

---

# Computation Graphs

Brian Thompson  
slides by Philipp Koehn

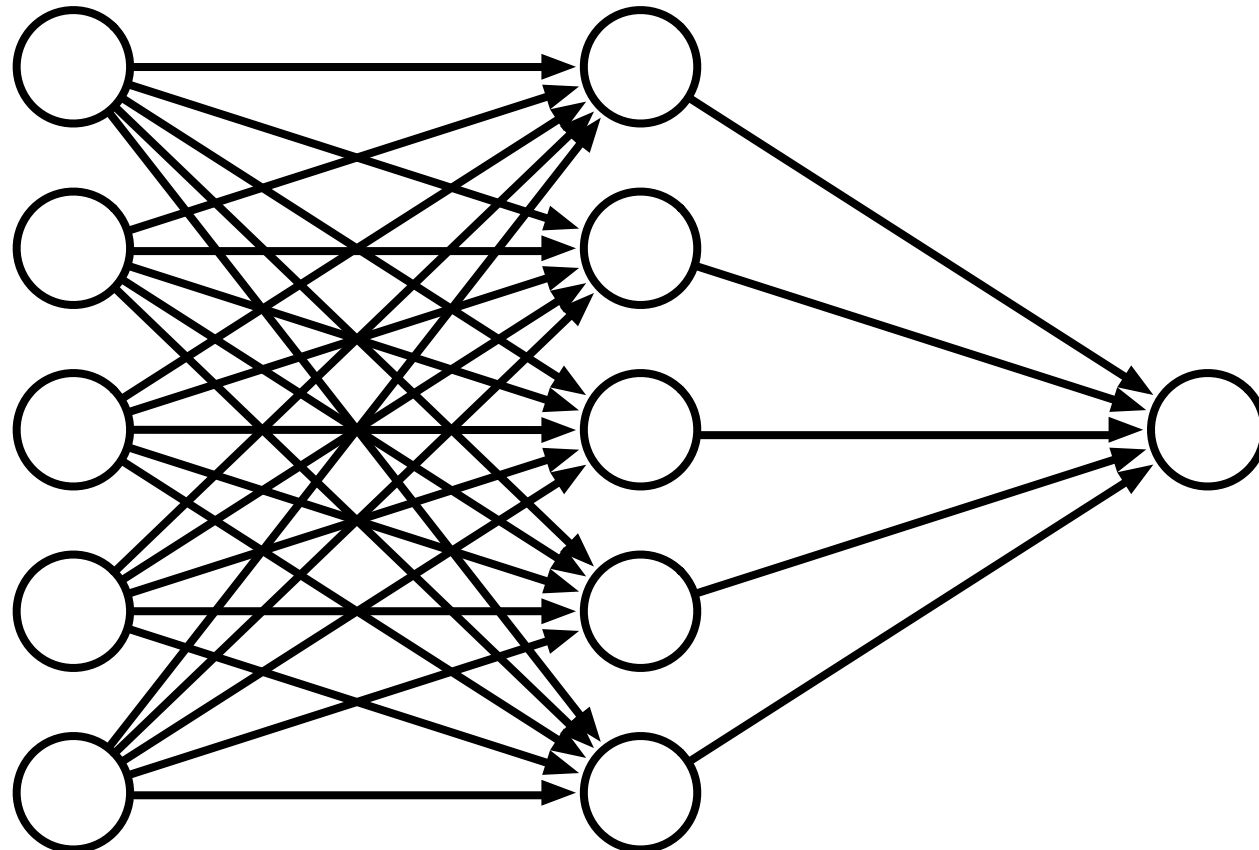
2 October 2018



# Neural Network Cartoon



- A common way to illustrate a neural network



# Neural Network Math



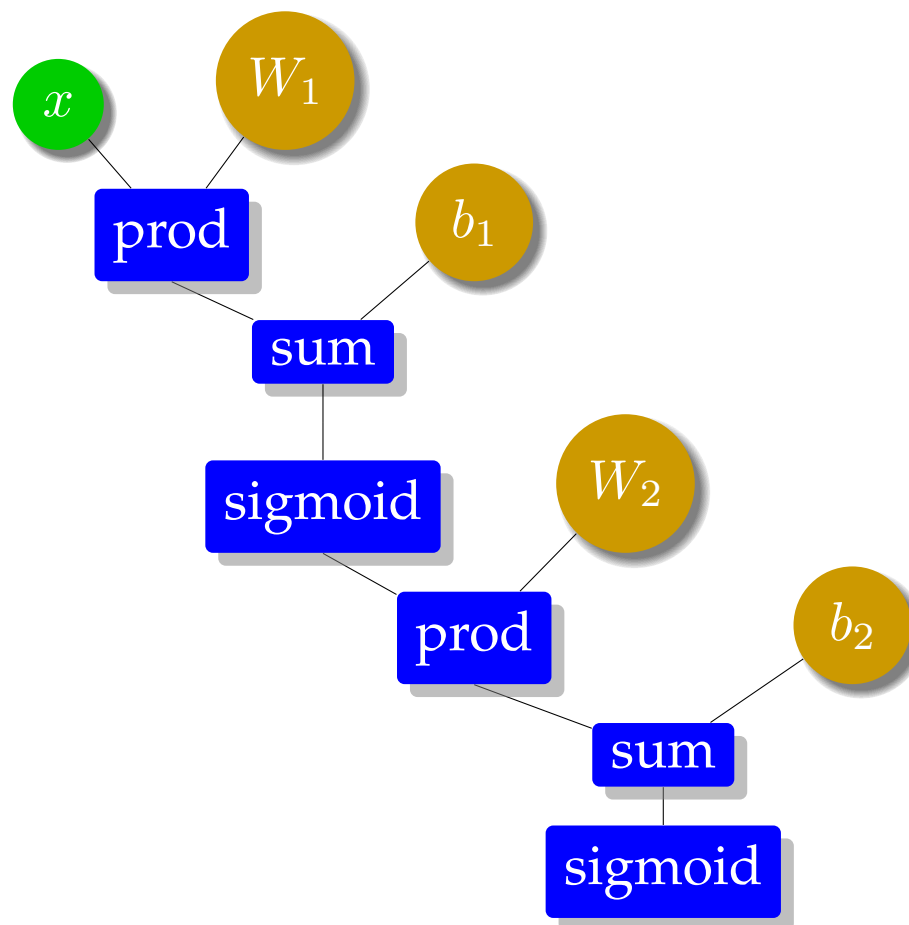
- Hidden layer

$$h = \text{sigmoid}(W_1x + b_1)$$

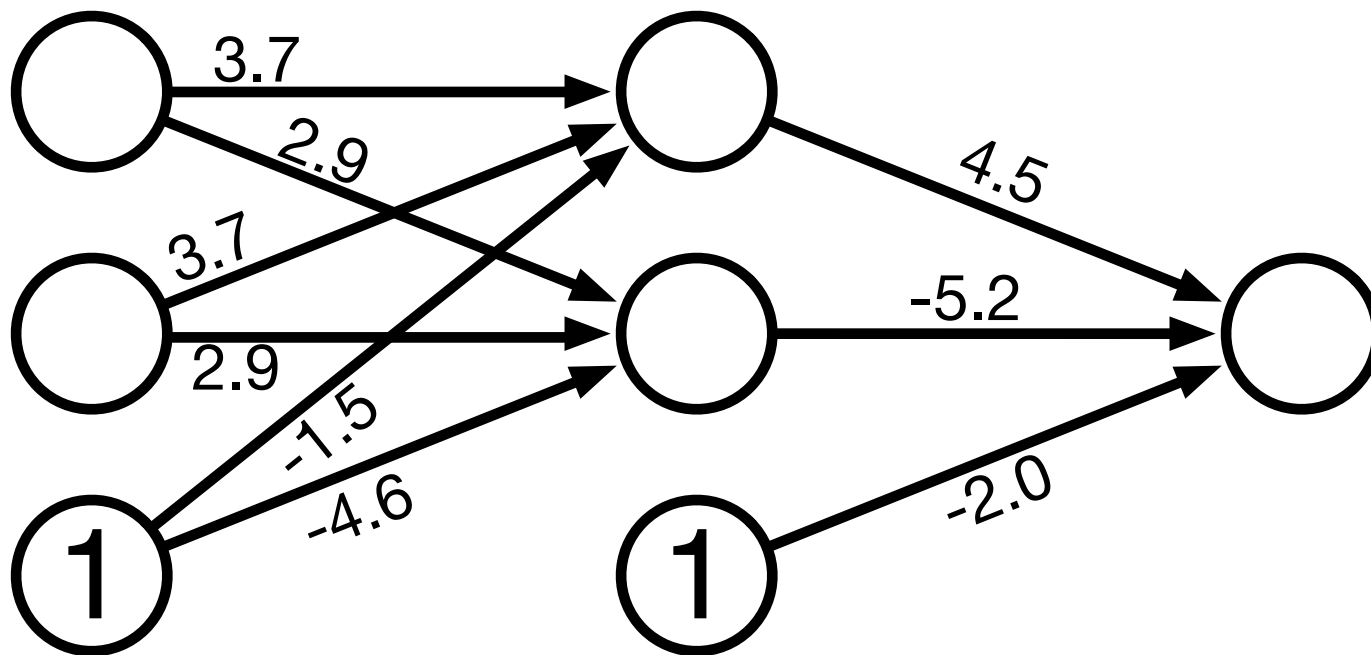
