Statistical Machine Translation LING-462/COSC-482 Week 7: Neural Networks

Achim Ruopp
achim.ruopp@Georgetown.edu

Agenda

- Language in ten minutes: Yushi Zhao
- Neural Networks

- Break -

- Neural Networks: Computation Graphs
- Deep Learning Frameworks
  - Tensorflow Playground
  - Keras

NEURAL NETWORKS

Weighted Model

- Described standard model consists of three sub-models
  - phrase translation model $\phi(\bar{f}|\bar{e})$
  - reordering model d
  - language model $p_{LM}(e)$

$$e_{\text{best}} = \text{argmax}_e \prod_{i=1}^{I} \phi(\bar{f}_i|\bar{e}_i)\; d(start_i - end_{i-1} - 1) \; \prod_{i=1}^{|\mathbf{e}|} p_{LM}(e_i|e_1...e_{i-1})$$

- Some sub-models may be more important than others

- Add weights λ_ϕ, λ_d, λ_{LM}

$$e_{\text{best}} = \text{argmax}_e \prod_{i=1}^{I} \phi(\bar{f}_i|\bar{e}_i)^{\lambda_\phi}\; d(start_i{-}end_{i-1}{-}1)^{\lambda_d} \; \prod_{i=1}^{|\mathbf{e}|} p_{LM}(e_i|e_1...e_{i-1})^{\lambda_{LM}}$$

Log-Linear Model

- Such a weighted model is a log-linear model:

$$p(x) = \exp \sum_{i=1}^{n} \lambda_i h_i(x)$$

- Our feature functions
  - number of feature function $n = 3$
  - random variable $x = (e, f, start, end)$
  - feature function $h_1 = \log \phi$
  - feature function $h_2 = \log d$
  - feature function $h_3 = \log p_{\text{LM}}$

Weighted Model as Log-Linear Model

$$p(e,a|f) = \exp(\lambda_\phi \sum_{i=1}^{I} \log \phi(\bar{f}_i|\bar{e}_i) +$$

$$\lambda_d \sum_{i=1}^{I} \log d(a_i - b_{i-1} - 1) +$$

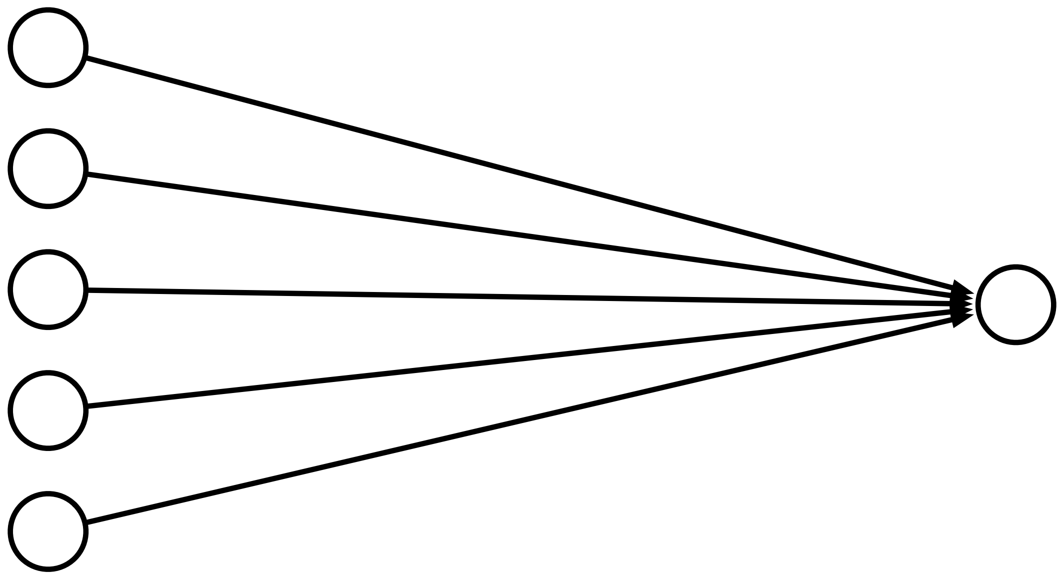
$$\lambda_{LM} \sum_{i=1}^{|\mathbf{e}|} \log p_{LM}(e_i|e_1...e_{i-1}))$$

Linear Models

- We used before weighted linear combination of feature values h_j and weights λ_j

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

- Such models can be illustrated as a "network"



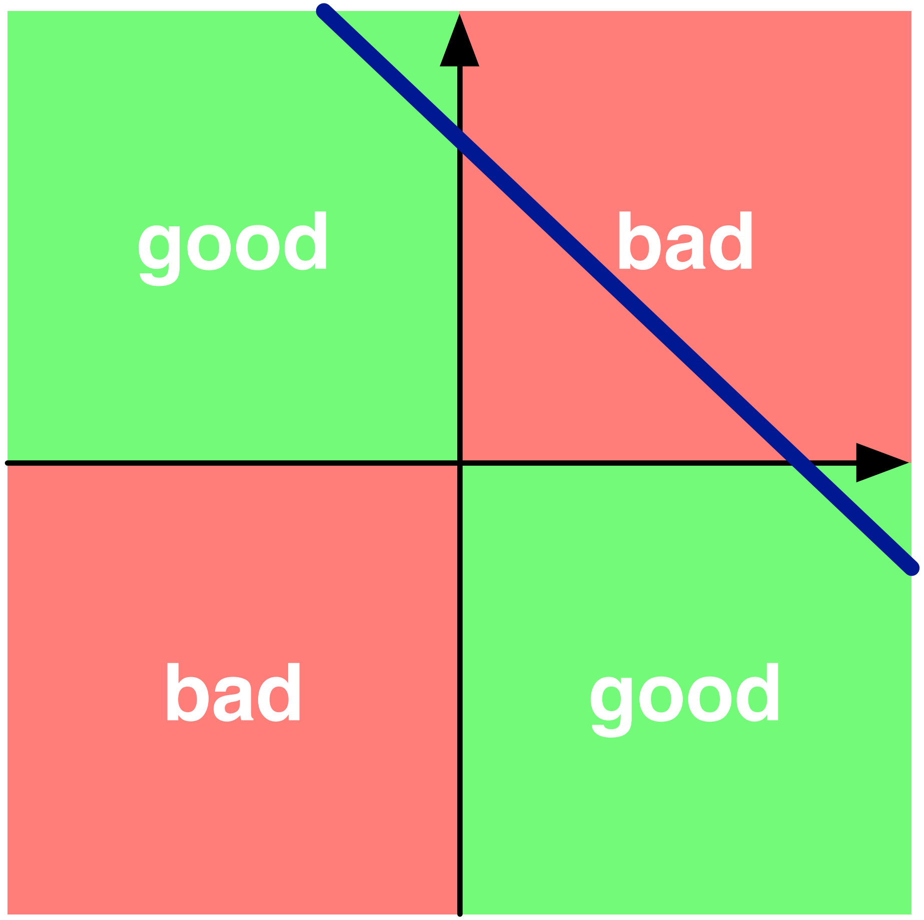
Limits of Linearity

- We can give each feature a weight

- But not more complex value relationships, e.g,
  - any value in the range [0;5] is equally good
  - values over 8 are bad
  - higher than 10 is not worse

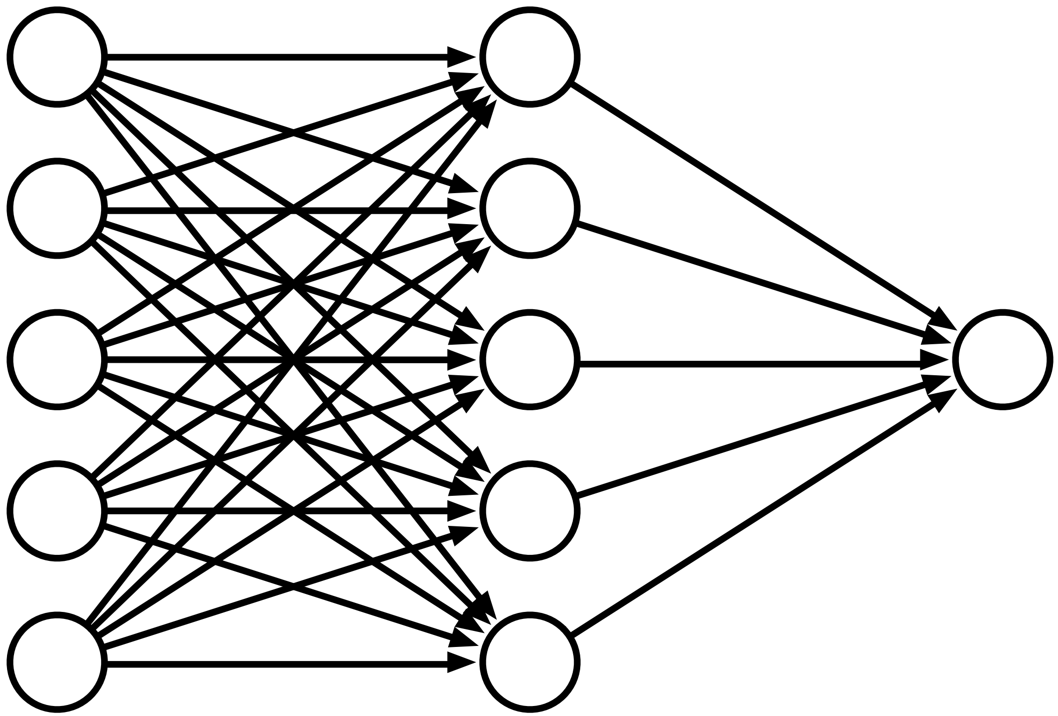
XOR

- 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

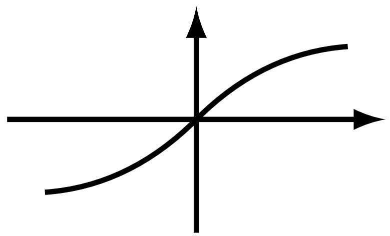
$$\text{score}(\lambda, \mathbf{d}_i) = \sum_j \lambda_j \, h_j(\mathbf{d}_i)$$

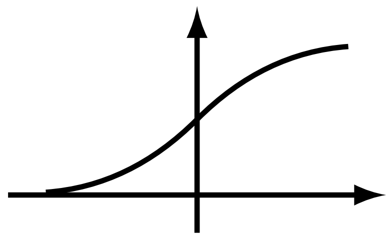
- Add a non-linear function

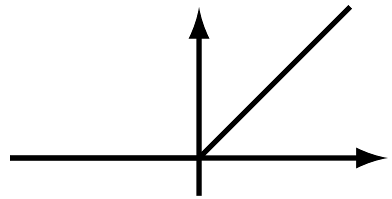
$$\text{score}(\lambda, \mathbf{d}_i) = f\Big(\sum_j \lambda_j \, h_j(\mathbf{d}_i)\Big)$$

- Popular choices

tanh(x)   sigmoid(x) = $\frac{1}{1+e^{-x}}$ relu(x) = max(0,x)



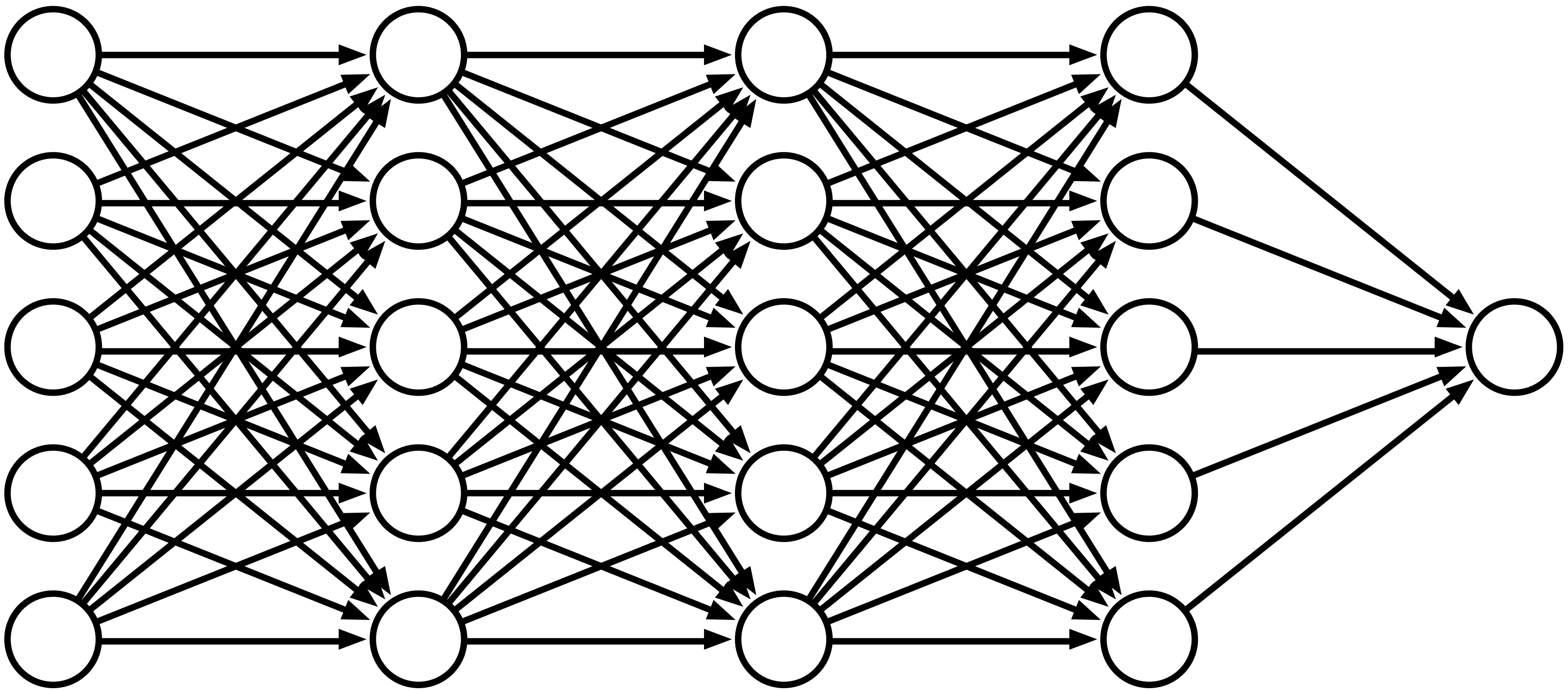




(sigmoid is also called the "logistic function")

Deep Learning

- More layers = deep learning

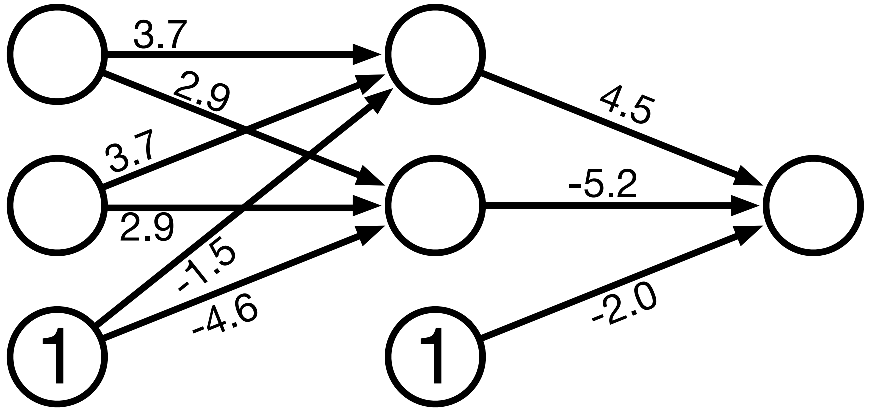


What Depths Holds

- Each layer is a processing step
- Having multiple processing steps allows complex functions
- Metaphor: NN and computing circuits
  - computer = sequence of Boolean gates
  - neural computer = sequence of layers
- Deep neural networks can implement complex functions
  e.g., sorting on input values

