# Neural Machine Translation

Philipp Koehn

13 June 2024

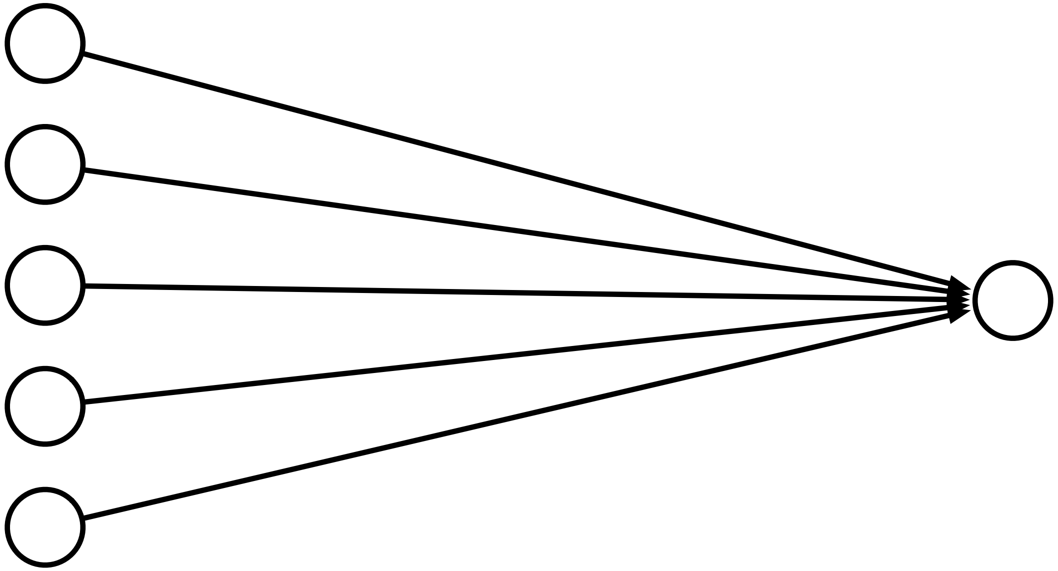
# Linear Models



- Linear combination of input values $x_i$ and weights $\lambda_i$

$$\mathrm{y}(\lambda, \mathbf{x}) = \sum_i \lambda_i \, x_i$$

- Such models can be illustrated as a "network"



# Limits of Linearity

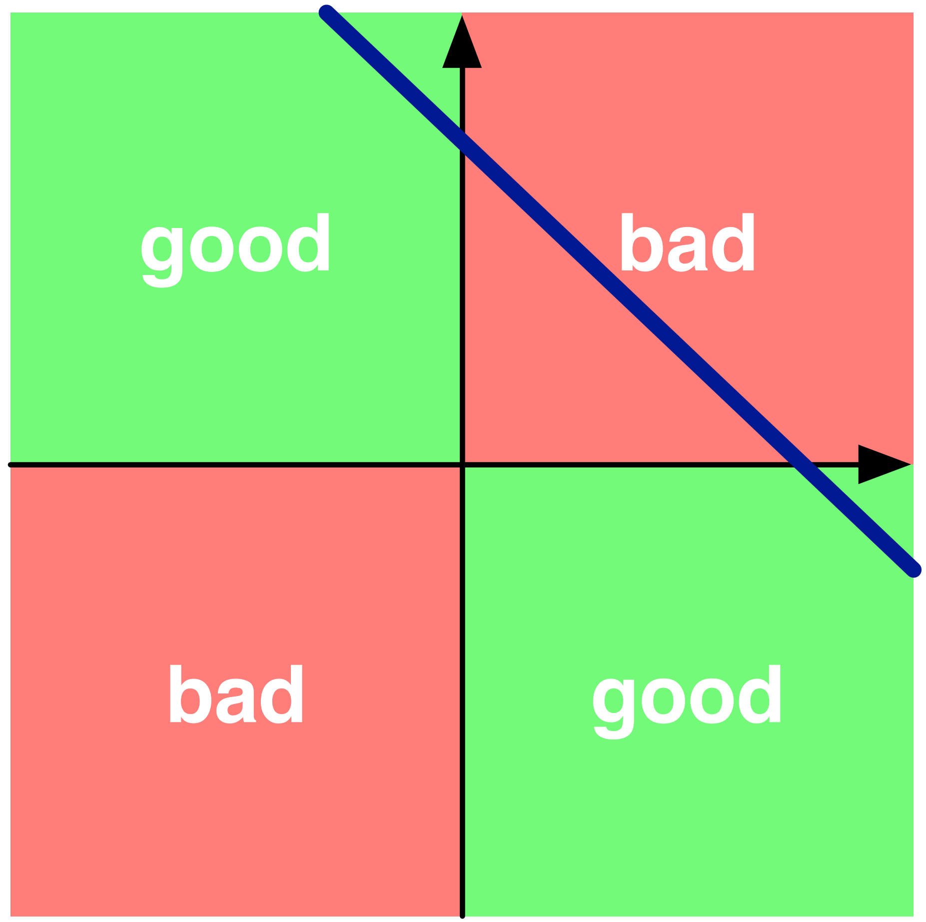


- We can give each feature a weight

- But not more complex value relationships, e.g.,
  - only a critical range of the feature value matters
  - feature interactions

# XOR

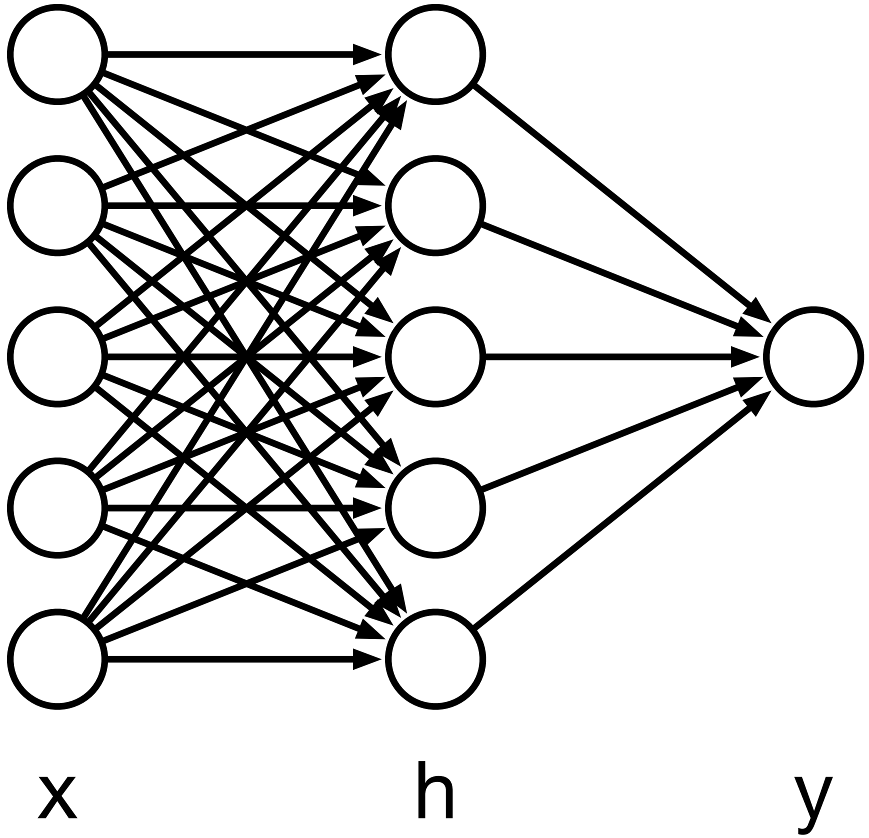
- Linear models cannot model XOR



# 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

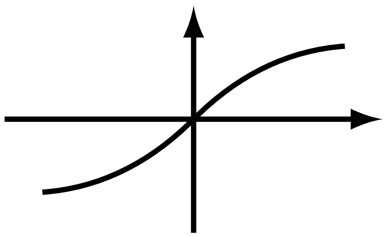
$$y(\lambda, \mathbf{x}) = \sum_i \lambda_i \, x_i$$

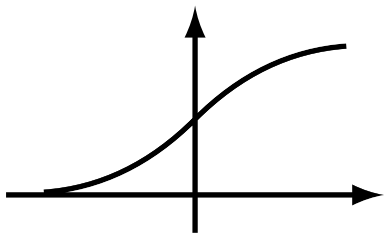
- Add a non-linear function

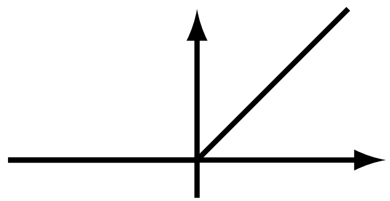
$$y(\lambda, \mathbf{x}) = f\Big(\sum_i \lambda_i \, x_i\Big)$$

- Popular choices

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





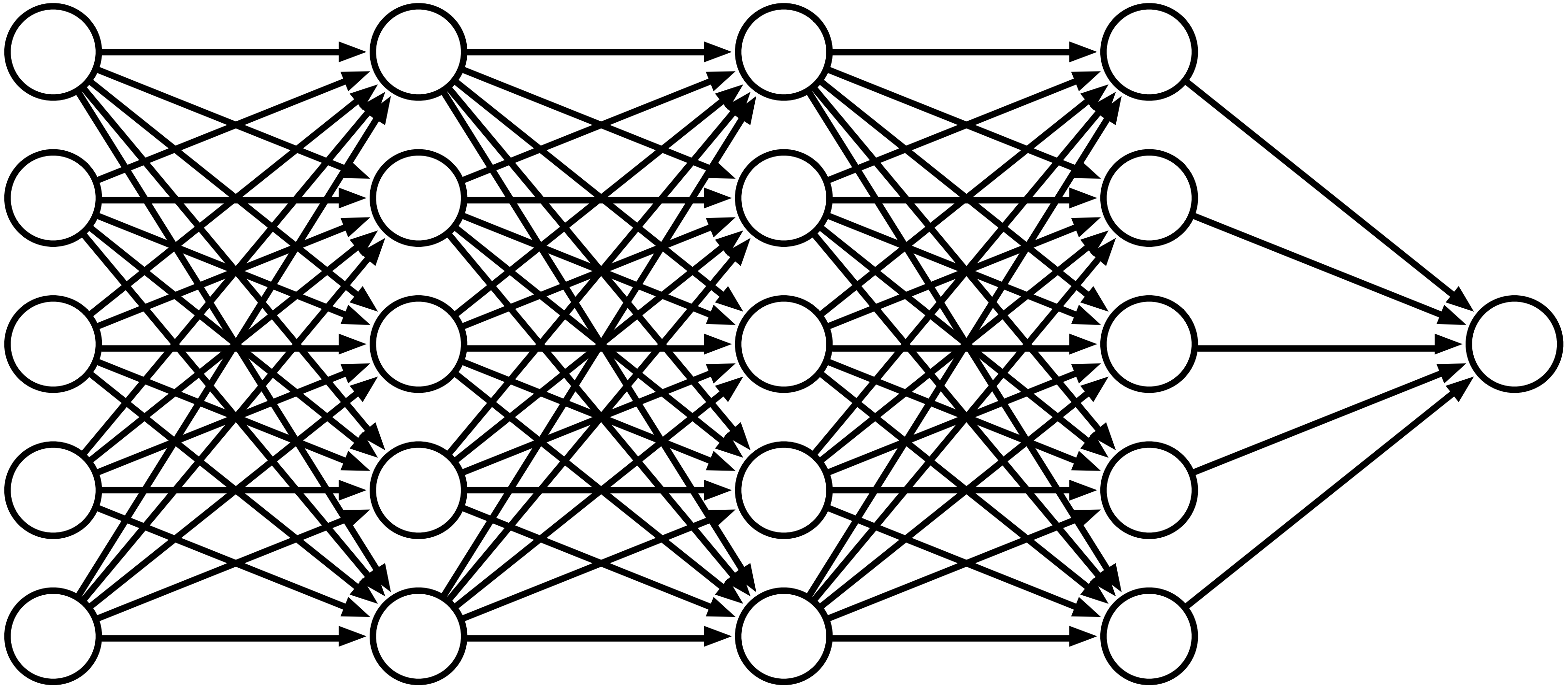


(sigmoid is also called the "logistic function")

# Deep Learning



- More layers = deep learning



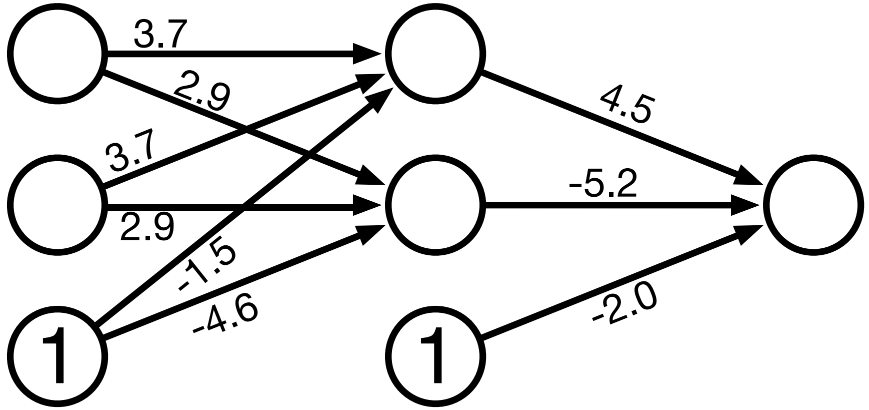
# What Depth Enables

- 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

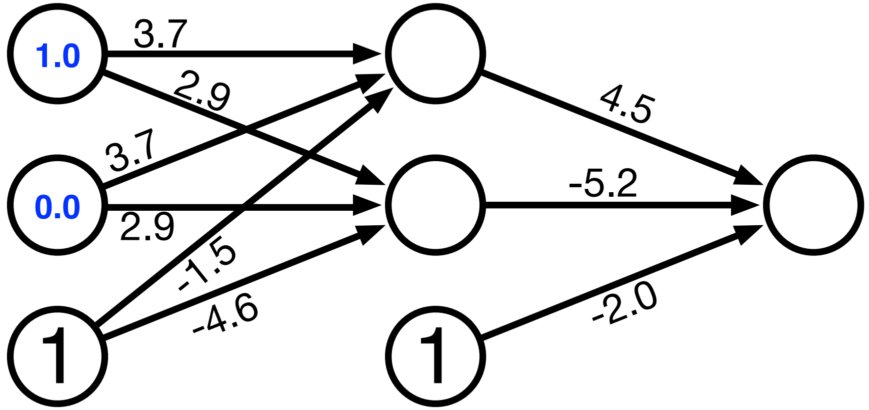





- One innovation: bias units (no inputs, always value 1)

# Sample Input






- Try out two input values

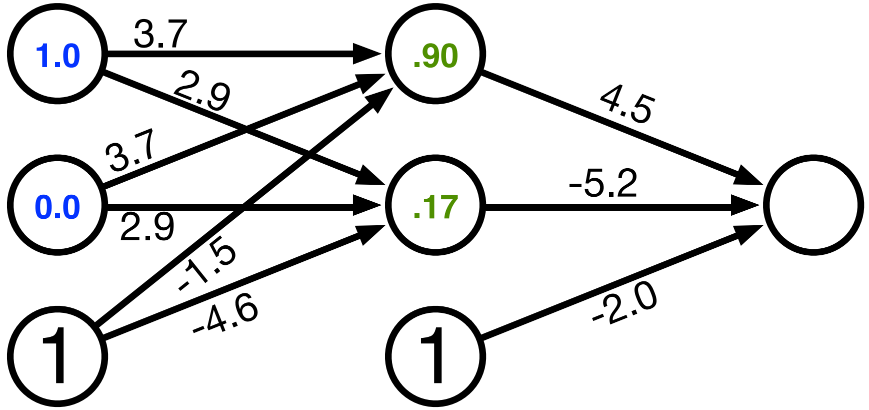
- Hidden unit computation

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

$$\text{sigmoid}(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times -4.6) = \text{sigmoid}(-1.7) = \frac{1}{1+e^{1.7}} = 0.17$$

# Computed Hidden






- Try out two input values

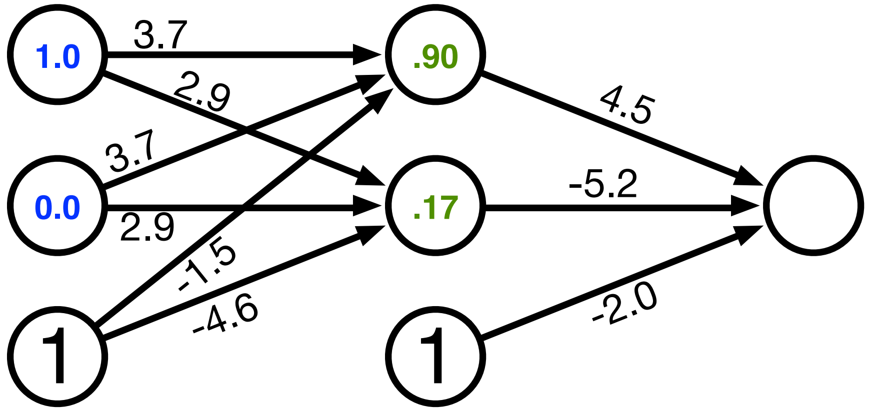
- Hidden unit computation

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

$$\text{sigmoid}(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times -4.6) = \text{sigmoid}(-1.7) = \frac{1}{1+e^{1.7}} = 0.17$$

# Compute Output



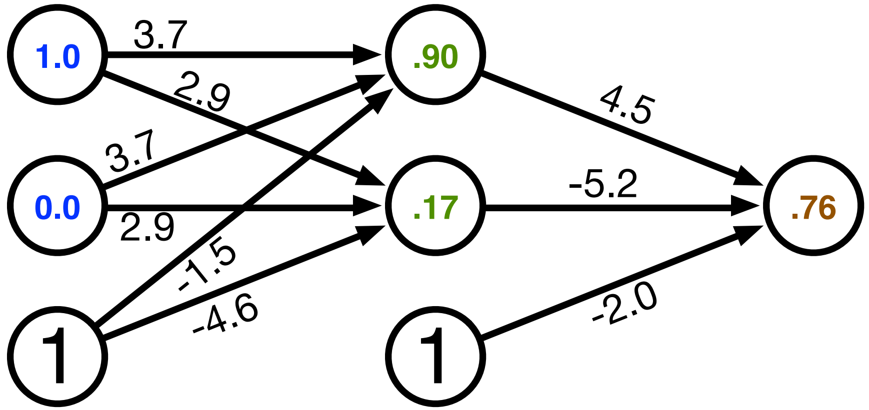



- Output unit computation

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

# 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



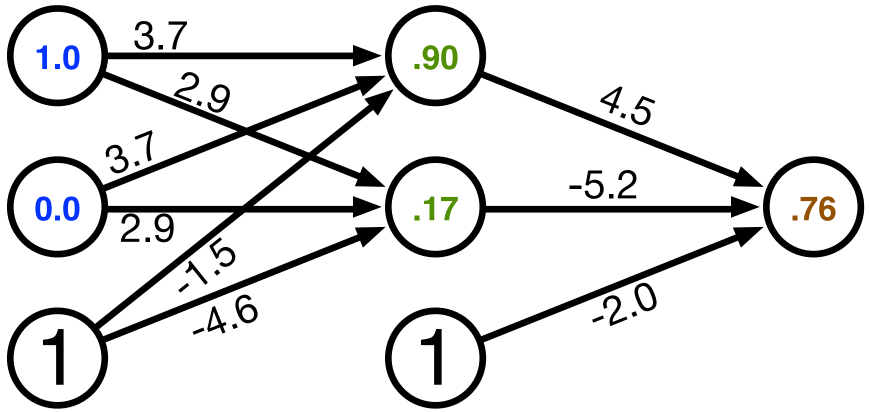
| Input $x_0$ | Input $x_1$ | Hidden $h_0$ | Hidden $h_1$ | Output $y_0$ |
|---|---|---|---|---|
| 0 | 0 | 0.12 | 0.02 | $0.18 \rightarrow 0$ |
| 0 | 1 | 0.88 | 0.27 | $0.74 \rightarrow 1$ |
| 1 | 0 | 0.73 | 0.12 | $0.74 \rightarrow 1$ |
| 1 | 1 | 0.99 | 0.73 | $0.33 \rightarrow 0$ |

- 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

# back-propagation training

# Error




- Computed output: $y = .76$

- Correct output: $t = 1.0$

$\Rightarrow$ How do we adjust the weights?

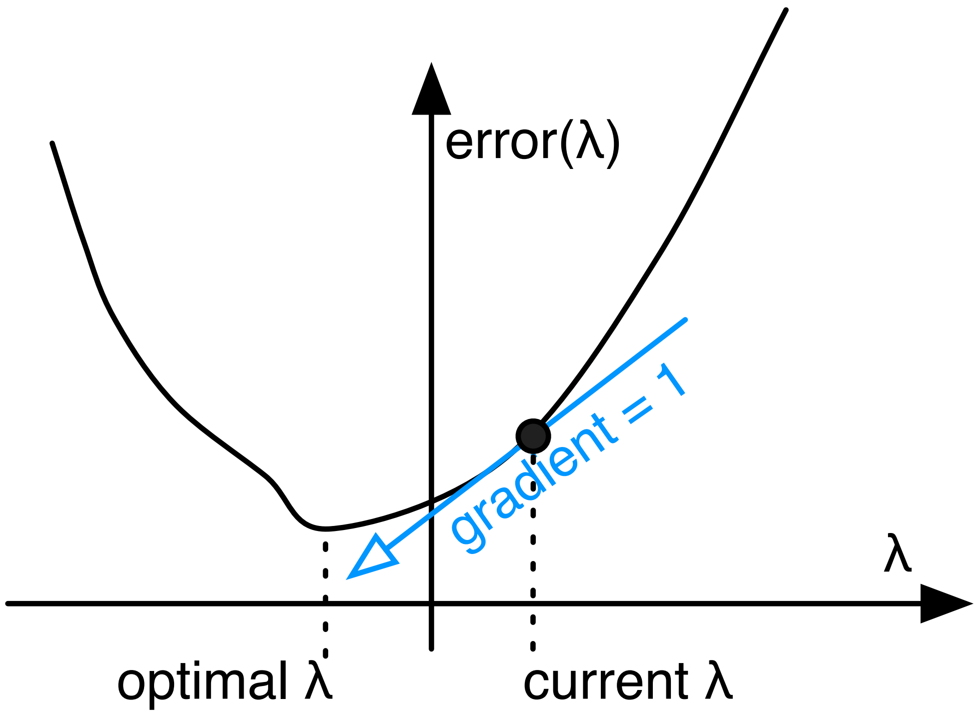
# Key Concepts



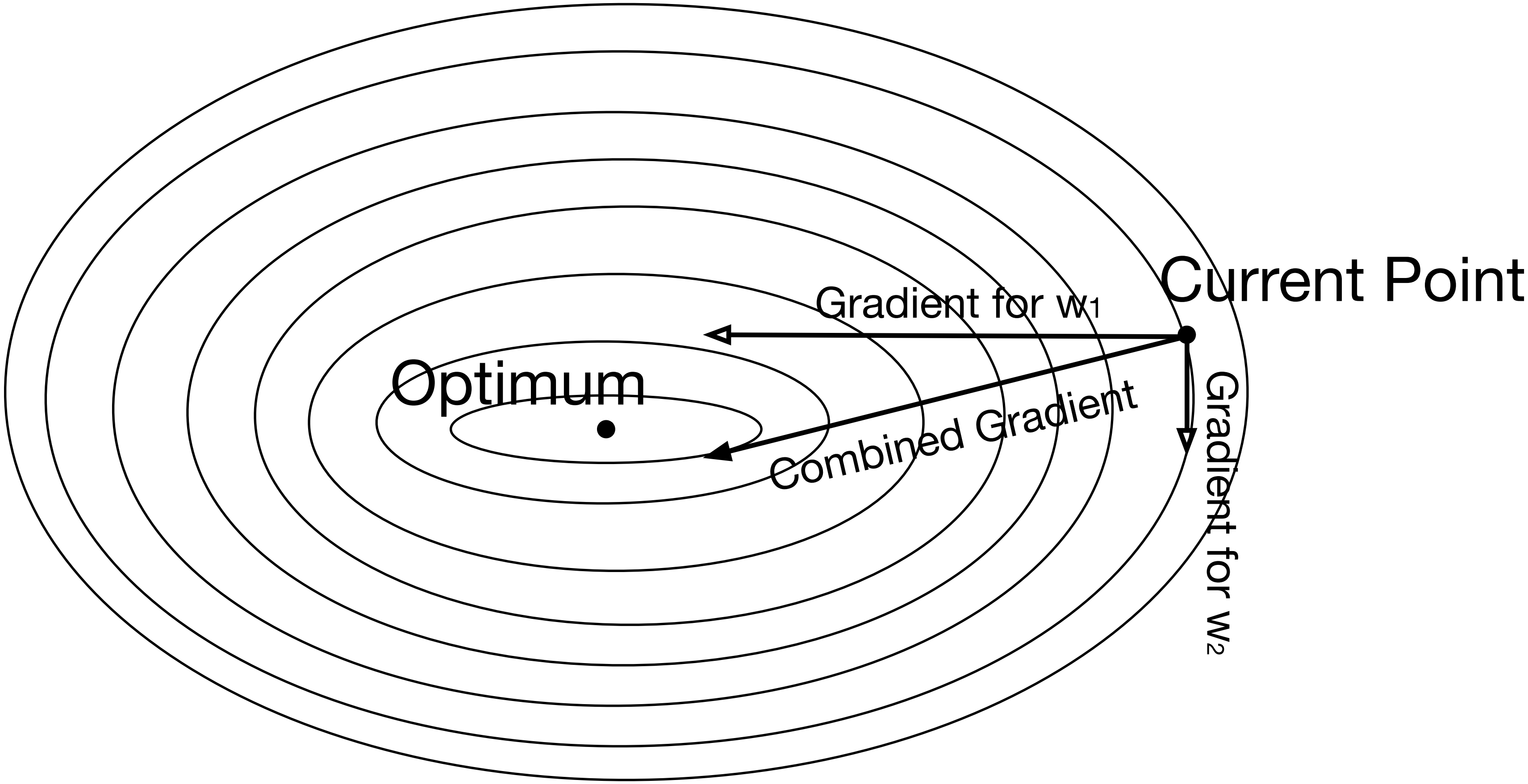
- Gradient descent
  - error is a function of the weights
  - we want to reduce the error
  - gradient descent: move towards the error minimum
  - compute gradient $\rightarrow$ get direction to the error minimum
  - adjust weights towards direction of lower error