- Final layer

$$y = \text{sigmoid}(W_2h + b_2)$$

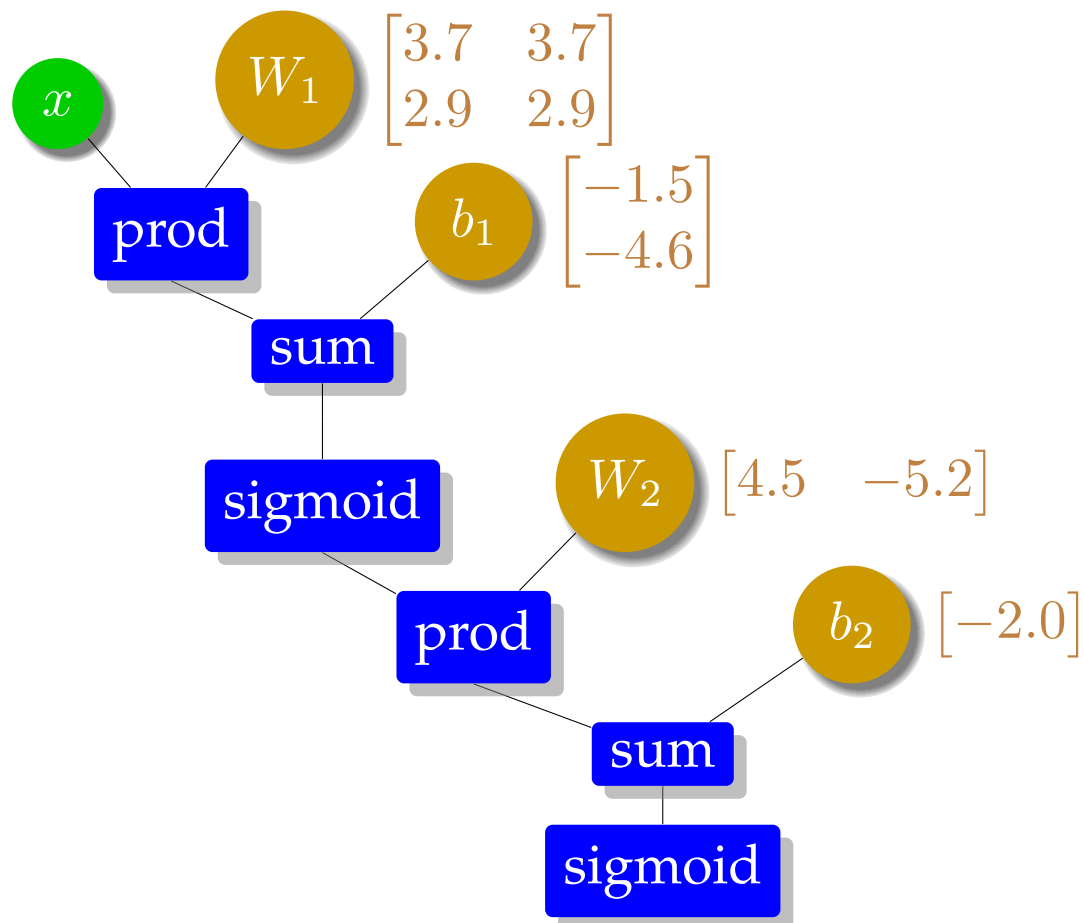
# Computation Graph



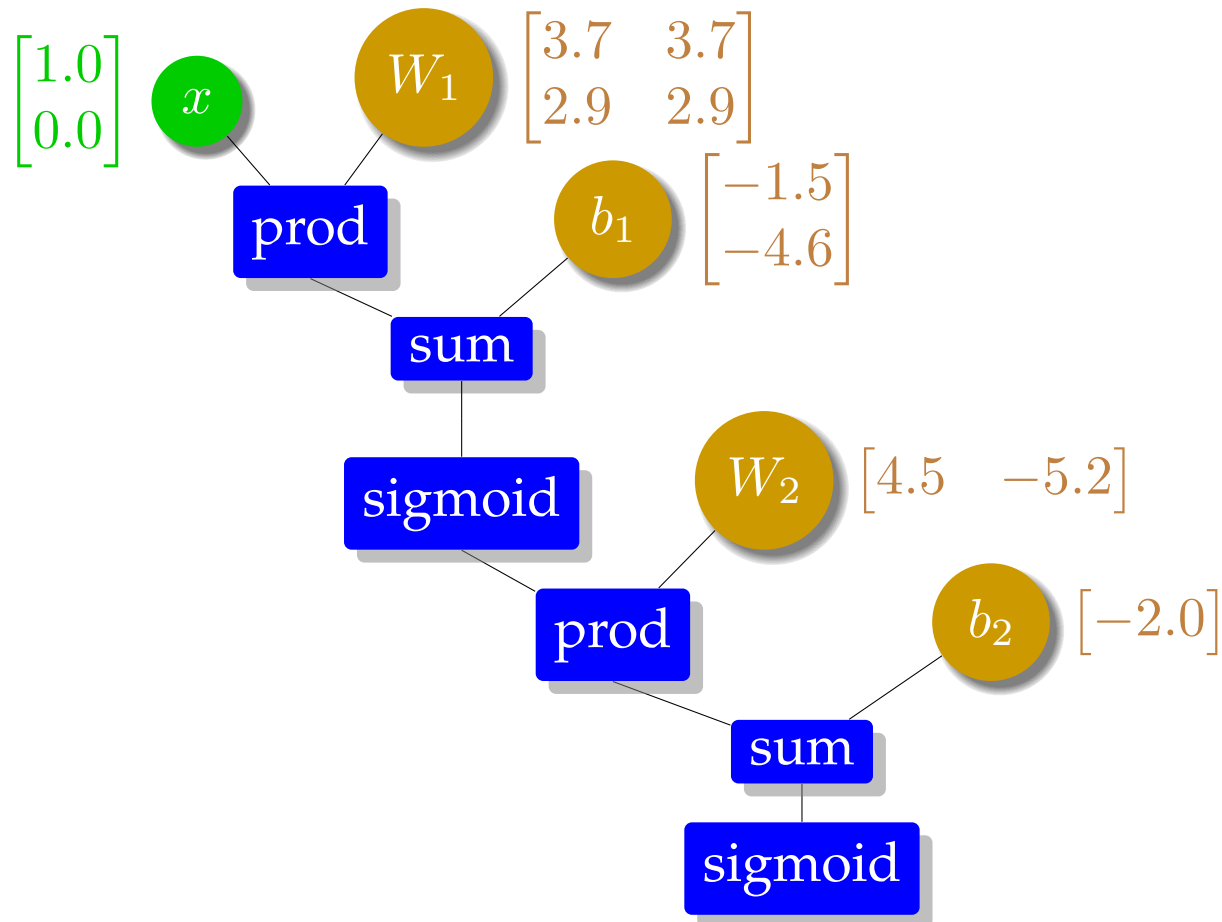
# Simple Neural Network



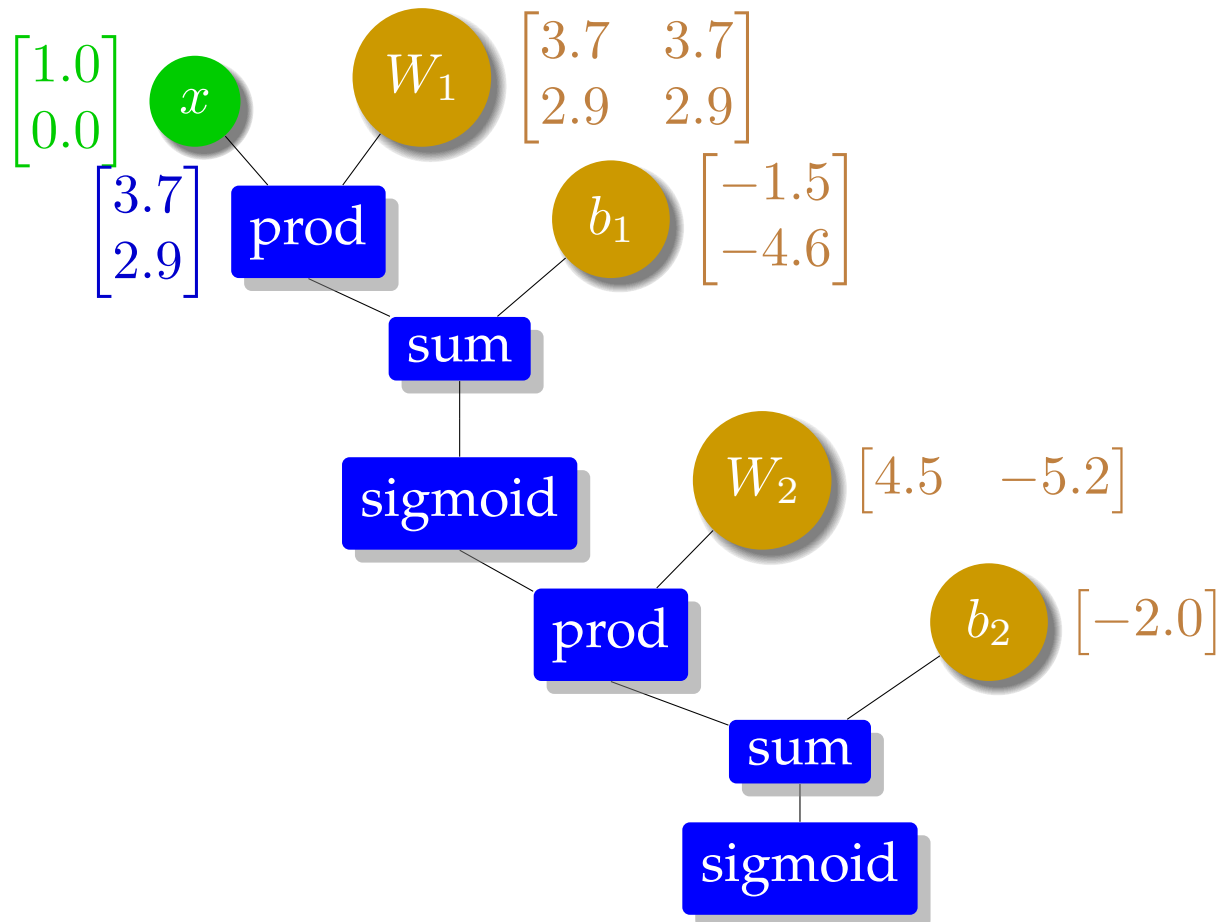
# Computation Graph



# Processing Input

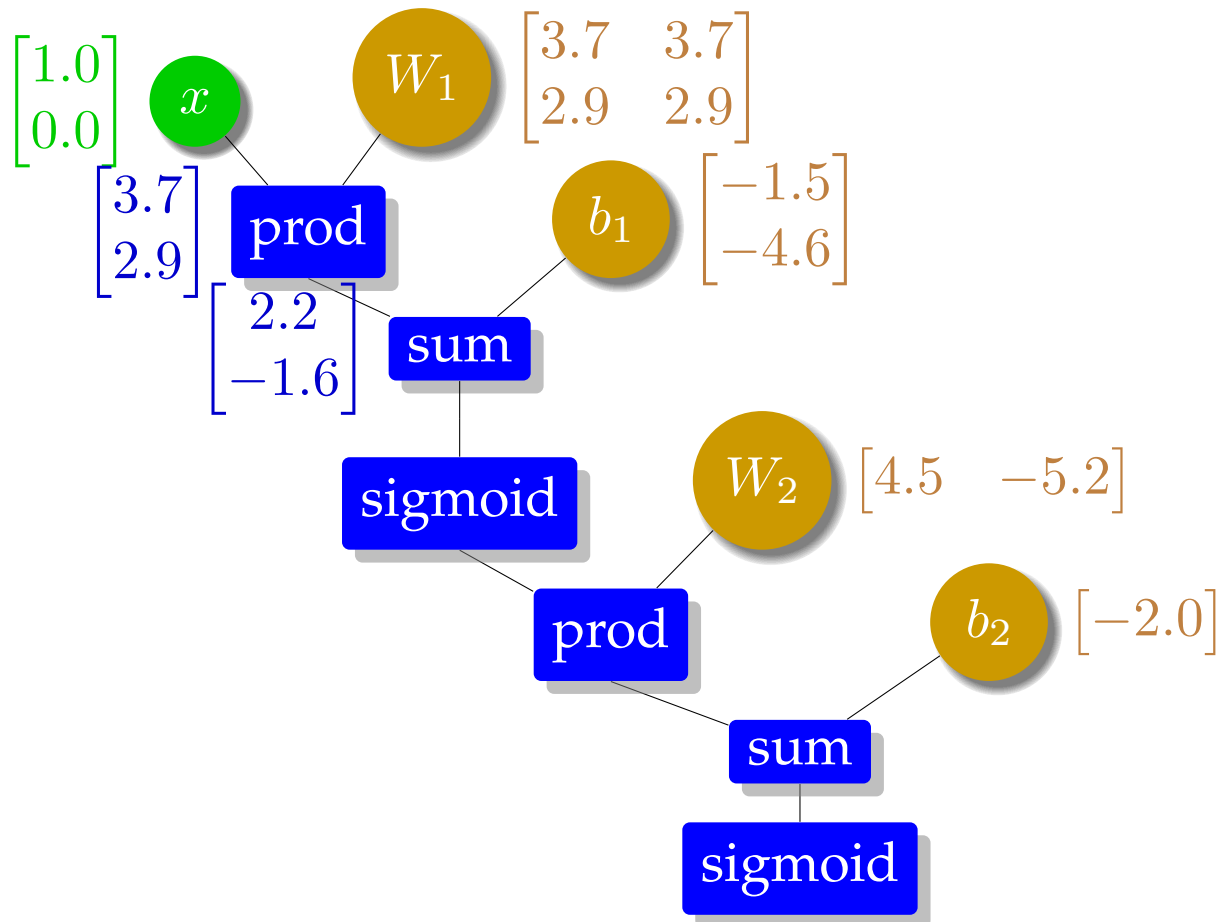


# Processing Input

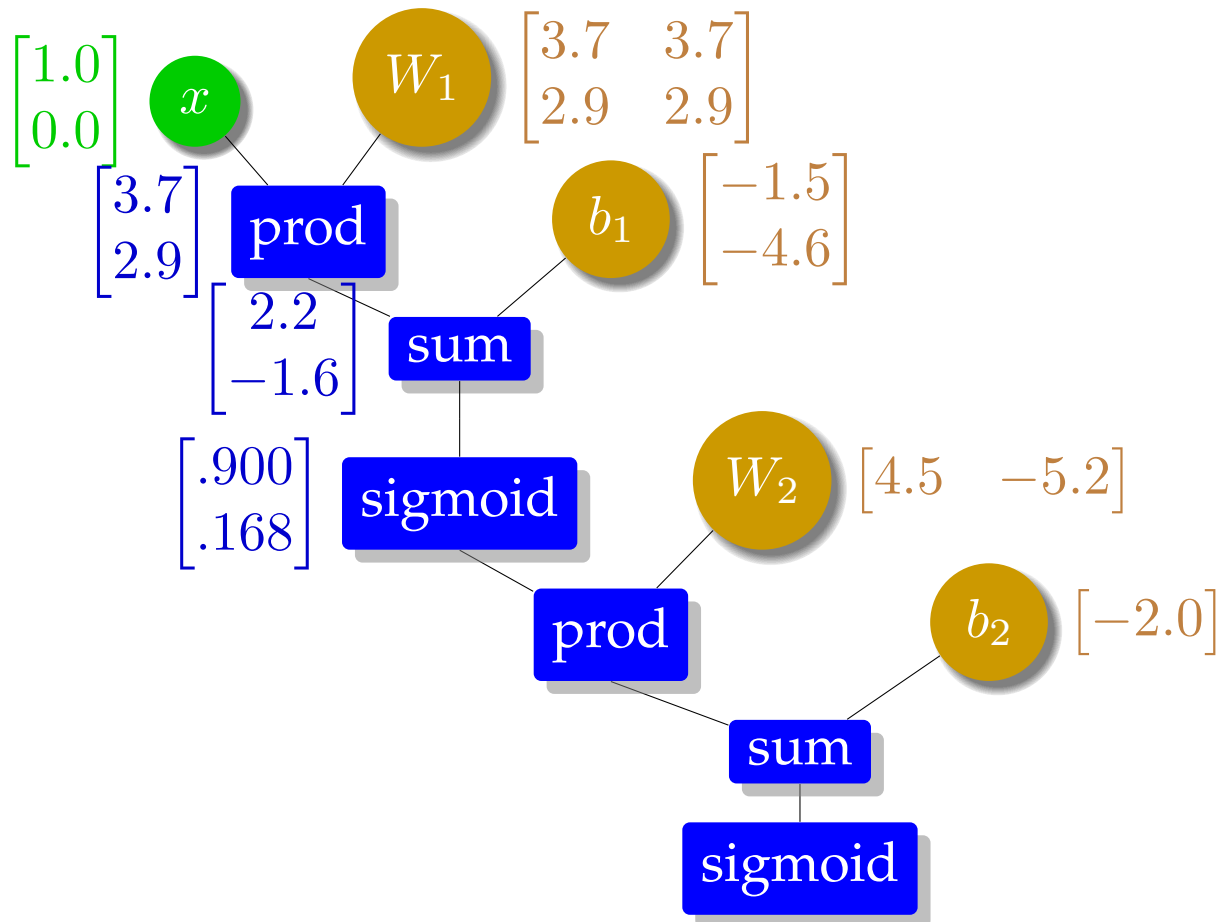