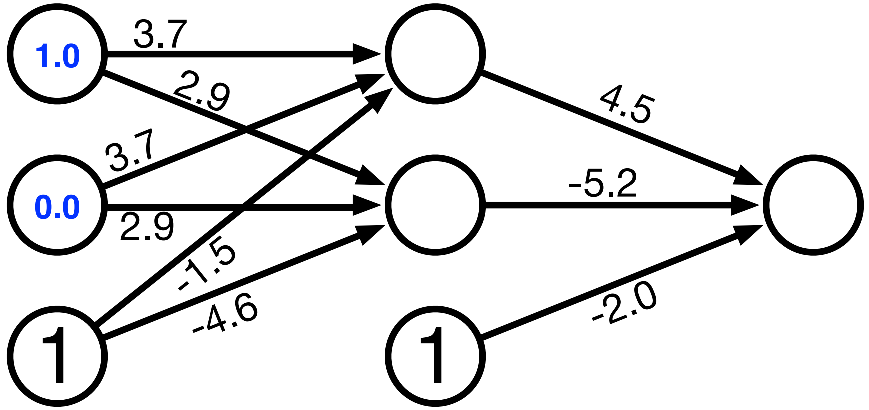
example

Simple Neural Network



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

Sample Input

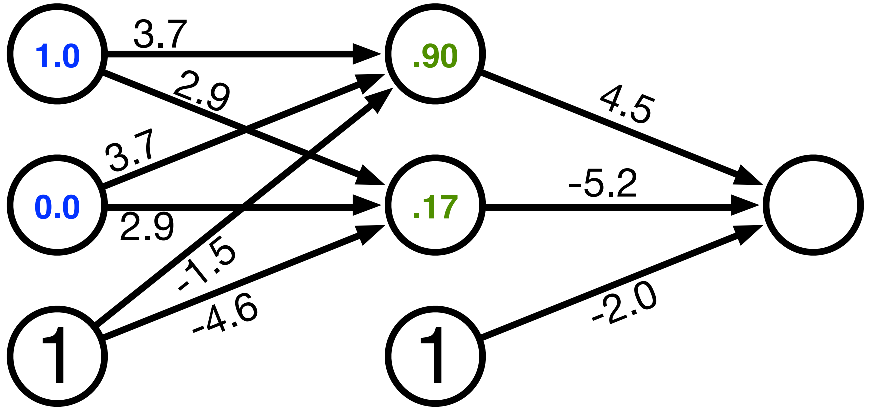


- Try out two input values
- Hidden unit computation

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

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

Computed Hidden

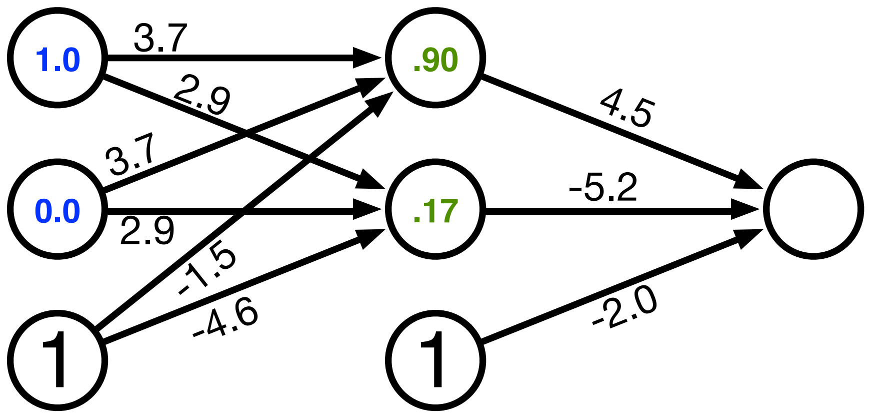


- Try out two input values
- Hidden unit computation

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

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

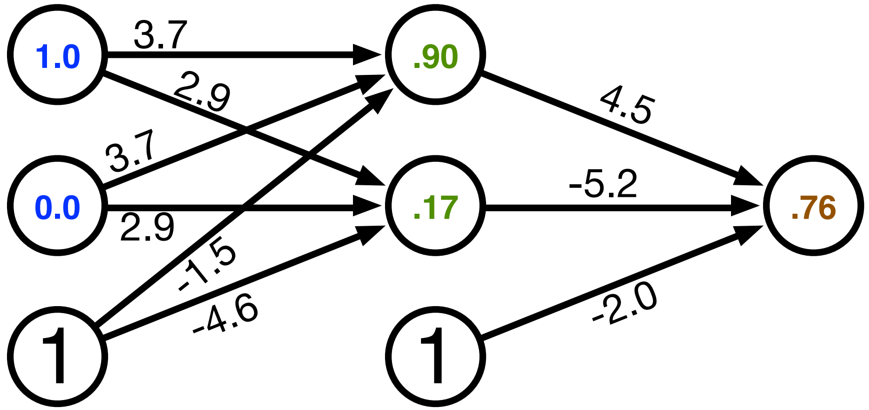
Compute Output



- 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

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

Output for all Binary Inputs

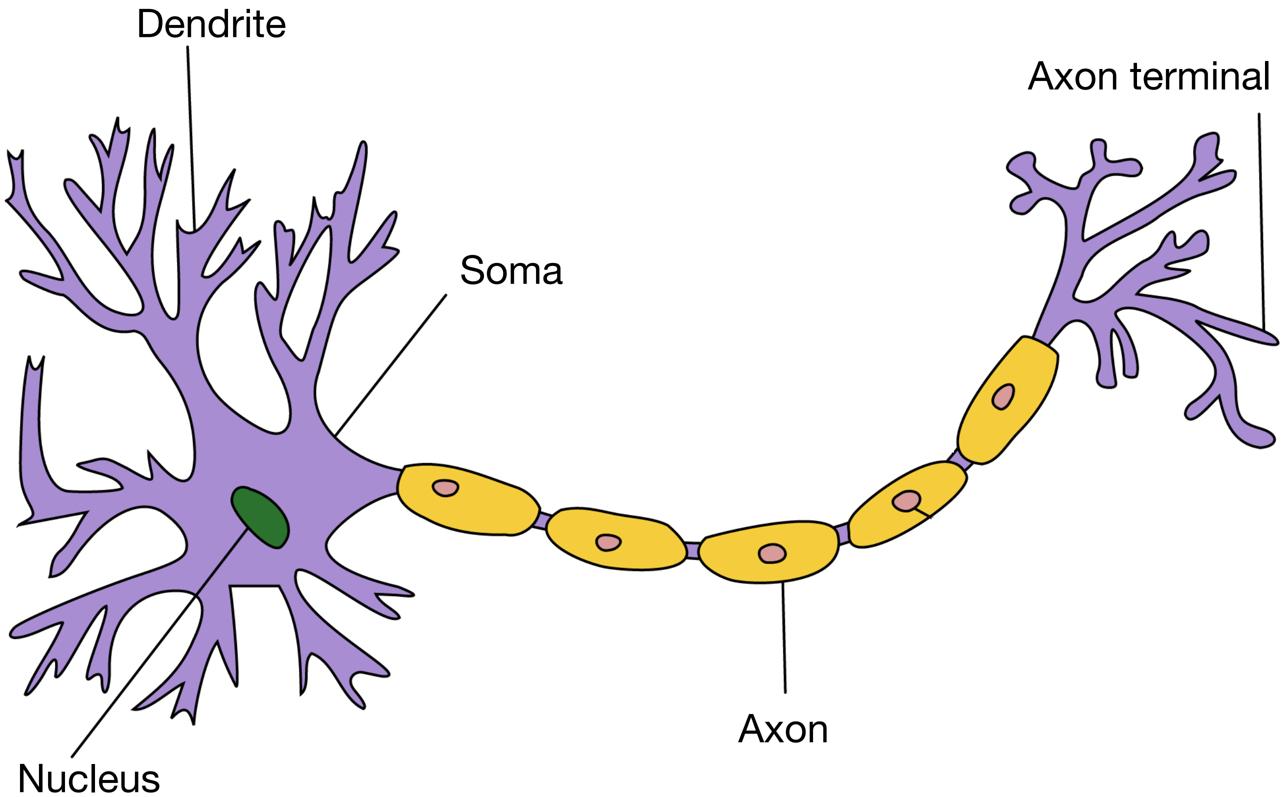
Input x_0	Input x_1	Hidden h_0	Hidden h_1	Output y_0
0	0	0.12	0.02	$0.18 \rightarrow 0$
0	1	0.88	0.27	$0.74 \rightarrow 1$
1	0	0.73	0.12	$0.74 \rightarrow 1$
1	1	0.99	0.73	$0.33 \rightarrow 0$

- Network implements XOR
  - hidden node h_0 is OR
  - hidden node h_1 is AND
  - final layer operation is $h_0 - -h_1$
- Power of deep neural networks: chaining of processing steps
  just as: more Boolean circuits $\rightarrow$ more complex computations possible

why ”neural” networks?

Neuron in the Brain

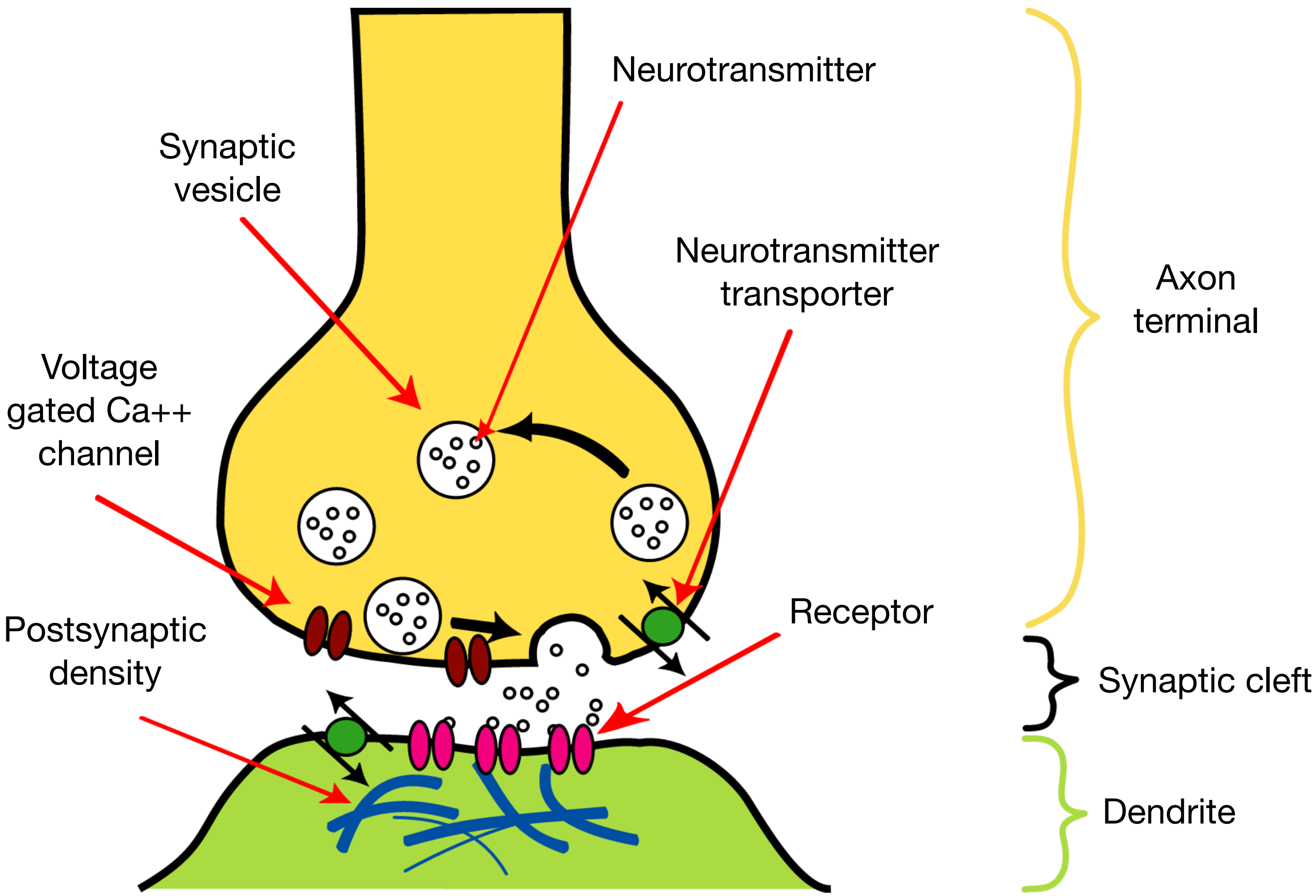
- The human brain is made up of about 100 billion neurons



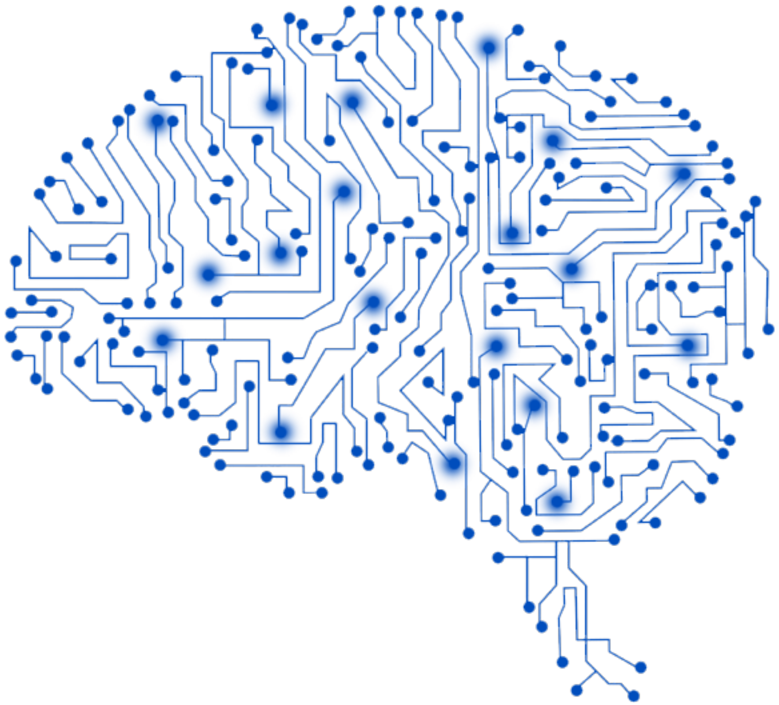
- Neurons receive electric signals at the dendrites and send them to the axon

Neural Communication

- The axon of the neuron is connected to the dendrites of many other neurons



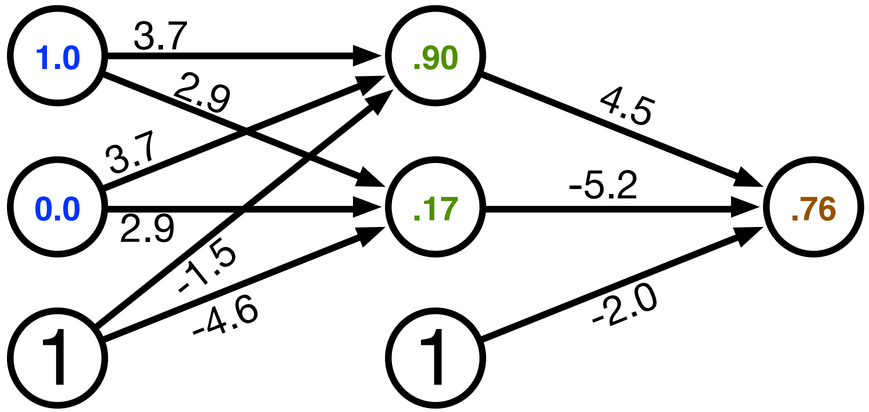
The Brain vs. Artificial Neural Networks



- 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: $t = 1.0$

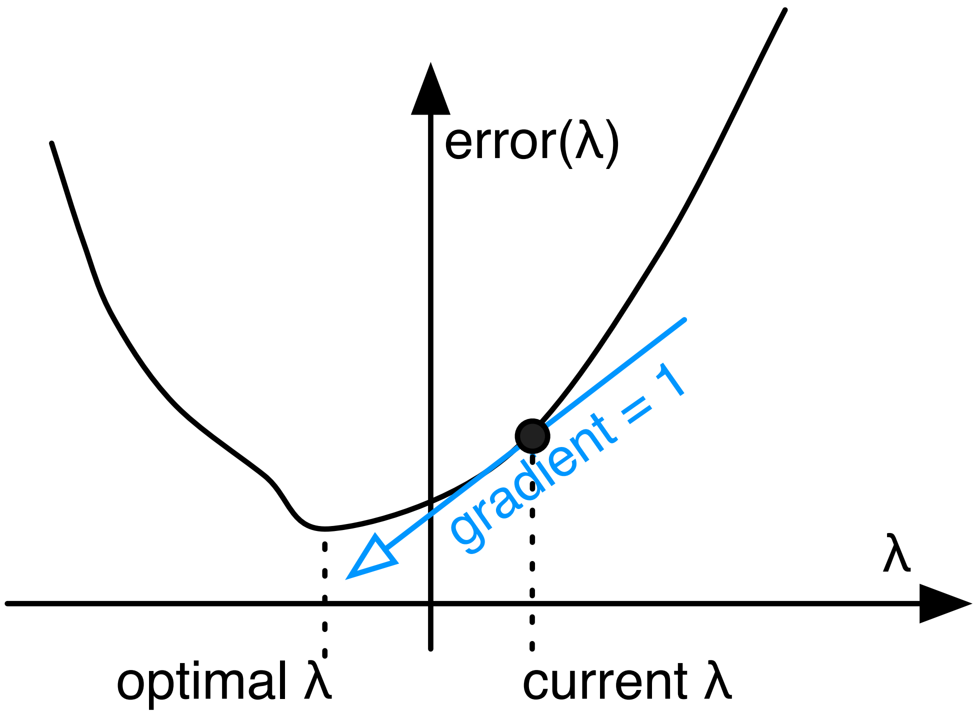
$\Rightarrow$ How do we adjust the weights?