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

# Gradient Descent



# Gradient Descent

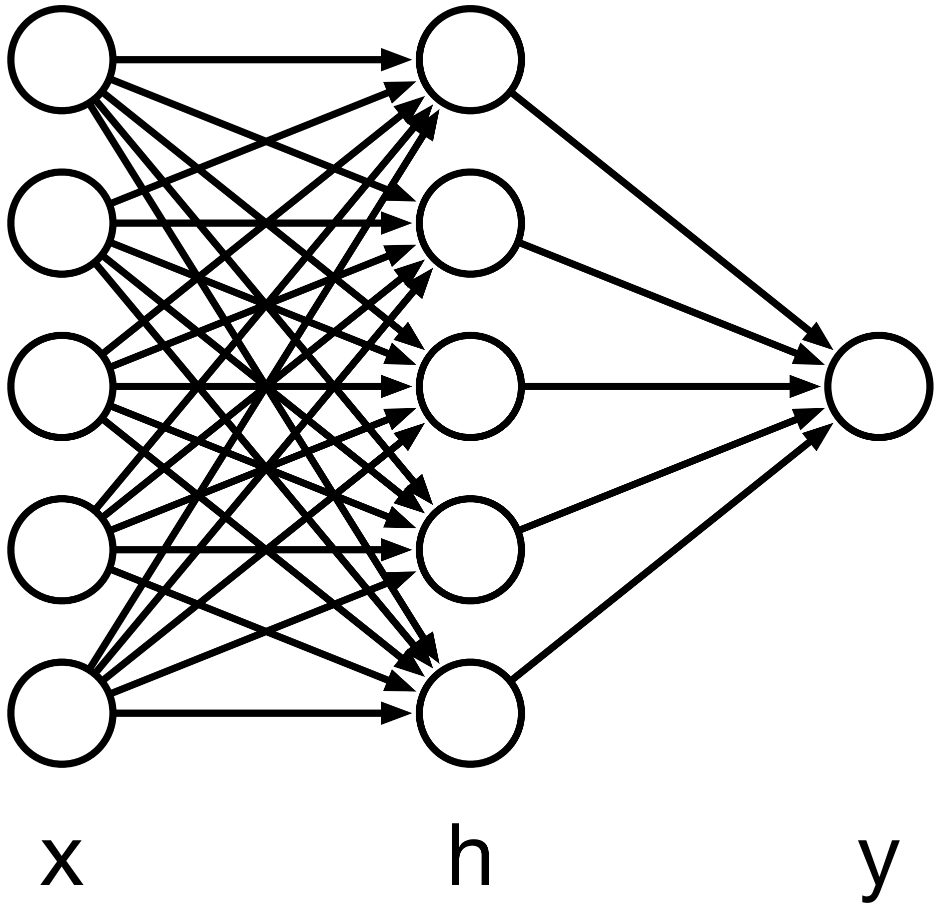


# computation graphs

# Neural Network Cartoon



- A common way to illustrate a neural network



# Neural Network Math

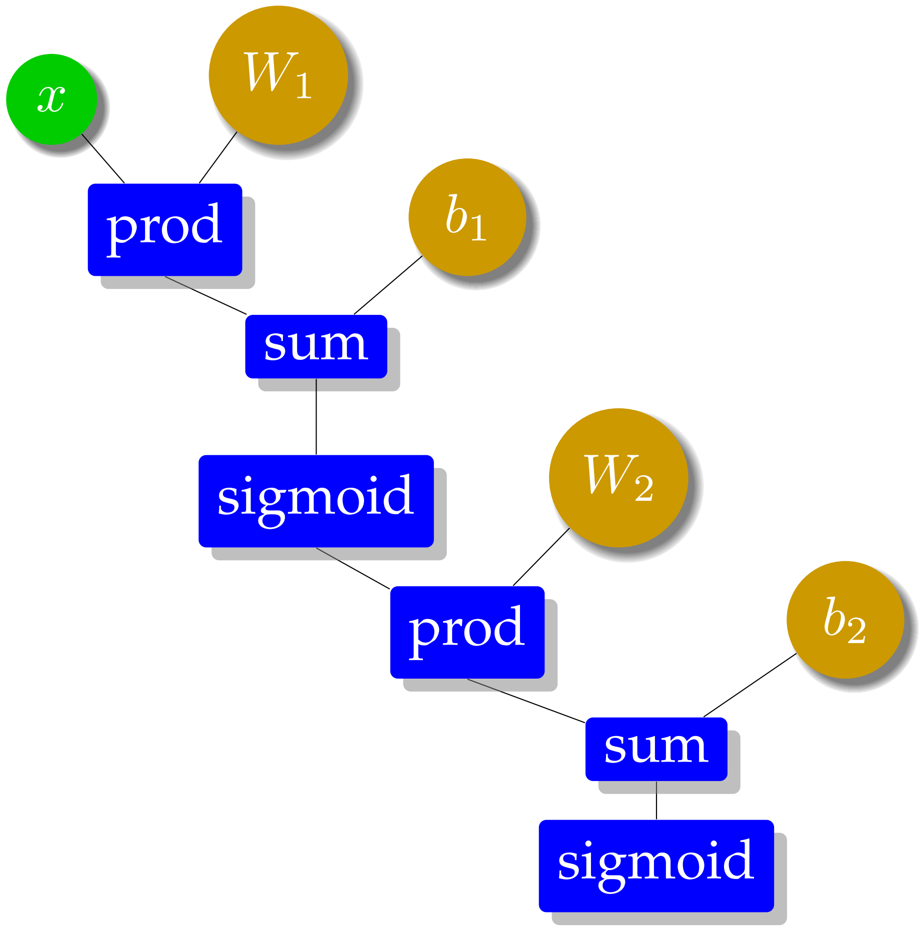
- Hidden layer

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

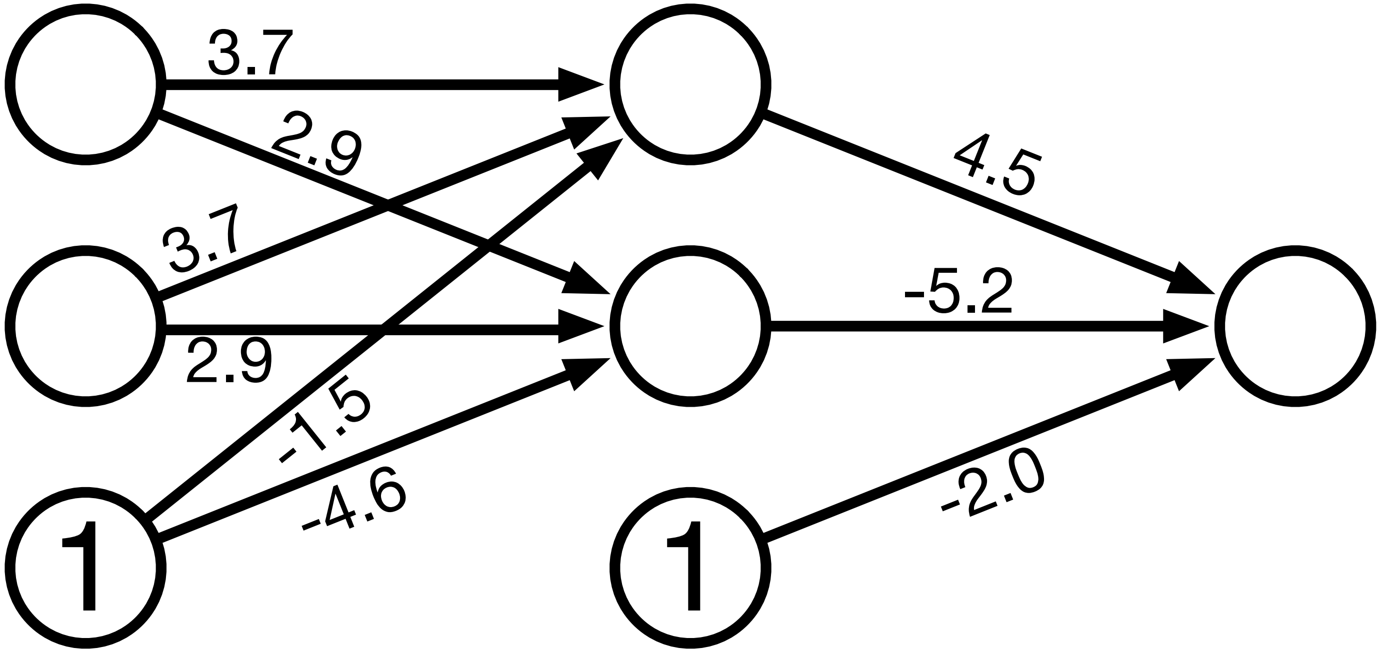
- Final layer

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

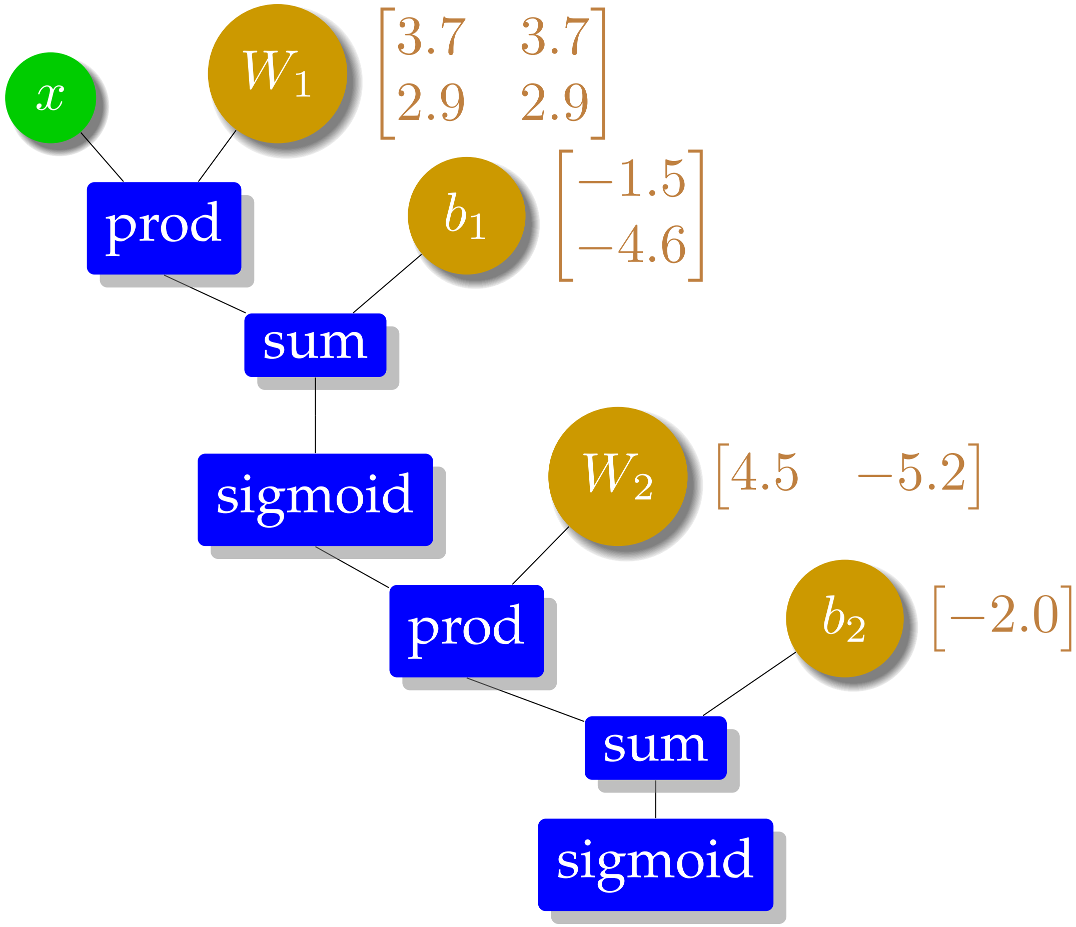
# Computation Graph



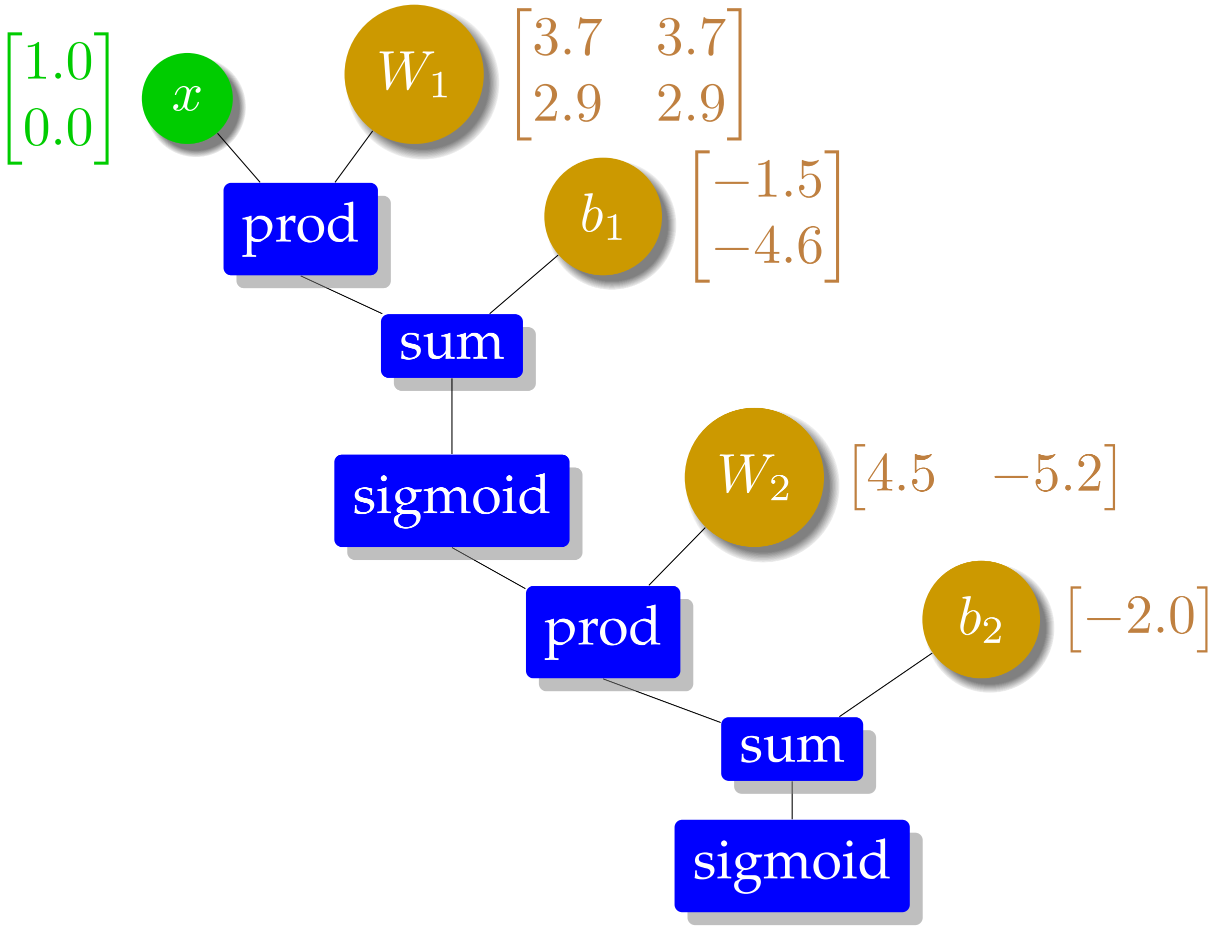
# Simple Neural Network



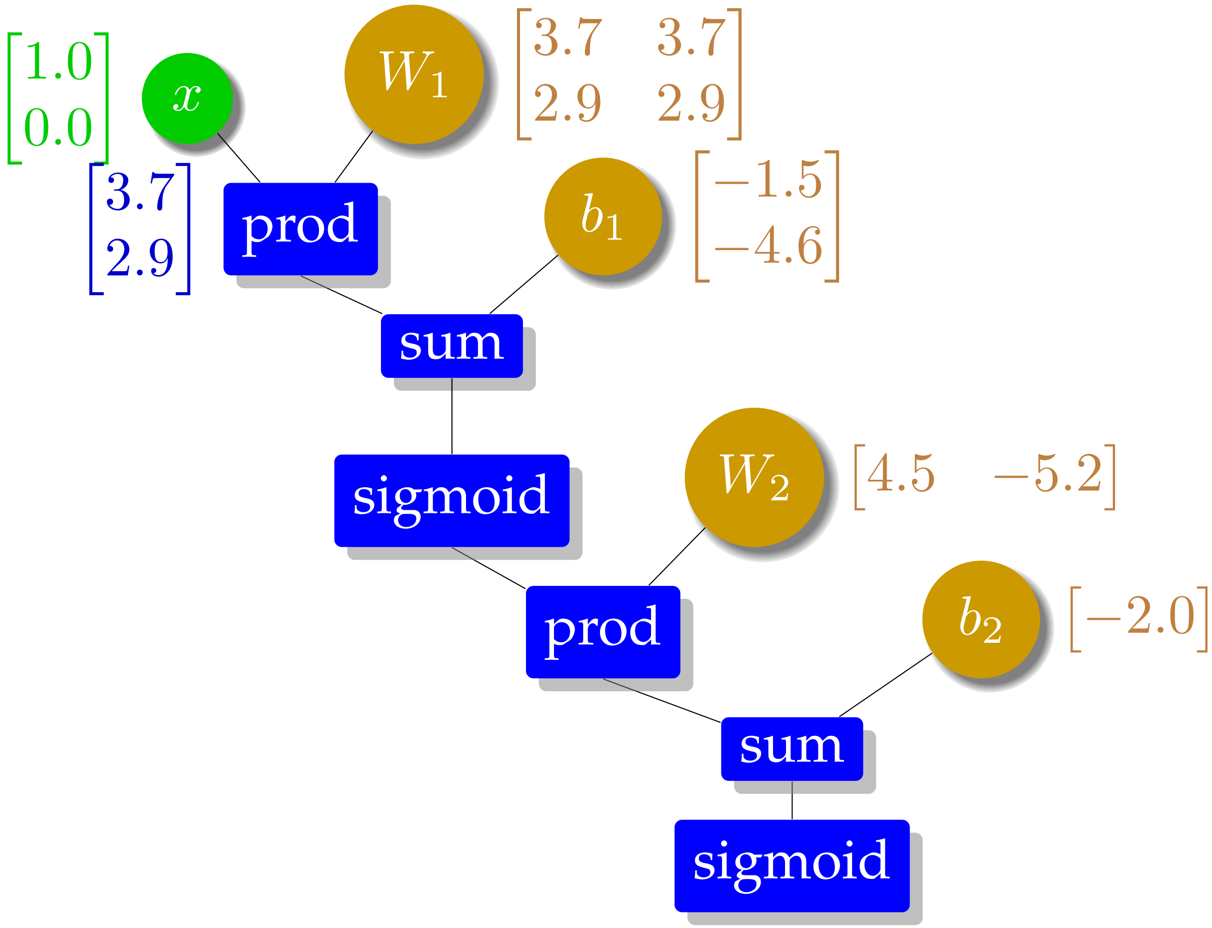
# Computation Graph



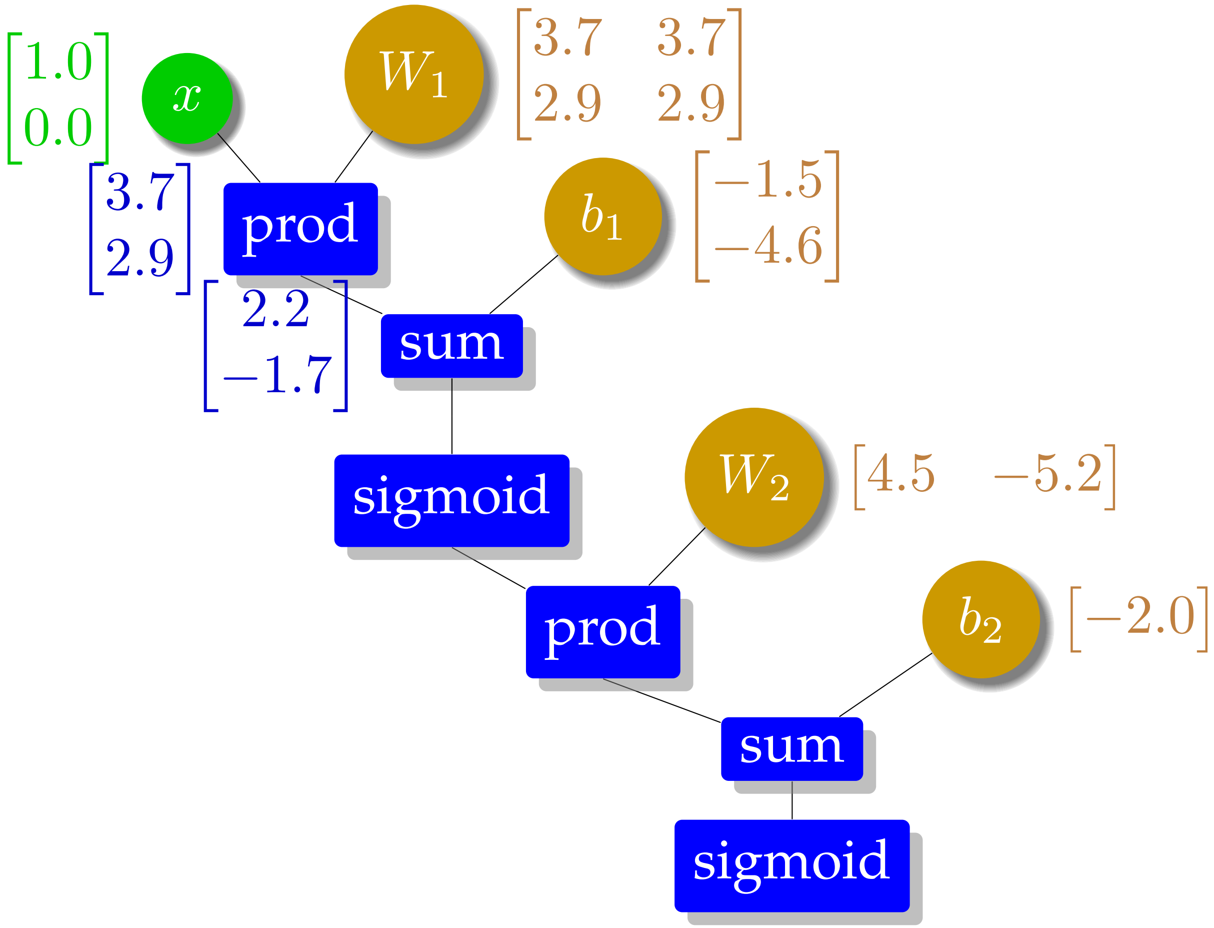
# Processing Input



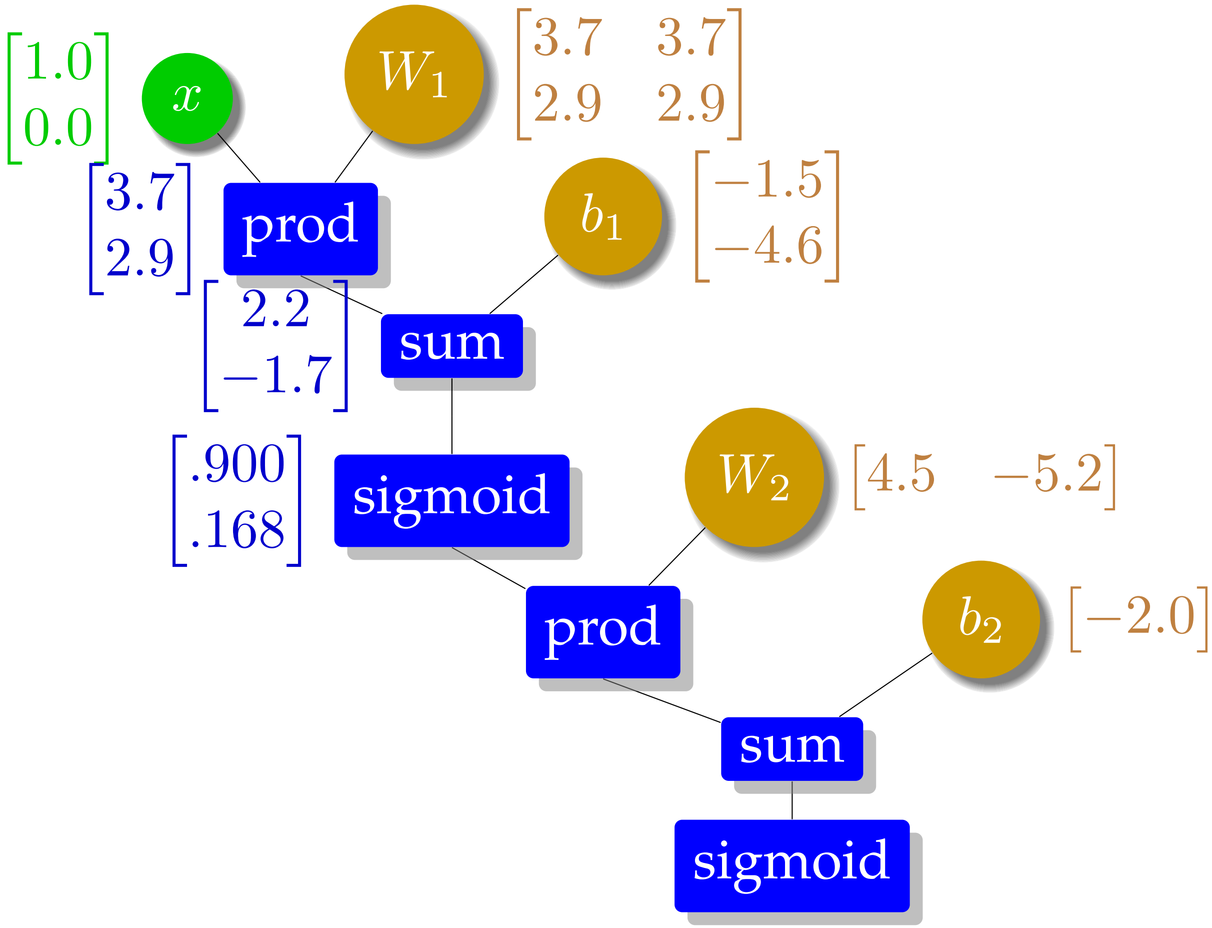
# Processing Input



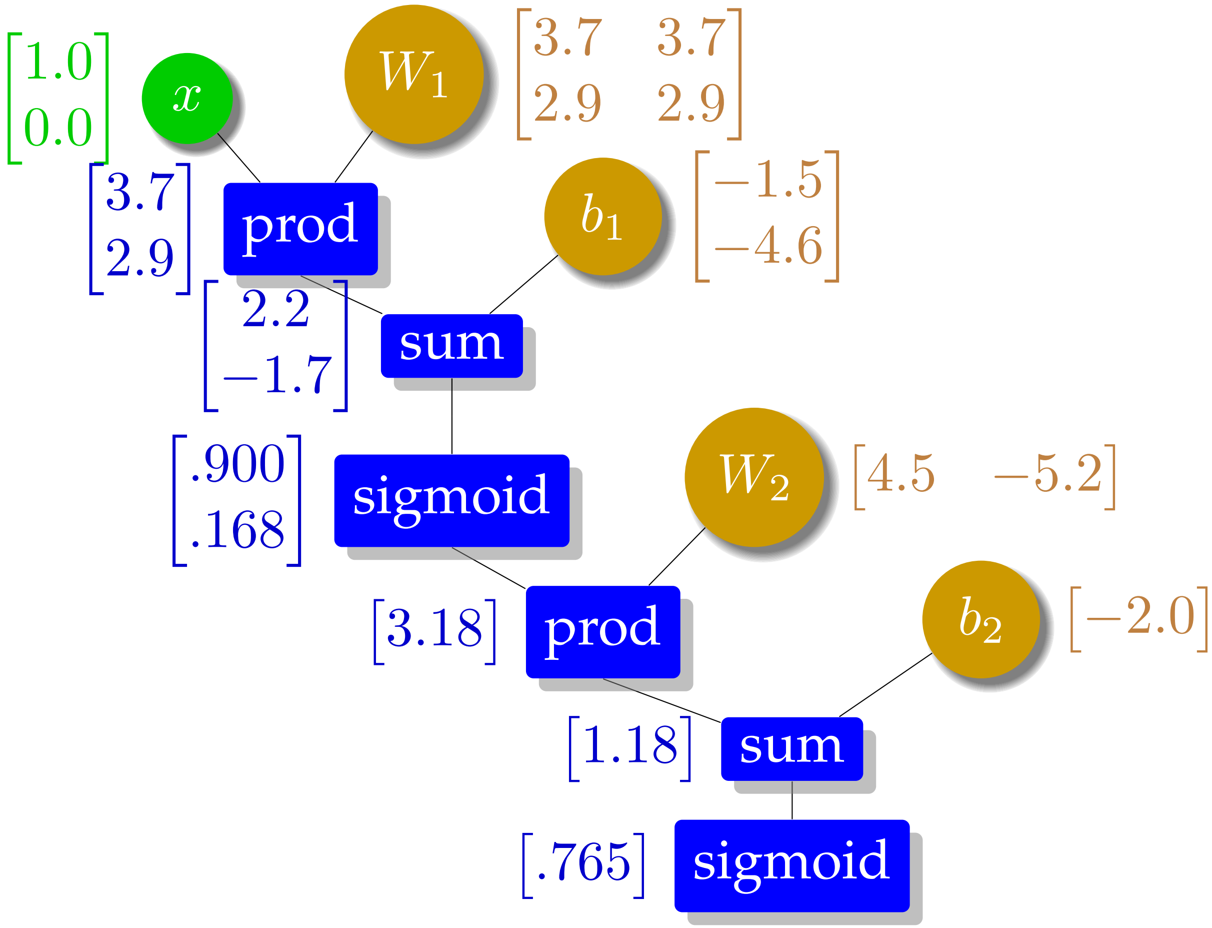
# Processing Input



# Processing Input



# Processing Input



# Error Function



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

$\Rightarrow$ Cost function

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

- For instance L2 norm

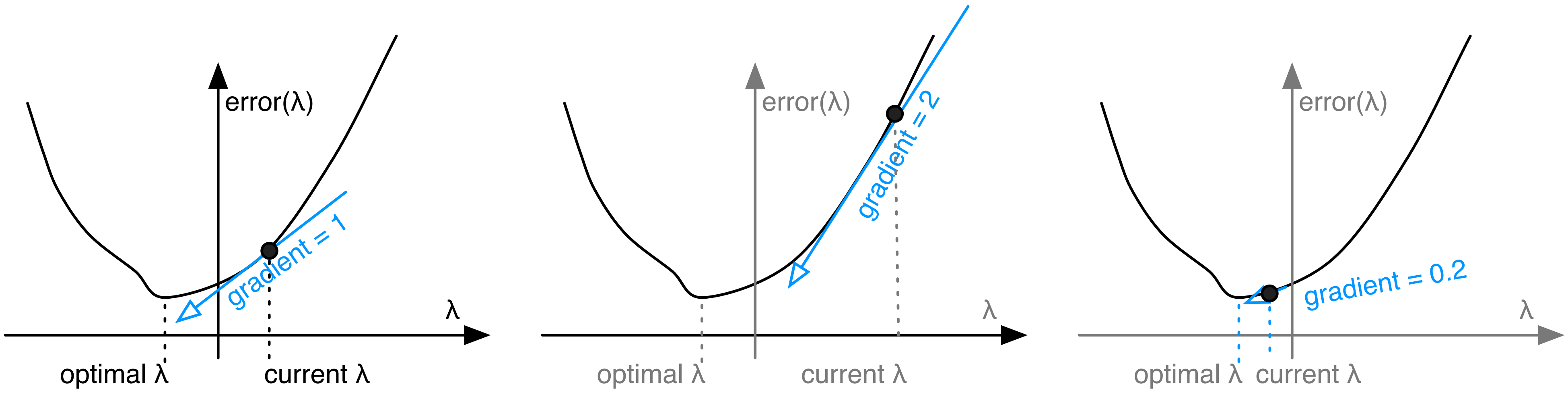
$$\text{error} = \frac{1}{2}(t - y)^2$$

# 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

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

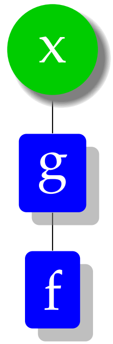
- Leibniz's notation

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

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

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

# Chain Rule in the Computation Graph



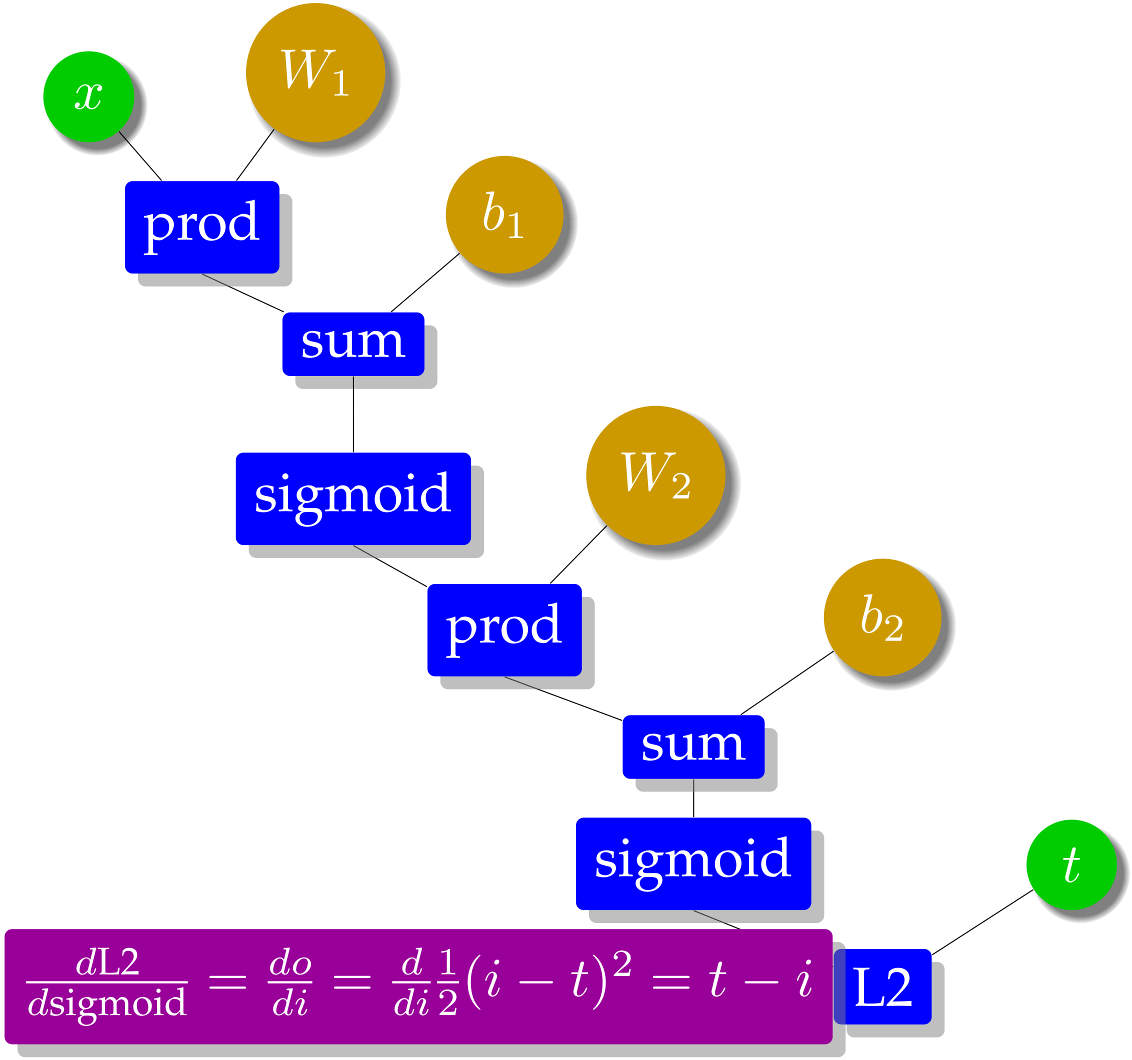

$g(x)$   g   $g'(x) \times \downarrow$

$f(g(x))$   f   $f'(g(x))$

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

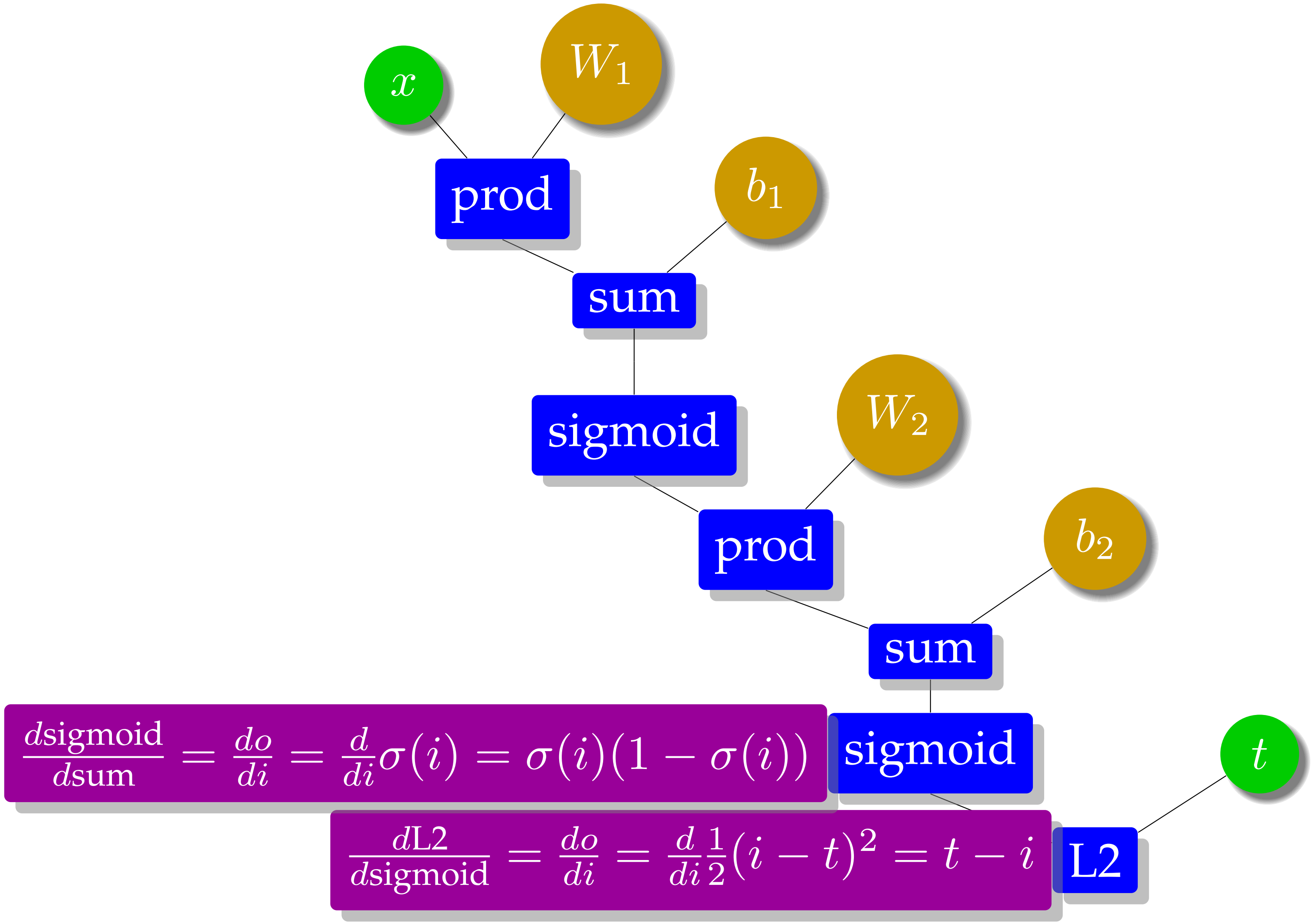
# Derivatives for Each Node



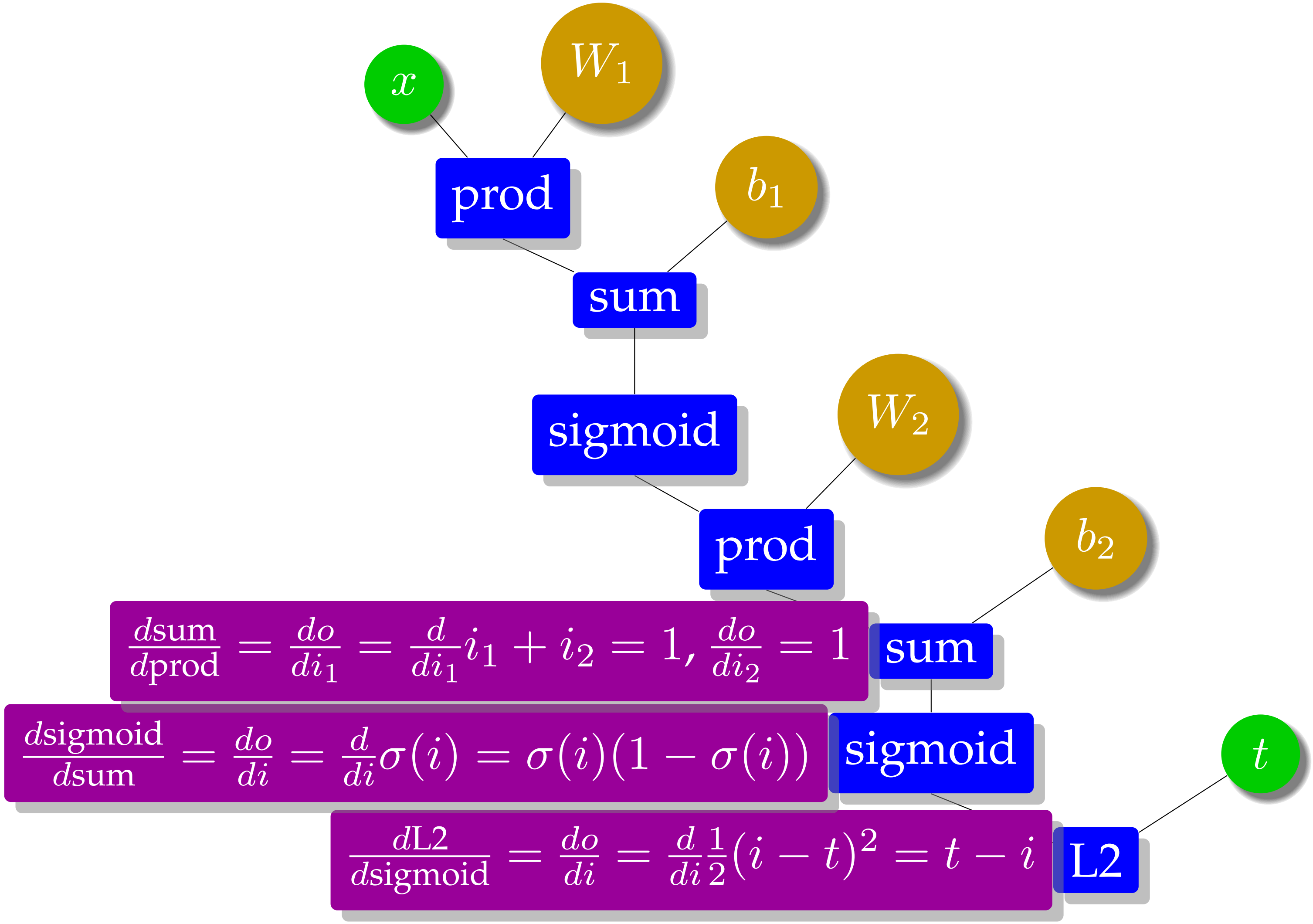


# Derivatives for Each Node



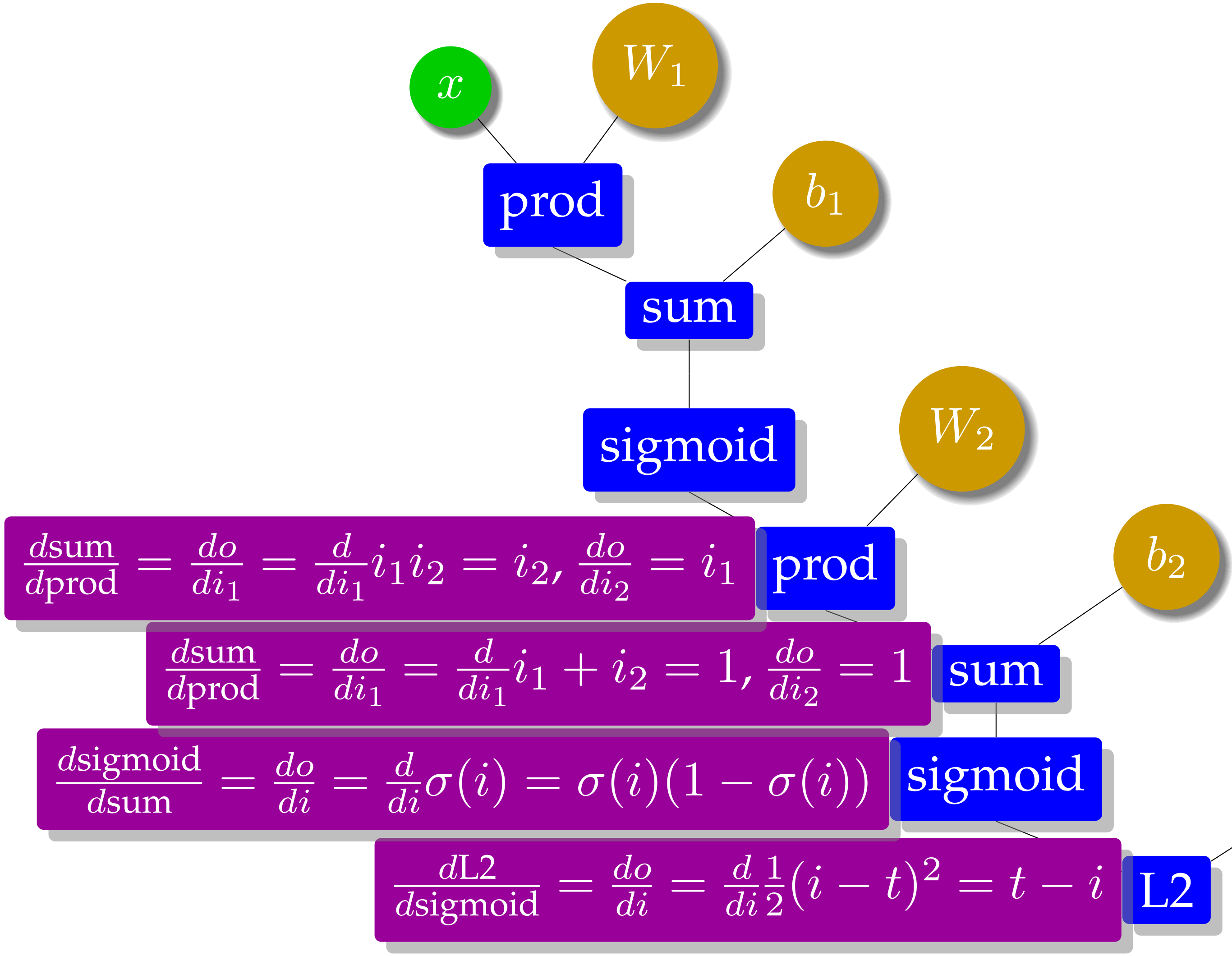


# Derivatives for Each Node

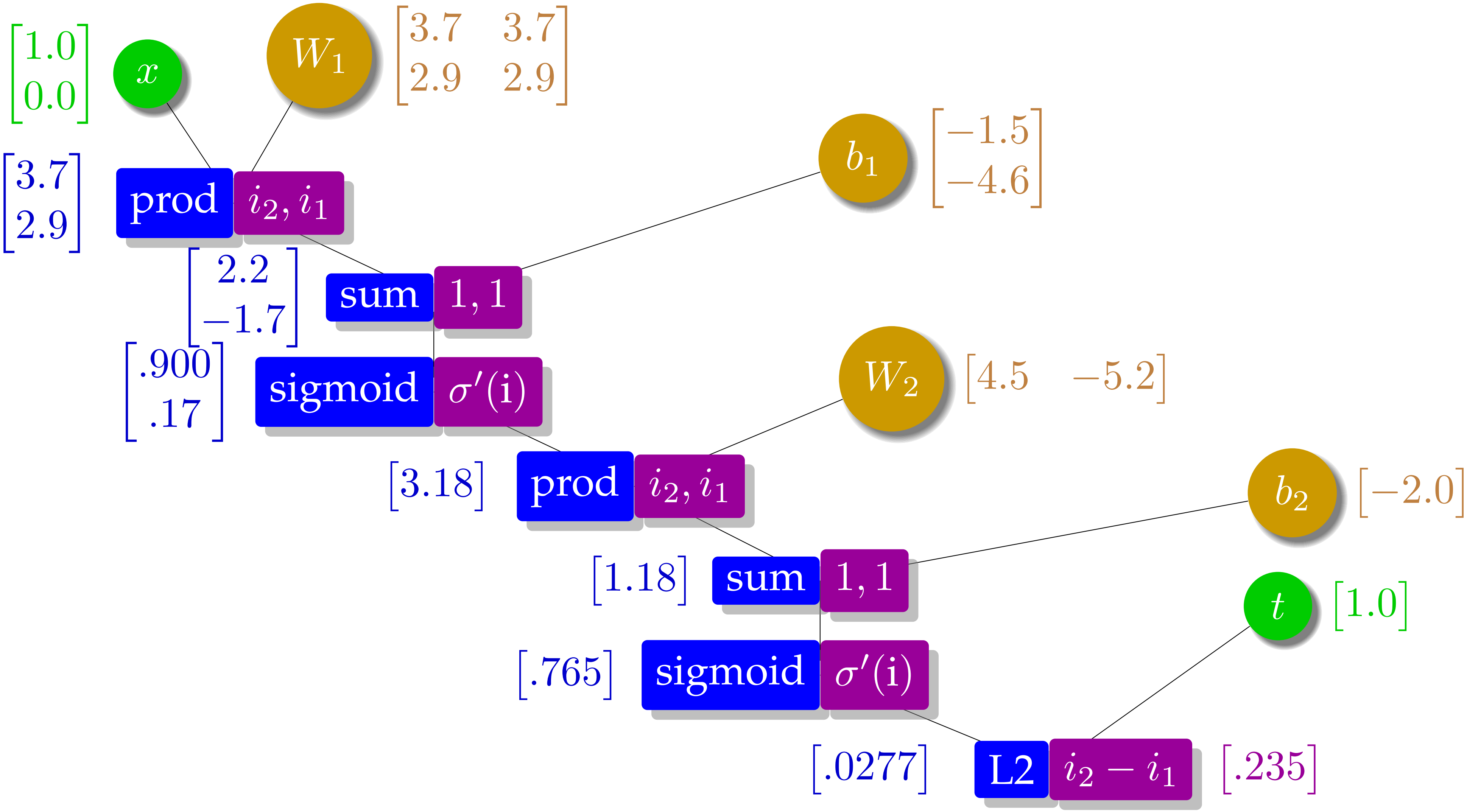


# Derivatives for Each Node

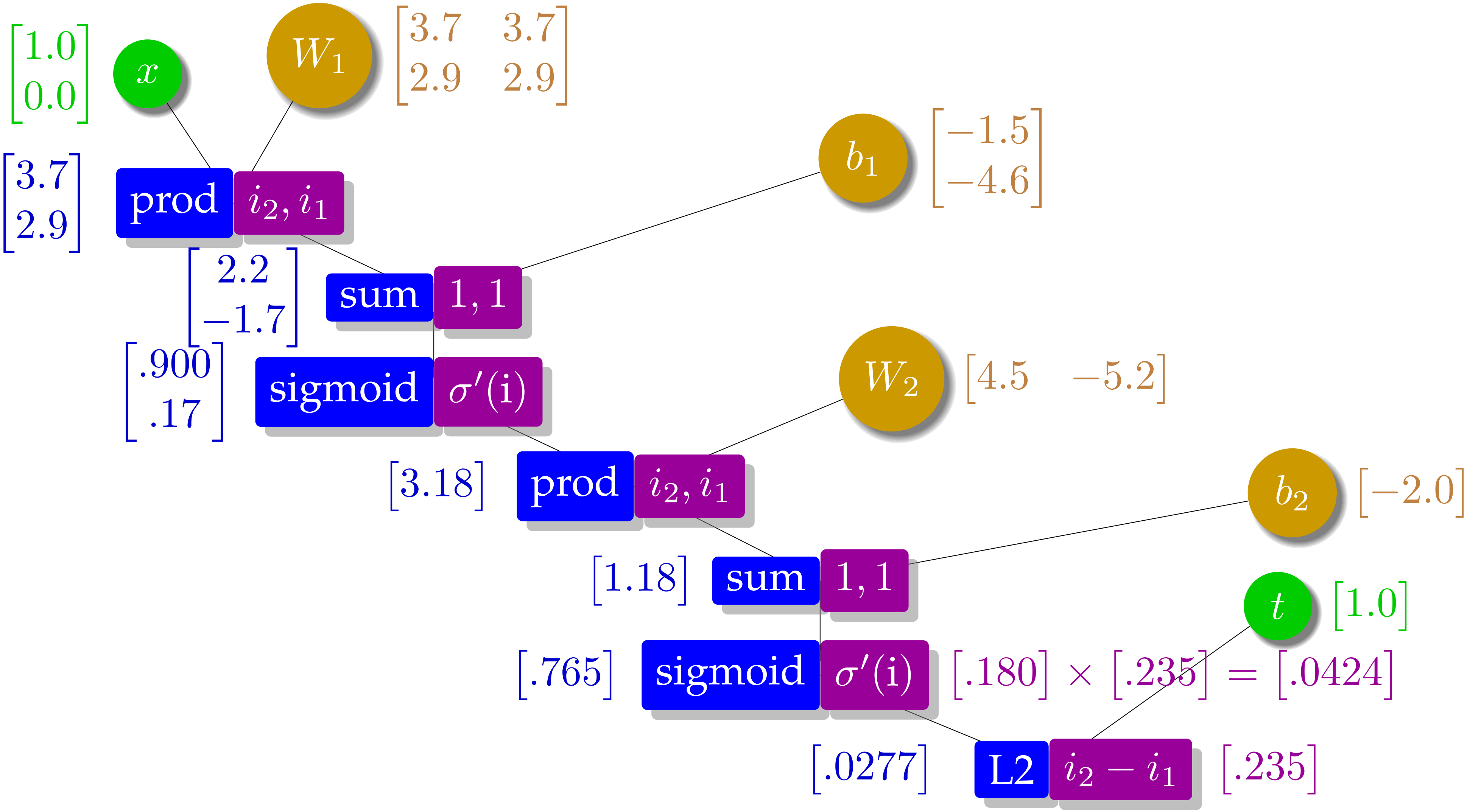




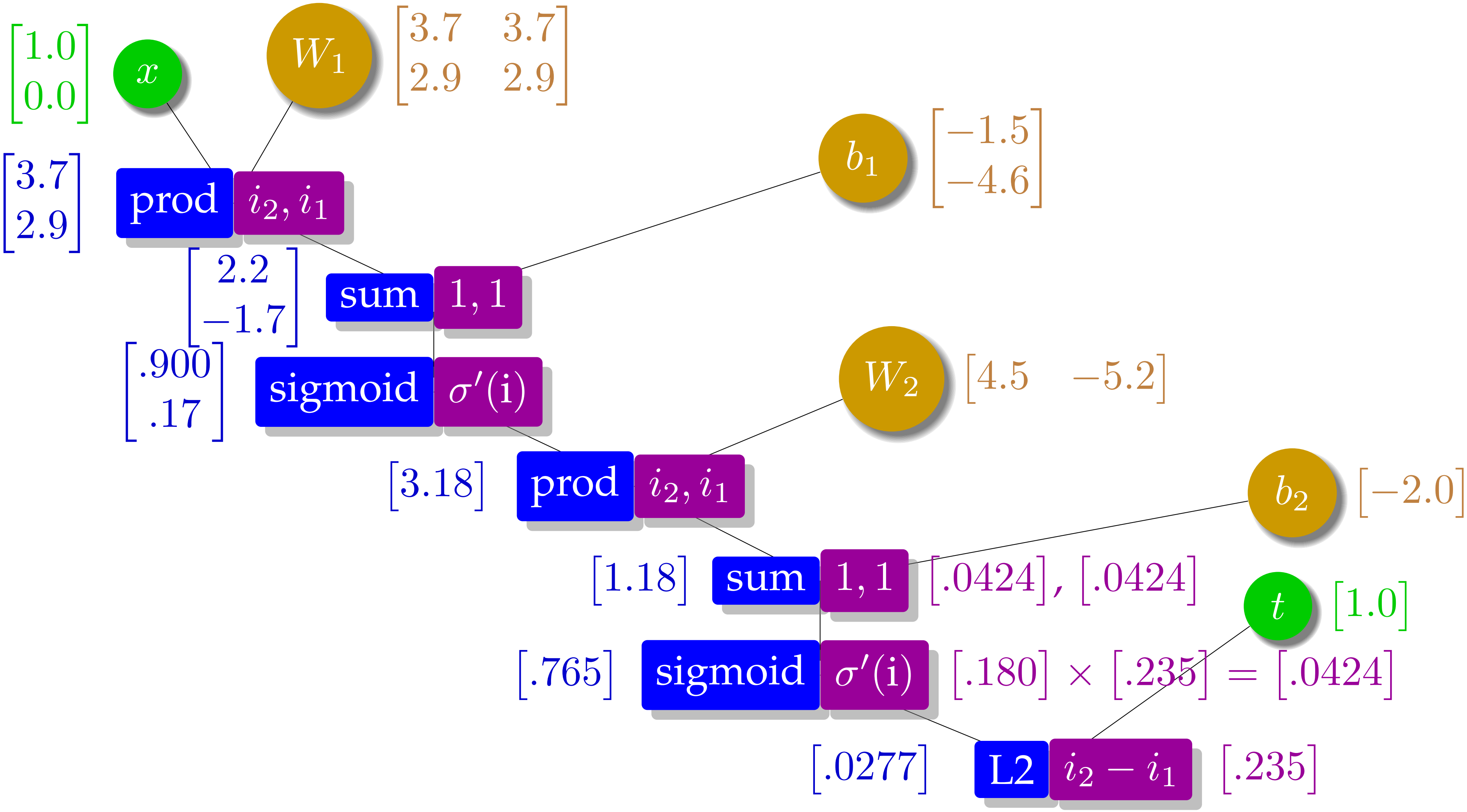
# Backward Pass: Derivative Computation



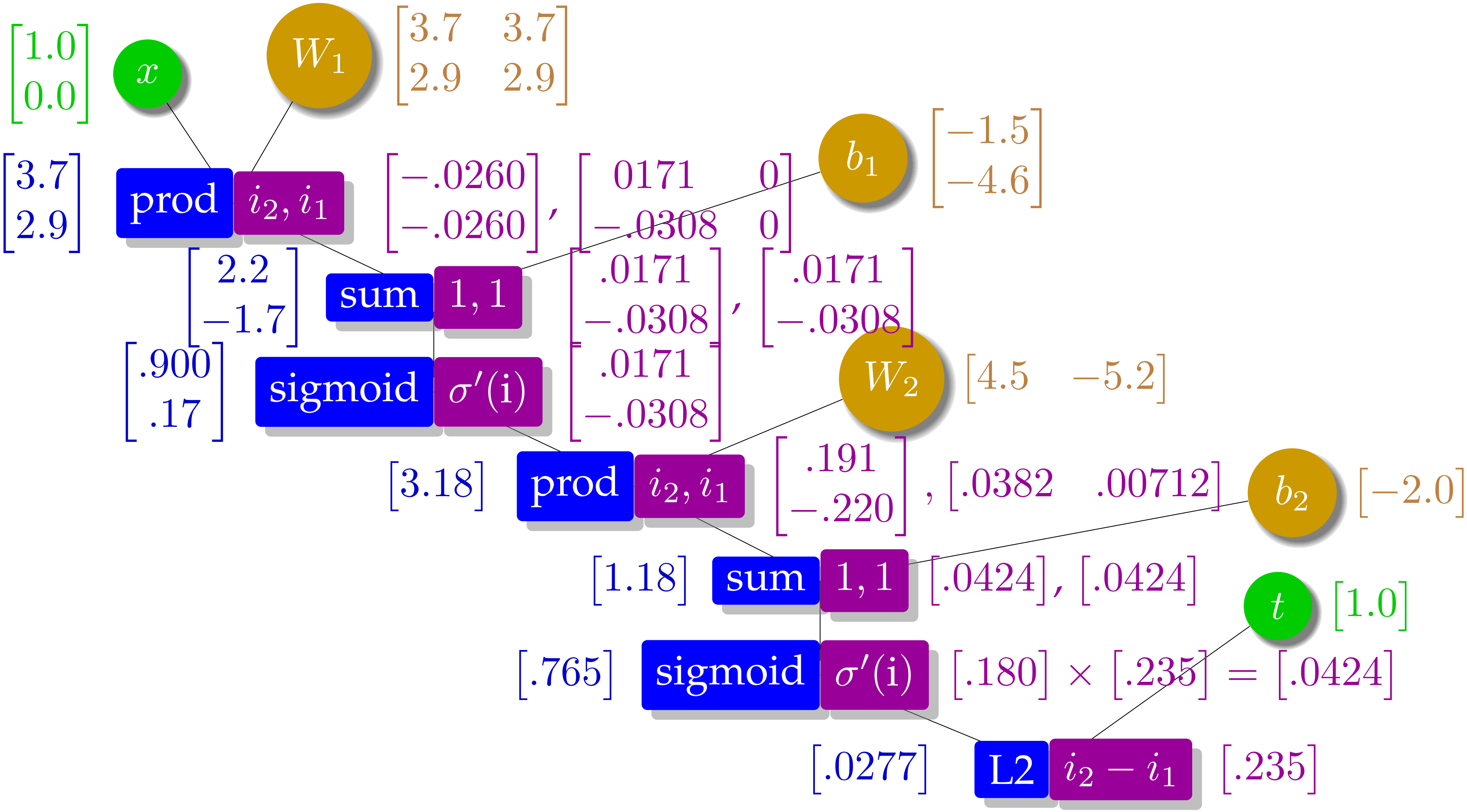
# Backward Pass: Derivative Computation



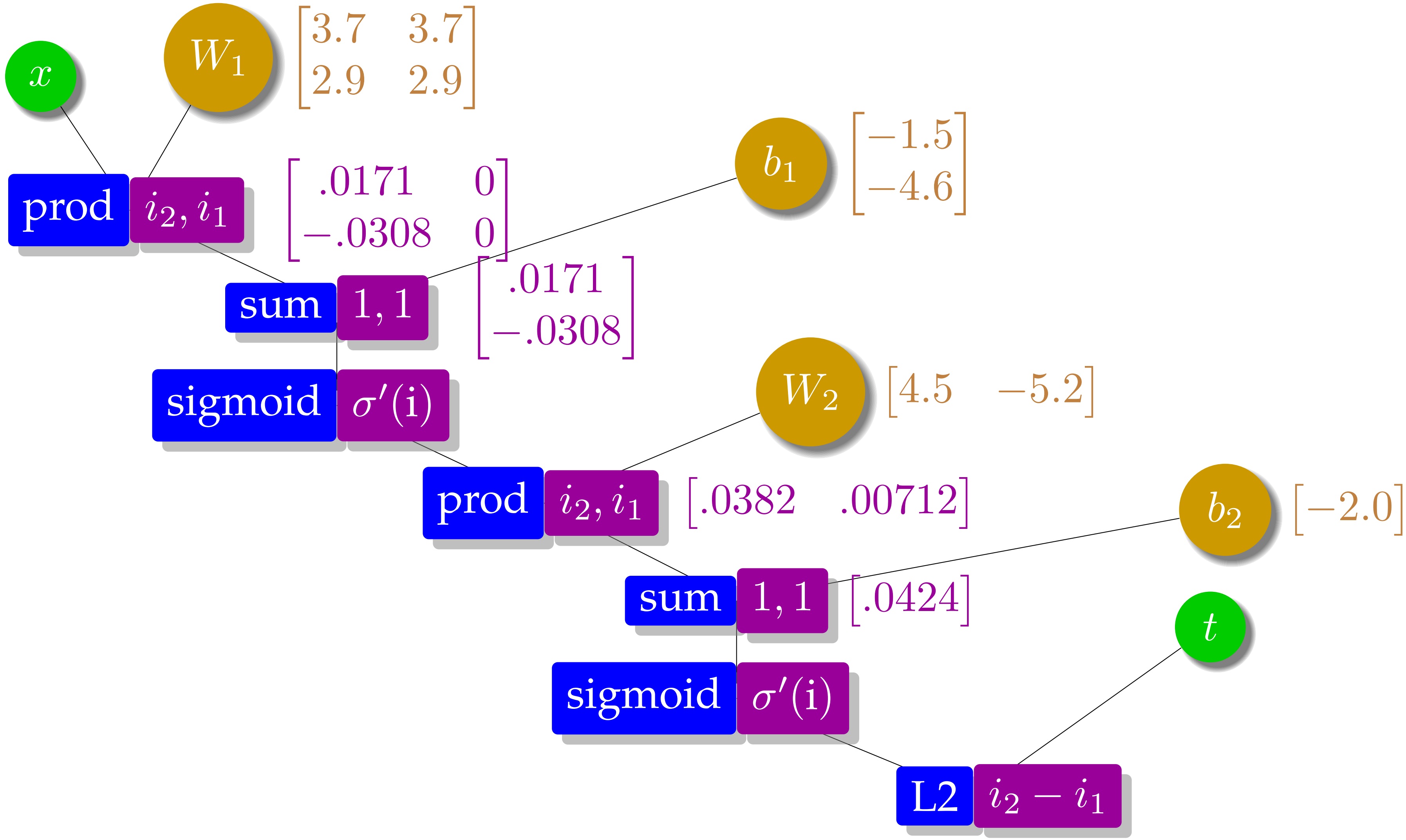
# Backward Pass: Derivative Computation



# Backward Pass: Derivative Computation

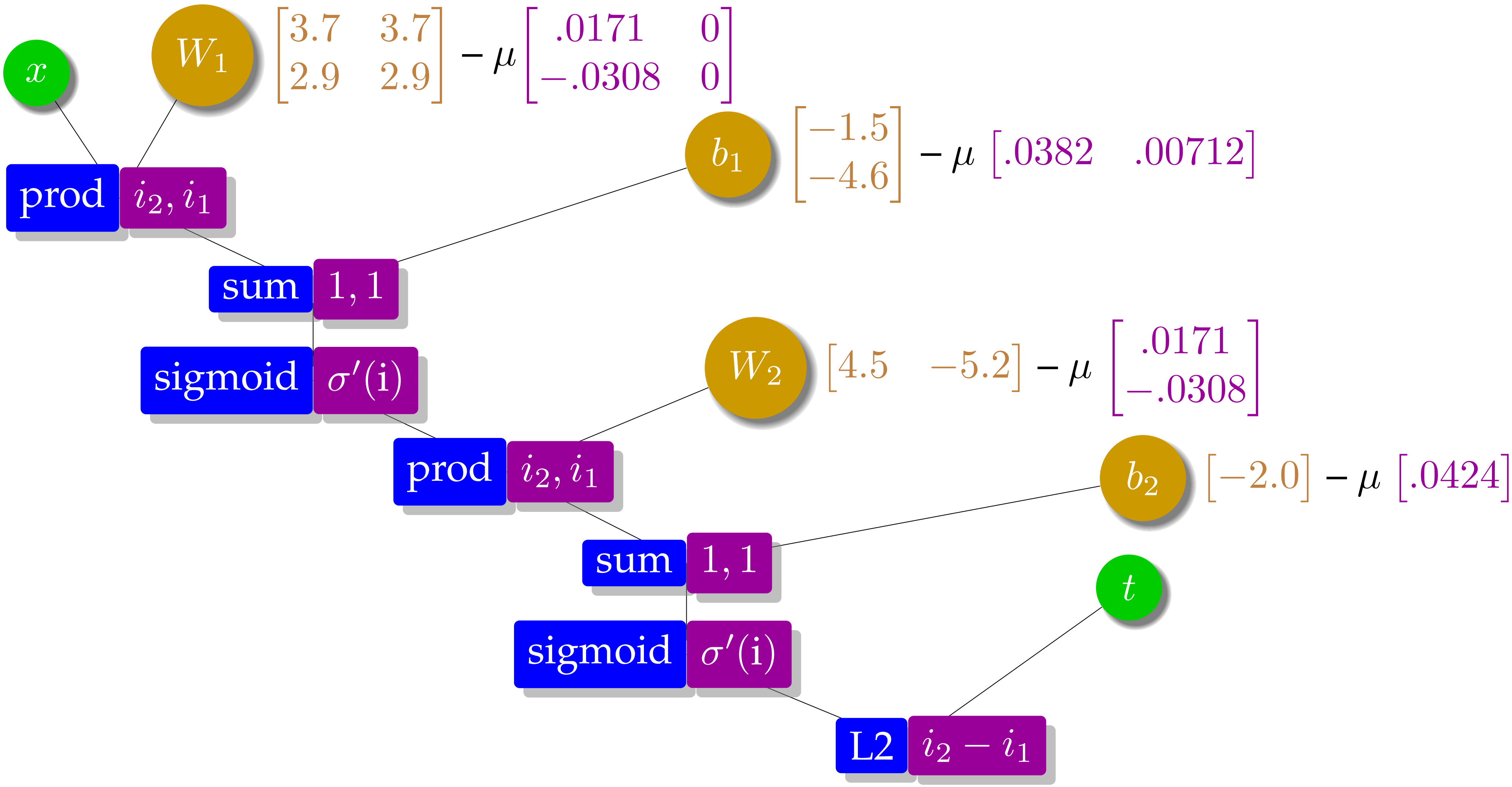


# Gradients for Parameter Update



# Parameter Update





# toolkits

# Explosion of Deep Learning Toolkits



- University of Montreal: Theano (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 - y) ** 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()