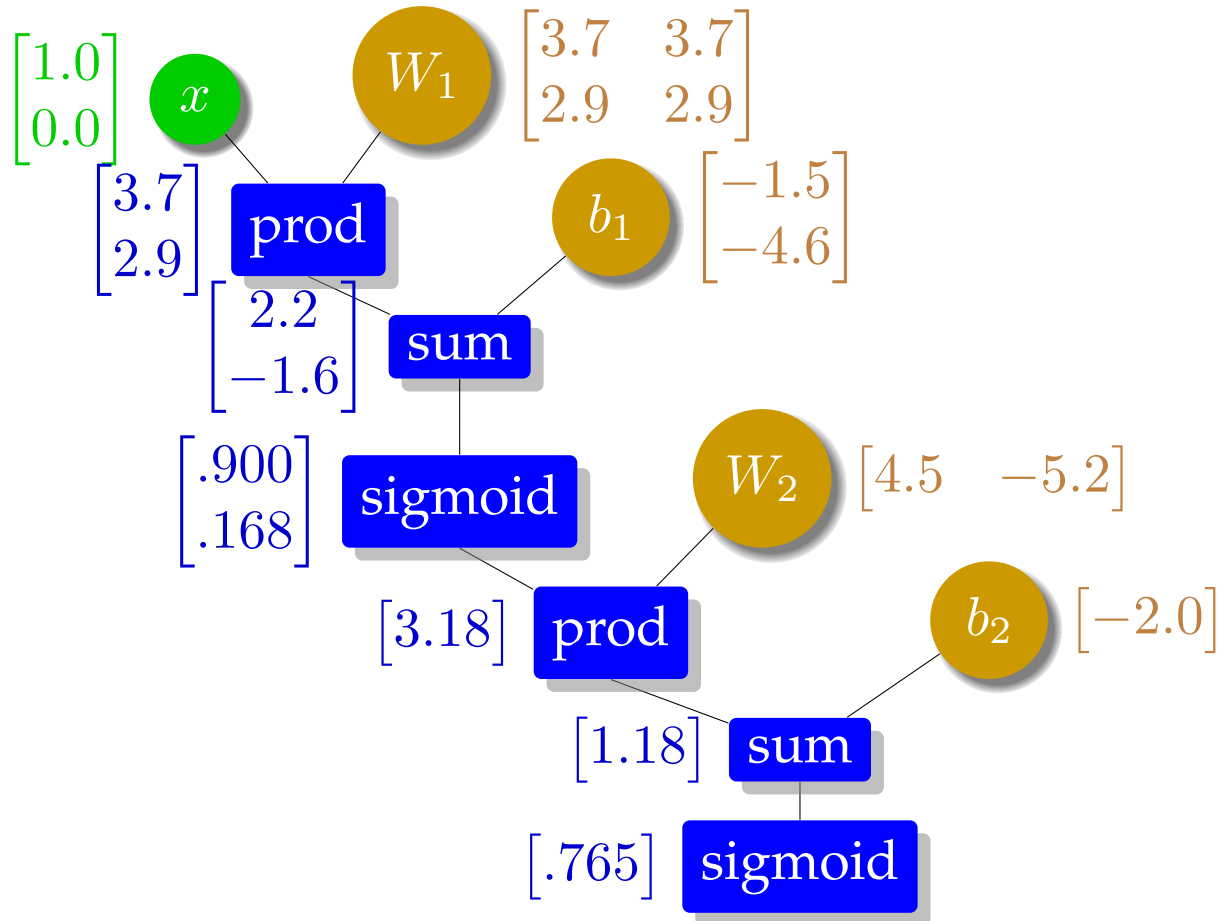
# Processing Input



# Processing Input



# Processing Input



# Error Function

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

⇒ Cost function

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

- For instance L2 norm

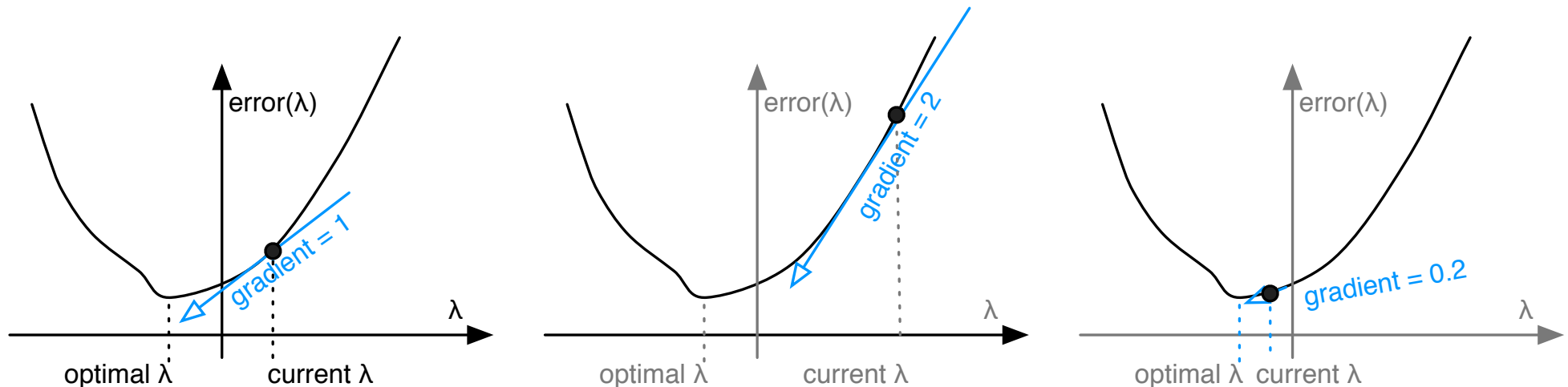