Key Concepts

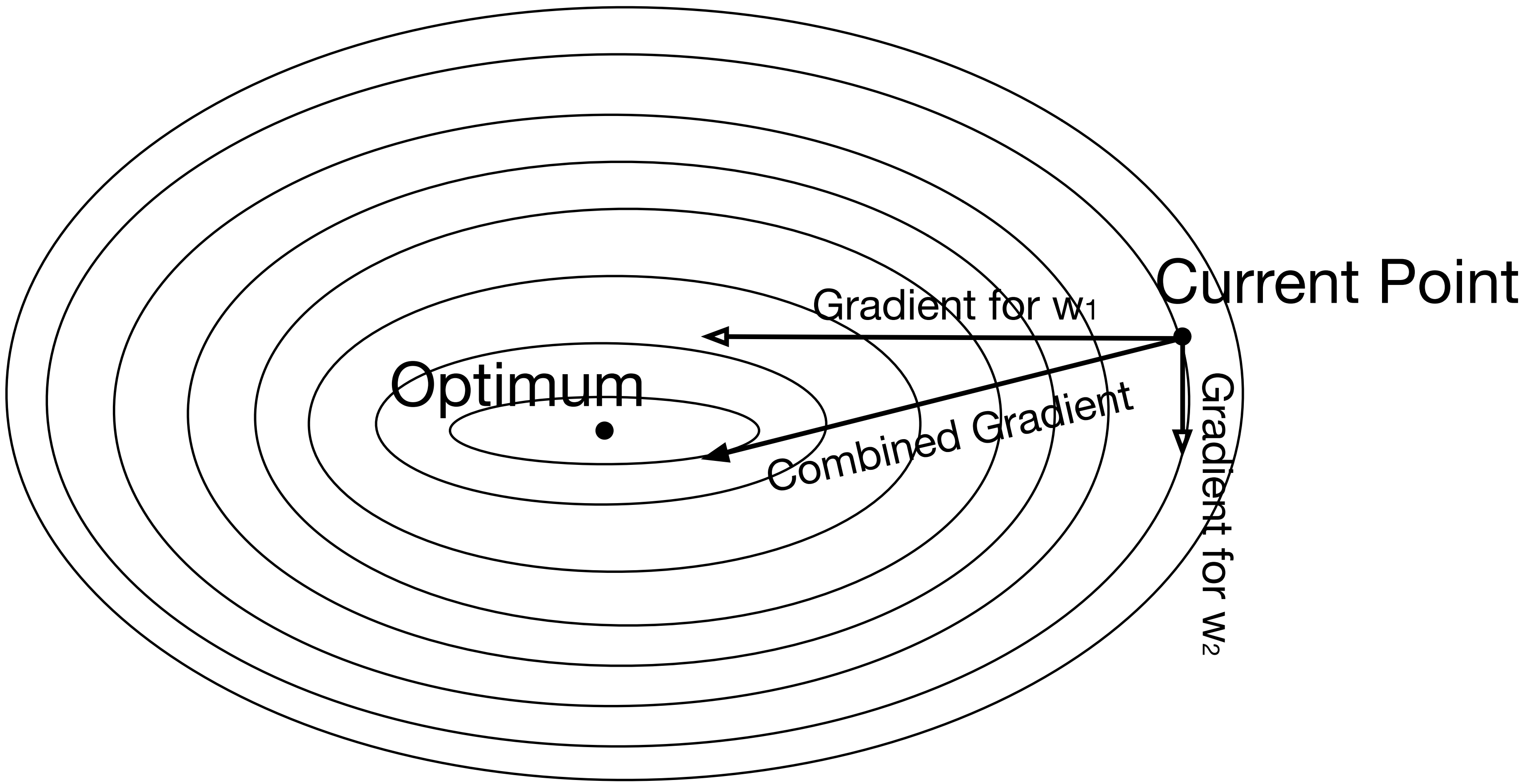
- Gradient descent
  - error is a function of the weights
  - we want to reduce the error
  - gradient descent: move towards the error minimum
  - compute gradient $\rightarrow$ get direction to the error minimum
  - adjust weights towards direction of lower error

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

Gradient Descent



Gradient Descent



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

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

Multiple Output Nodes

- Our example only had one output node
- Typically neural networks have multiple output nodes
- Error is computed over all j output nodes

$$E = \sum_j \frac{1}{2}(t_j - y_j)^2$$

- Weights $k \rightarrow j$ are adjusted according to the node they point to

$$\Delta w_{j \leftarrow k} = \mu(t_j - y_j)\; y'_j \; h_k$$

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

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

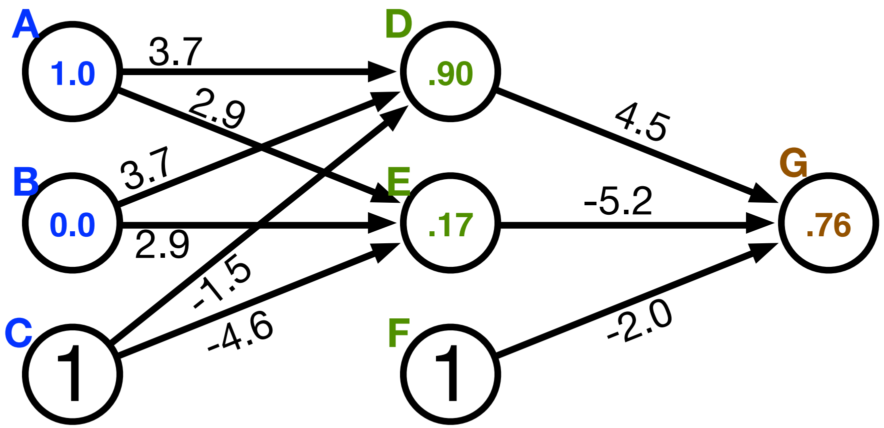
- Back-propagate the error term
  (why this way? there is math to back it up...)

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

- Universal update formula

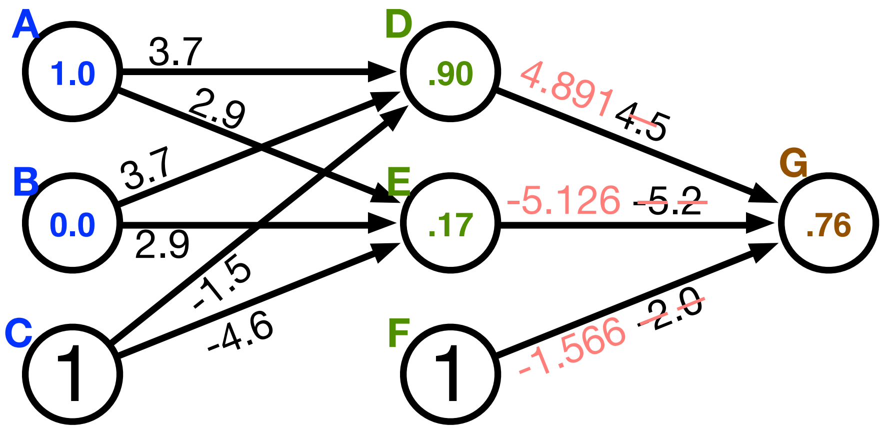
$$\Delta w_{j \leftarrow k} = \mu\ \delta_j\ h_k$$

Our Example



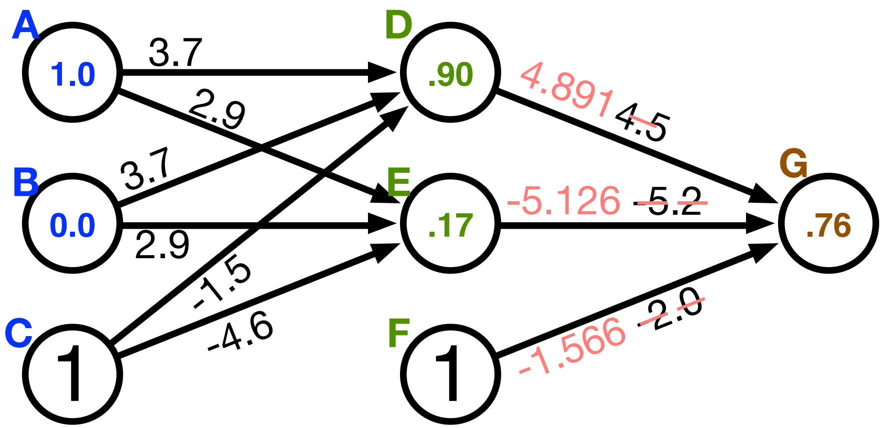
- Computed output: $y = .76$
- Correct output: $t = 1.0$
- Final layer weight updates (learning rate $\mu = 10$)
  - $\delta_{\mathsf{G}} = (t - y)\, y' = (1 - .76)\; 0.181 = .0434$
  - $\Delta w_{\mathsf{GD}} = \mu\; \delta_{\mathsf{G}}\; h_{\mathsf{D}} = 10 \times .0434 \times .90 = .391$
  - $\Delta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

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**
  - $\delta_{\text{E}} = \left(\sum_j w_{j \leftarrow i} \delta_j\right) y'_{\text{E}} = w_{\text{GE}}\ \delta_{\text{G}}\ y'_{\text{E}} = -5.2 \times .0434 \times 0.2055 = -.0464$
  - $\Delta w_{\text{EA}} = \mu\ \delta_{\text{E}}\ h_{\text{A}} = 10 \times -.0464 \times 1.0 = -.464$
  - etc.

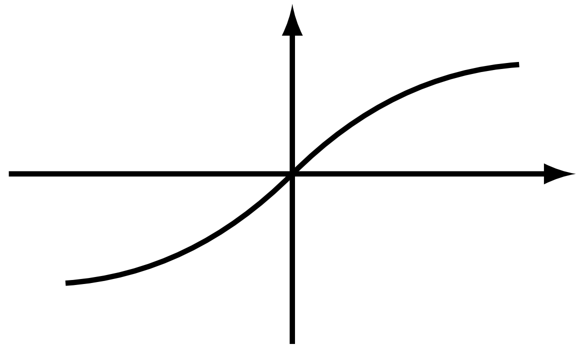
some additional aspects

Initialization of Weights

- Weights are initialized randomly
  e.g., uniformly from interval $[-0.01, 0.01]$

- Glorot and Bengio (2010) suggest
  - for shallow neural networks

$$[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$$



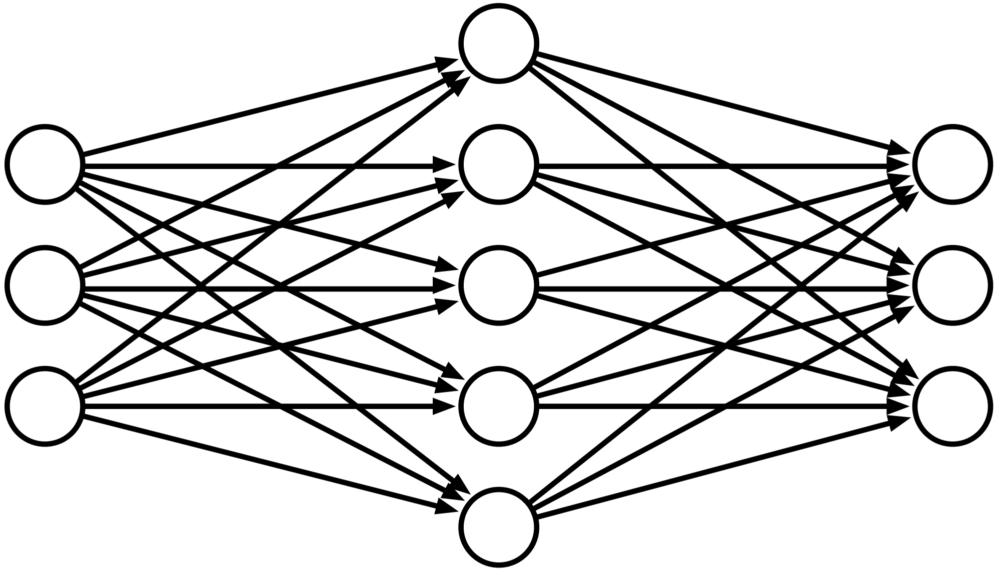
n is the size of the previous layer

  - for deep neural networks

$$[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}]$$

n_j is the size of the previous layer, n_j size of next layer

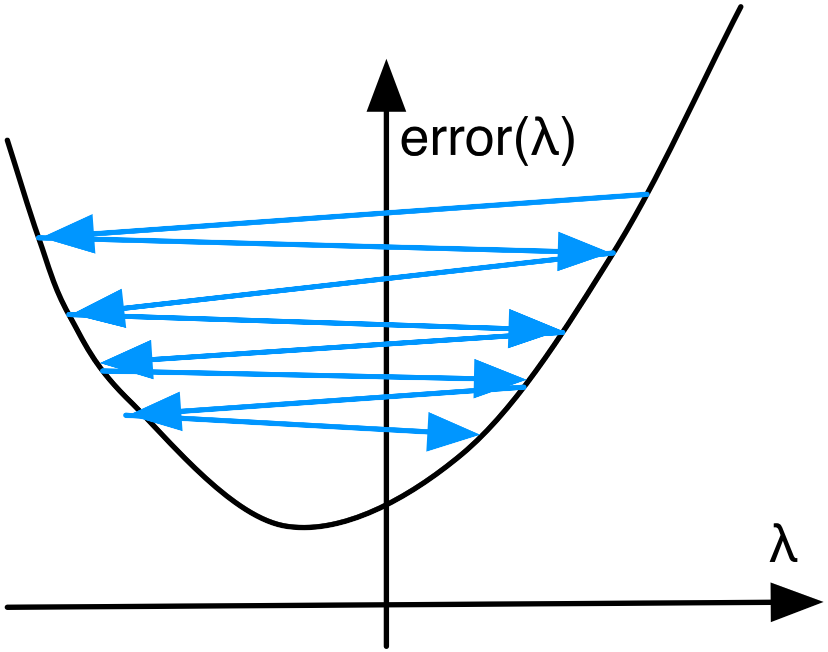
Neural Networks for Classification



- Predict class: one output node per class
- Training data output: "One-hot vector", e.g., $\vec{y} = (0, 0, 1)^T$
- Prediction
  - predicted class is output node y_i with highest value
  - obtain posterior probability distribution by soft-max