```

# language models

# N-Gram Backoff Language Model



- Previously, we approximated

$$p(W) = p(w_1, w_2, ..., w_n)$$

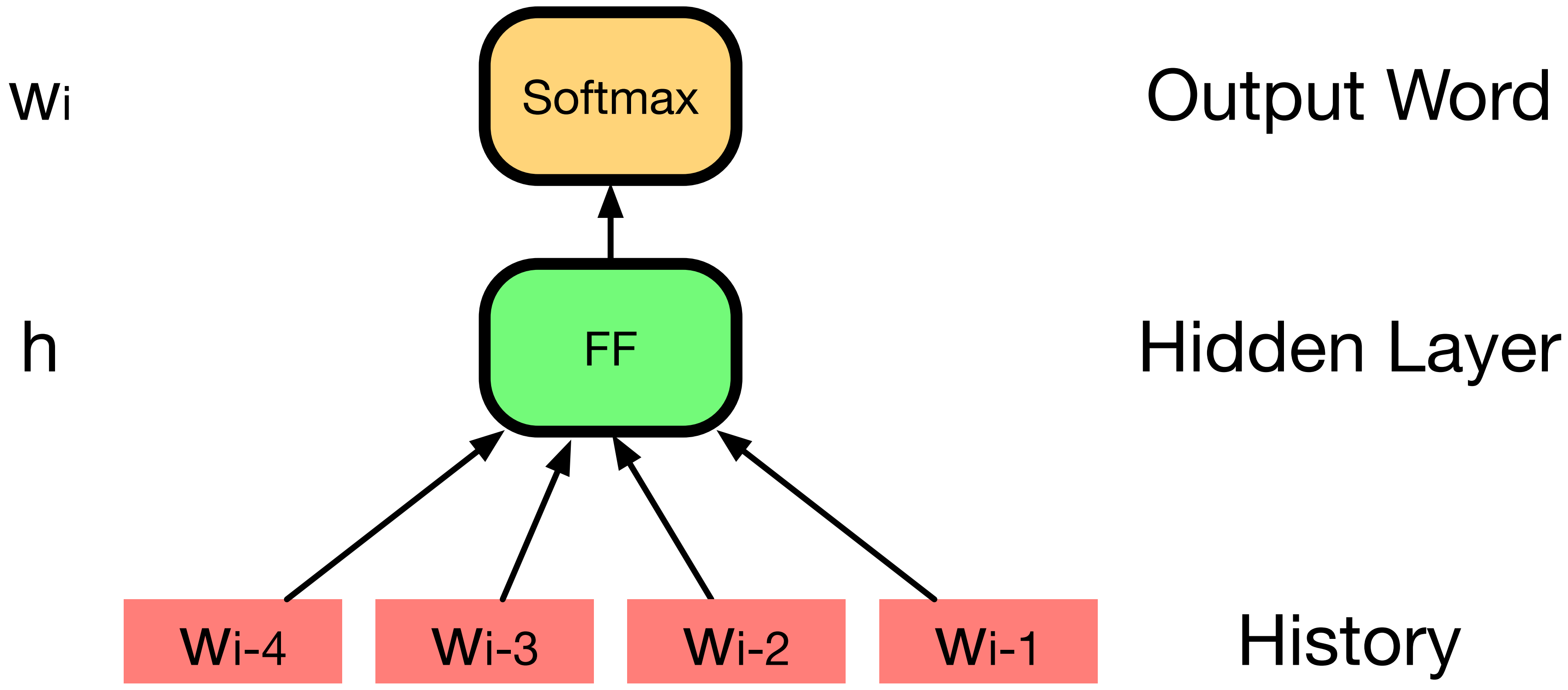
- ... by applying the chain rule

$$p(W) = \sum_i p(w_i | w_1, ..., w_{i-1})$$

- ... and limiting the history (Markov order)

$$p(w_i | w_1, ..., w_{i-1}) \simeq p(w_i | w_{i-4}, w_{i-3}, w_{i-2}, w_{i-1})$$

# First Sketch



# Representing Words



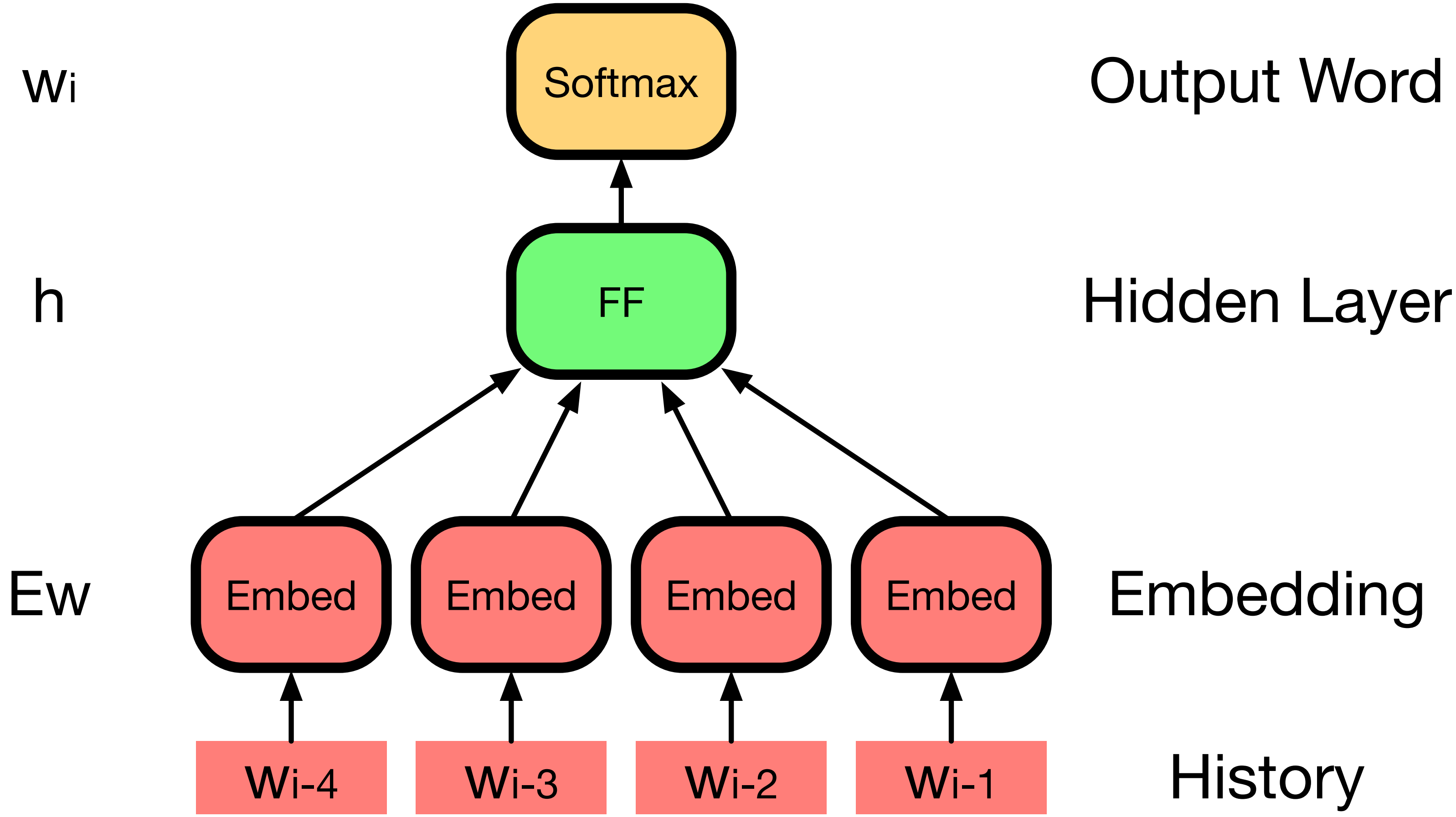
- Words are represented with a one-hot vector, e.g.,
  - dog = (0,0,0,0,1,0,0,0,0,....)
  - cat = (0,0,0,0,0,0,0,1,0,....)
  - eat = (0,1,0,0,0,0,0,0,0,....)

- That's a large vector!

- Remedies
  - limit to, say, 20,000 most frequent words, rest are OTHER
  - place words in $\sqrt{n}$ classes, so each word is represented by
    - 1 class label
    - 1 word in class label
  - splitting rare words into subwords
  - character-based models

# word embeddings

# Add an Embedding Layer



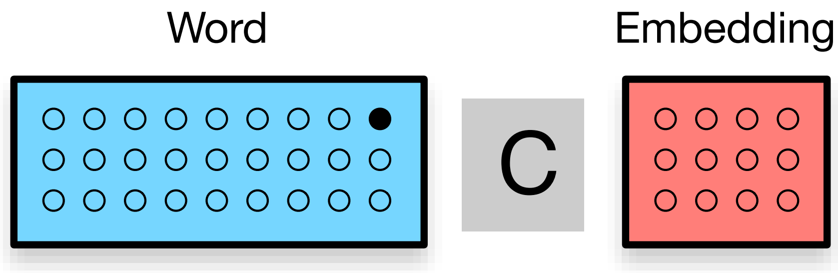

- Map each word first into a lower-dimensional real-valued space
- Shared weight matrix $E$

# Details (Bengio et al., 2003)



- Add direct connections from embedding layer to output layer
- Activation functions
  - input→embedding: none
  - embedding→hidden: tanh
  - hidden→output: softmax
- Training
  - loop through the entire corpus
  - update between predicted probabilities and 1-hot vector for output word

# Word Embeddings

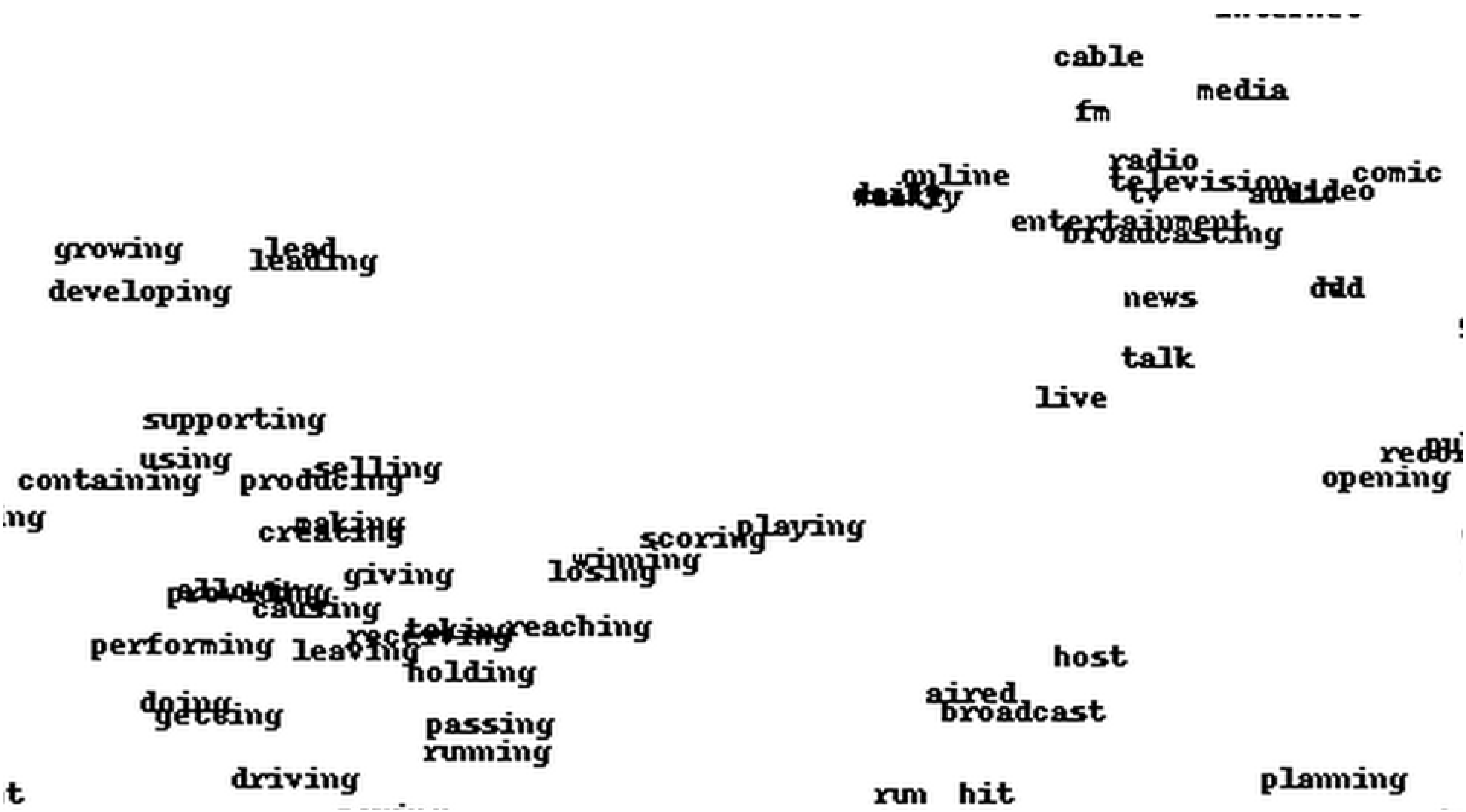



- By-product: embedding of word into continuous space
- Similar contexts → similar embedding
- Recall: distributional semantics

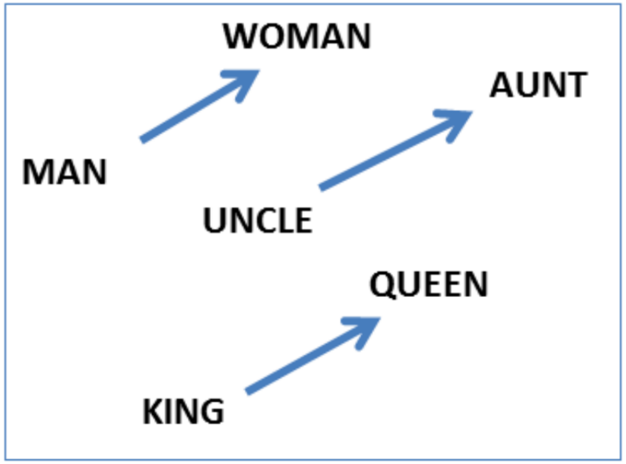
# Word Embeddings

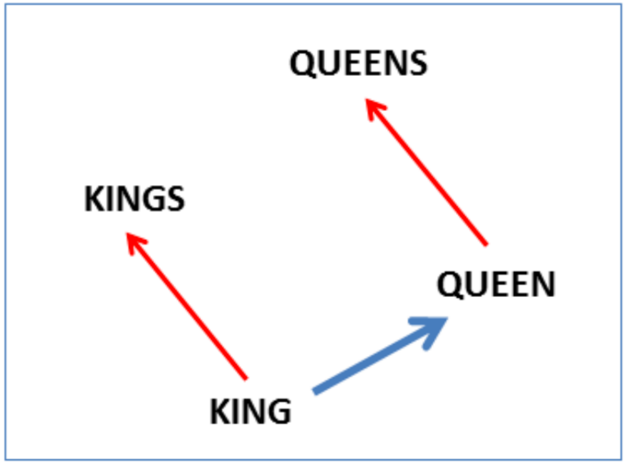
# Word Embeddings





# Are Word Embeddings Magic?




- Morphosyntactic regularities (Mikolov et al., 2013)
  - adjectives base form vs. comparative, e.g., good, better
  - nouns singular vs. plural, e.g., year, years
  - verbs present tense vs. past tense, e.g., see, saw

- Semantic regularities
  - clothing is to shirt as dish is to bowl
  - evaluated on human judgment data of semantic similarities

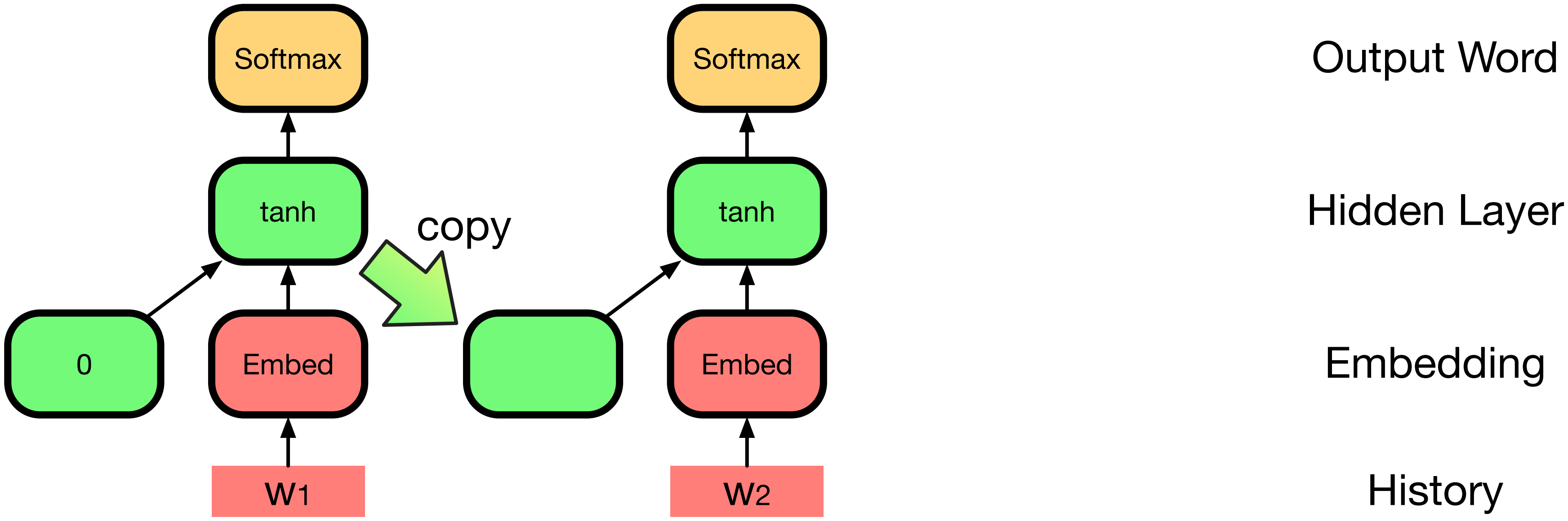
# recurrent neural networks

# Recurrent Neural Networks



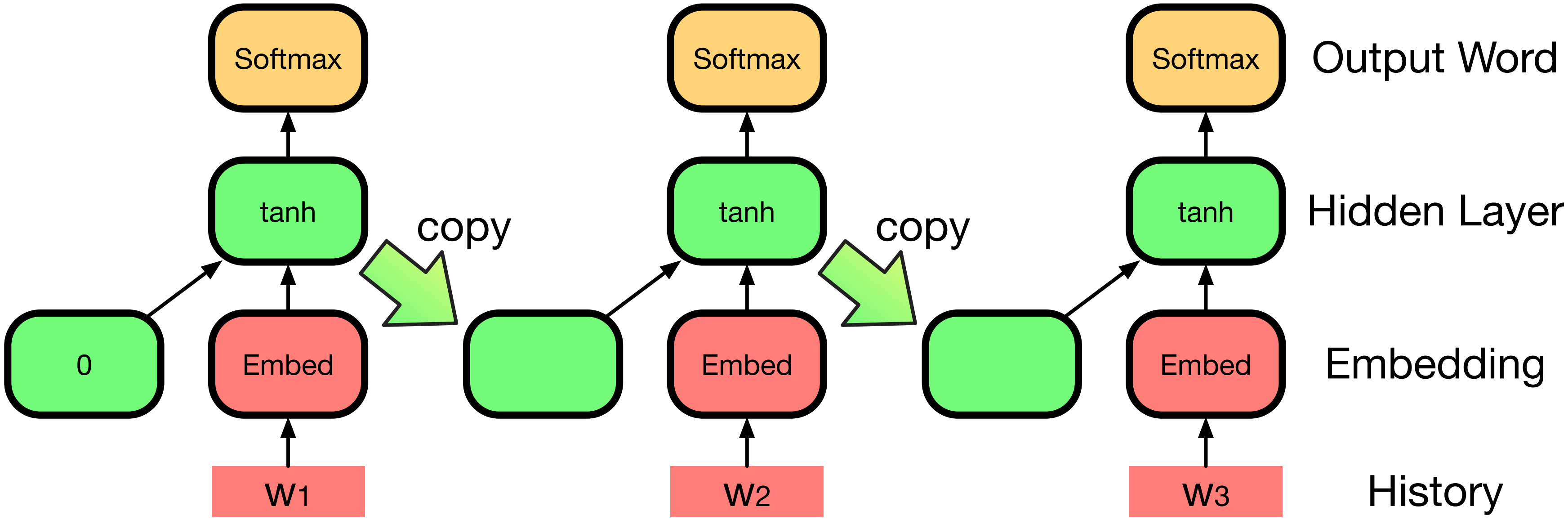

- Start: predict second word from first
- Mystery layer with nodes all with value 1

# Recurrent Neural Networks

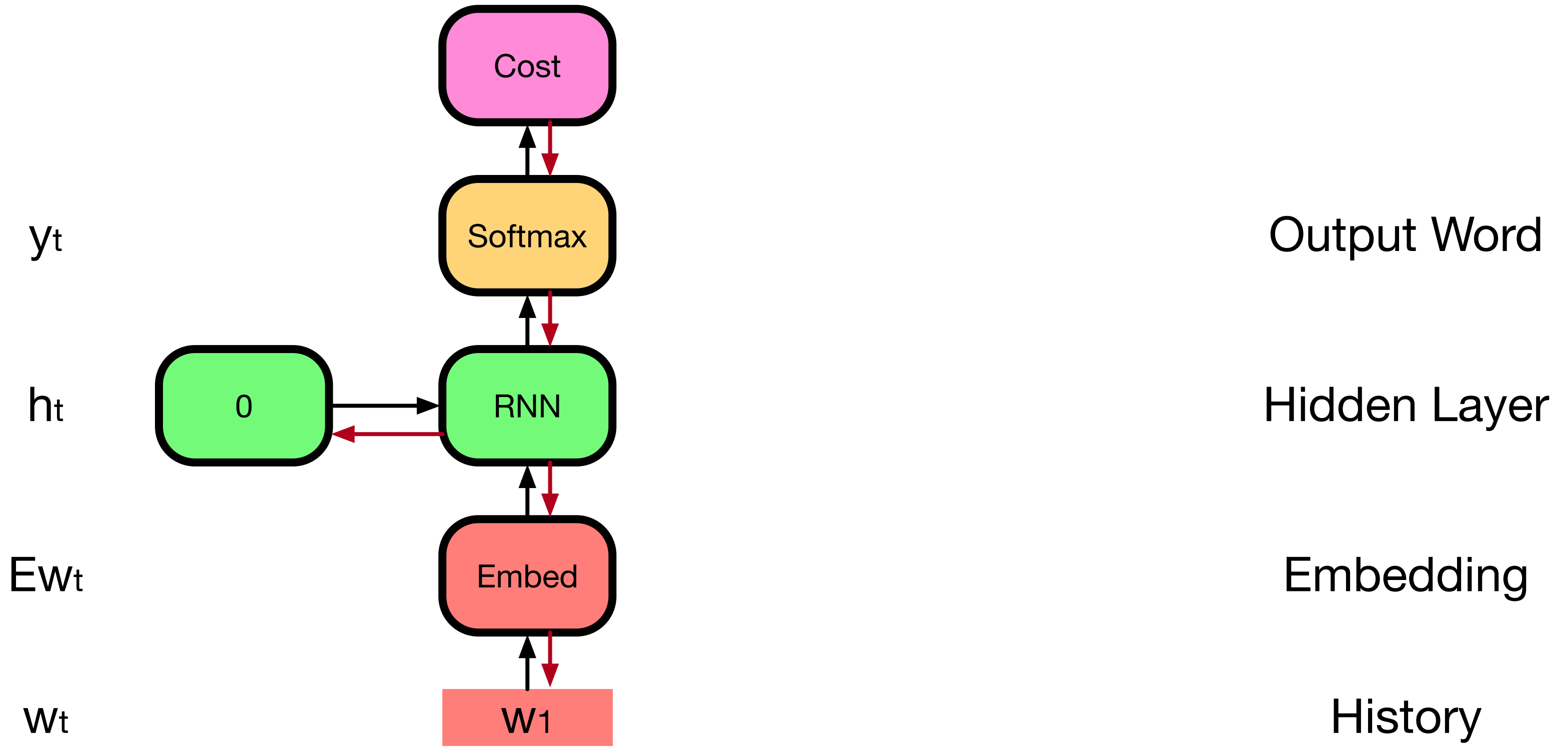


# Recurrent Neural Networks





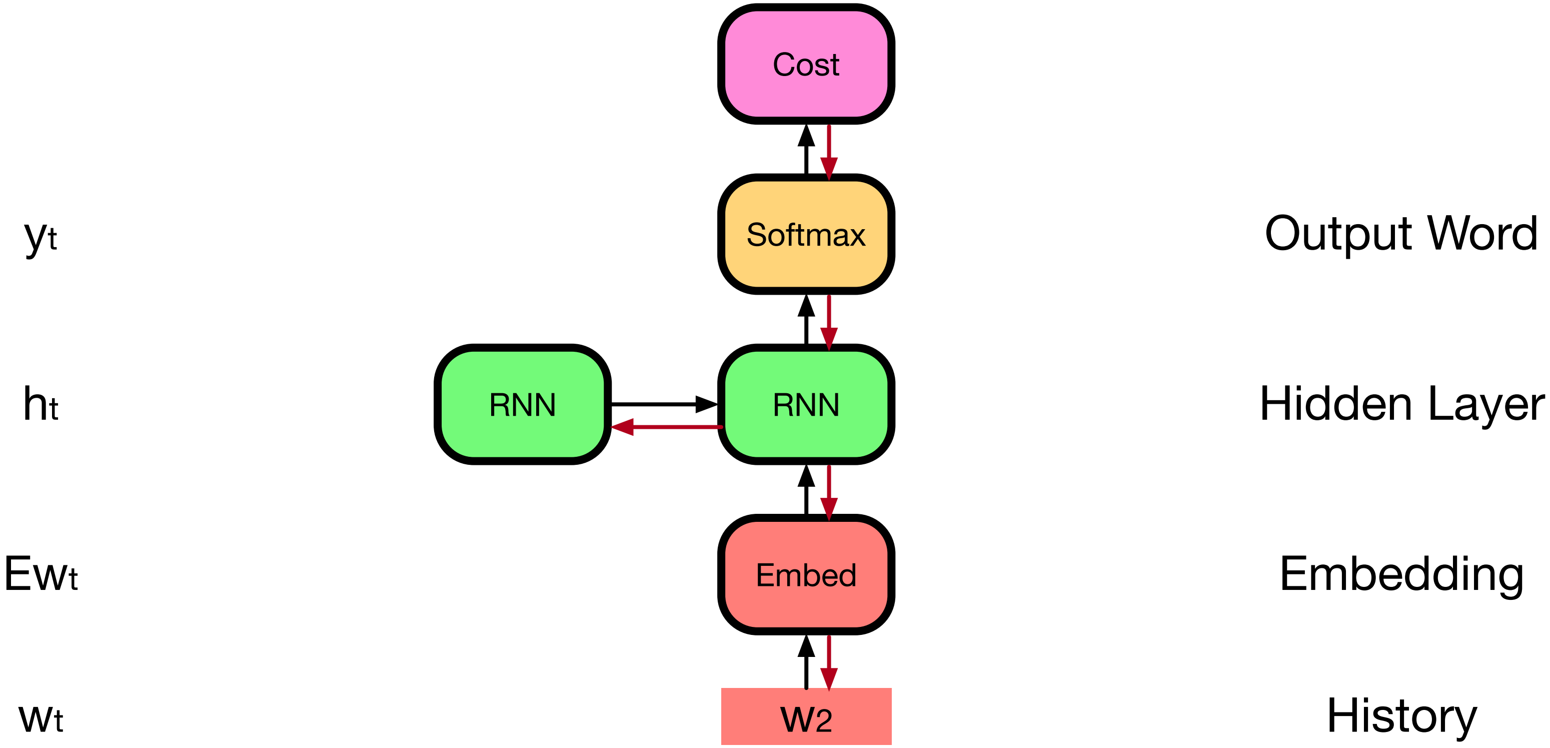
# Training




- Process first training example
- Update weights with back-propagation

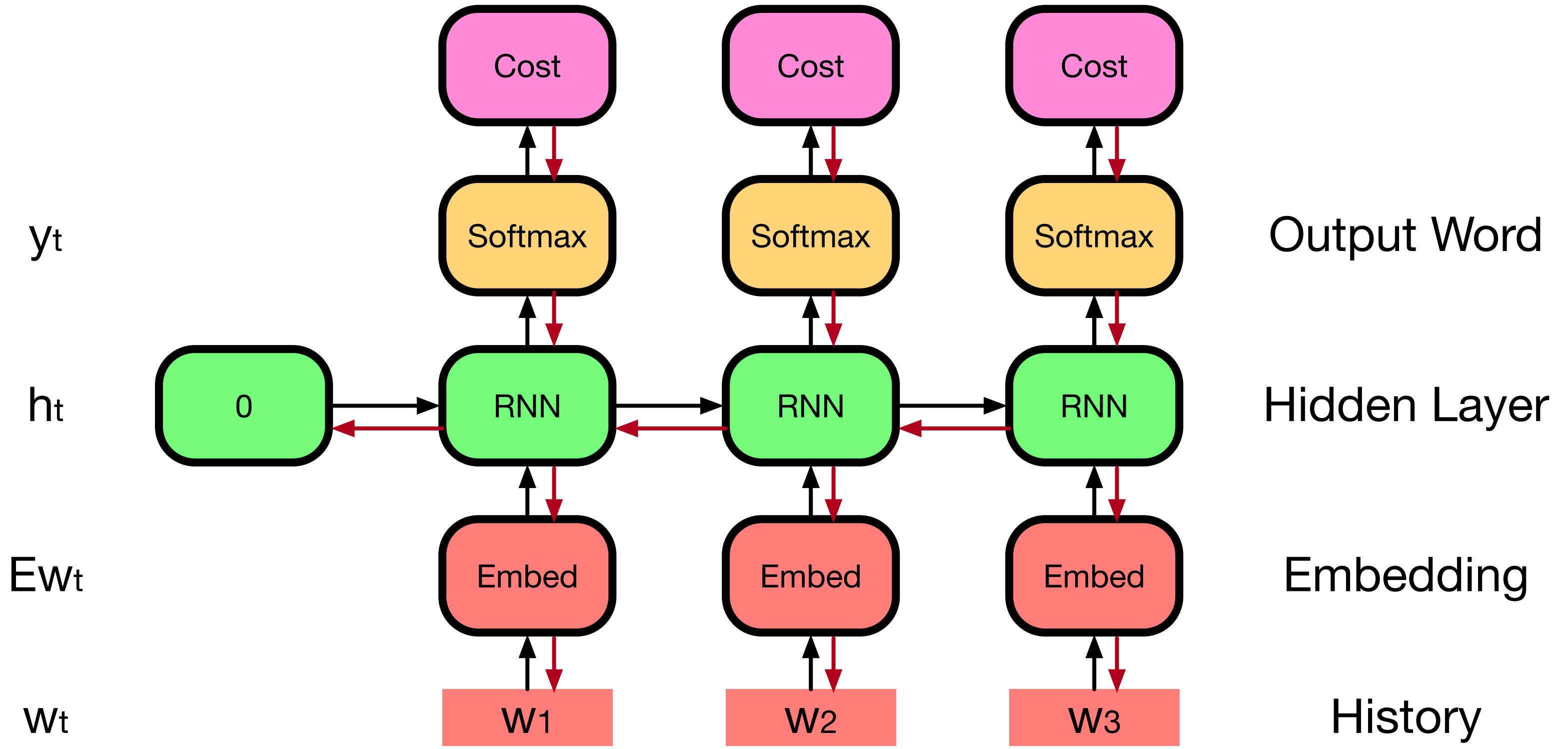
# Training






- Process second training example