$$\text{error} = \frac{1}{2}(t - y)^2$$

# 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

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

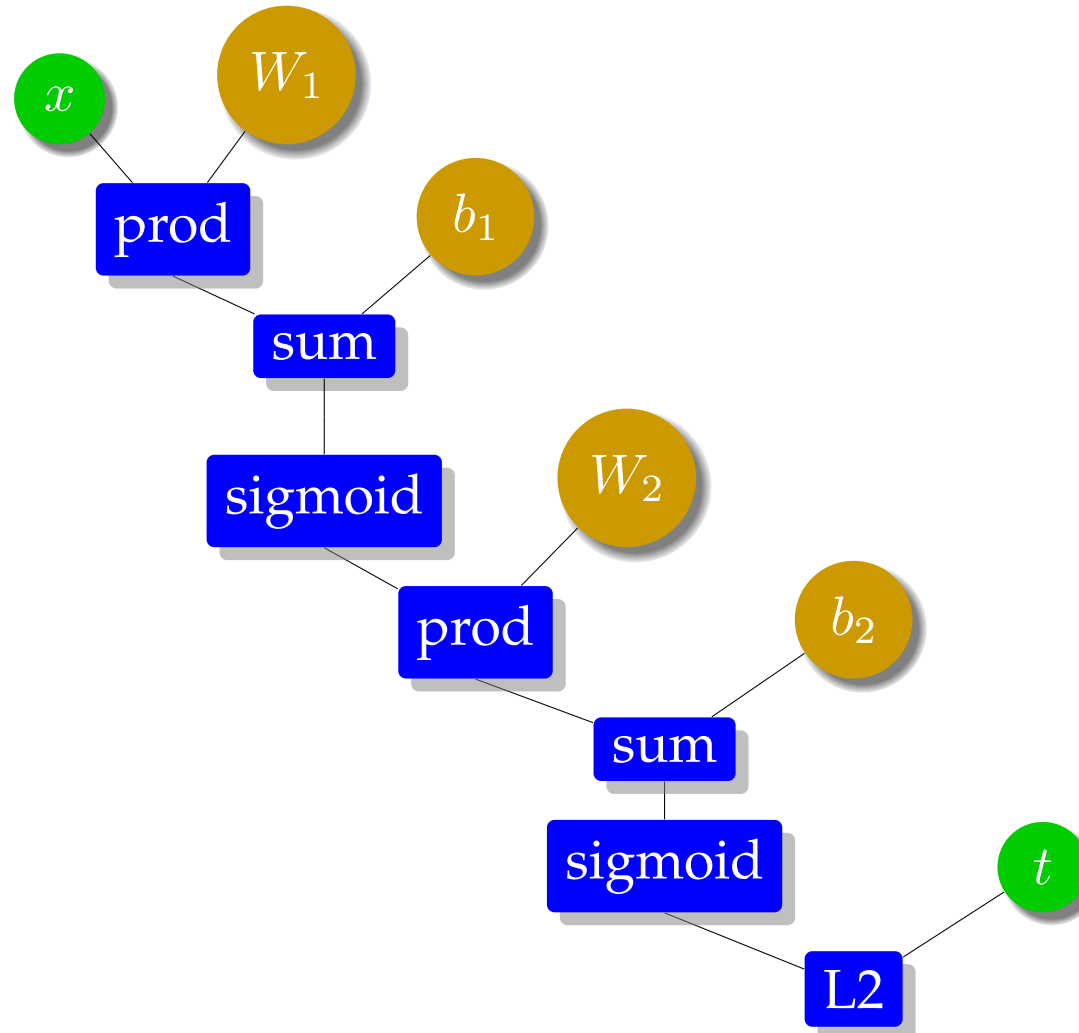
# Final Layer Update

- 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

15





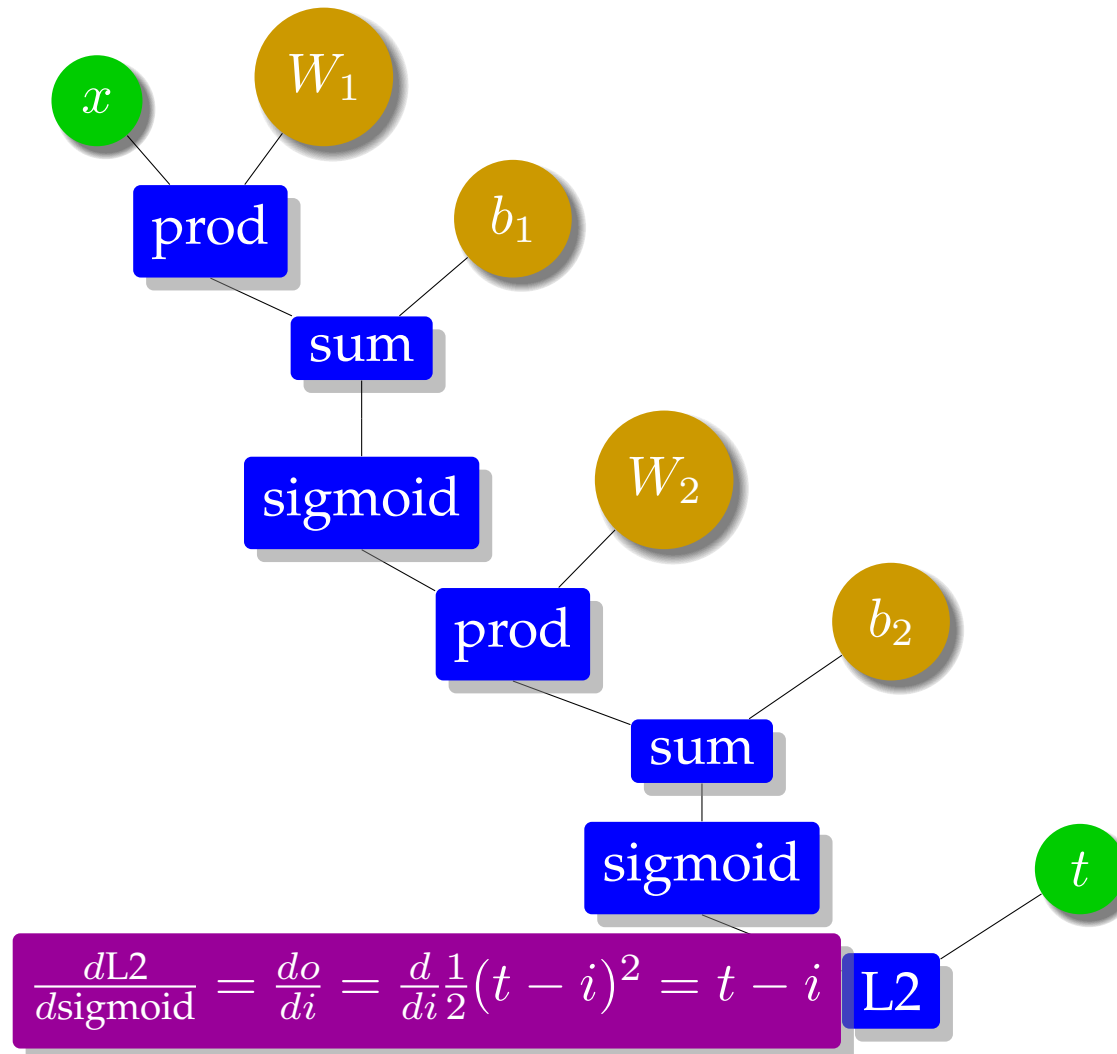
# Error Propagation in Computation Graph