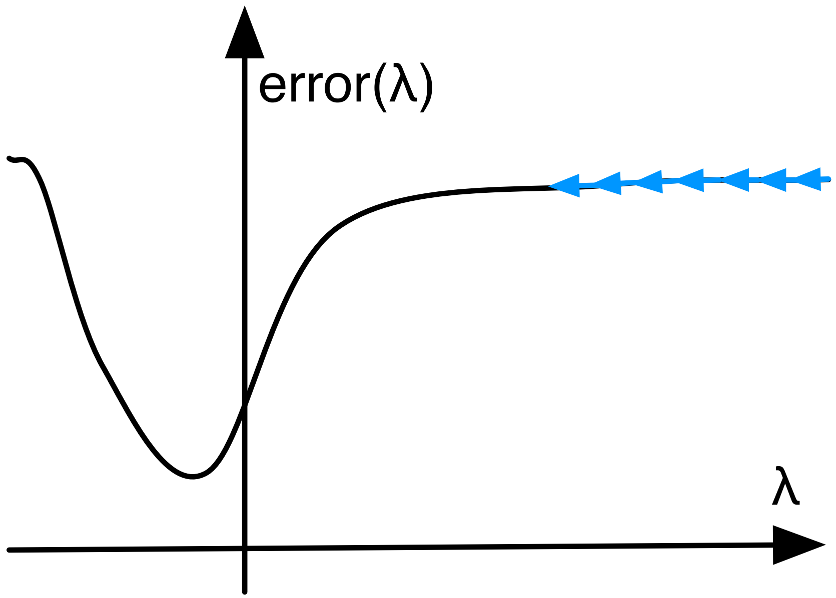
$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Problems with Gradient Descent Training



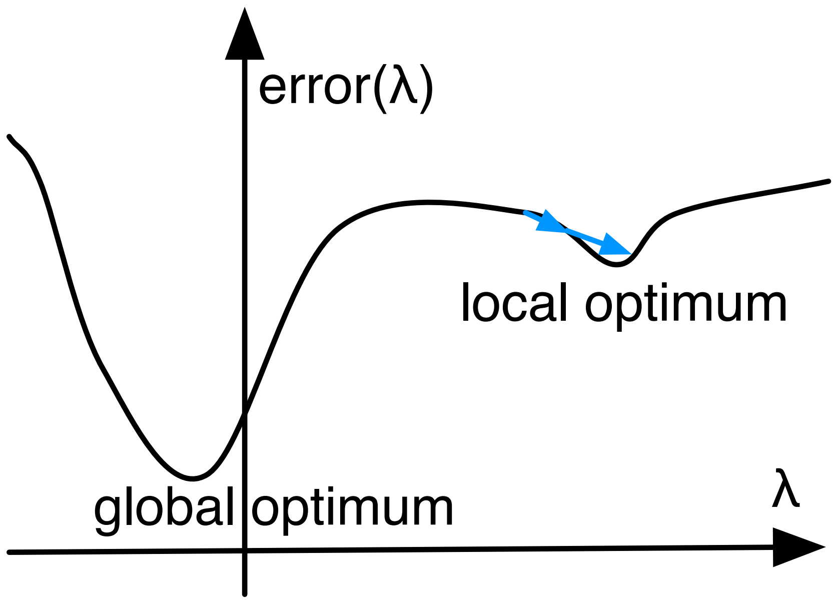
Too high learning rate

Problems with Gradient Descent Training



Bad initialization

Problems with Gradient Descent Training



Local optimum

Speedup: Momentum Term

- Updates may move a weight slowly in one direction
- To speed this up, we can keep a memory of prior updates

$$\Delta w_{j \leftarrow k}(n-1)$$

- ... and add these to any new updates (with decay factor ρ)

$$\Delta w_{j \leftarrow k}(n) = \mu \; \delta_j \; h_k + \rho \Delta w_{j \leftarrow k}(n-1)$$

Adagrad

- Typically reduce the learning rate μ over time
  - at the beginning, things have to change a lot
  - later, just fine-tuning

- Adapting learning rate per parameter

- Adagrad update
  based on error E with respect to the weight w at time t = $g_t = \frac{dE}{dw}$

$$\Delta w_t = \frac{\mu}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2}} g_t$$

Dropout

- A general problem of machine learning: overfitting to training data
  (very good on train, bad on unseen test)
- Solution: **regularization**, e.g., keeping weights from having extreme values
- Dropout: randomly remove some hidden units during training
  - mask: set of hidden units dropped
  - randomly generate, say, 10–20 masks
  - alternate between the masks during training
- Why does that work?
  $\rightarrow$ bagging, ensemble, ...

Mini Batches

- Each training example yields a set of weight updates Δw_i.

- Batch up several training examples
  - sum up their updates
  - apply sum to model

- Mostly done or speed reasons

computational aspects

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: $\Delta W = \mu\vec{\delta}\vec{h}^T$

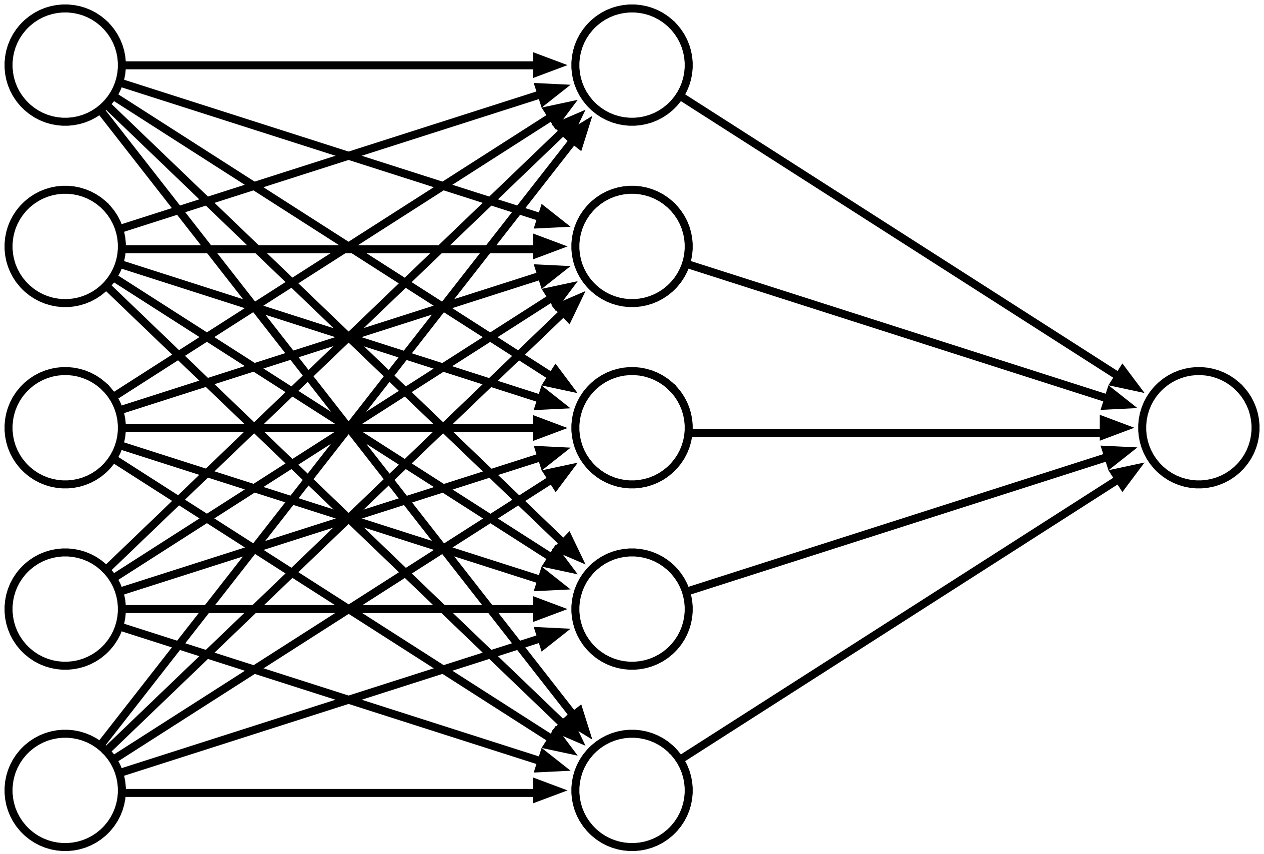
GPU

- 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 Tesla K20c GPU provides 2496 thread processors
- Extensions to C to support programming of GPUs, such as CUDA