- Update weights with back-propagation
- And so on...
- But: no feedback to previous history

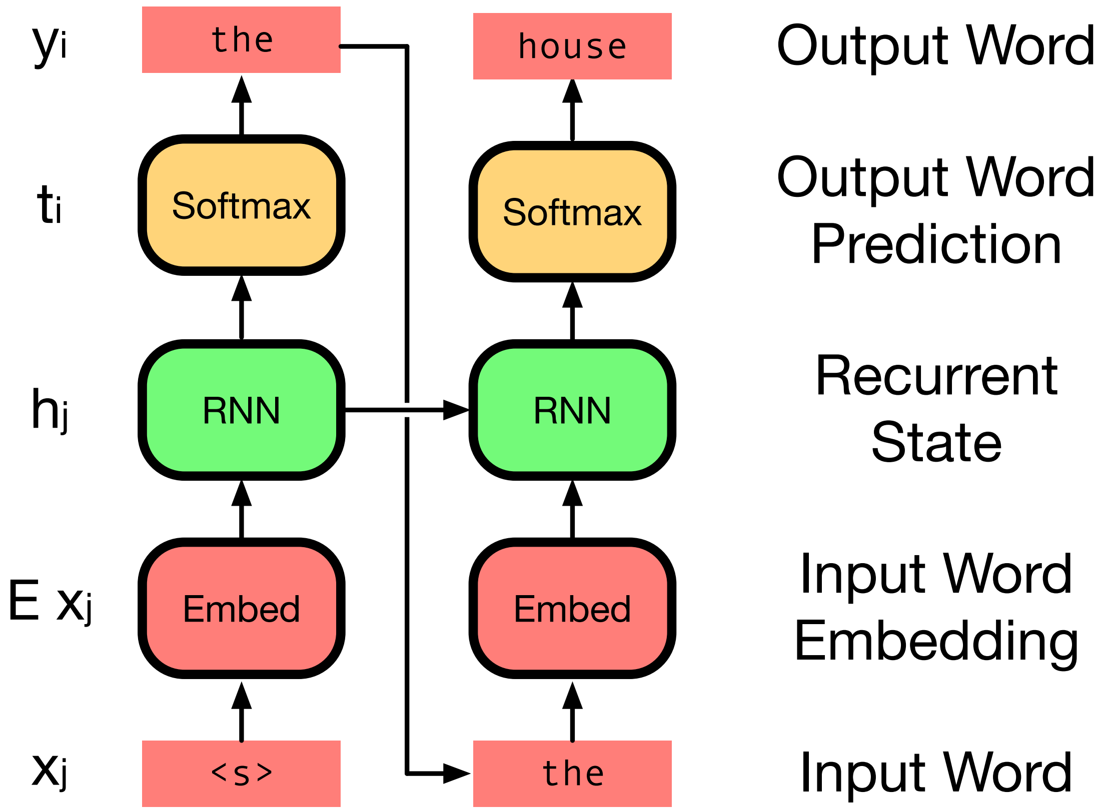
# Back-Propagation Through Time




- Unfolded recurrent neural network for a sentence

# neural translation models

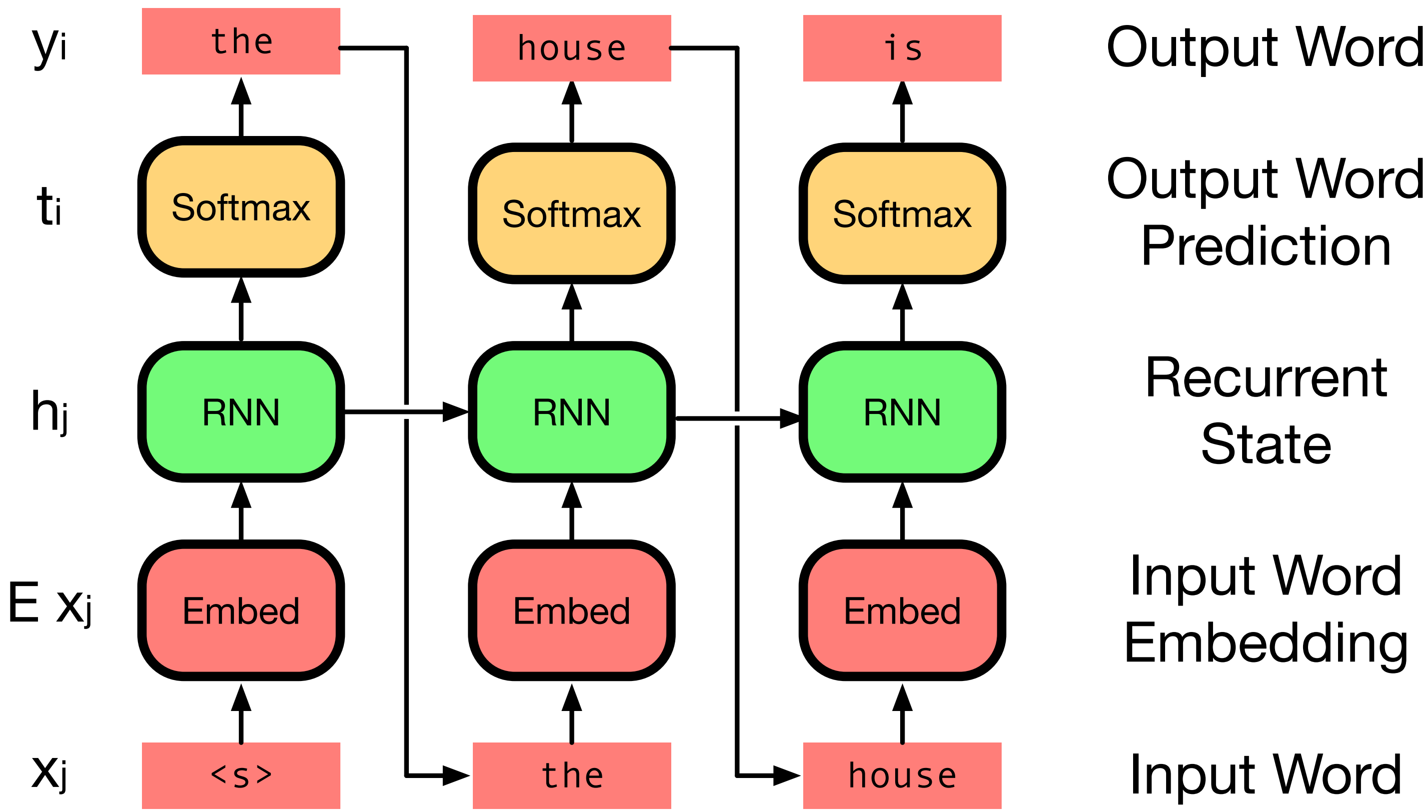
# Recurrent Neural Language Model




Predict the second word of a sentence

Re-use hidden state from first word prediction

# Recurrent Neural Language Model

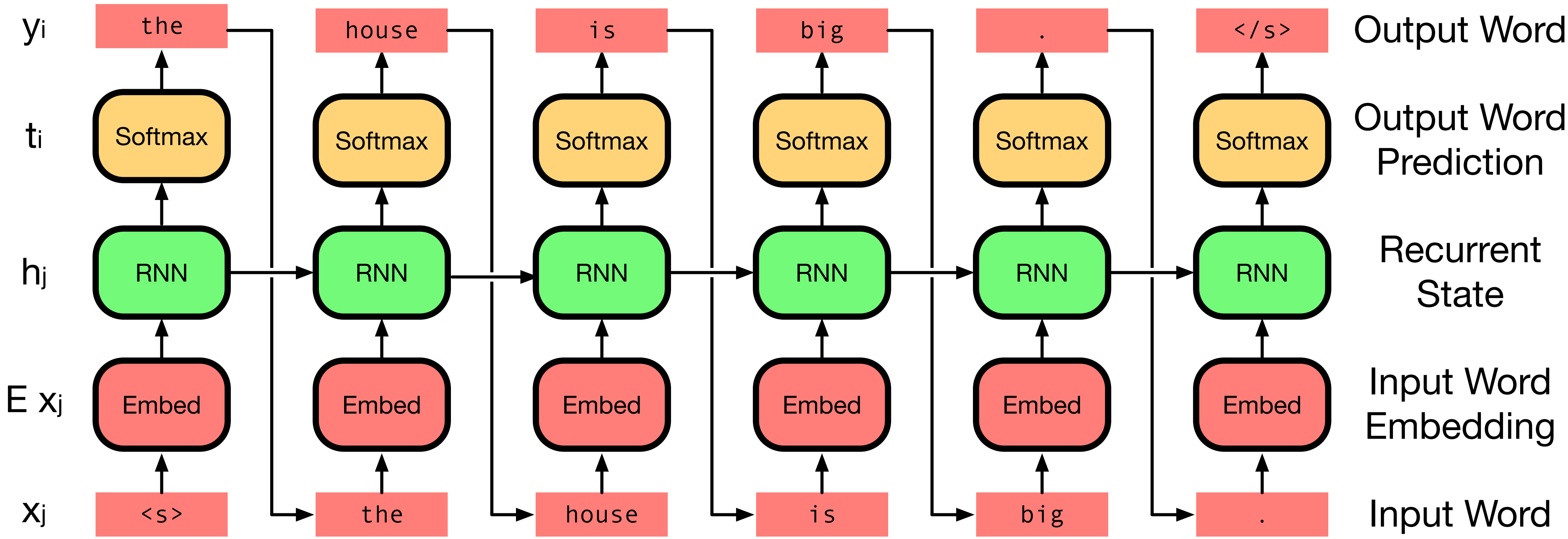



Predict the third word of a sentence

... and so on

# Recurrent Neural Language Model





# Recurrent Neural Translation Model

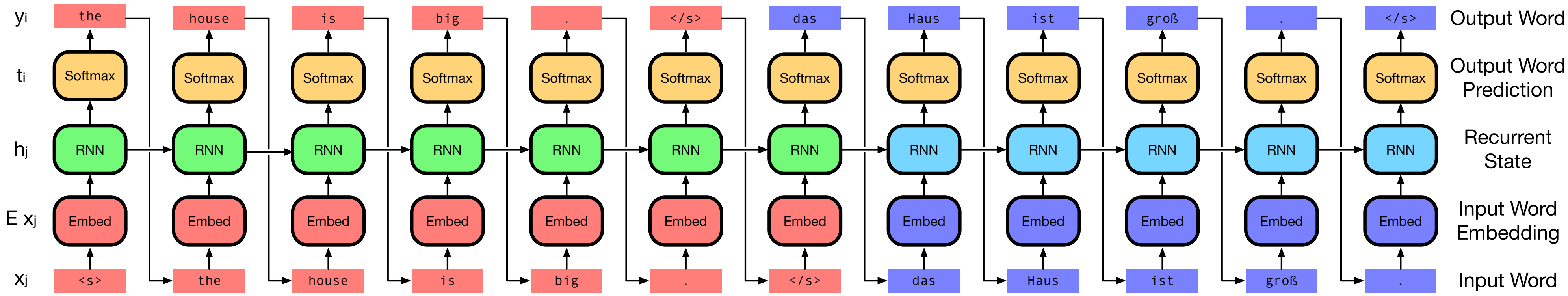


- We predicted the words of a sentence

- Why not also predict their translations?

# Encoder-Decoder Model






- Obviously madness
- Proposed by Google (Sutskever et al. 2014)

# What is Missing?

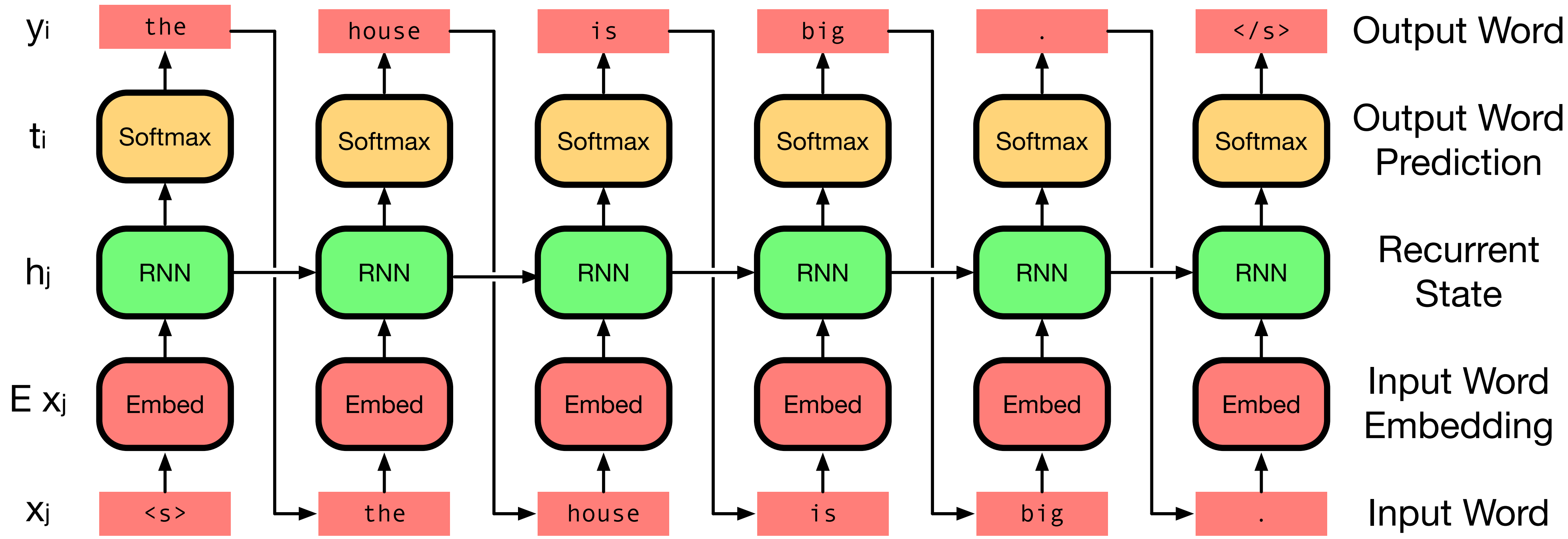


- Alignment of input words to output words

$\Rightarrow$ Solution: attention mechanism

# neural translation model with attention

# Input Encoding

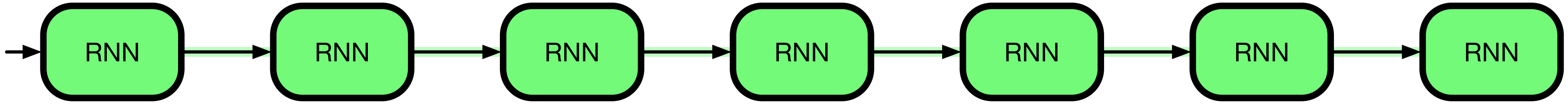



- Inspiration: recurrent neural network language model on the input side

# Hidden Language Model States

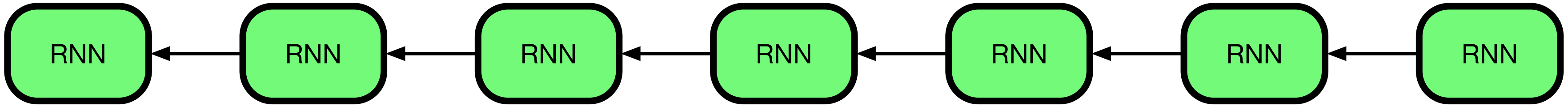


- This gives us the hidden states

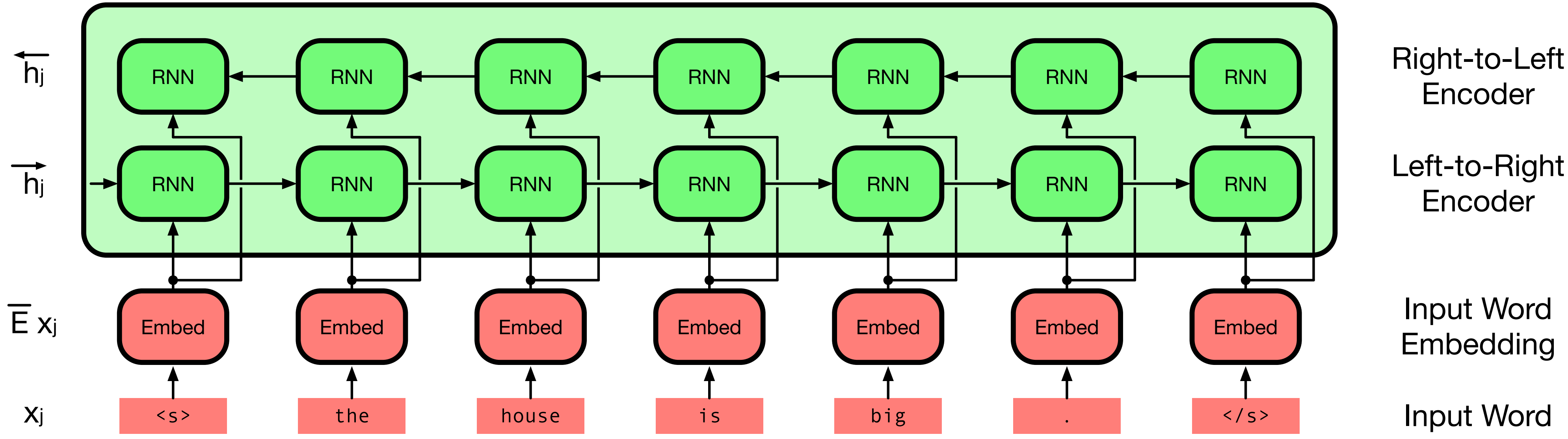



- These encode left context for each word

- Same process in reverse: right context for each word

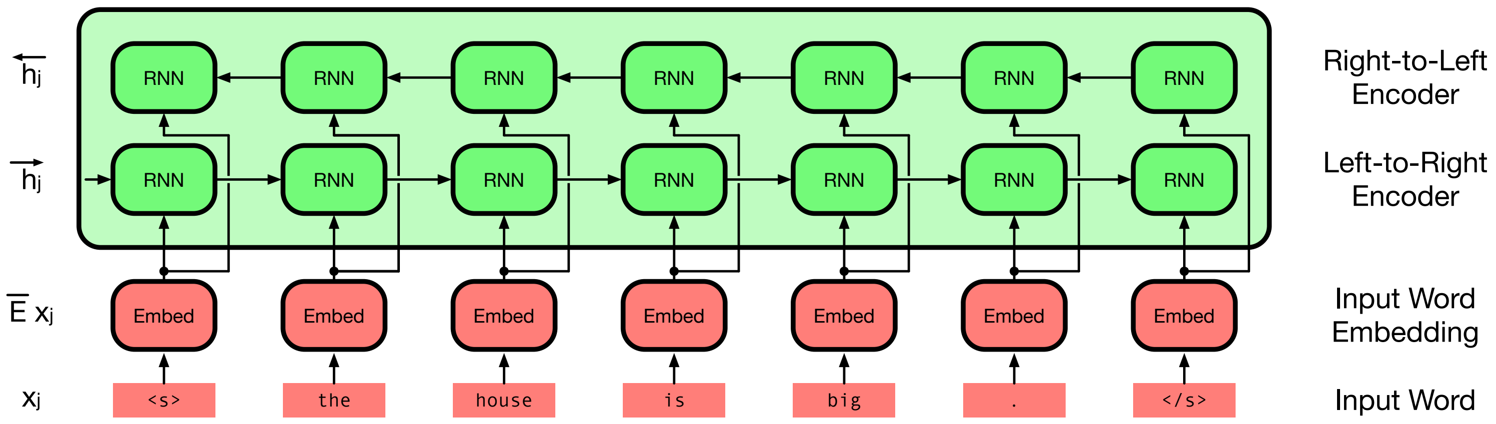


# Input Encoder




- Input encoder: concatenate bidrectional RNN states
- Each word representation includes full left and right sentence context

# Encoder: Math




- Input is sequence of words $x_j$, mapped into embedding space $\bar{E}\ x_j$

- Bidirectional recurrent neural networks

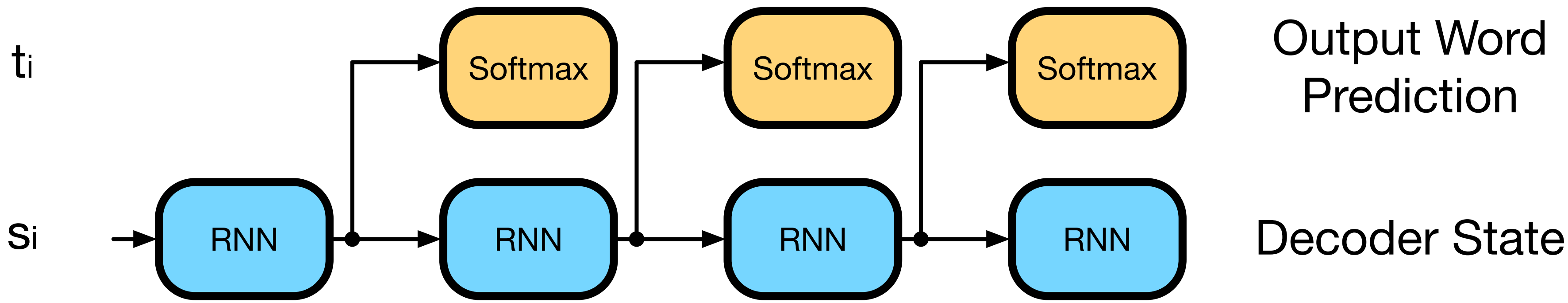
$$\overleftarrow{h_j} = f(\overleftarrow{h_{j+1}}, \bar{E}\ x_j)$$
$$\overrightarrow{h_j} = f(\overrightarrow{h_{j-1}}, \bar{E}\ x_j)$$

- Various choices for the function $f()$: feed-forward layer, GRU, LSTM, ...

# Decoder



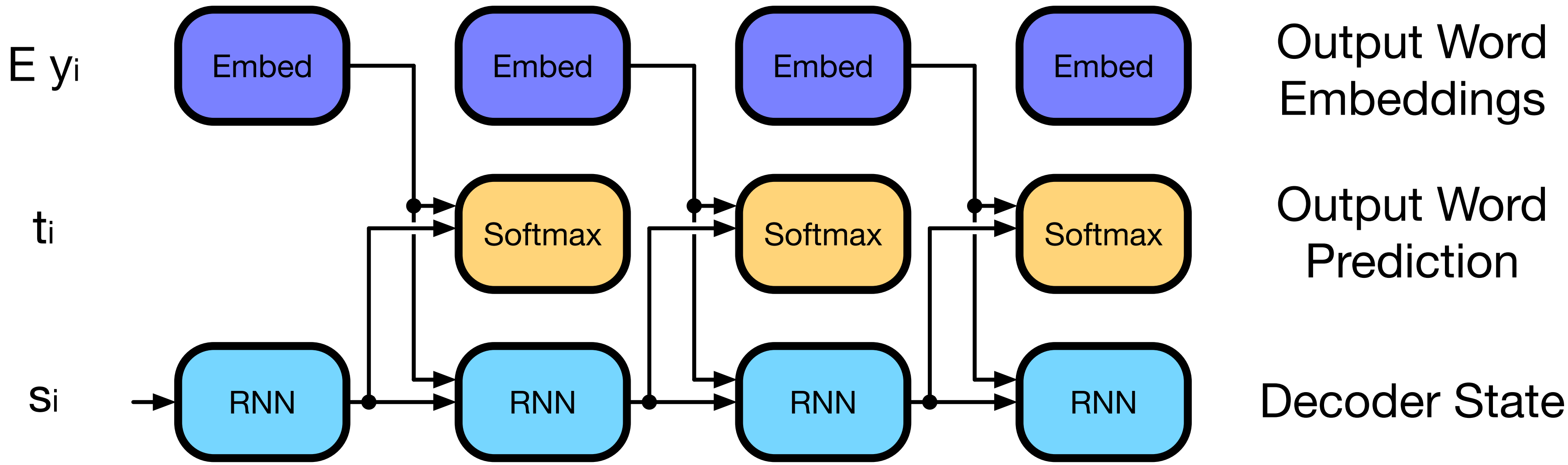
- We want to have a recurrent neural network predicting output words



# Decoder



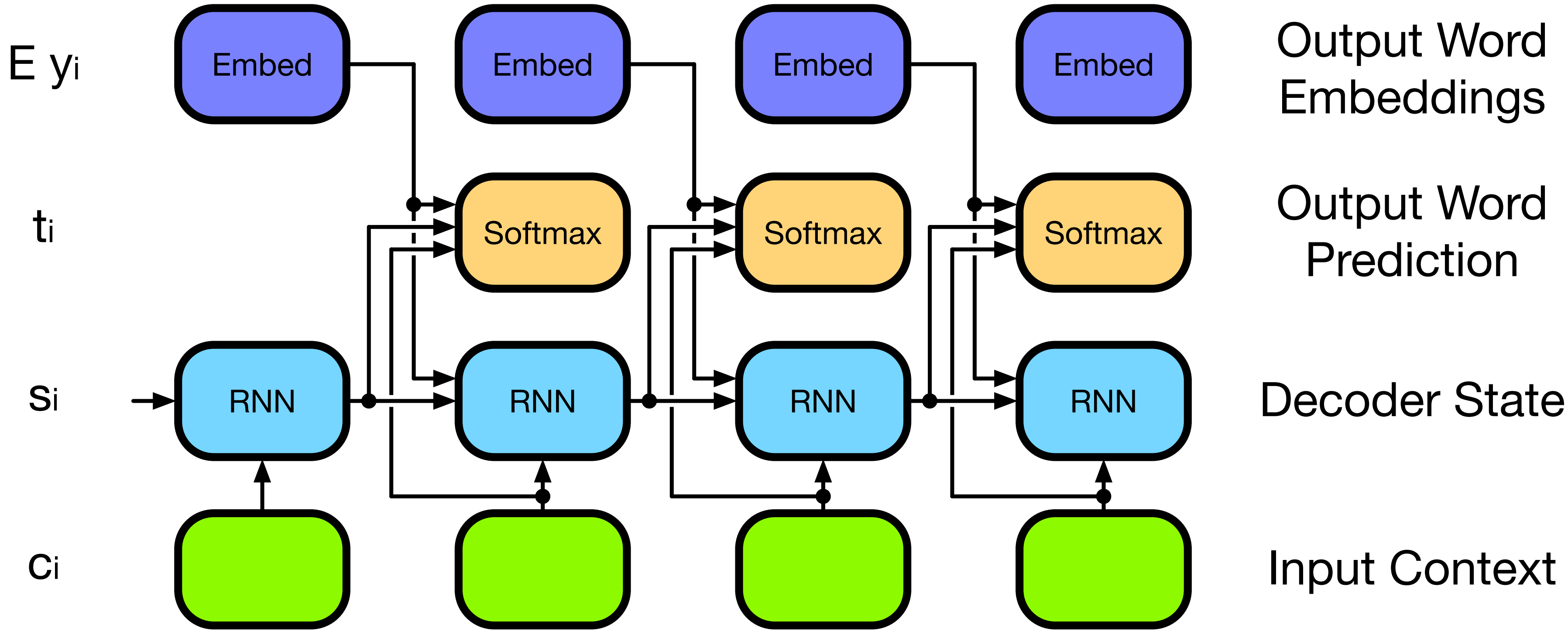
- We want to have a recurrent neural network predicting output words




- We feed decisions on output words back into the decoder state

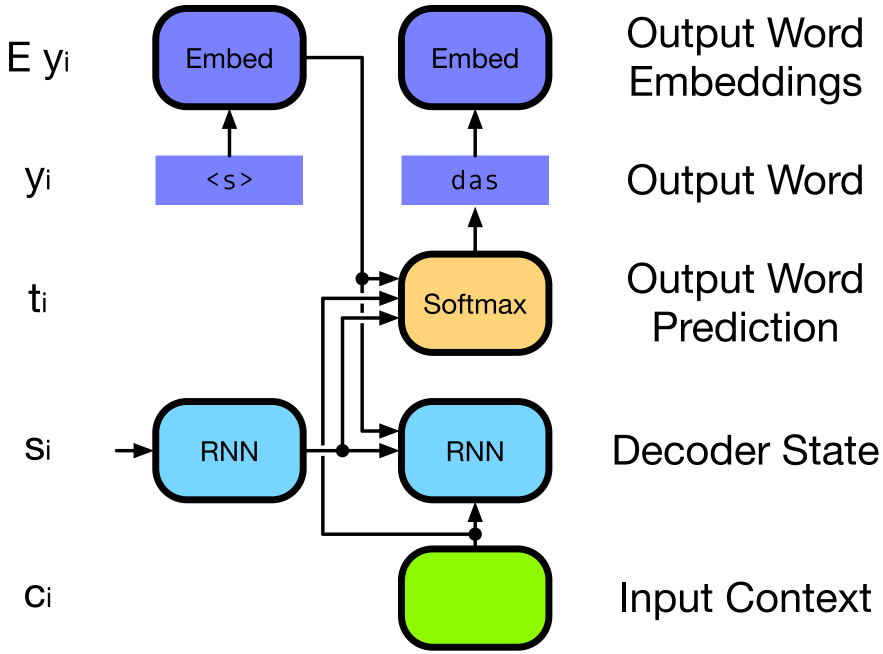
# Decoder

- We want to have a recurrent neural network predicting output words




- We feed decisions on output words back into the decoder state
- Decoder state is also informed by the input context

# More Detail




- Decoder is also recurrent neural network over sequence of hidden states $s_i$