16

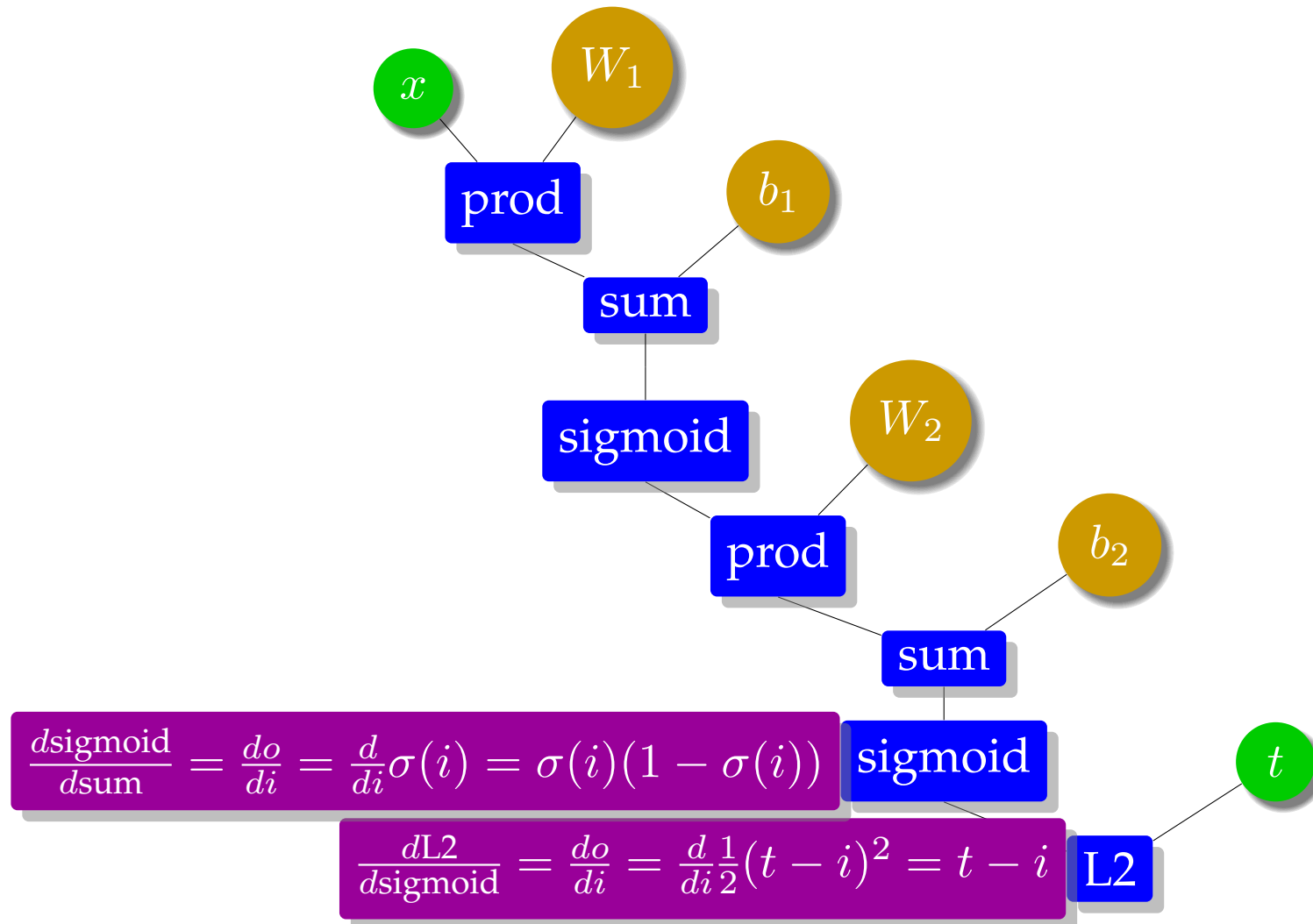


- Compute derivative at node  $A$ :  $\frac{dE}{dA} = \frac{dE}{dB} \frac{dB}{dA}$
- Assume that we already computed  $\frac{dE}{dB}$  (backward pass through graph)
- So now we only have to get the formula for  $\frac{dB}{dA}$
- For instance  $B$  is a square node
  - forward computation:  $B = A^2$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