NEURAL NETWORKS: COMPUTATIONAL GRAPHS

Neural Network Cartoon

- A common way to illustrate a neural network



Neural Network Math

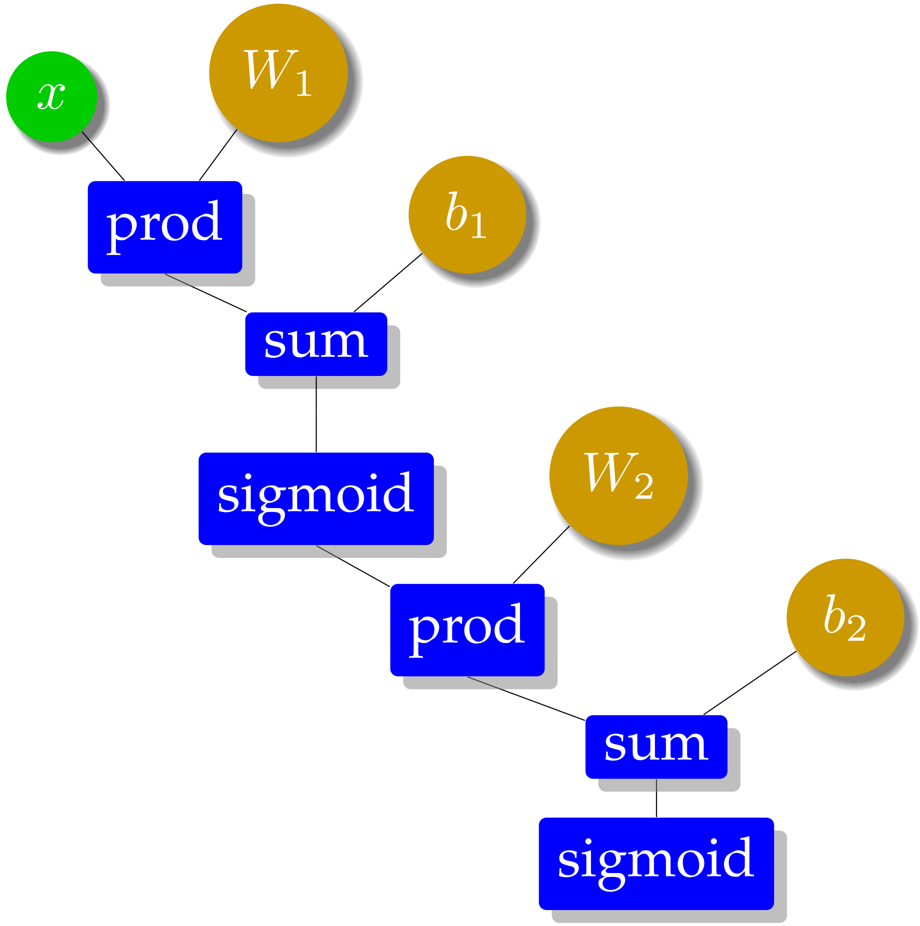
- Hidden layer

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

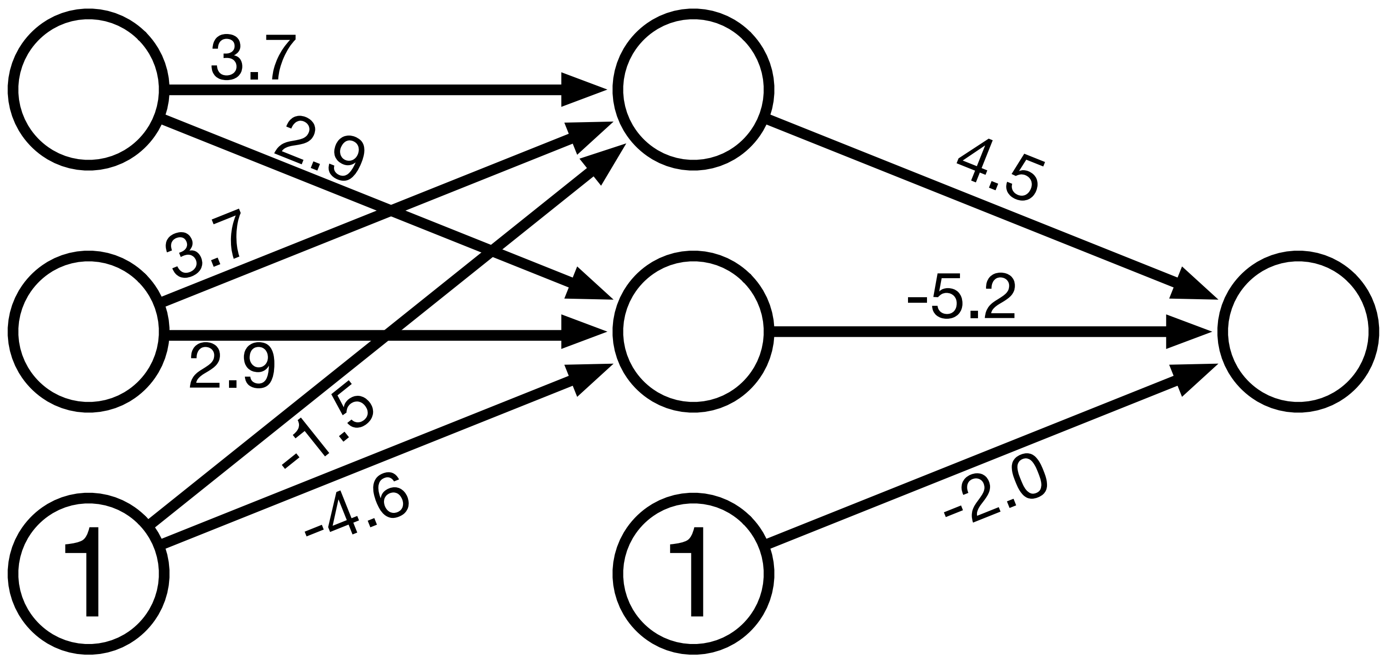
- Final layer

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

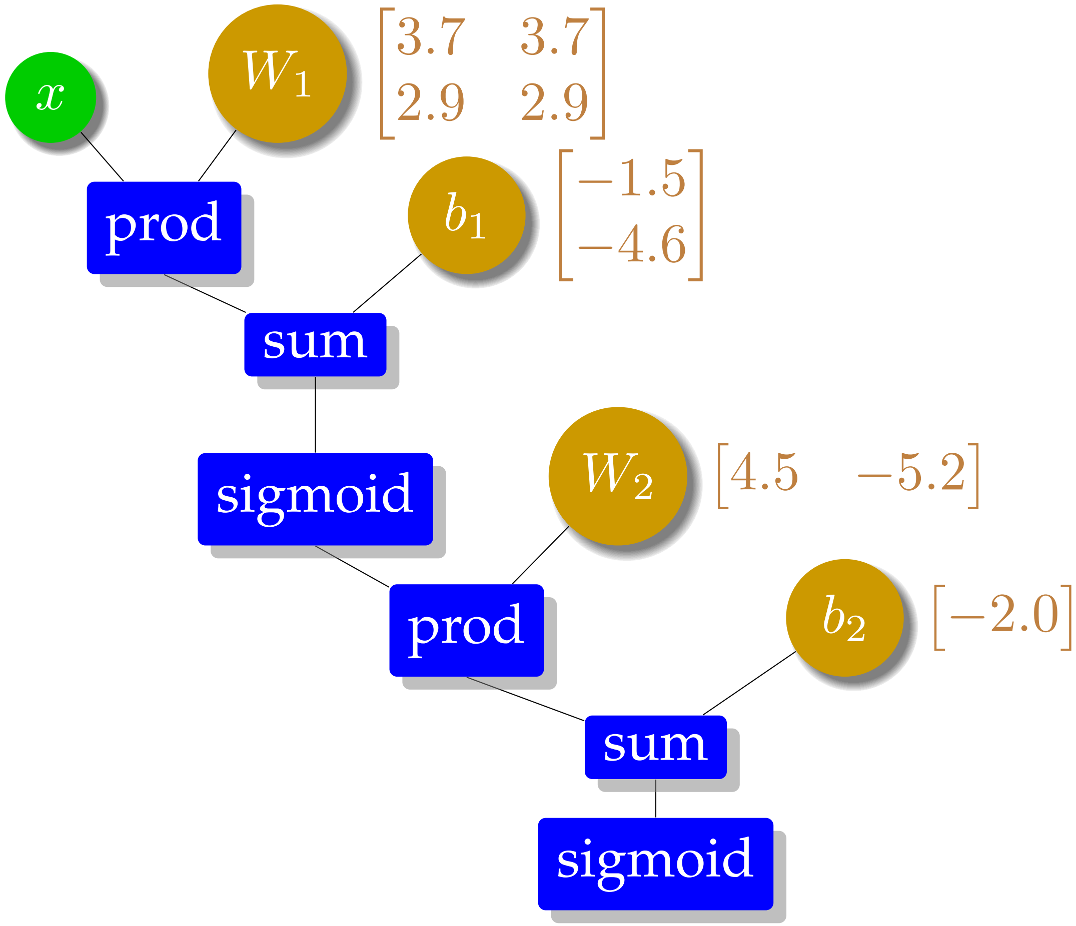
Computation Graph



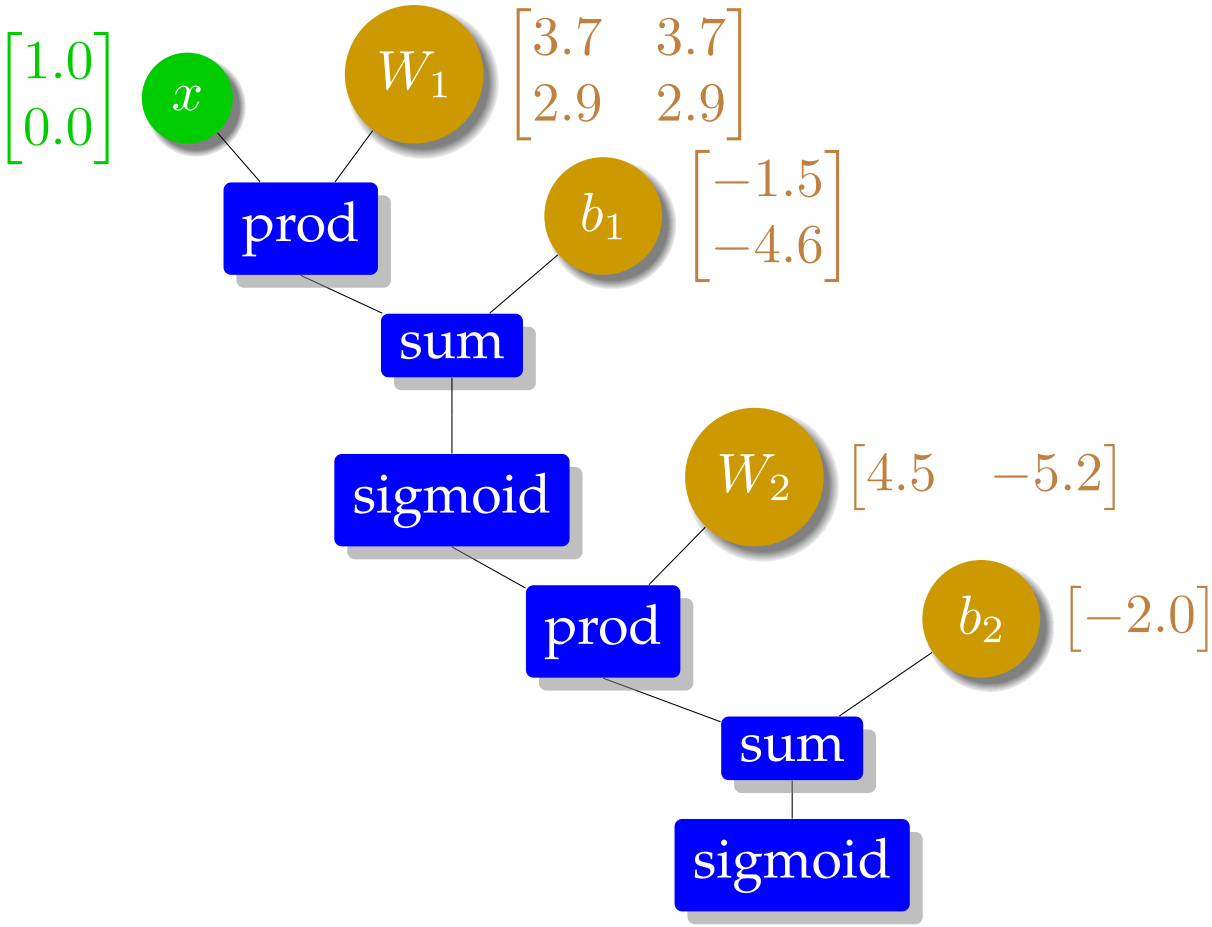
Simple Neural Network



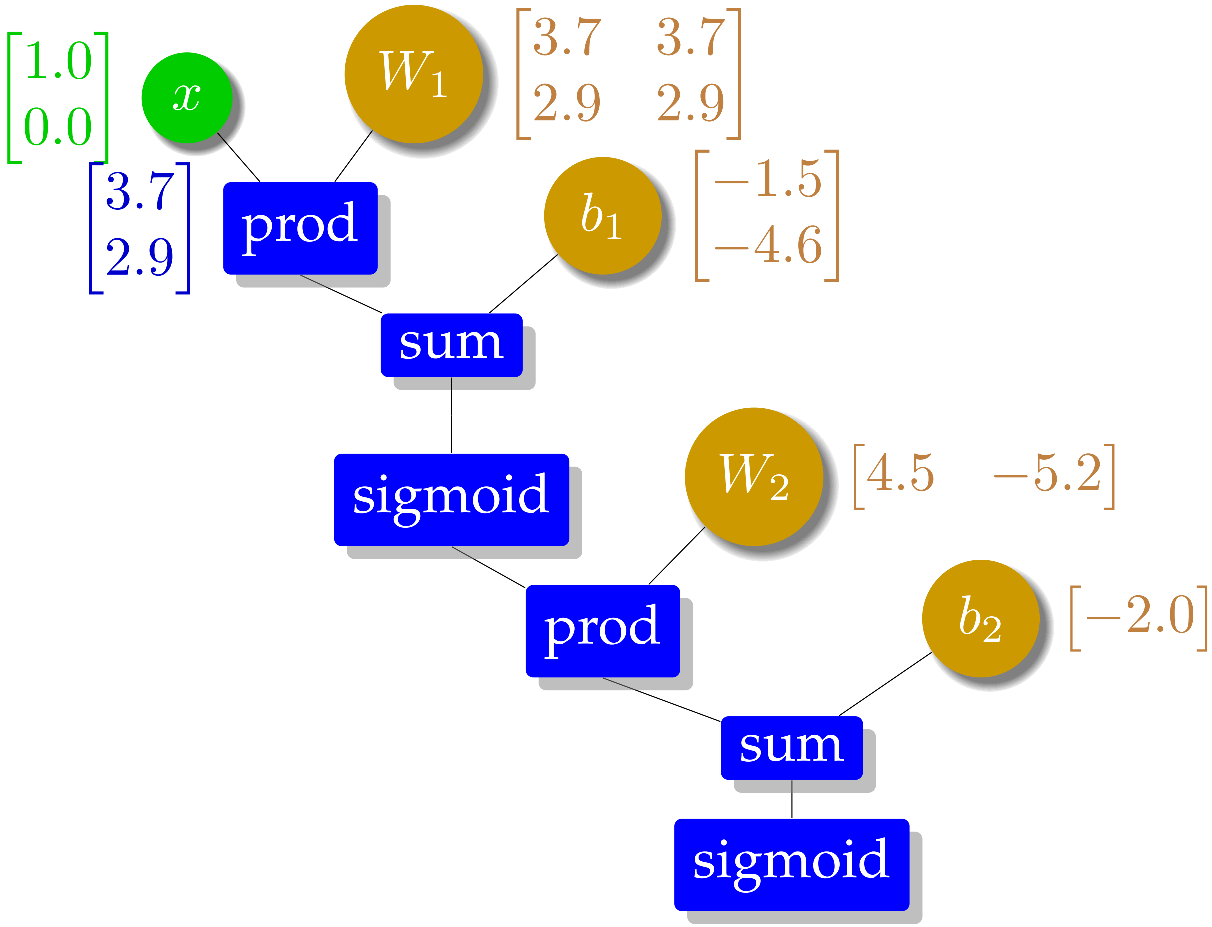
Computation Graph



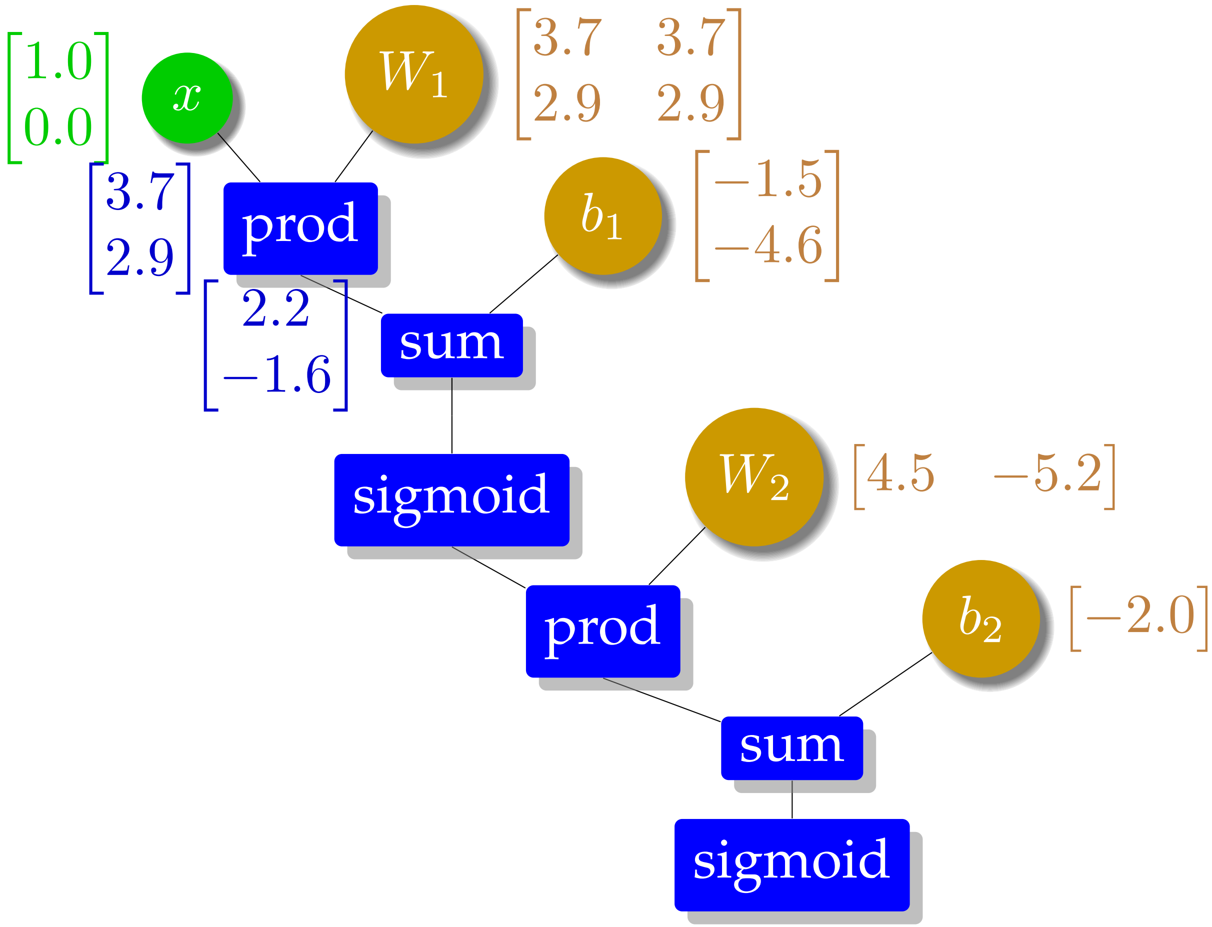
Processing Input



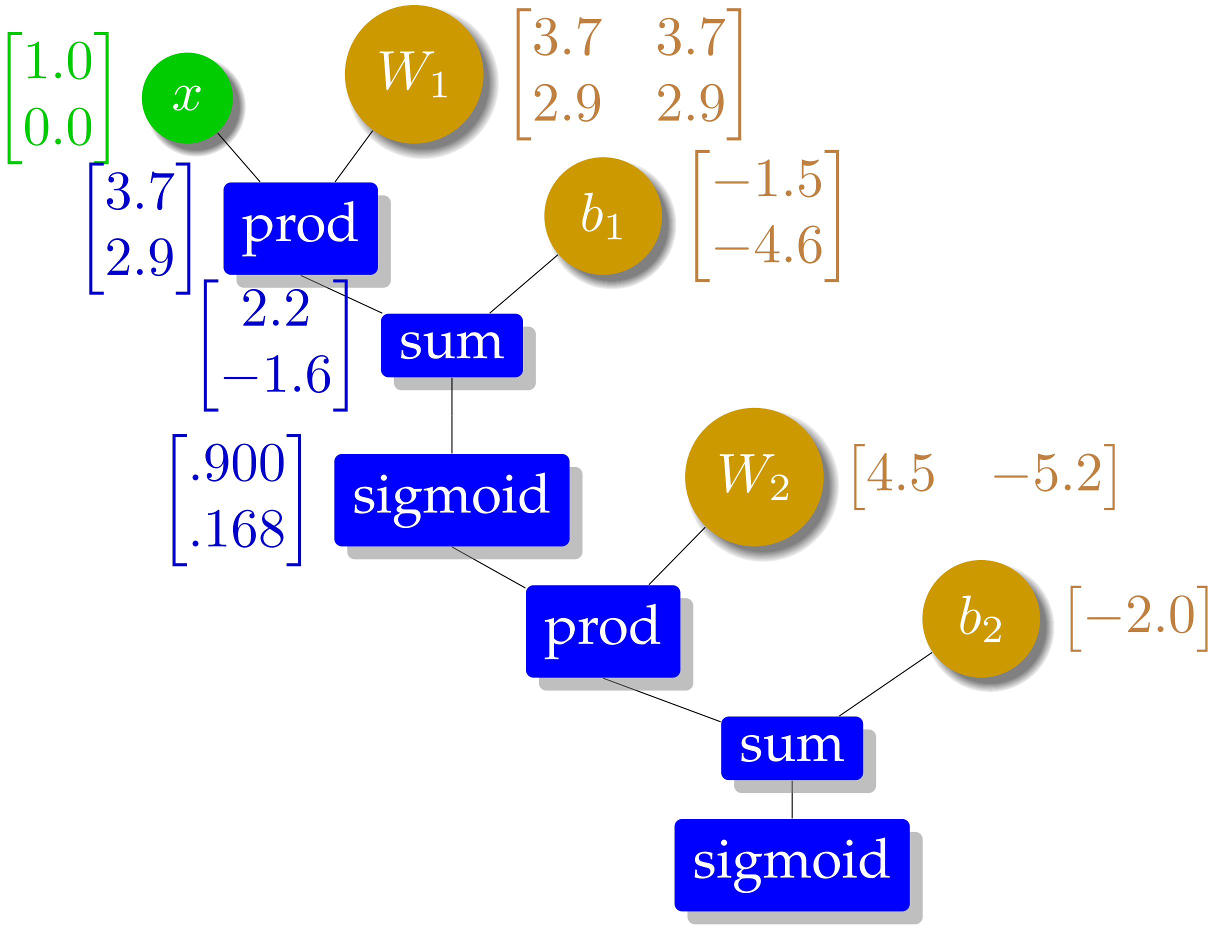
Processing Input



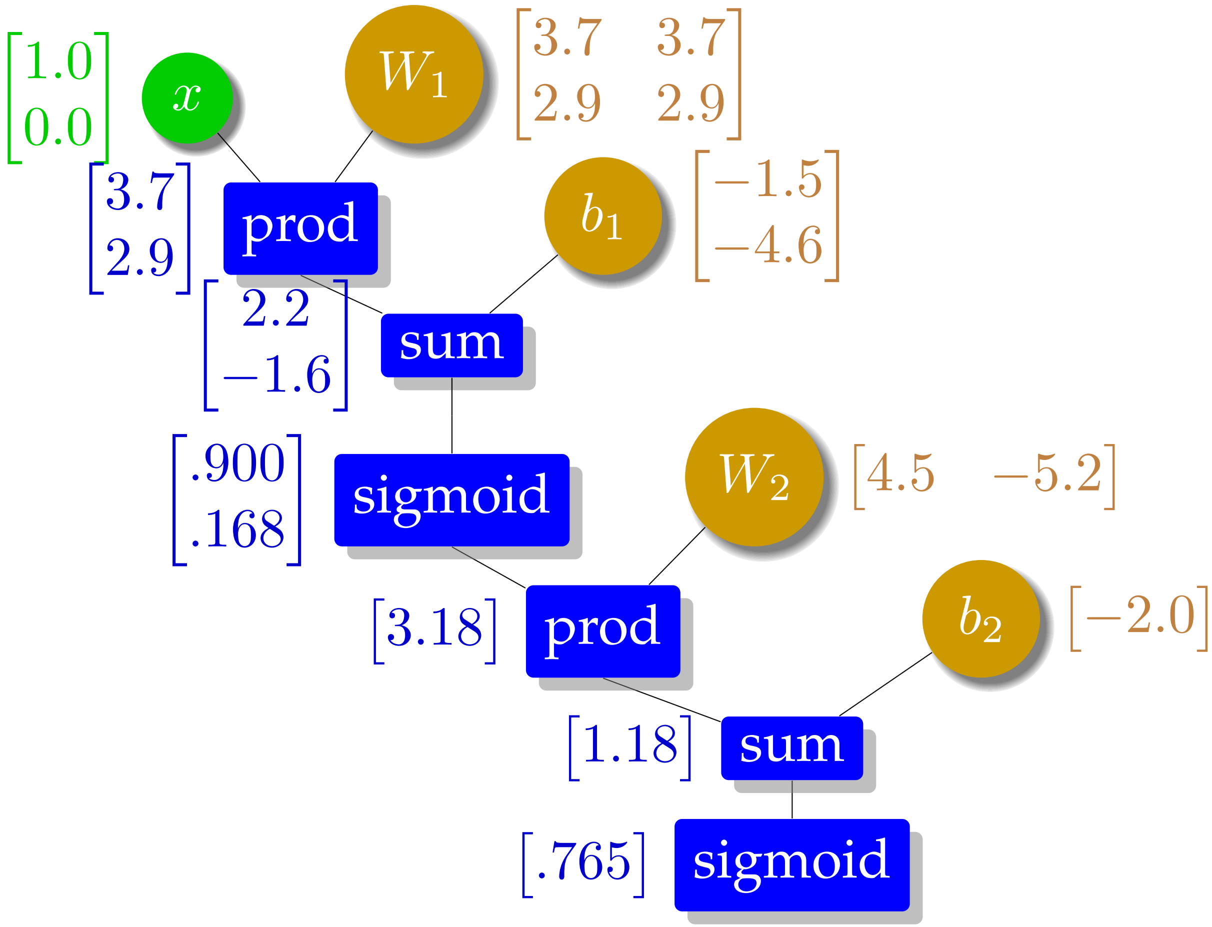
Processing Input



Processing Input



Processing Input



Error Function

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

⇒ 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

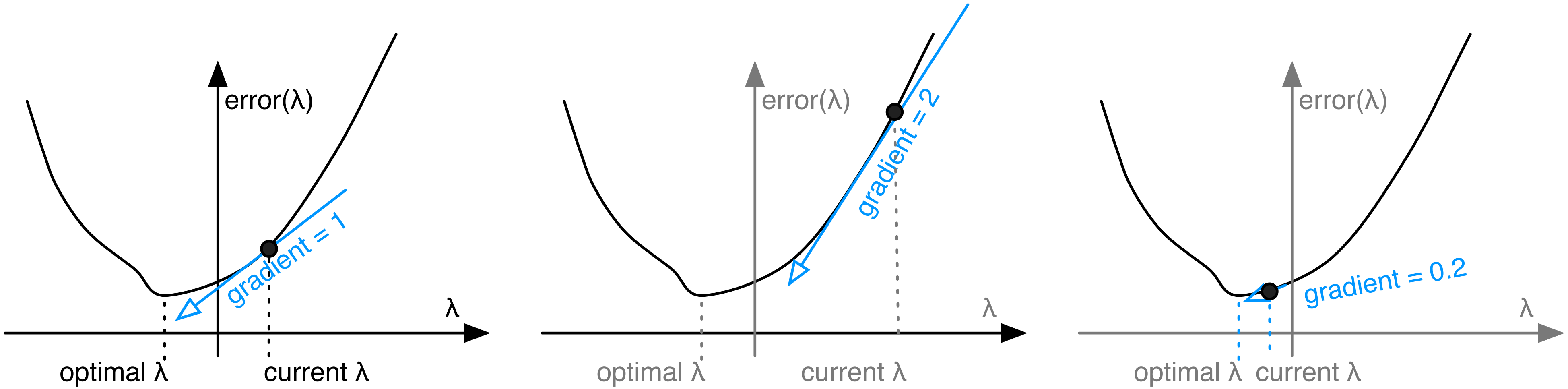
$$\text{error}(\lambda)$$

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



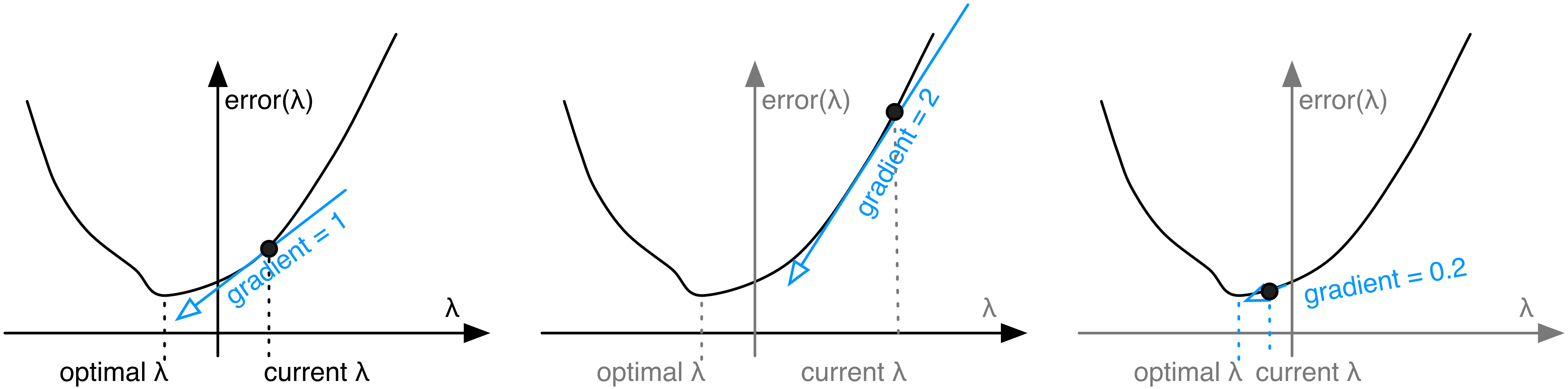