$$s_i = f(s_{i-1},\ Ey_{-1}, c_i)$$

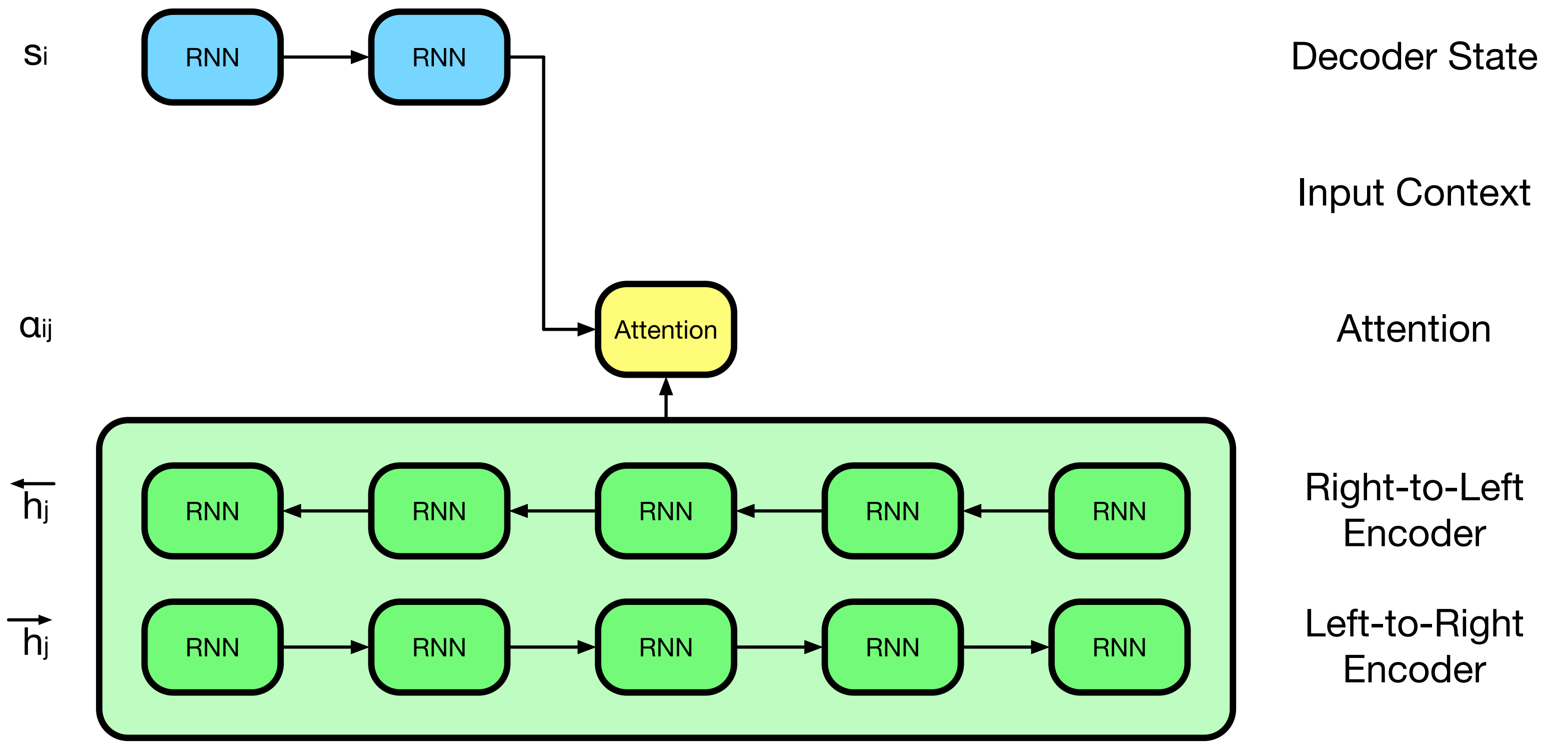
- Again, various choices for the function $f()$: feed-forward layer, GRU, LSTM, ...
- Output word $y_i$ is selected by computing a vector $t_i$ (same size as vocabulary)

$$t_i = W(Us_{i-1} + VEy_{i-1} + Cc_i)$$

then finding the highest value in vector $t_i$

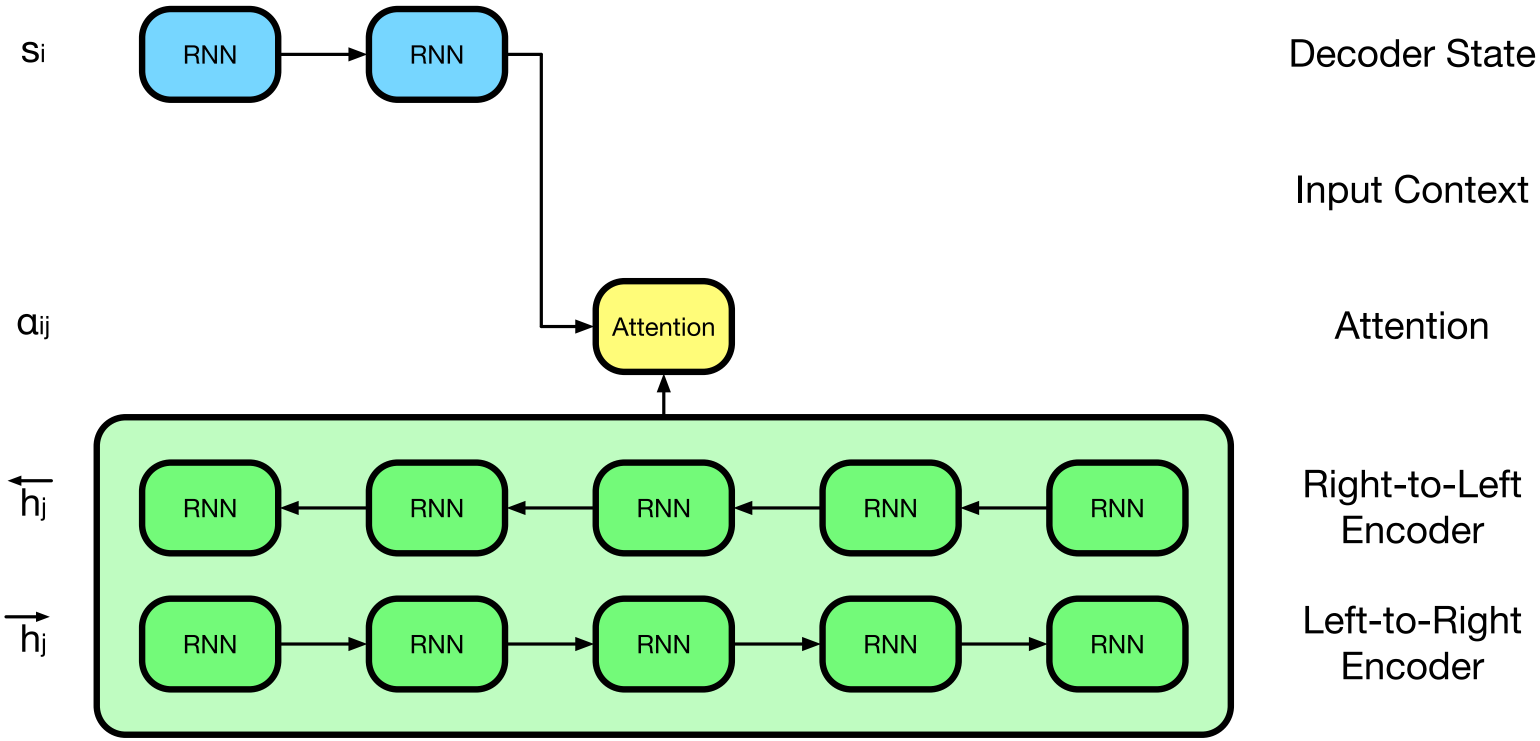
- If we normalize $t_i$, we can view it as a probability distribution over words
- $Ey_i$ is the embedding of the output word $y_i$

# Attention




- Given what we have generated so far (decoder hidden state)
- ... which words in the input should we pay attention to (encoder states)?

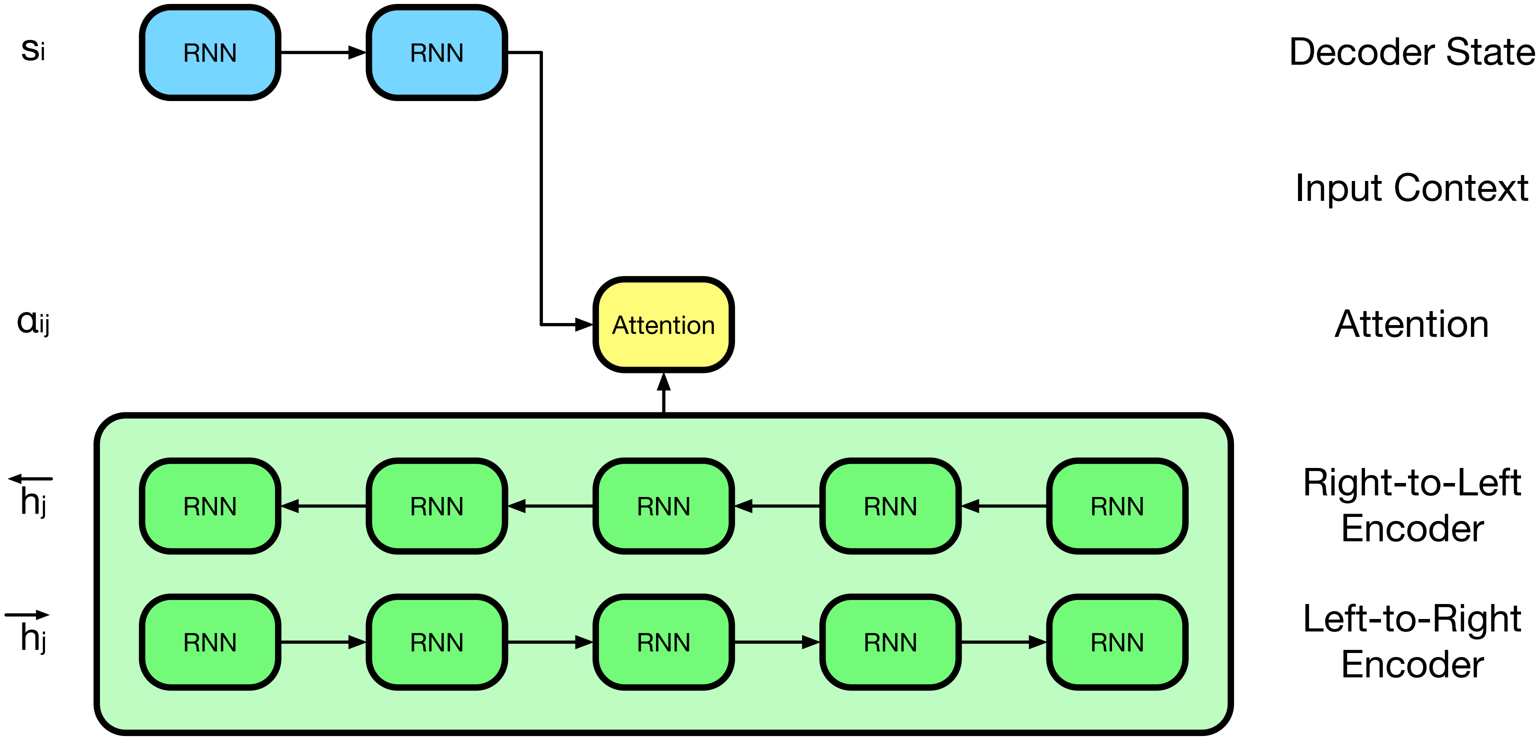
# Attention




- Given: – the previous hidden state of the decoder $s_{i-1}$
  – the representation of input words $h_j = (\overleftarrow{h_j}, \overrightarrow{h_j})$
- Predict an alignment probability $a(s_{i-1}, h_j)$ to each input word $j$ (modeled with with a feed-forward neural network layer)

# Attention

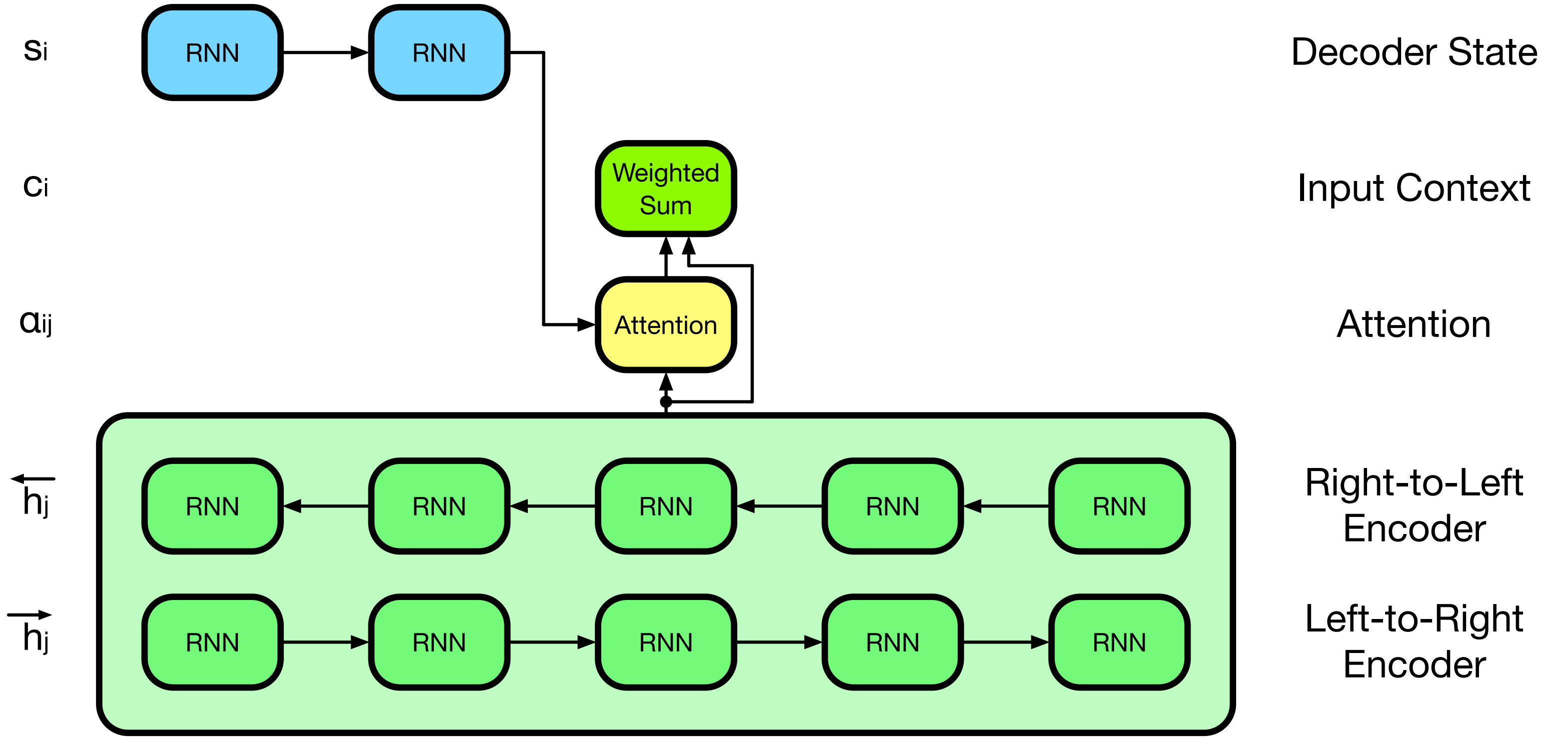





- Normalize attention (softmax)

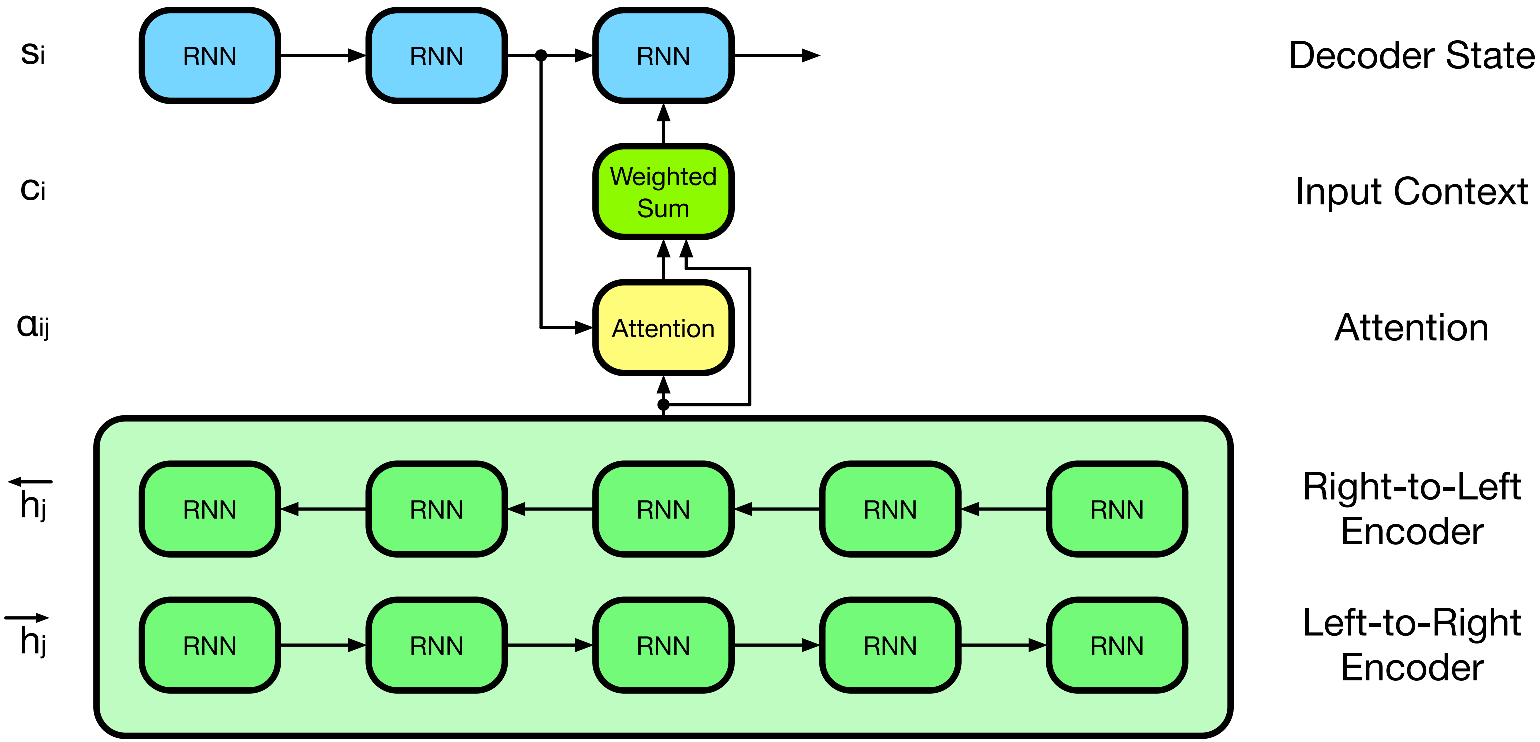
$$\alpha_{ij} = \frac{\exp(a(s_{i-1}, h_j))}{\sum_k \exp(a(s_{i-1}, h_k))}$$

# Attention




- Relevant input context: weigh input words according to attention: $c_i = \sum_j \alpha_{ij} h_j$

# Attention

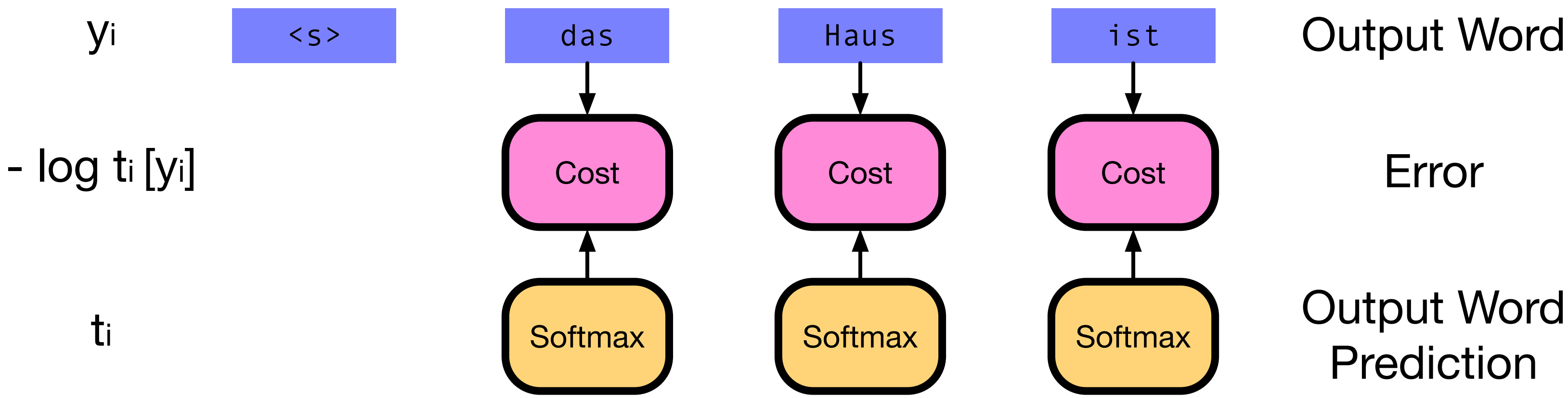



- Use context to predict next hidden state and output word

# training

# Comparing Prediction to Correct Word



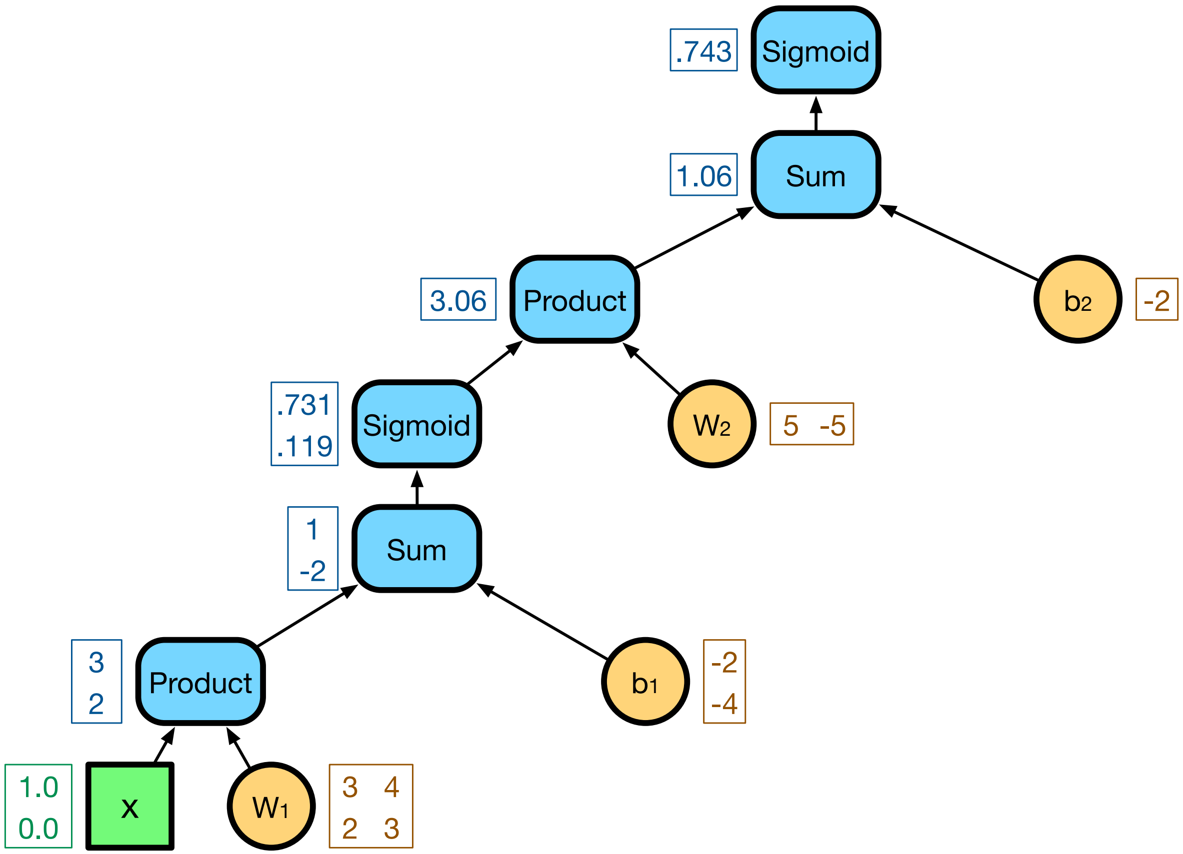



- Current model gives some probability $t_i[y_i]$ to correct word $y_i$
- We turn this into an error by computing cross-entropy: $-\log t_i[y_i]$

# Computation Graph

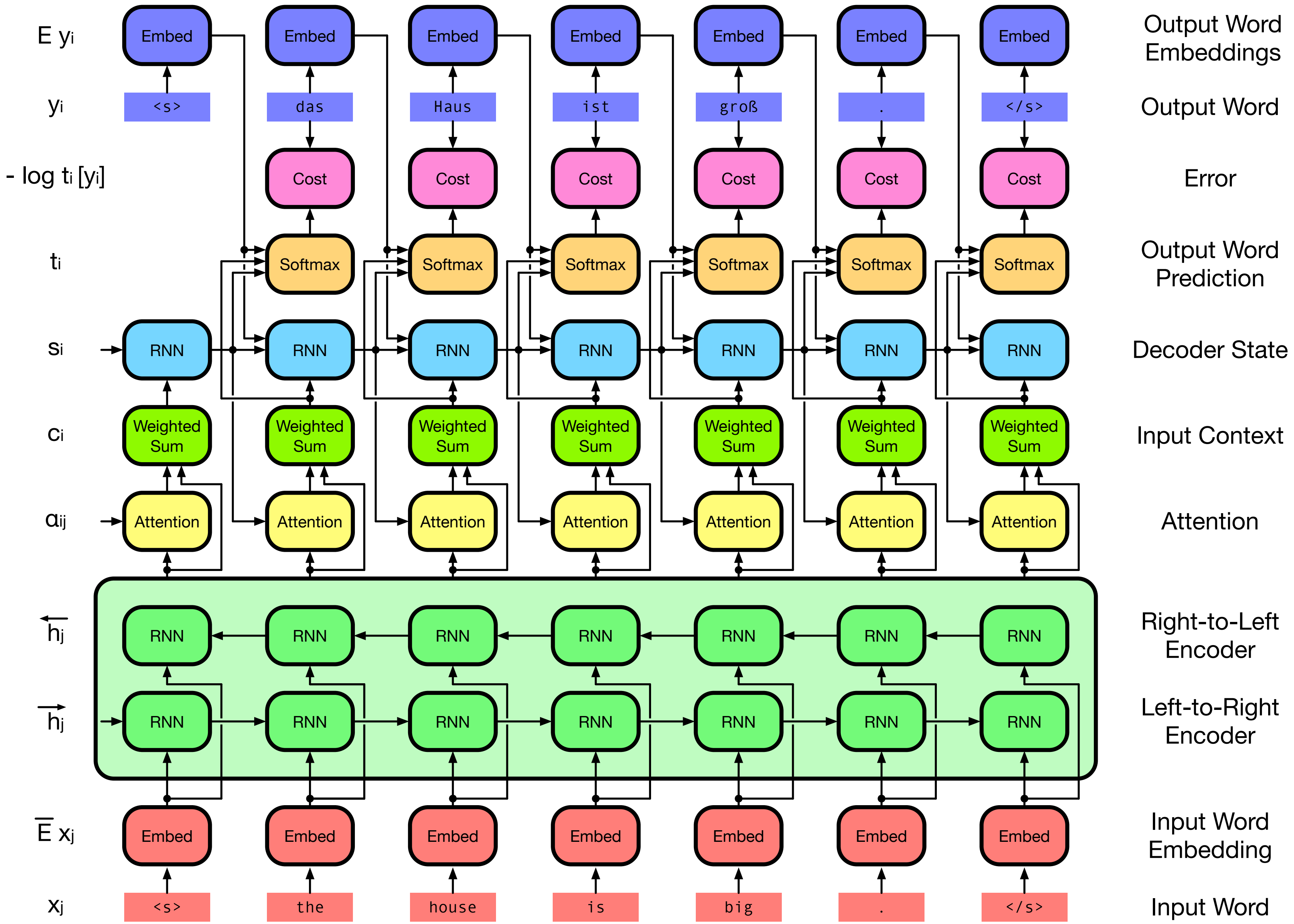


- Math behind neural machine translation defines a computation graph
- Forward and backward computation to compute gradients for model training



# Unrolled Computation Graph





# attention

# Attention



- Machine translation is a structured prediction task
  - output is not a single label
  - output structure needs to be built, word by word

- Relevant information for each word prediction varies

- Human translators pay attention to different parts of the input sentence when translating

$\Rightarrow$ Attention mechanism

# Computing Attention



- Attention mechanism in neural translation model (Bahdanau et al., 2015)
  - previous hidden state $s_{i-1}$
  - input word embedding $h_j$
  - trainable parameters $b$, $W_a$, $U_a$, $v_a$

$$a(s_{i-1}, h_j) = v_a^T \tanh(W_a s_{i-1} + U_a h_j + b)$$

- Other ways to compute attention