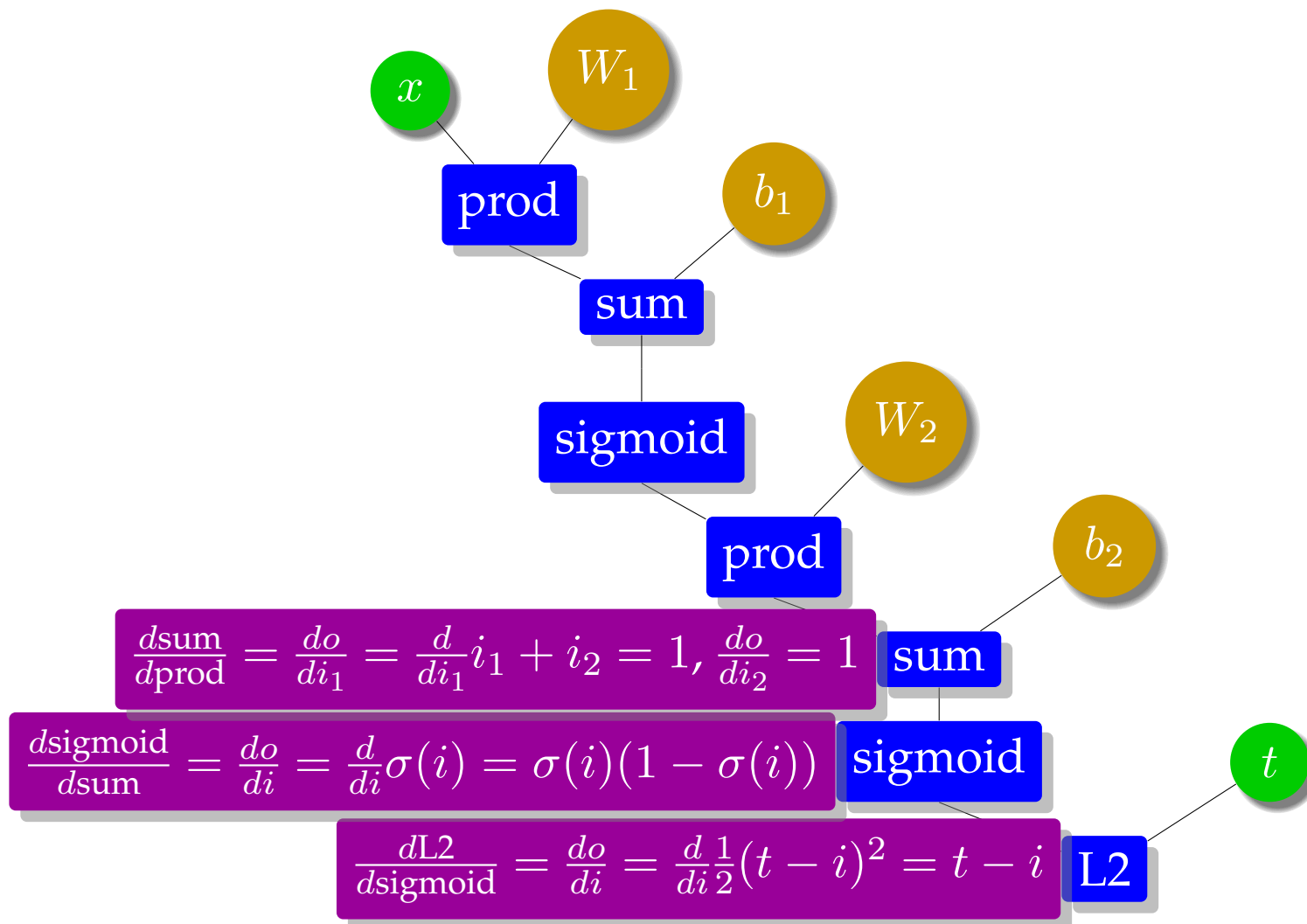
# Derivatives for Each Node



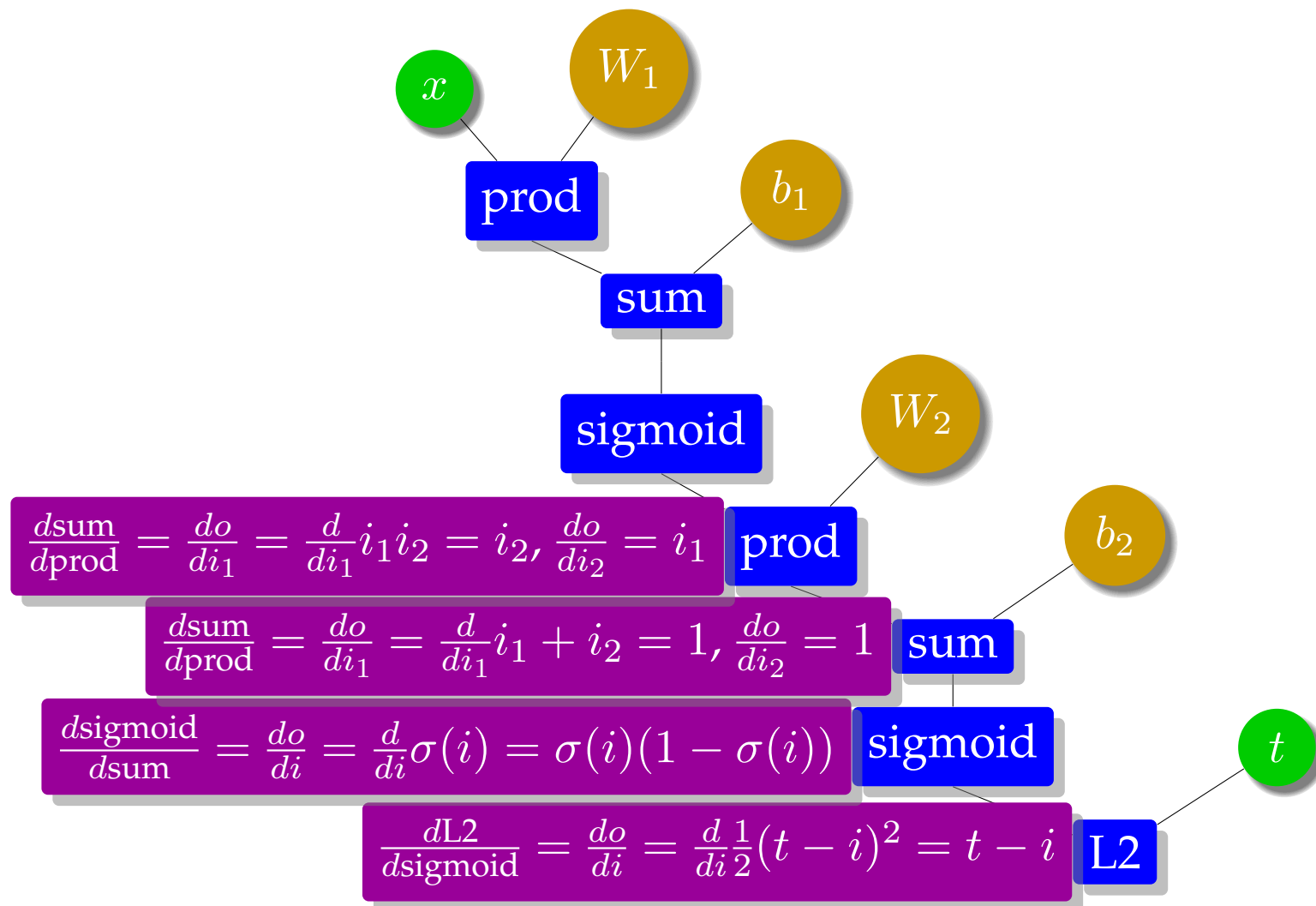
# Derivatives for Each Node



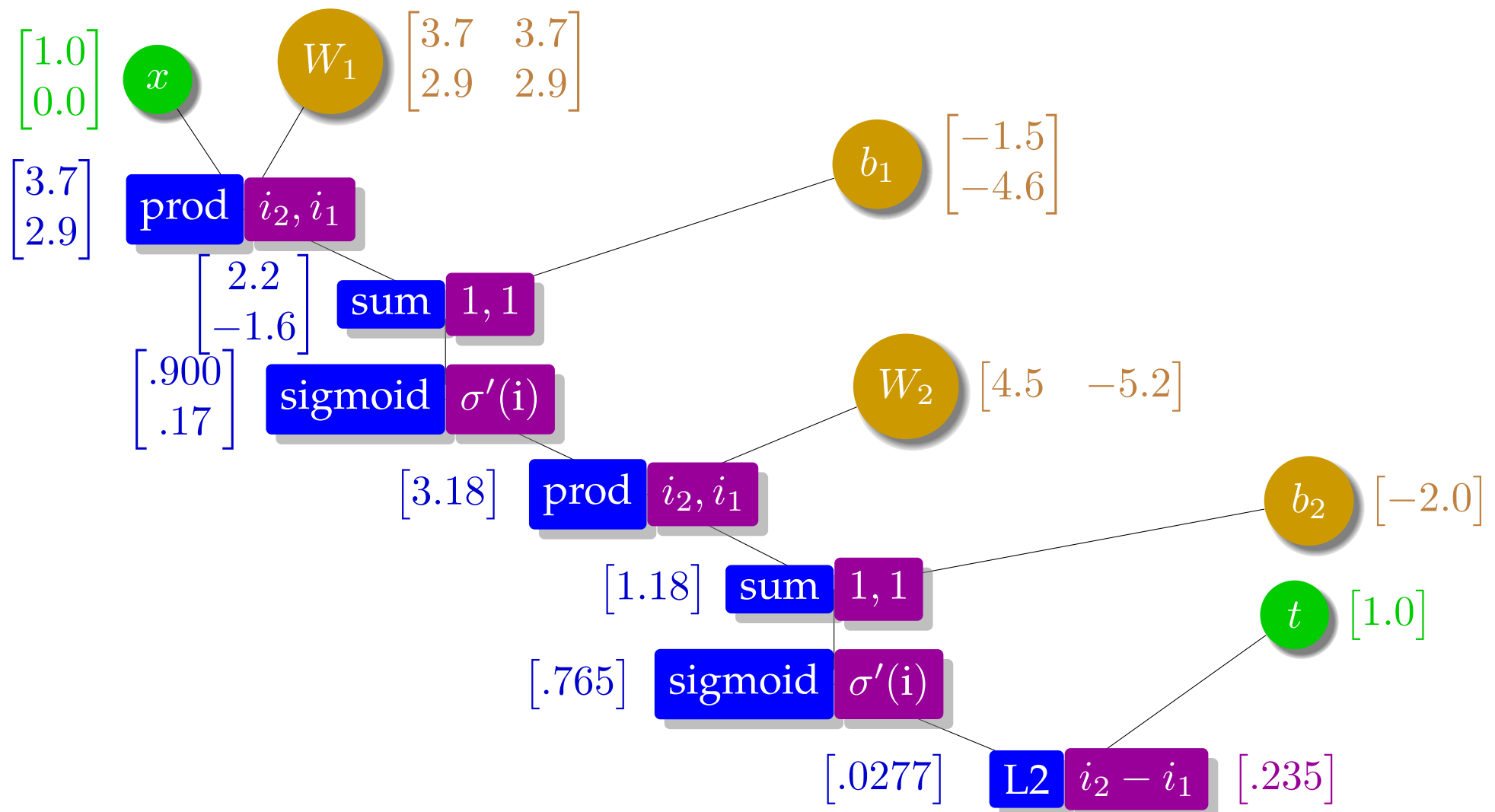
# Derivatives for Each Node



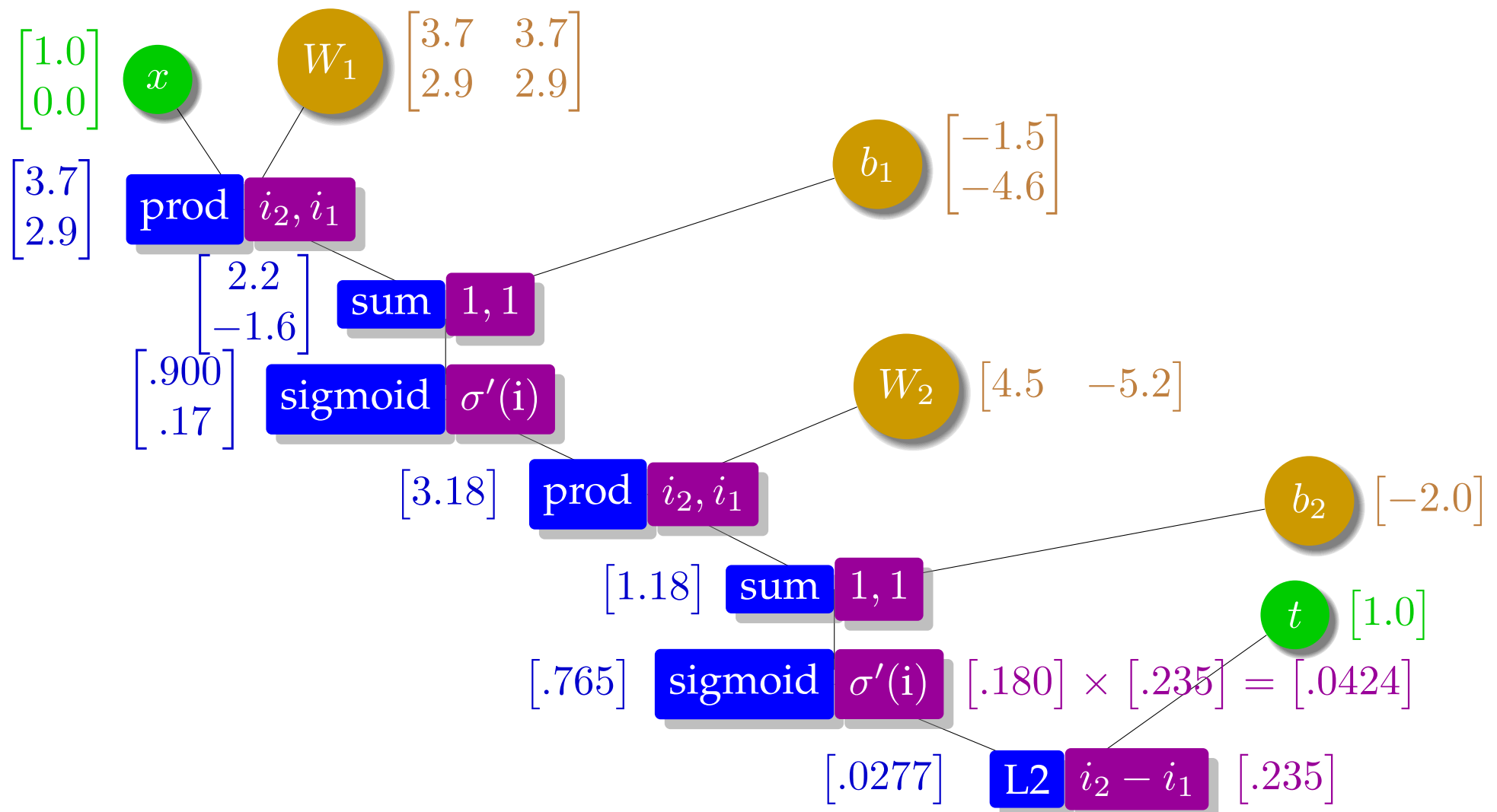
# Derivatives for Each Node



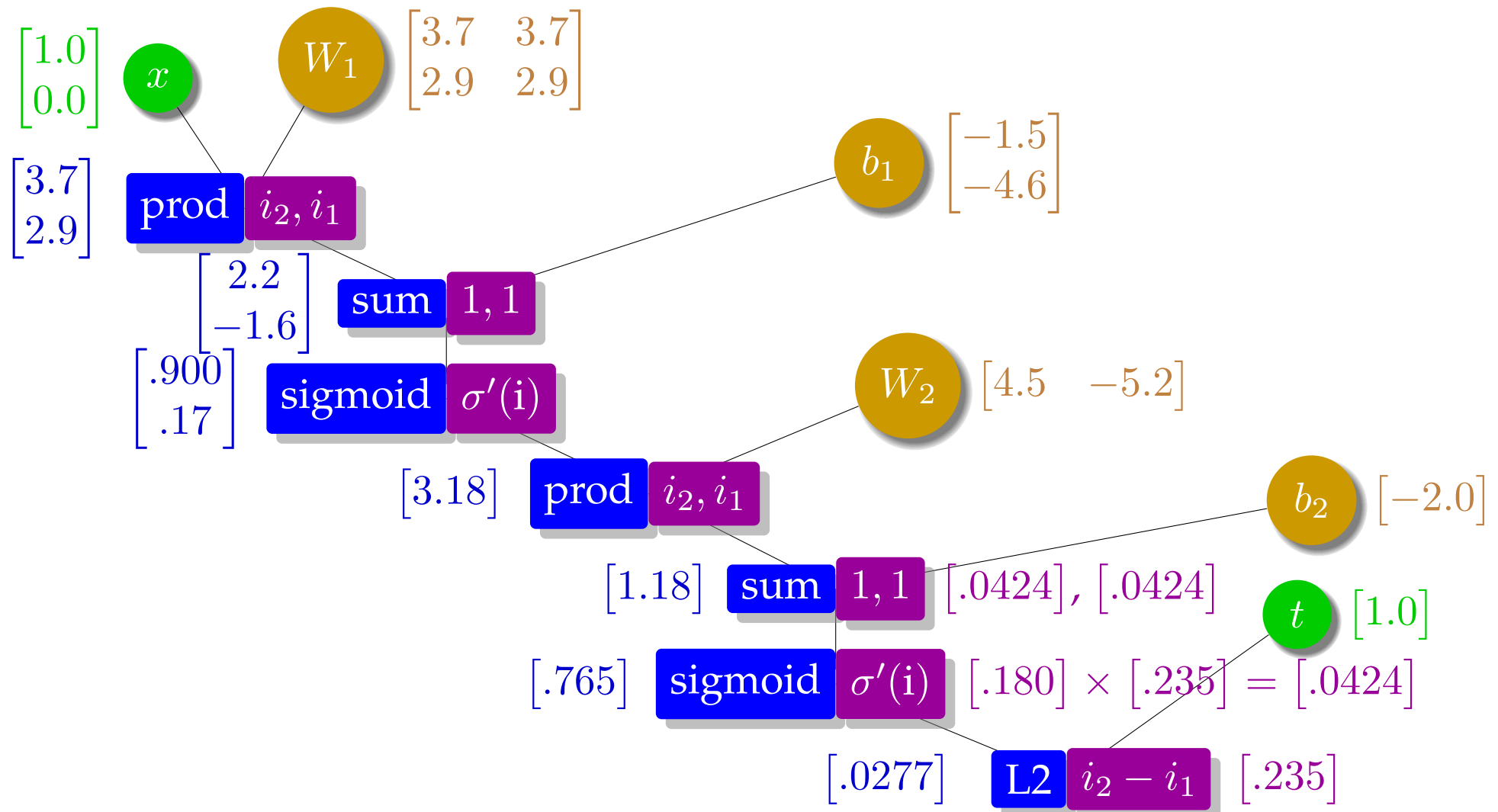
# Backward Pass: Derivative Computation



# Backward Pass: Derivative Computation

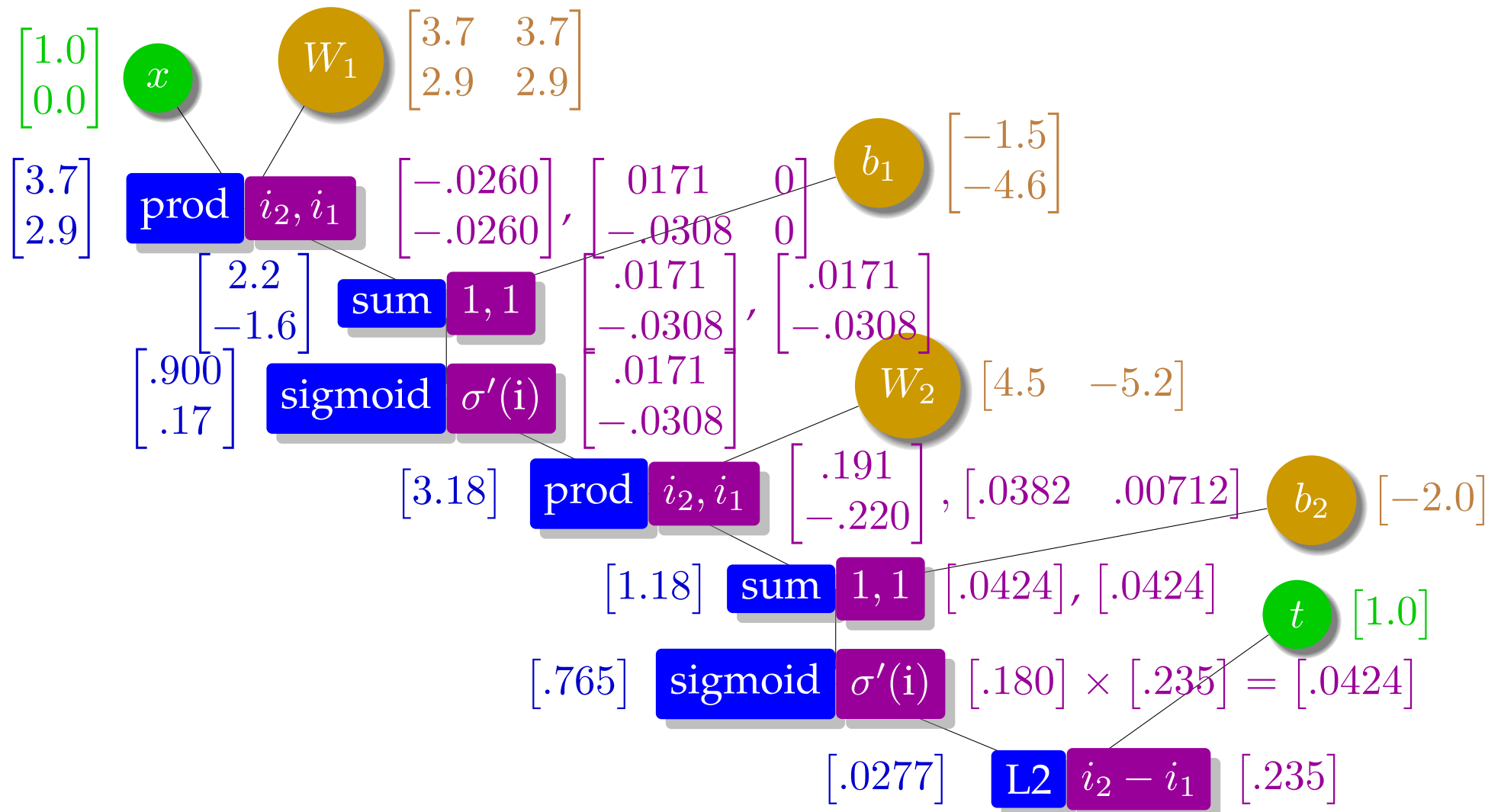


# Backward Pass: Derivative Computation

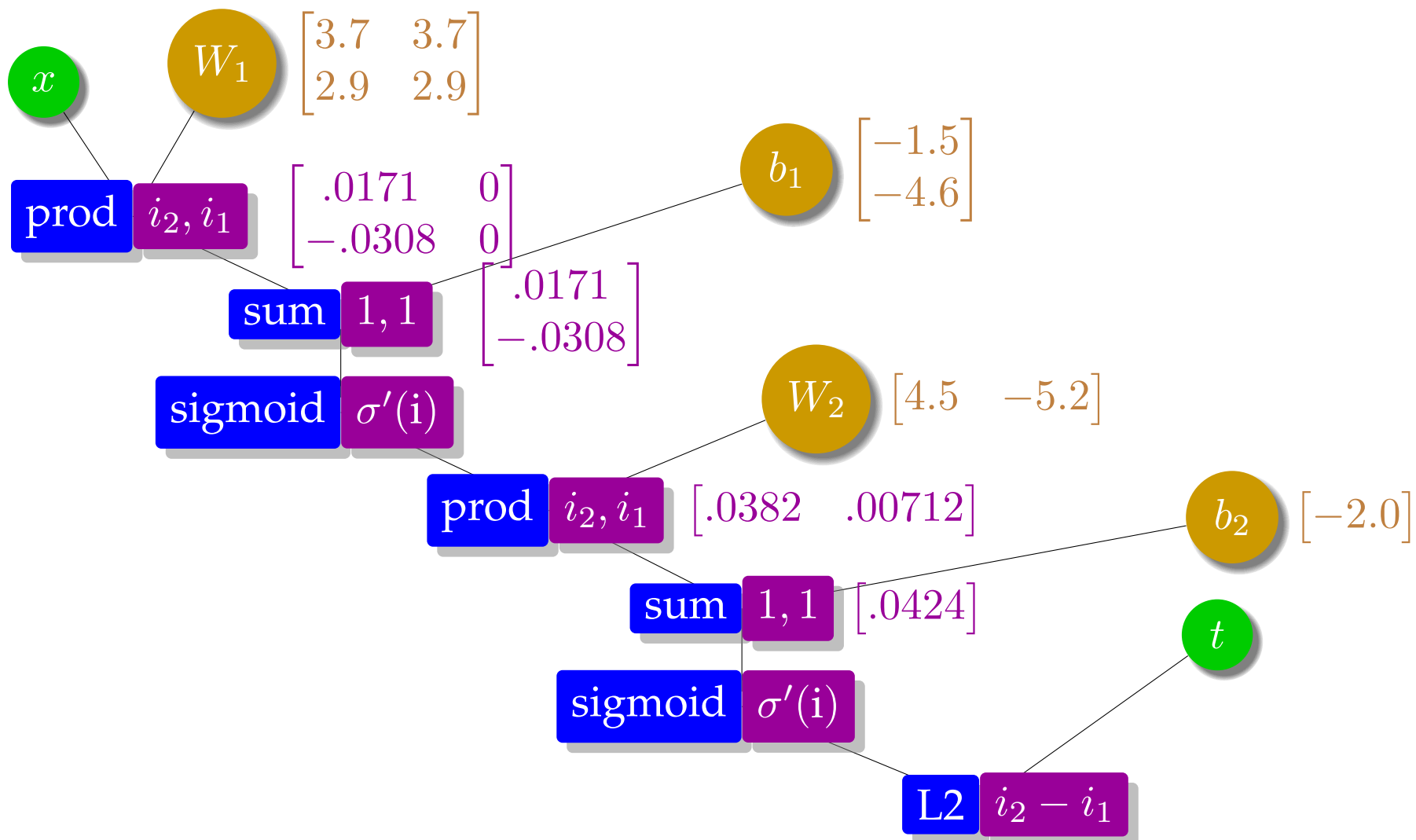




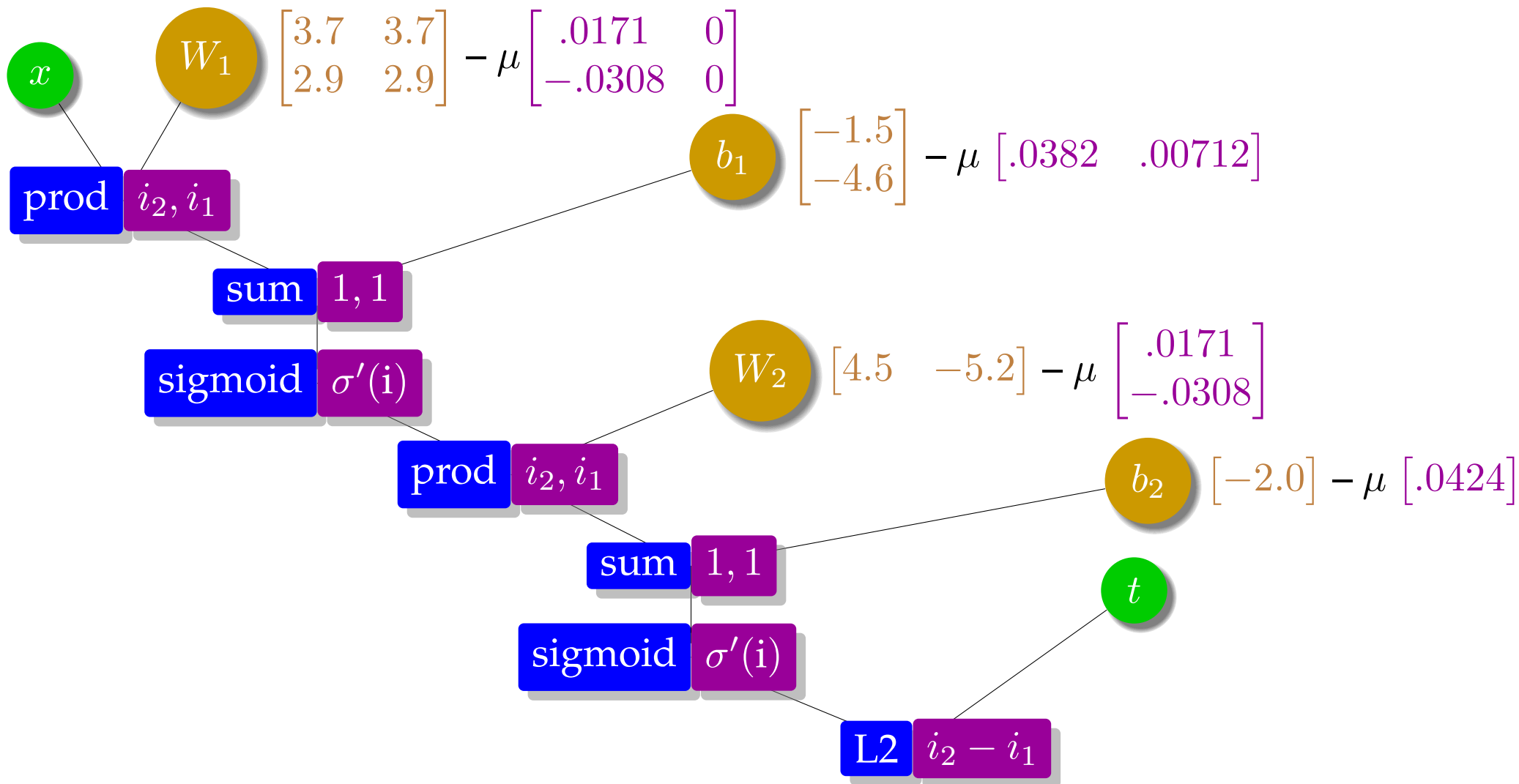
# Backward Pass: Derivative Computation



# Gradients for Parameter Update



# Parameter Update



# toolkits

# Explosion of Deep Learning Toolkits



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

- 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

# Example: Theano

- Deep learning toolkit for Python
- Included as library
  - > `import numpy`
  - > `import theano`
  - > `import theano.tensor as T`

# Example: Theano

- Definition of parameters

```
> x = T.dmatrix()  
> W = theano.shared(value=numpy.array([[3.0, 2.0], [4.0, 3.0]]))  
> b = theano.shared(value=numpy.array([-2.0, -4.0]))
```

- Definition of feed-forward layer

```
> h = T.nnet.sigmoid(T.dot(x,W)+b)
```

note:  $x$  is matrix  $\rightarrow$  process several training examples (sequence of vectors).

- Define as callable function

```
> h_function = theano.function([x], h)
```

- Apply to data

```
> h_function([[1,0]])  
array([[ 0.73105858,  0.11920292]])
```



# Example: Theano

- Same setup for hidden→output layer

```
W2 = theano.shared(value=numpy.array([5.0, -5.0] ))  
b2 = theano.shared(-2.0)  
y_pred = T.nnet.sigmoid(T.dot(h,W2)+b2)
```

- Define as callable function > `predict = theano.function([x], y_pred)`

- Apply to data

```
> predict([[1,0]])  
array([[ 0.7425526]])
```

# Model Training

- First, define the variable for the correct output

```
> y = T.dvector()
```

- Definition of a cost function (we use the L2 norm).

```
> l2 = (y-y_pred)**2
```

```
> cost = l2.mean()
```

- Gradient descent training: computation of the derivative

```
> gW, gb, gW2, gb2 = T.grad(cost, [W,b,W2,b2])
```

- Update rule (with learning rate 0.1)

```
> train = theano.function(inputs=[x,y], outputs=[y_pred, cost],  
    updates=((W, W-0.1*gW), (b, b-0.1*gb),  