- Why not just set it to its optimum?

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 $f \cdot g$ 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

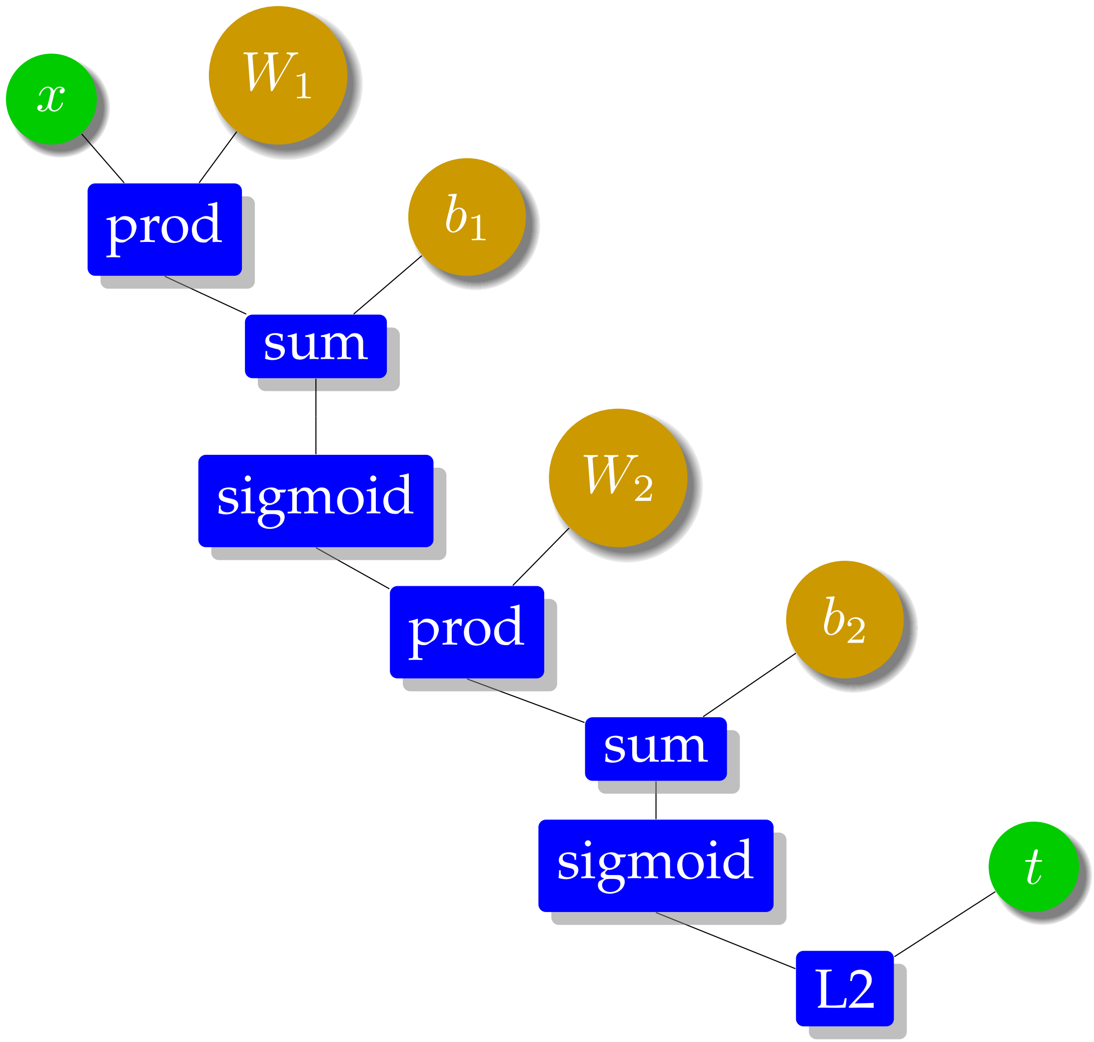
$$\frac{dE}{dw_k} =$$

Final Layer Update

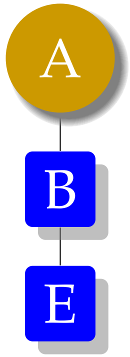
- Linear combination of weights $s = \sum_k w_k h_k$
- Activation function $y = \text{sigmoid}(s)$
- Error (L2 norm) $E = \frac{1}{2}(t - y)^2$
- Derivative of error with regard to one weight w_k

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

Error Computation in Computation Graph

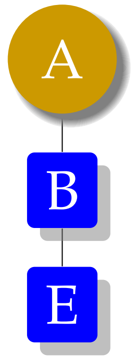


Error Propagation in Computation Graph



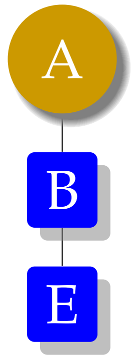
- Compute derivative at node A: $\frac{dE}{dA} = \frac{dE}{dB} \frac{dB}{dA}$

Error Propagation in Computation Graph



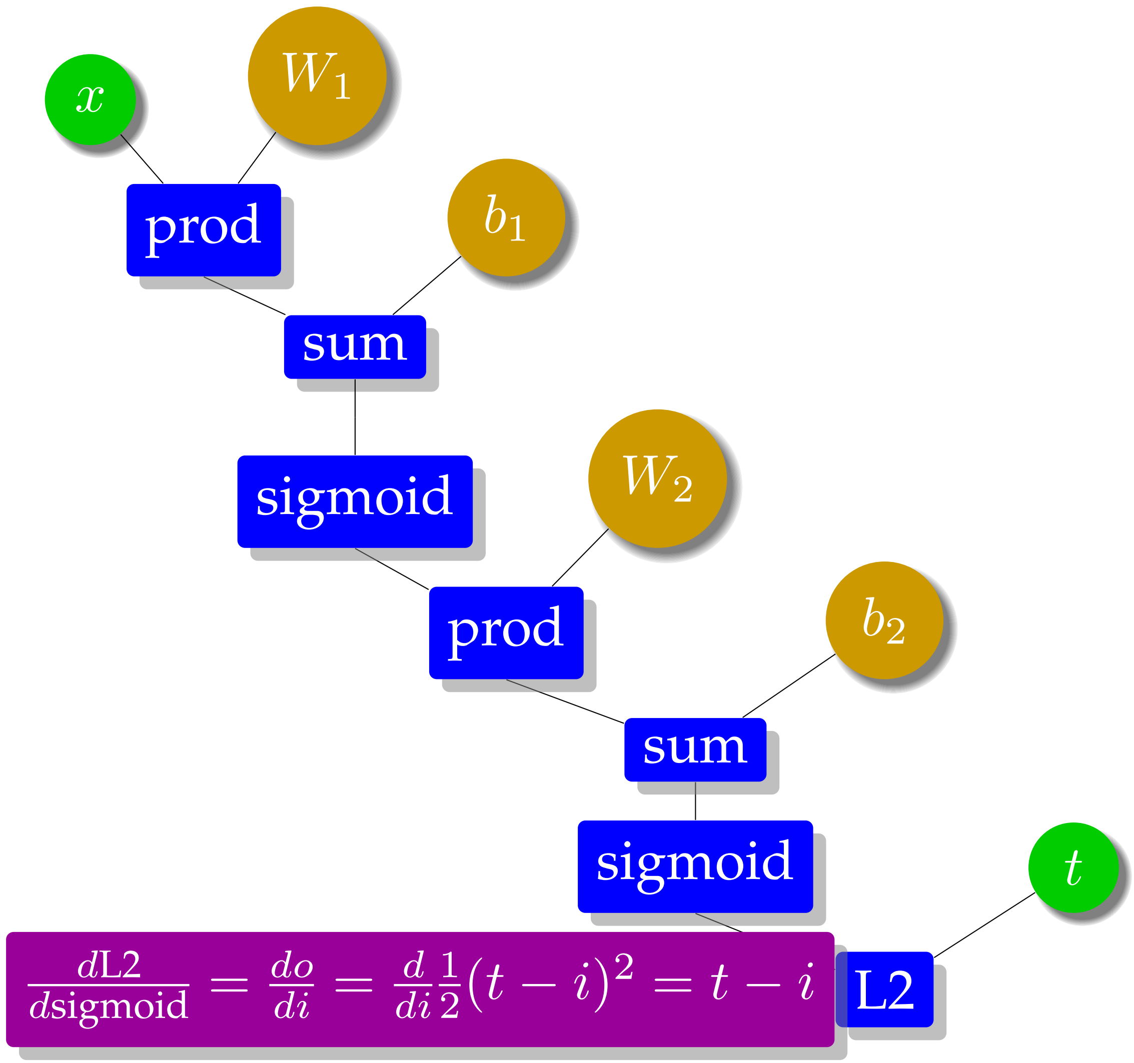
- Compute derivative at node A: $\frac{dE}{dA} = \frac{dE}{dB} \frac{dB}{dA}$
- Assume that we already computed $\frac{dE}{dB}$ (backward pass through graph)
- So now we only have to get the formula for $\frac{dB}{dA}$