  - Dot product: $a(s_{i-1}, h_j) = s_{i-1}^T h_j$
  - Scaled dot product: $a(s_{i-1}, h_j) = \frac{1}{\sqrt{|h_j|}} s_{i-1}^T h_j$
  - General: $a(s_{i-1}, h_j) = s_{i-1}^T W_a h_j$
  - Local: $a(s_{i-1}) = W_a s_{i-1}$

# Attention of Luong et al. (2015)



- Luong et al. (2015) demonstrate good results with the dot product

$$a(s_{i-1}, h_j) = s_{i-1}^T h_j$$

- No trainable parameters

- Additional changes

- Currently more popular

# General View of Dot-Product Attention



- Three element

  **Query** : decoder state
  **Key** : encoder state
  **Value** : encoder state

- Intuition
  - given a query (the decoder state)
  - we check how well it matches keys in the database (the encoder states)
  - and then use the matching score to scale the retrieved value (also the encoder state)

- Computation

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

# Scaled Dot-Product Attention



- Refinement of query and key
- Scale it down to lower-dimensional vectors (e.g., 512 from 4096)
- Using a weight matrix for each: $QW^Q$, $KW^K$

# Multi-Head Attention

- Add redundancy
  - say, 16 attention weights
  - each based on its own parameters $W$
  - matrix $W$ also reduces the dimensionality

- Formally:

$$\begin{aligned} \text{head}_i &= \text{Attention}(QW_i^Q, KW_i^K, V) \\ \text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O \end{aligned}$$

- Multi-head attention is a form of ensembling

# Self Attention



- Finally, a very different take at attention

- Motivation so far: need for alignment between input words and output words

- Now: refine representation of input words in the encoder
  - representation of an input word mostly depends on itself
  - but also informed by the surrounding context
  - previously: recurrent neural networks (considers left or right context)
  - now: attention mechanism

- Self attention:

  Which of the surrounding words is most relevant to refine representation?

# Self Attention



- Formal definition (based on sequence of vectors $h_j$, packed into matrix $H$

$$\text{self-attention}(H) = \text{Attention}(HW_i^Q, HW_i^K, H)$$

- Association between every word representation $h_j$ any other context word $h_k$

- Resulting vector of normalized association values used to weigh context words

# transformer

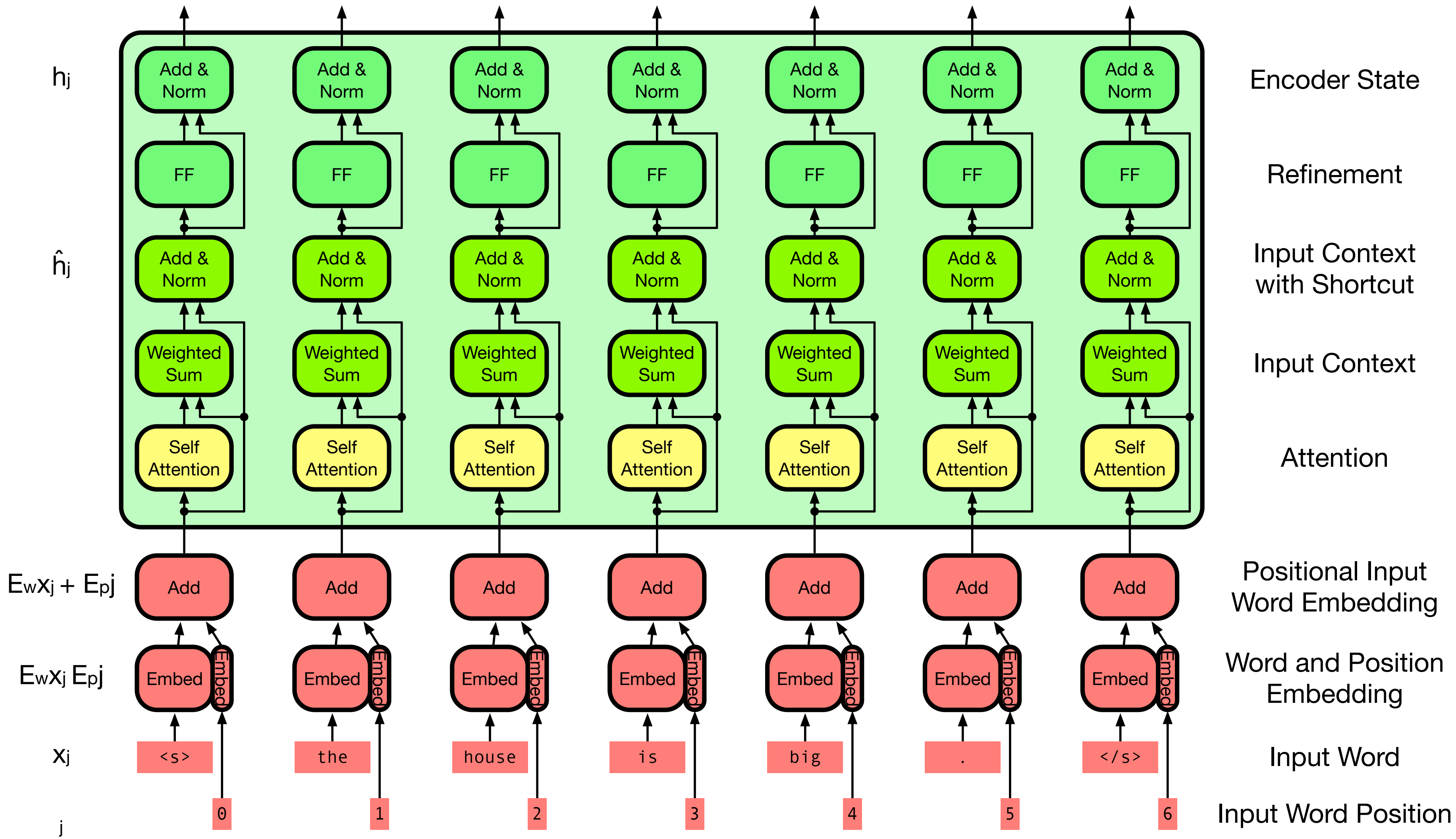
# Self Attention: Transformer



- Self-attention in encoder
  - refine word representation based on relevant context words
  - relevance determined by self attention
- Self-attention in decoder
  - refine output word predictions based on relevant previous output words
  - relevance determined by self attention
- Also regular attention to encoder states in decoder