            (W2, W2-0.1*gW2), (b2, b2-0.1*gb2)))
```

# Model Training

- Training data

```
> DATA_X = numpy.array([[0,0],[0,1],[1,0],[1,1]])  
> DATA_Y = numpy.array([0,1,1,0])
```

- Predict output for training data

```
> predict(DATA_X)  
array([ 0.18333462, 0.7425526 , 0.7425526 , 0.33430961])
```

- Train with training data

```
> train(DATA_X,DATA_Y)
[array([ 0.18333462, 0.7425526 , 0.7425526 , 0.33430961]),
array(0.06948320612438118)]
```

- Prediction after training

```
> train(DATA_X,DATA_Y)
[array([ 0.18353091, 0.74260499, 0.74321824, 0.33324929]),
array(0.06923193686092949)]
```

example: dynet

- Our example: static computation graph, fixed set of data
  - But: language requires different computation data for different data items (sentences have different length)
- ⇒ Dynamically create a computation graph for each data item

## Example: Dynet

```
model = dy.model()
W_p = model.add_parameters((20, 100))
b_p = model.add_parameters(20)
E = model.add_lookup_parameters((20000, 50))
trainer = dy.SimpleSGDTrainer(model)
for epoch in range(num_epochs):
    for in_words, out_label in training_data:
        dy.renew_cg()
        W = dy.parameter(W_p)
        b = dy.parameter(b_p)
        score_sym = dy.softmax(
            W*dy.concatenate([E[in_words[0]],E[in_words[1]]])+b)
        loss_sym = dy.pickneglogsoftmax(score_sym, out_label)
        loss_sym.forward()
        loss_sym.backward()
        trainer.update()
```

# Model Parameters

```
model = dy.model()
W_p = model.add_parameters((20, 100))
b_p = model.add_parameters(20)
E = model.add_lookup_parameters((20000, 50))
trainer = dy.SimpleSGDTrainer(model)
for epoch in range(num_epochs):
    for in_words, out_label in training_data:
        dy.renew_cg()
        W = dy.parameter(W_p)
        b = dy.parameter(b_p)
        score_sym = dy.softmax(
            W*dy.concatenate([E[in_words[0]],E[in_words[1]]])+b)
        loss_sym = dy.pickneglogsoftmax(score_sym, out_label)
        loss_sym.forward()
        loss_sym.backward()
        trainer.update()
```

Model holds the values for the weight matrices and weight vectors



# Training Setup

```
model = dy.model()
W_p = model.add_parameters((20, 100))
b_p = model.add_parameters(20)
E = model.add_lookup_parameters((20000, 50))
trainer = dy.SimpleSGDTrainer(model)
for epoch in range(num_epochs):
    for in_words, out_label in training_data:
        dy.renew_cg()
        W = dy.parameter(W_p)
        b = dy.parameter(b_p)
        score_sym = dy.softmax(
            W*dy.concatenate([E[in_words[0]],E[in_words[1]]])+b)
        loss_sym = dy.pickneglogsoftmax(score_sym, out_label)
        loss_sym.forward()
        loss_sym.backward()
        trainer.update()
```

Defines the model update function (could be also Adagrad, Adam, ...)

# Setting up Computation Graph



```
model = dy.model()
W_p = model.add_parameters((20, 100))
b_p = model.add_parameters(20)
E = model.add_lookup_parameters((20000, 50))
trainer = dy.SimpleSGDTrainer(model)
for epoch in range(num_epochs):
    for in_words, out_label in training_data:
        dy.renew_cg()
        W = dy.parameter(W_p)
        b = dy.parameter(b_p)
        score_sym = dy.softmax(
            W*dy.concatenate([E[in_words[0]],E[in_words[1]]])+b)
        loss_sym = dy.pickneglogsoftmax(score_sym, out_label)
        loss_sym.forward()
        loss_sym.backward()
        trainer.update()
```

Create a new computation graph. Inform it about parameters.

# Operations

```
model = dy.model()
W_p = model.add_parameters((20, 100))
b_p = model.add_parameters(20)
E = model.add_lookup_parameters((20000, 50))
trainer = dy.SimpleSGDTrainer(model)
for epoch in range(num_epochs):
    for in_words, out_label in training_data:
        dy.renew_cg()
        W = dy.parameter(W_p)
        b = dy.parameter(b_p)
        score_sym = dy.softmax(
            W*dy.concatenate([E[in_words[0]],E[in_words[1]]])+b)
        loss_sym = dy.pickneglogsoftmax(score_sym, out_label)
        loss_sym.forward()
        loss_sym.backward()
        trainer.update()
```

Builds the computation graph by defining operations.

# Training Loop

```
model = dy.model()
W_p = model.add_parameters((20, 100))
b_p = model.add_parameters(20)
E = model.add_lookup_parameters((20000, 50))
trainer = dy.SimpleSGDTrainer(model)
for epoch in range(num_epochs):
    for in_words, out_label in training_data:
        dy.renew_cg()
        W = dy.parameter(W_p)
        b = dy.parameter(b_p)
        score_sym = dy.softmax(
            W*dy.concatenate([E[in_words[0]],E[in_words[1]]])+b)
        loss_sym = dy.pickneglogsoftmax(score_sym, out_label)
        loss_sym.forward()
        loss_sym.backward()
        trainer.update()
```

Process training data. Computations are done in forward and backward.