Error Propagation in Computation Graph

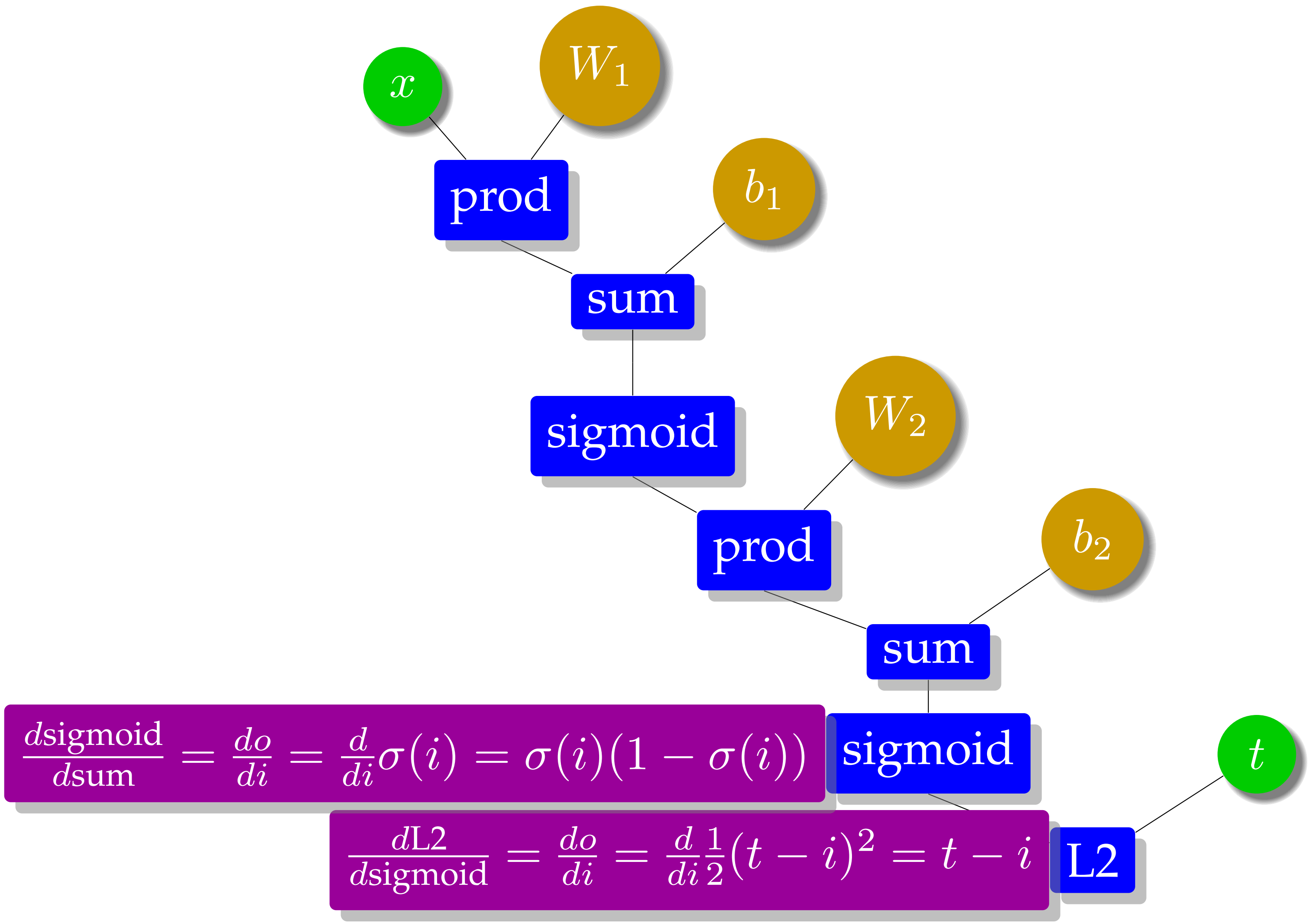


- Compute derivative at node A: $\frac{dE}{dA} = \frac{dE}{dB}\,\frac{dB}{dA}$
- Assume that we already computed $\frac{dE}{dB}$ (backward pass through graph)
- So now we only have to get the formula for $\frac{dB}{dA}$
- For instance B is a square node
  - forward computation: $B = A^2$
  - backward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