- Currently most successful model

  (maybe only with self attention in decoder, but regular recurrent decoder)

# Encoder




Sequence of self-attention layers

# Self Attention Layer



- Given: input word representations $h_j$, packed into a matrix $H = (h_1, ..., h_j)$
- Self attention
$$\text{self-attention}(H) = \text{MultiHead}(H, H, H)$$
- Shortcut connection
$$\text{self-attention}(h_j) + h_j$$
- Layer normalization
$$\hat{h}_j = \text{layer-normalization}(\text{self-attention}(h_j) + h_j)$$
- Feed-forward step with ReLU activation function
$$\text{relu}(W\hat{h}_j + b)$$
- Again, shortcut connection and layer normalization
$$\text{layer-normalization}(\text{relu}(W\hat{h}_j + b) + \hat{h}_j)$$

# Stacked Self Attention Layers



- Stack several such layers (say, $D = 6$)

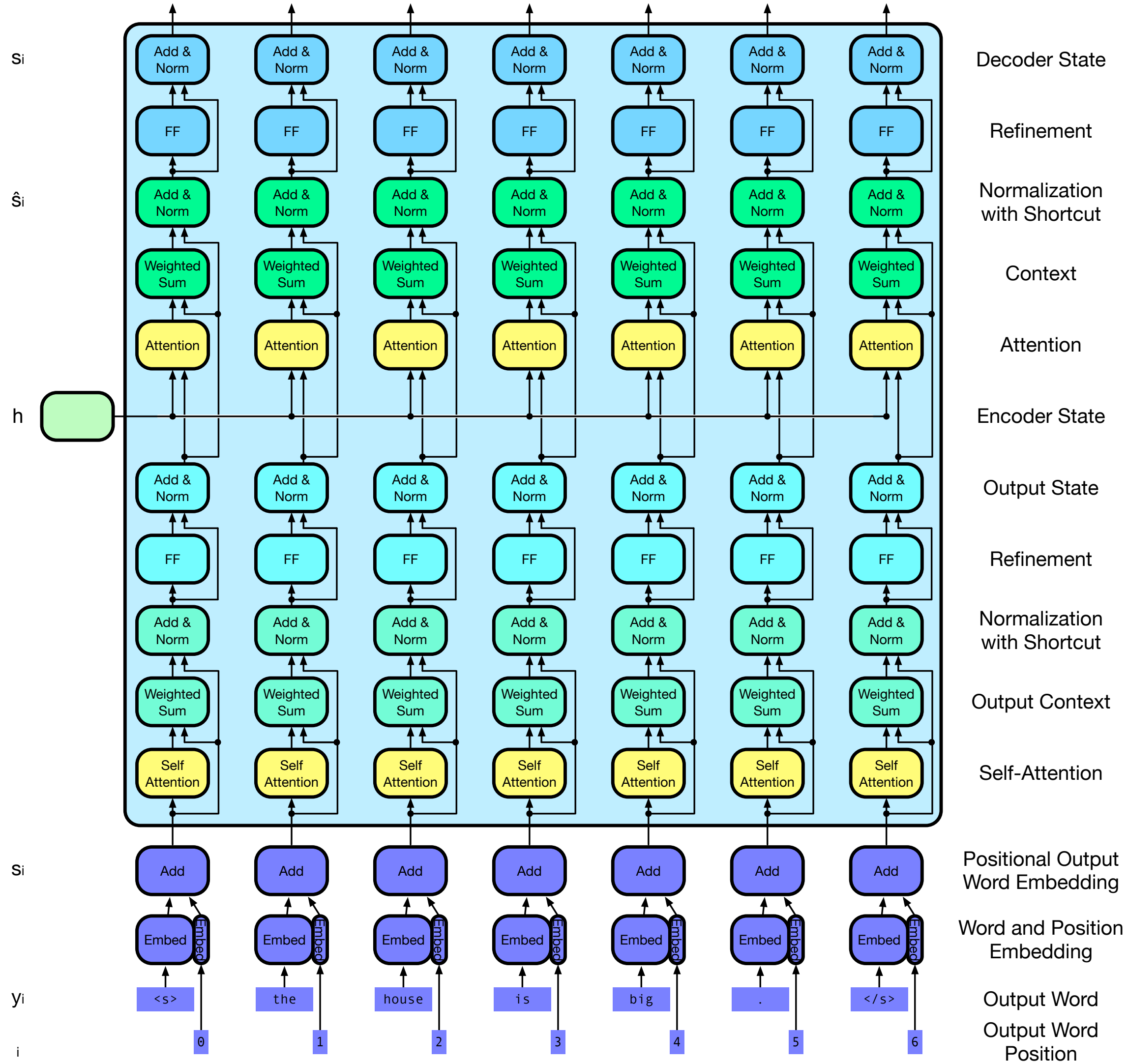
- Start with input word embedding

$$h_{0,j} = Ex_j$$

- Stacked layers

$$h_{d,j} = \text{self-attention-layer}(h_{d-1,j})$$

# Decoder




Decoder computes attention-based representations of the output in several layers, initialized with the embeddings of the previous output words

# Self-Attention in the Decoder



- Same idea as in the encoder
- Output words are initially encoded by word embeddings $s_i = Ey_i$.
- Self attention is computed over previous output words
  - association of a word $s_i$ is limited to words $s_k$ $(k \leq i)$
  - resulting representation $\tilde{s}_i$

$$\text{self-attention}(\tilde{S}) = \text{MultiHead}(\tilde{S}, \tilde{S}, \tilde{S})$$

# Attention in the Decoder

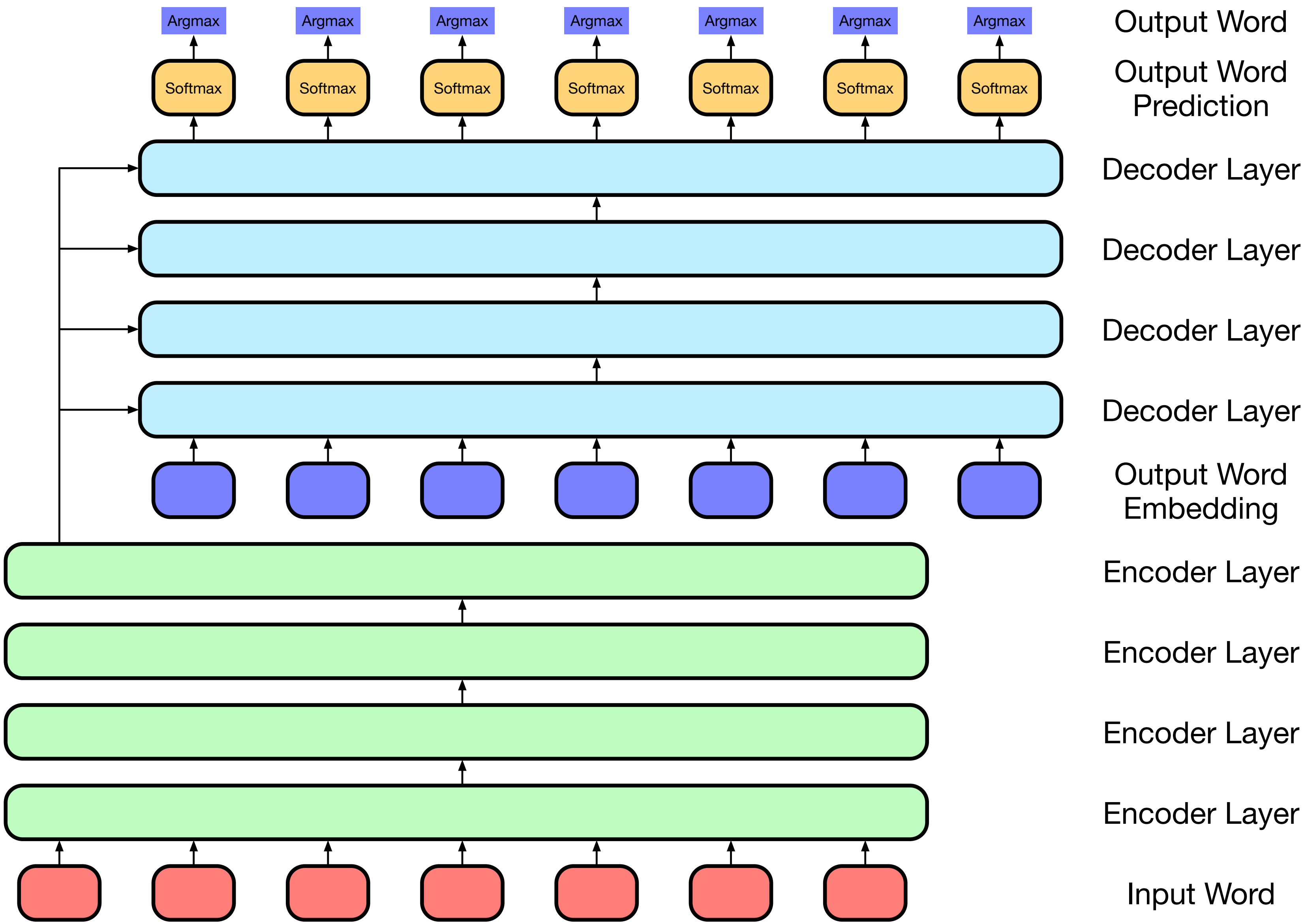
- Original intuition of attention mechanism: focus on relevant input words
- Compute attention between the decoder states $\tilde{S}$ and the final encoder states $H$

$$\text{attention}(\tilde{S}, H) = \text{MultiHead}(\tilde{S}, H, H)$$

- Note: attention mechanism formally mirrors self-attention

# Full Decoder





# Full Decoder



- Self-attention

$$\text{self-attention}(\tilde{S}) = \text{MultiHead}(\tilde{S}, \tilde{S}, \tilde{S})$$

  - shortcut connections
  - layer normalization
  - feed-forward layer

- Attention

$$\text{attention}(\tilde{S}, H) = \text{softmaxMultiHead}(\tilde{S}, H, H)$$

  - shortcut connections
  - layer normalization
  - feed-forward layer

- Multiple stacked layers

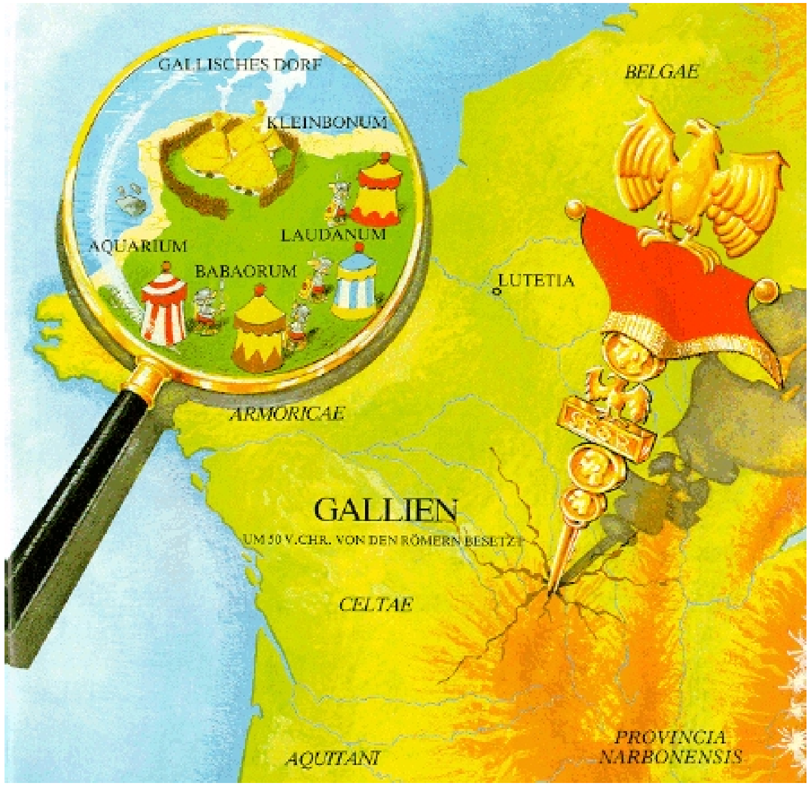
# machine translation

# and

# large language models

# The Large Language Model Wave

- Large language models have overtaken much of NLP
- So far, Machine Translation is still a hold-out: dedicated models are trained from scratch
- How long will this still be the case?



