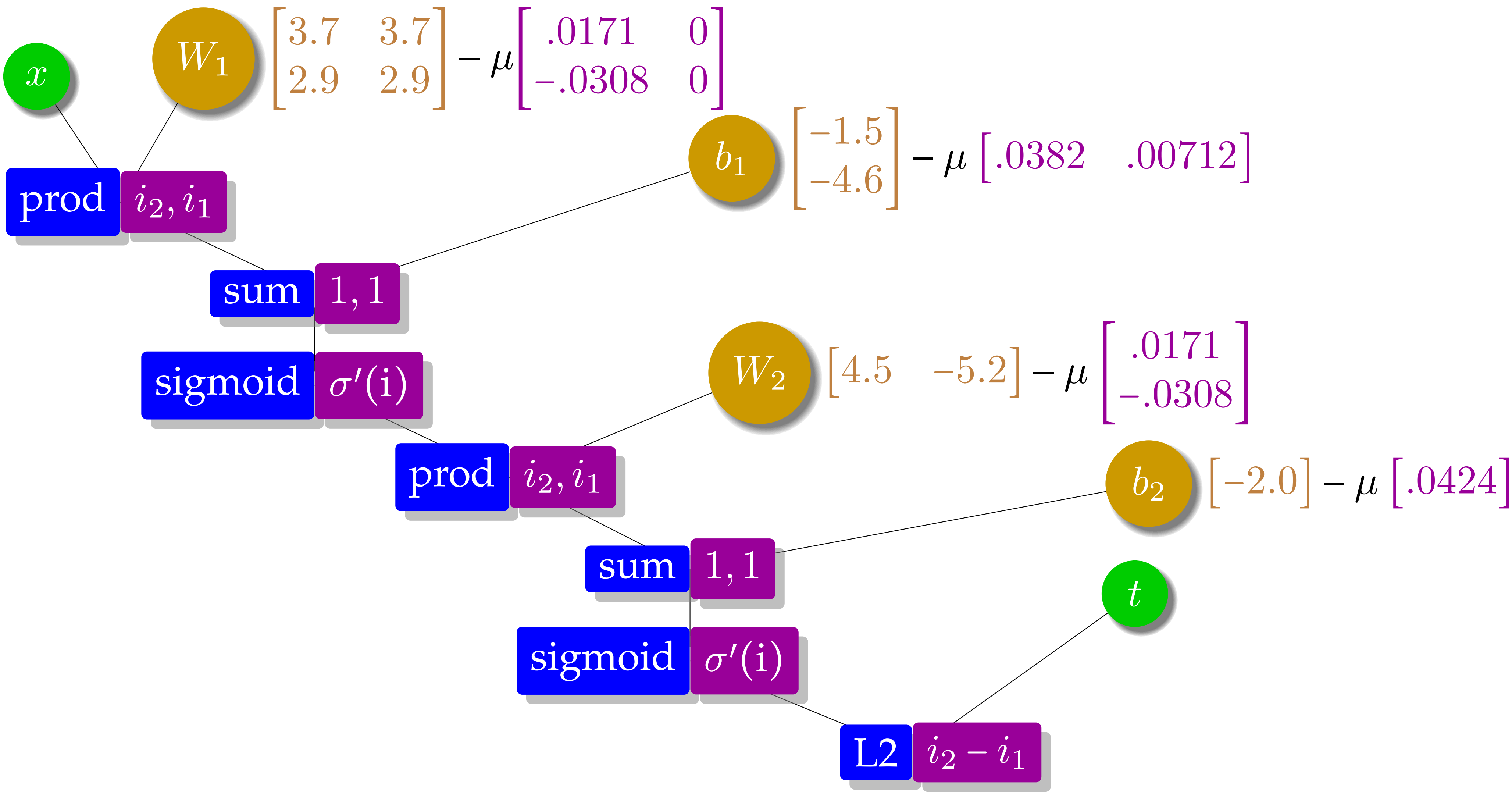
Backward Pass: Derivative Computation



Gradients for Parameter Update



Parameter Update



toolkits

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
- Popular today
  - PyTorch
  - Huggingface
  - Tensorflow

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**	**x ⊕ 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()
```

GPU

- Neural network layers may have, say, 200 nodes
- Computations such as $W\vec{h}$ require $200 \times 200 = 40,000$ multiplications
- Graphics Processing Units (GPU) are designed for such computations
  - image rendering requires such vector and matrix operations
  - massively mulit-core but lean processing units
  - example: NVIDIA H100 GPU provides 14,592 thread processors
- Extensions to C to support programming of GPUs, such as CUDA