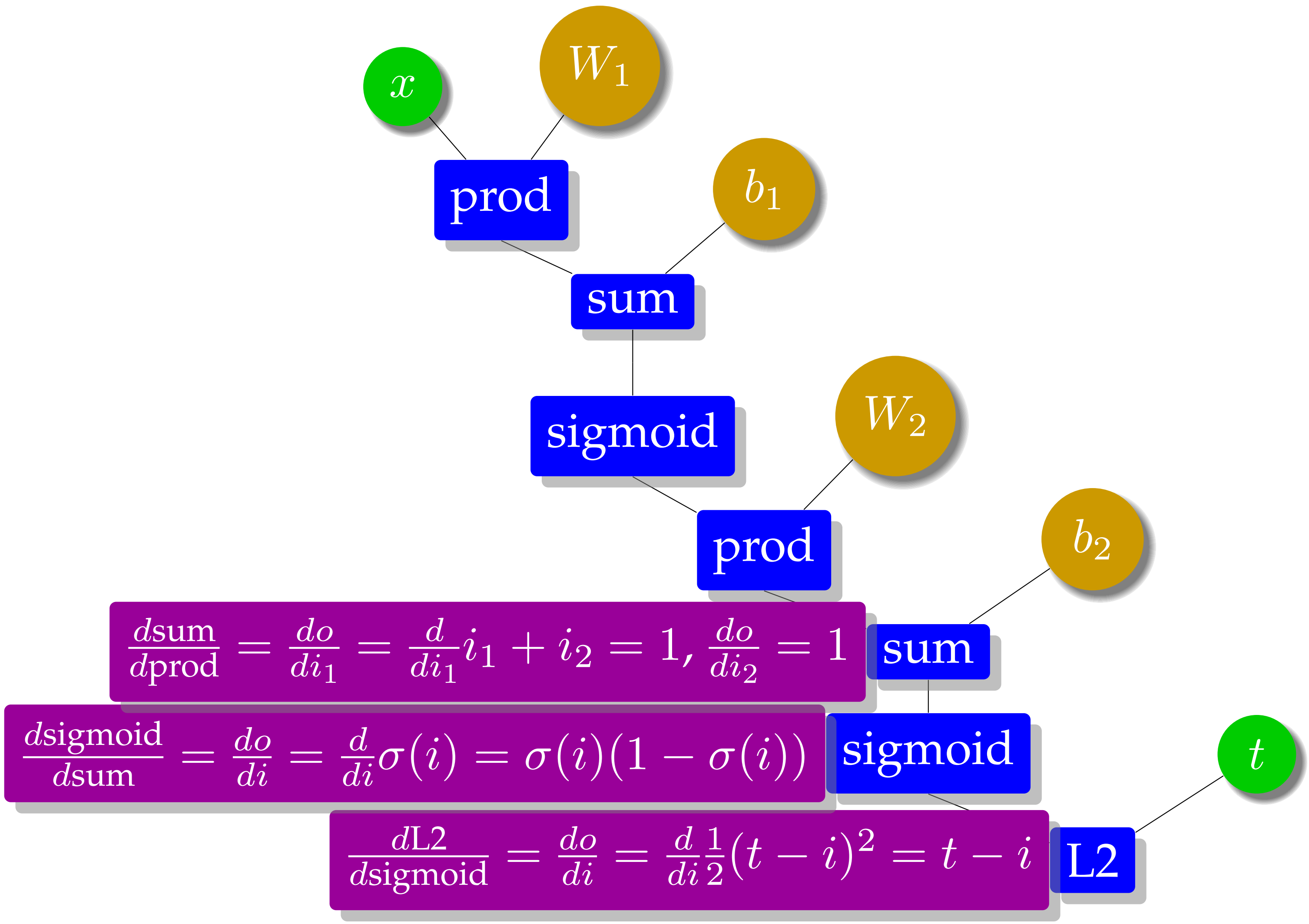
Derivatives for Each Node



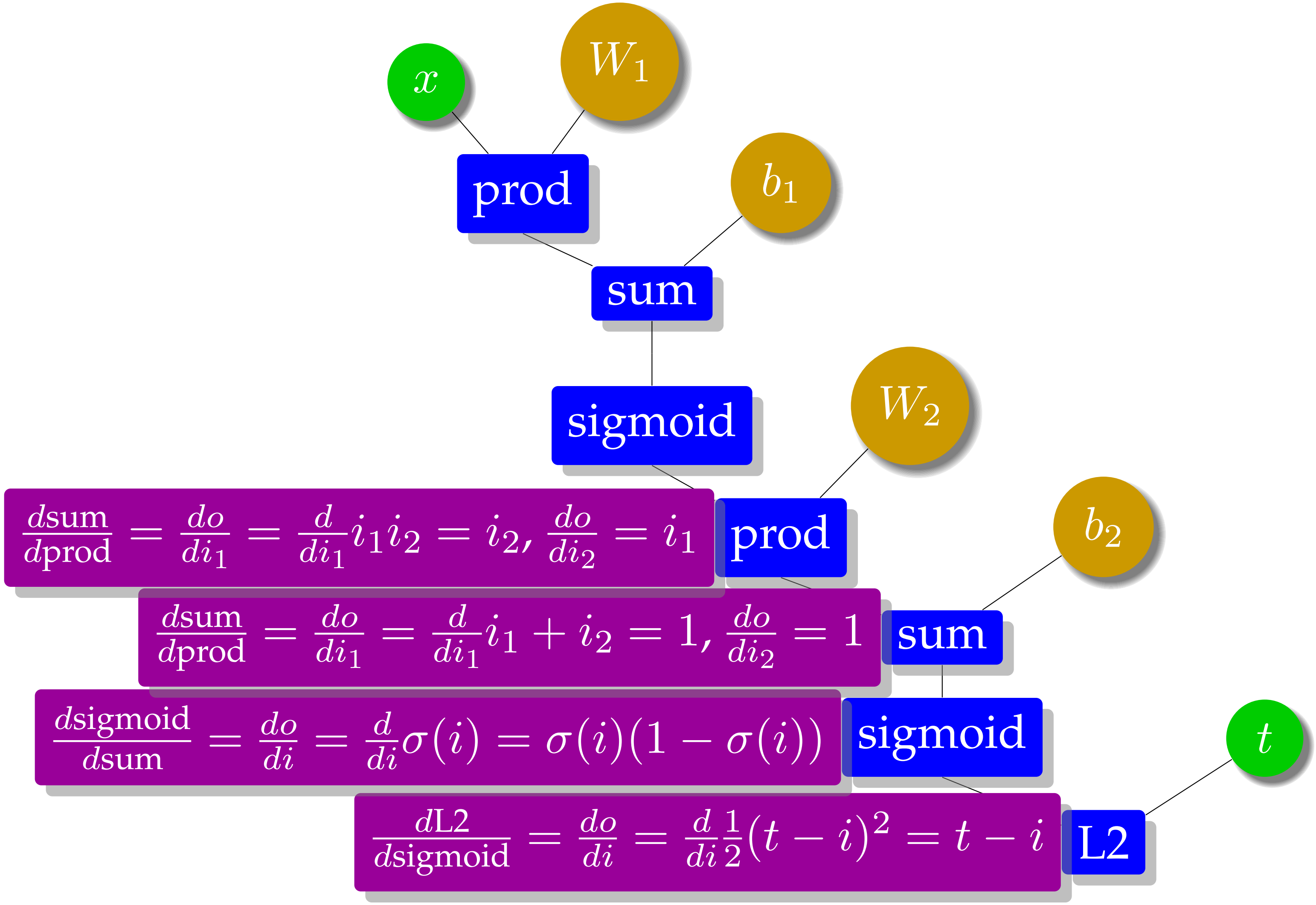
Derivatives for Each Node



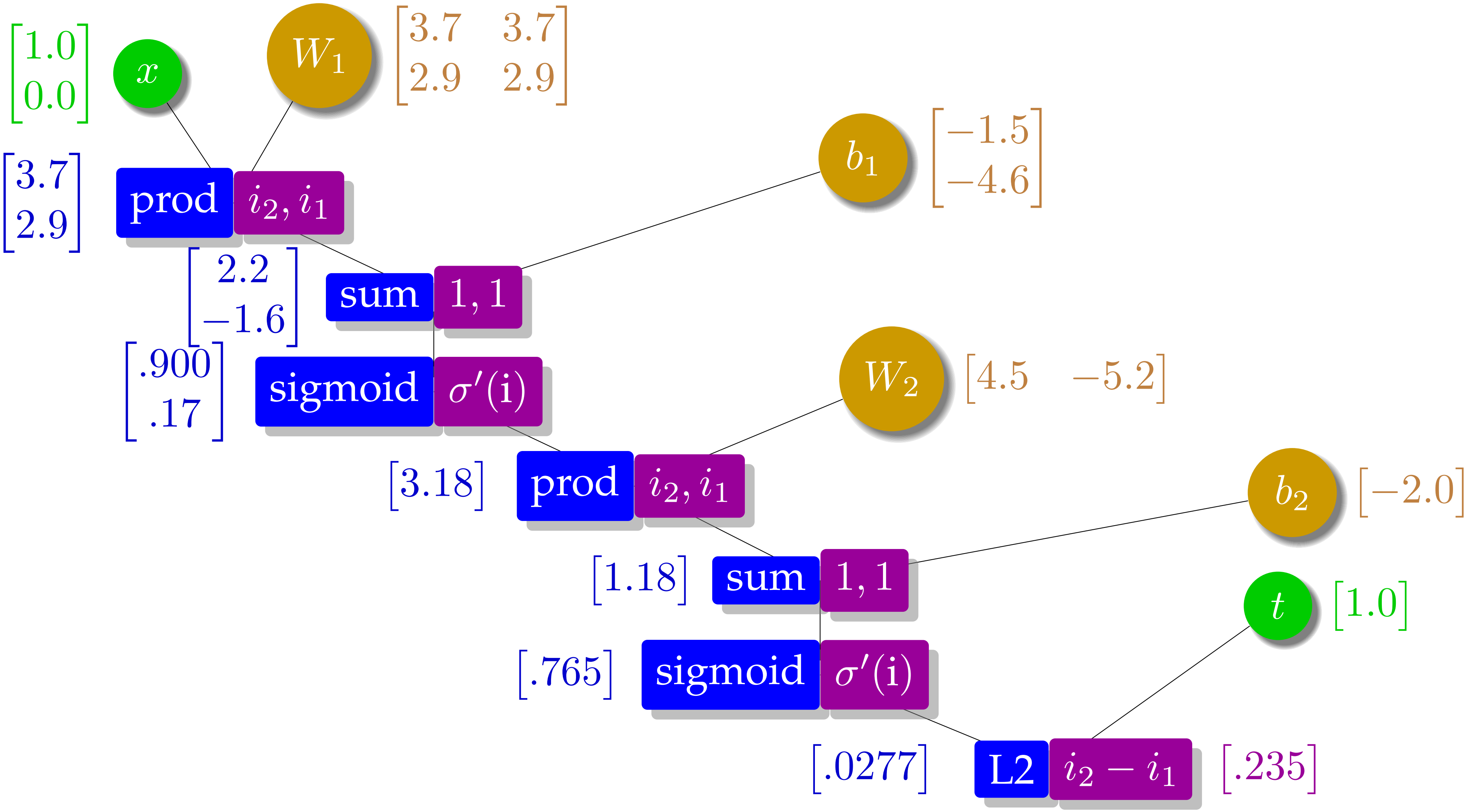
Derivatives for Each Node



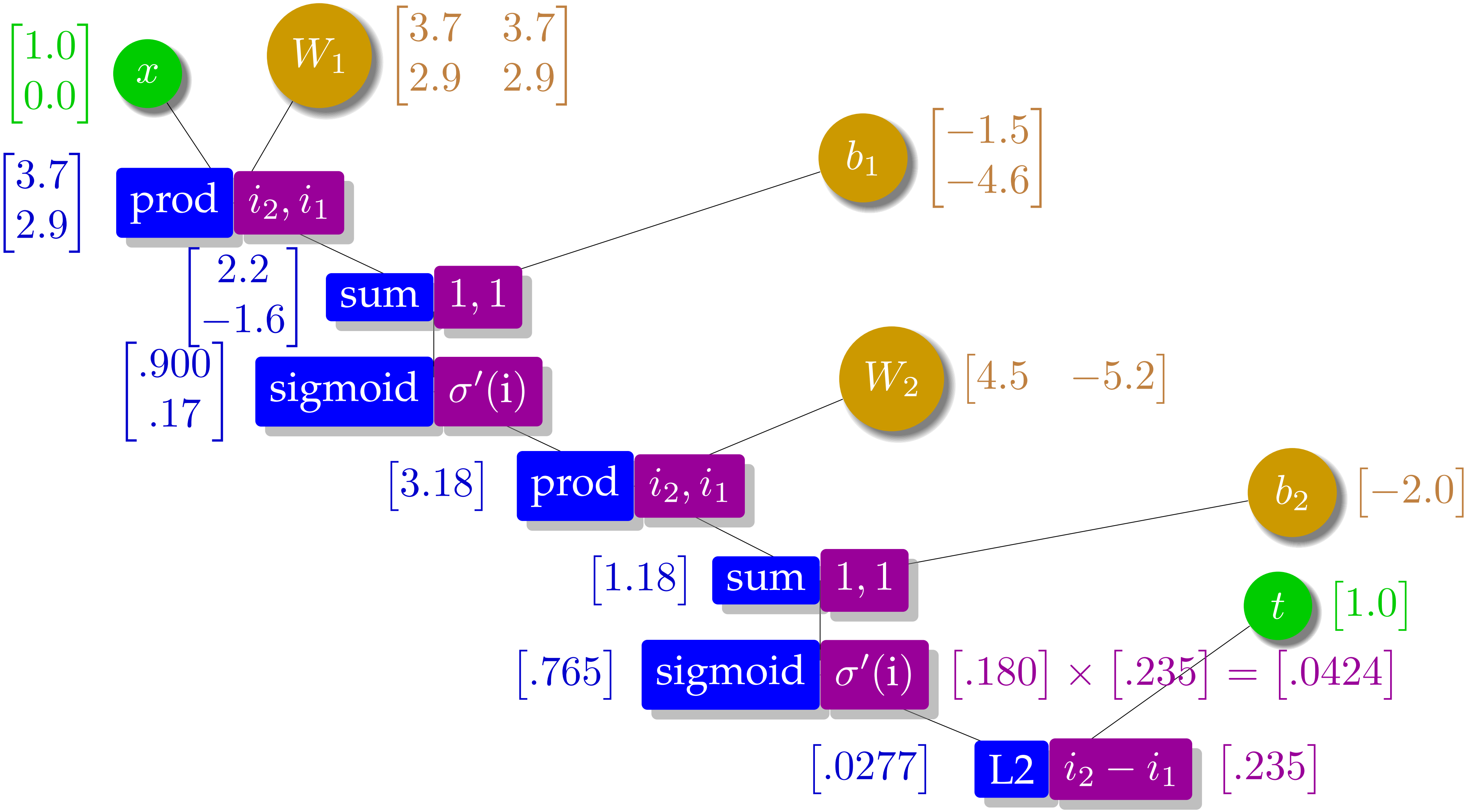
Derivatives for Each Node



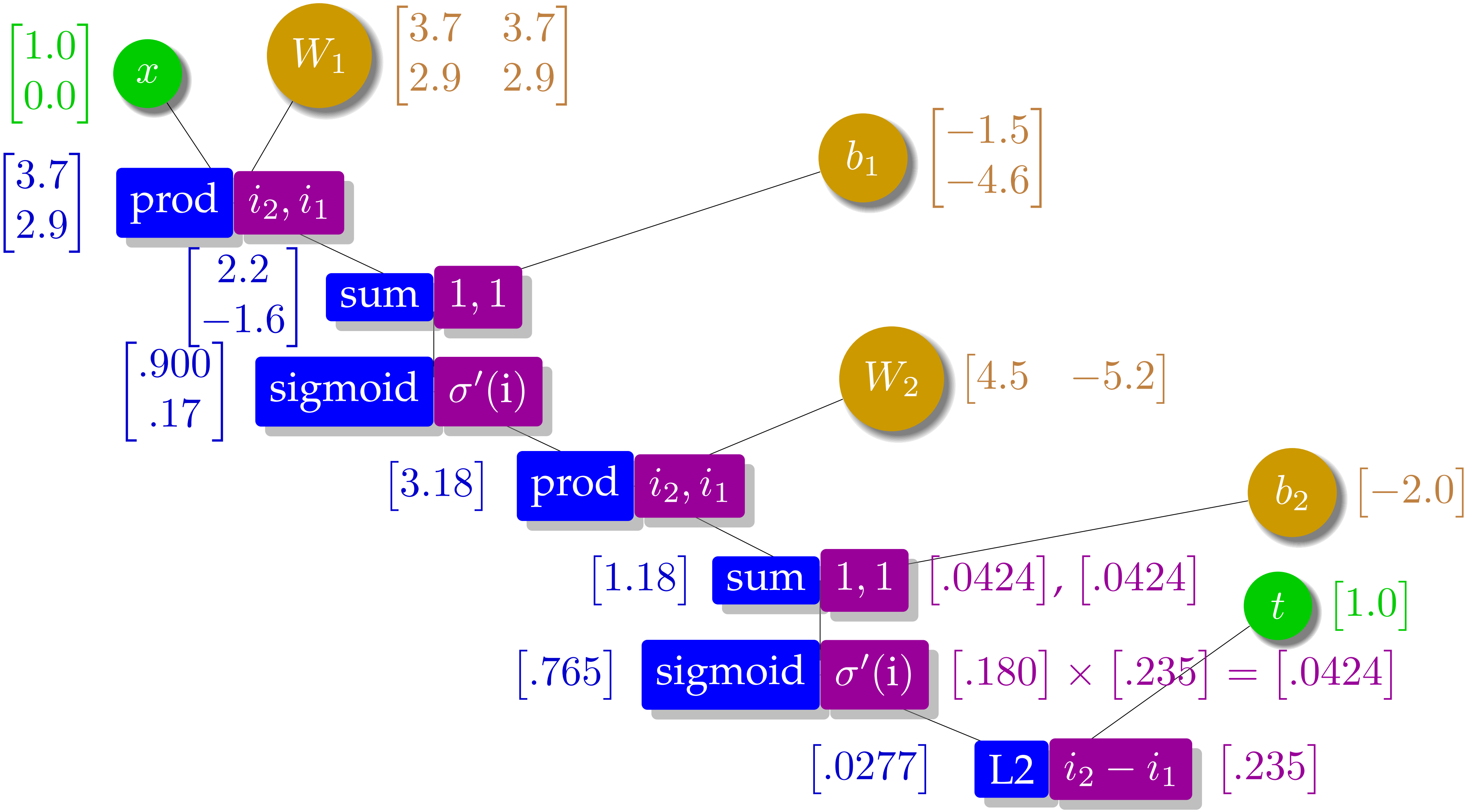
Backward Pass: Derivative Computation



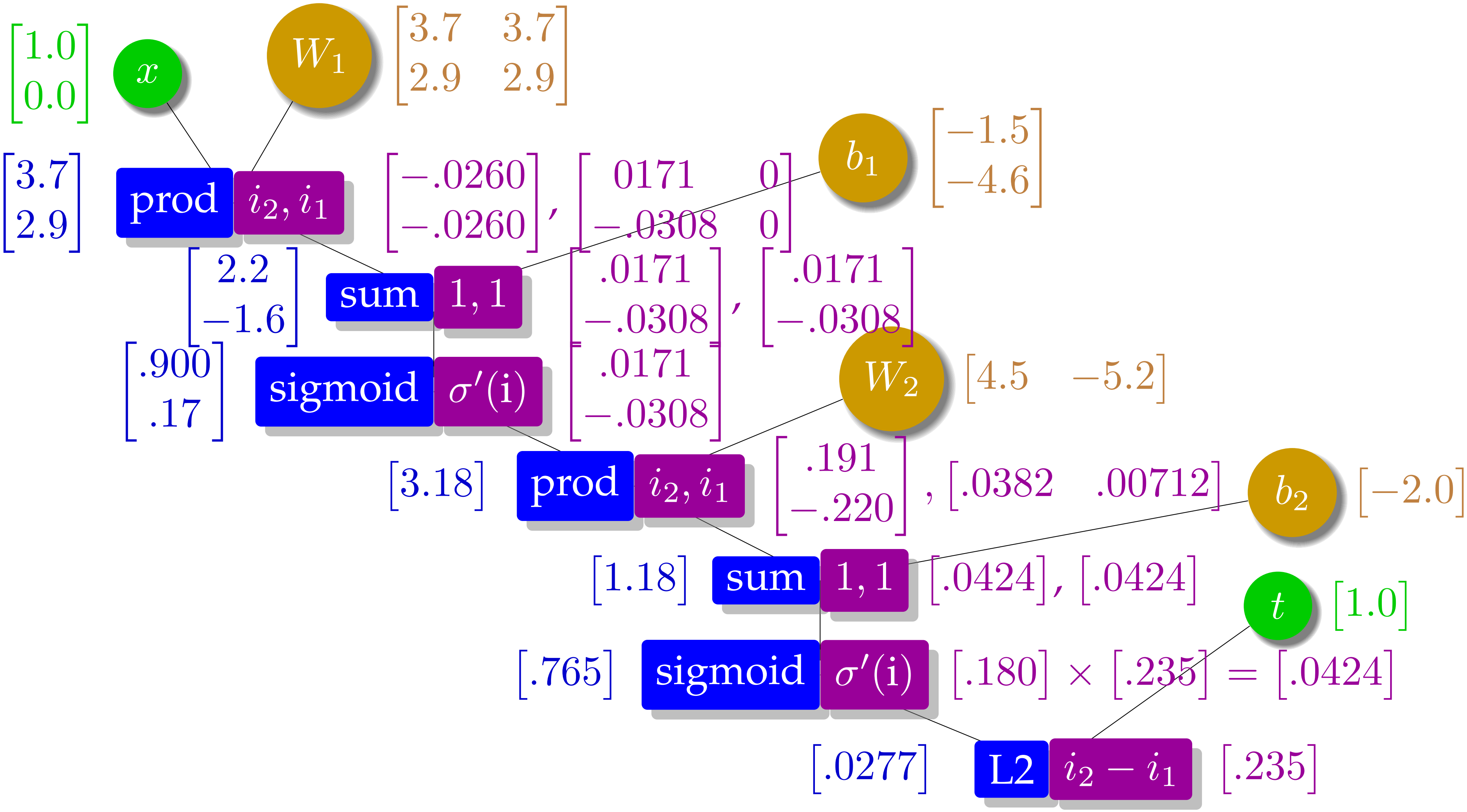
Backward Pass: Derivative Computation



Backward Pass: Derivative Computation

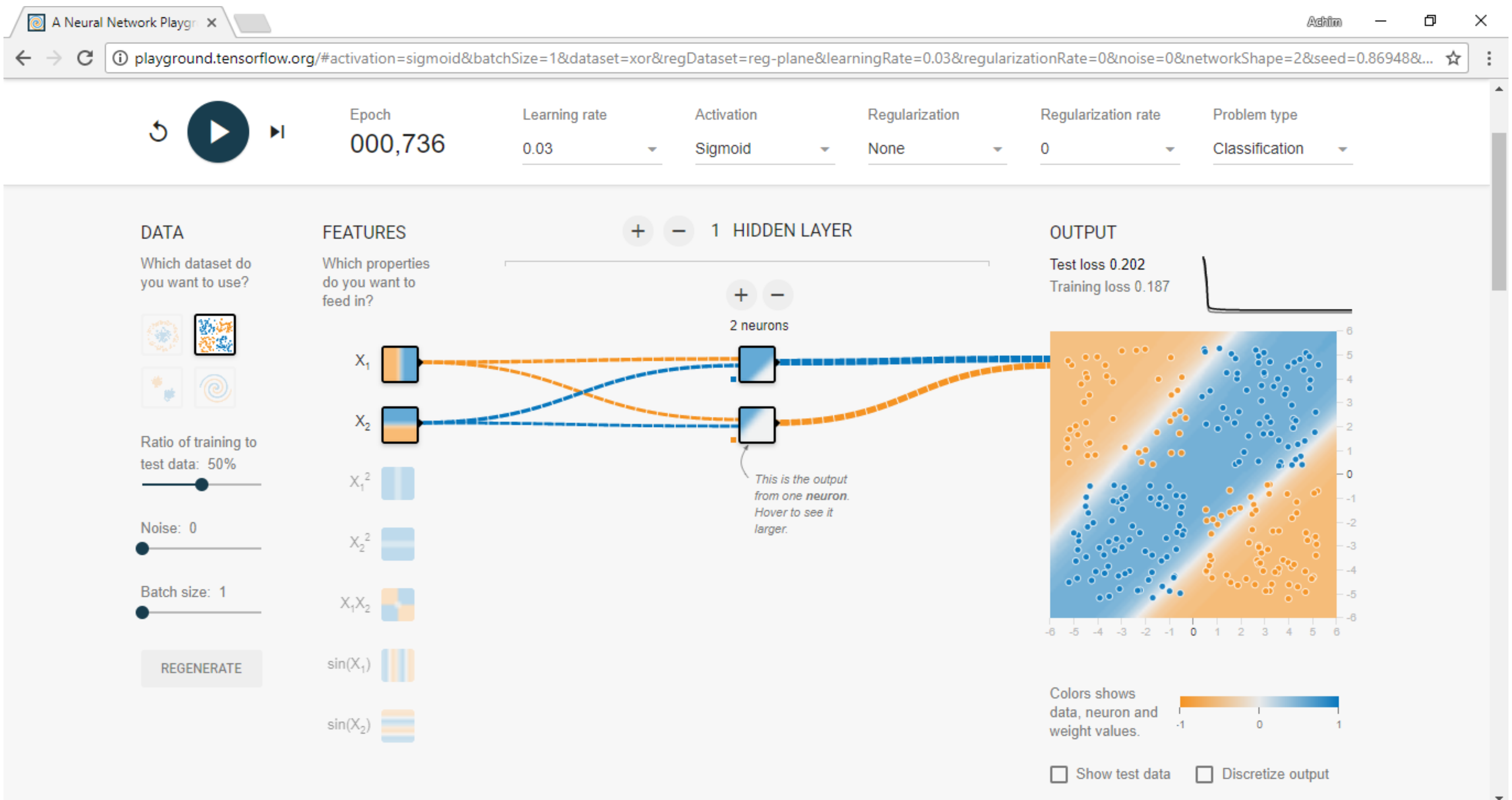


Backward Pass: Derivative Computation



Deep Learning Demo: Tensorflow Playground

http://playground.tensorflow.org



Deep Learning Framework: Keras

- High-level framework with Theano, Tensorflow and CNTK backends
- Provides many best practices defaults
- Terse declaration of static network
- Best integrated with Tensorflow

Keras: XOR example

```
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD
x = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])
model = Sequential()
model.add(Dense(2, input_dim=2, activation='sigmoid'))
model.add(Dense(1, activation='sigmoid'))
sgd=SGD(lr=0.3)
model.compile(loss='binary_crossentropy',optimizer=sgd)
model.fit(x,y,epochs=1000,batch_size=1,verbose=2)
print(model.predict(x,verbose=1))
```

Keras: XOR example Imports

```
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD
x = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])
model = Sequential()
model.add(Dense(2, input_dim=2, activation='sigmoid'))
model.add(Dense(1, activation='sigmoid'))
sgd=SGD(lr=0.3)
model.compile(loss='binary_crossentropy',optimizer=sgd)
model.fit(x,y,epochs=1000,batch_size=1,verbose=2)
print(model.predict(x,verbose=1))
```

Keras: XOR example
Data Definition

```
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD
x = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])
model = Sequential()
model.add(Dense(2, input_dim=2, activation='sigmoid'))
model.add(Dense(1, activation='sigmoid'))
sgd=SGD(lr=0.3)
model.compile(loss='binary_crossentropy',optimizer=sgd)
model.fit(x,y,epochs=1000,batch_size=1,verbose=2)
print(model.predict(x,verbose=1))
```

Keras: XOR example
Network Definition

```
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD
x = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])
model = Sequential()
model.add(Dense(2, input_dim=2, activation='sigmoid'))
model.add(Dense(1, activation='sigmoid'))
sgd=SGD(lr=0.3)
model.compile(loss='binary_crossentropy',optimizer=sgd)
model.fit(x,y,epochs=1000,batch_size=1,verbose=2)
print(model.predict(x,verbose=1))
```

Keras: XOR example
Network Training

```
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD
x = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])
model = Sequential()
model.add(Dense(2, input_dim=2, activation='sigmoid'))
model.add(Dense(1, activation='sigmoid'))
sgd=SGD(lr=0.3)
model.compile(loss='binary_crossentropy',optimizer=sgd)
model.fit(x,y,epochs=1000,batch_size=1,verbose=2)
print(model.predict(x,verbose=1))
```

Keras: XOR example Inference

```
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD
x = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])
model = Sequential()
model.add(Dense(2, input_dim=2, activation='sigmoid'))
model.add(Dense(1, activation='sigmoid'))
sgd=SGD(lr=0.3)
model.compile(loss='binary_crossentropy',optimizer=sgd)
model.fit(x,y,epochs=1000,batch_size=1,verbose=2)
print(model.predict(x,verbose=1))
```

example: dynet

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`.