# LMs as Unsupervised Learners (2018)



**Language Models are Unsupervised Multitask Learners**

**Alec Radford** [*1] **Jeffrey Wu** [*1] **Rewon Child** [1] **David Luan** [1] **Dario Amodei** [**1] **Ilya Sutskever** [**1]

- Train language models on relatively clean text data (GPT-2)

- Convert any NLP problem into a text continuation problem
  - pre prompt engineering
  - goes into some detail of how each task is converted
  - impressive performance on many tasks

- Terrible at translation
  ... but all non-English text was removed from training corpus

# A Closer Look at PaLM for MT (2022)

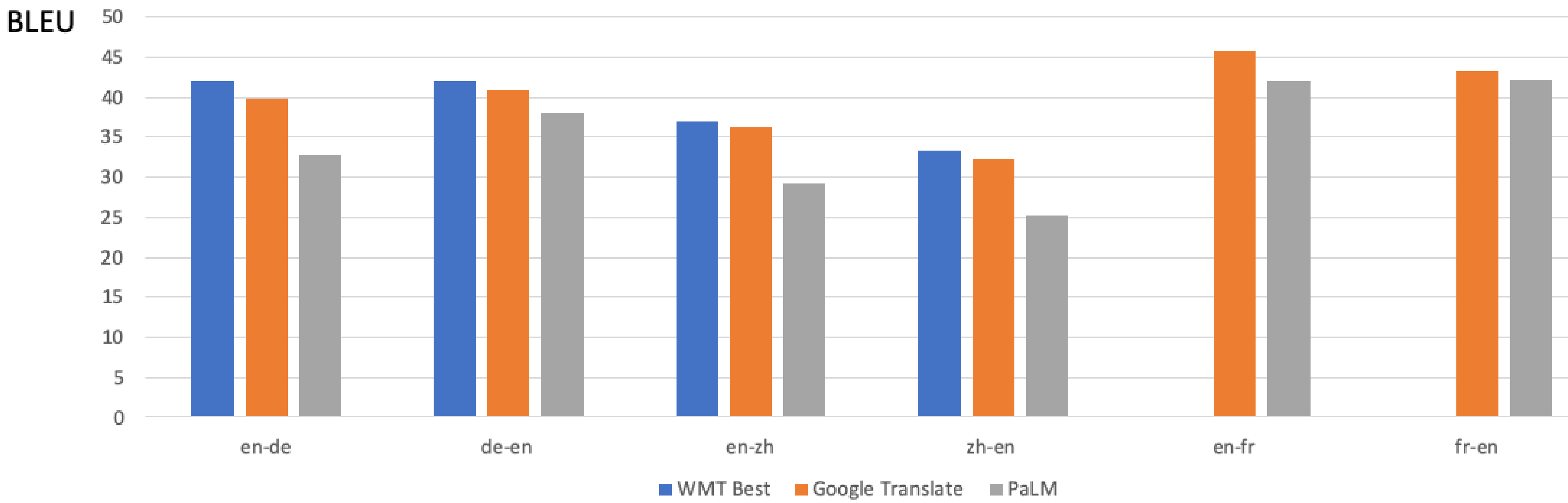
**Prompting PaLM for Translation: Assessing Strategies and Performance**

**David Vilar, Markus Freitag, Colin Cherry, Jiaming Luo, Viresh Ratnakar, George Foster**
Google Research
{vilar, freitag, colincherry, jmluo, vratnakar, fosterg}@google.com

- Exploration of examples used for prompting
- Evaluation with BLEU / BLEURT / MQM (human eval)
- WMT 2021 test set for de,zh→en, WMT 2014 for fr→en

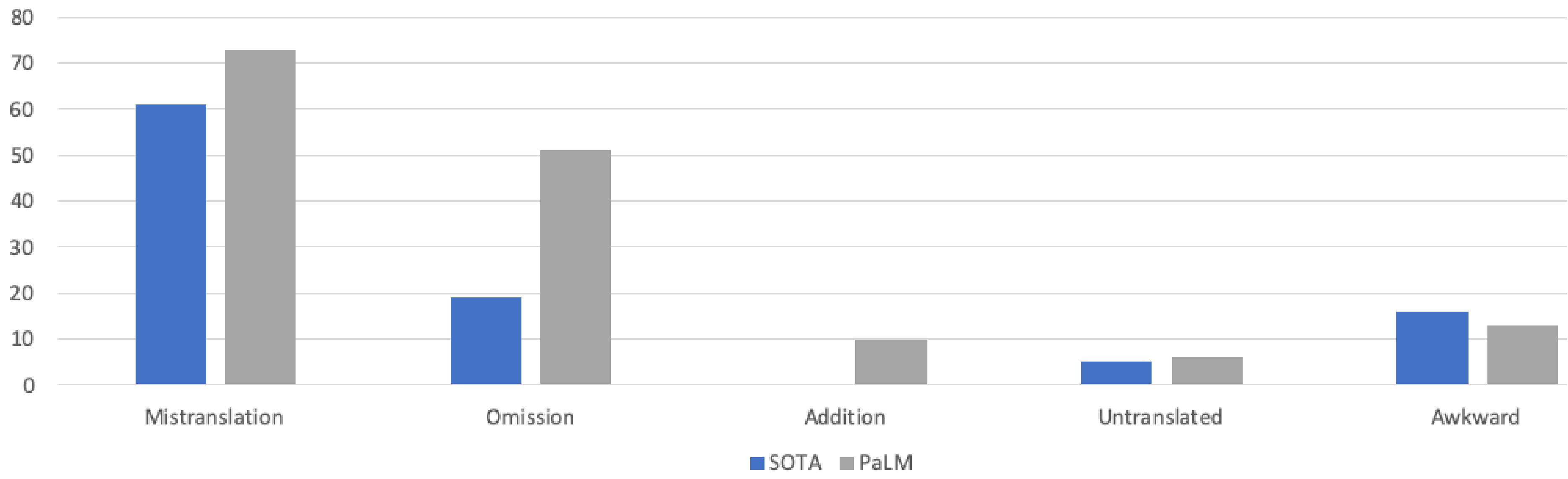
# Comparison to State of the Art





# Human Evaluation: MQM

- Language Models makes more adequacy errors, similar fluency
- German-English, MQM error categories (count of errors)



# Translation Data in Training? (2023)

***Searching for Needles in a Haystack:***
**On the Role of Incidental Bilingualism in PaLM's Translation Capability**

**Eleftheria Briakou** ebriakou@cs.umd.edu
**Colin Cherry** colincherry@google.com
**George Foster** fosterg@google.com

- PaLM is exposed to over 30 million translation pairs across at least 44 languages
  - 1.4% of training examples are bilingual
  - 0.34% have a translated sentence pair

- Most bilingual content is code-switched, about 20% contains translations

# Impact of Translation Data

- Sentence pairs can be extracted from bilingual samples
  - split sample into sentences
  - align English and French sentences with cross-lingual sentence embedding

  ⇒ parallel training corpus

- Training on mined parallel data (WMT fr-en): 38.1 BLEU
  Training on WMT training data: 42.0 BLEU

- Worse translation quality if bilingual content is removed from PaLM training

- Much worse translation quality with smaller (1B, 8B) PaLM models

# Convergence of LM and MT



- Both Language Models and Machine Translation are built with the same Transformer architecture

| TRANSLATION | LANGUAGE |
|---|---|
| *Der braune Hund is freundlich* | *The [MASK] dog is [MASK].* |
| *The brown dog is friendly* | The brown dog is friendly |

- This data can be mixed in any way

- Practical considerations: Large Language Models may be too big for use

# questions?