Neural Networks and Neural Language Models

- **Neural Networks**
- **Neural Language Models**

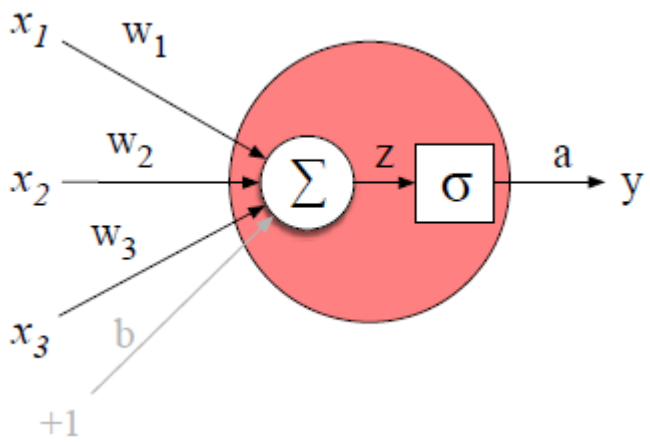
Neural Networks

Neural Networks

- A **neural network** is a network of small computing units, each of which takes a vector of input values and produces a single output value.
- **Neural networks** share much of the same mathematics as **logistic regression**.
  - Neural networks are a more powerful classifier than logistic regression,
  - A neural network can be shown to learn any function.
- In a **feedforward network**, the computation proceeds iteratively from one layer of units to the next.
  - The use of modern neural nets is often called deep learning, because modern networks are often deep (have many layers).

Neural Networks - Units

- The building block of a neural network is a single **computational unit**.
- A **unit** takes a set of real valued numbers as input, performs some computation on them, and produces an output.
- A neural unit is taking a weighted sum of its inputs, with one additional term in the sum called a bias term.
  - Given a **set of inputs $x_1 \ldots x_n$**, a unit has a set of corresponding **weights $w_1 \ldots w_n$** and a **bias b**, so the **weighted sum z** is computed, and **activation function f** is applied to the weighted sum z. The output of activation function is the **activation value** of the unit.

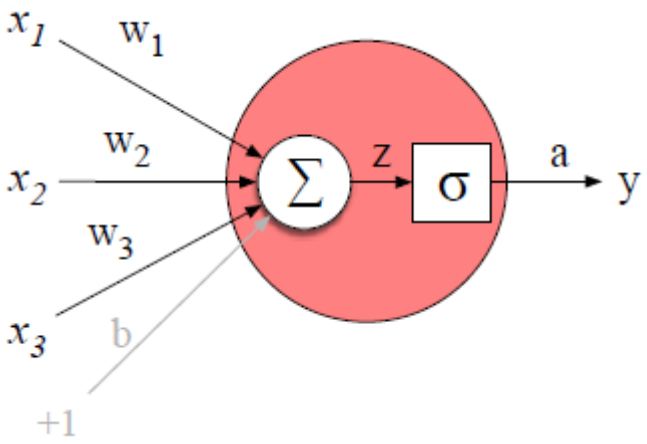


$$z = b + \sum_i w_i x_i \qquad z = \mathbf{w} \cdot \mathbf{x} + b$$

$$y = a = f(z) \qquad y = \sigma(z) = \frac{1}{1 + e^{-z}}$$

Neural Networks – Units

An Example



$$z = b + \sum_i w_i x_i \qquad z = \mathbf{w} \cdot \mathbf{x} + b$$

$$y = a = f(z) \qquad y = \sigma(z) = \frac{1}{1 + e^{-z}}$$

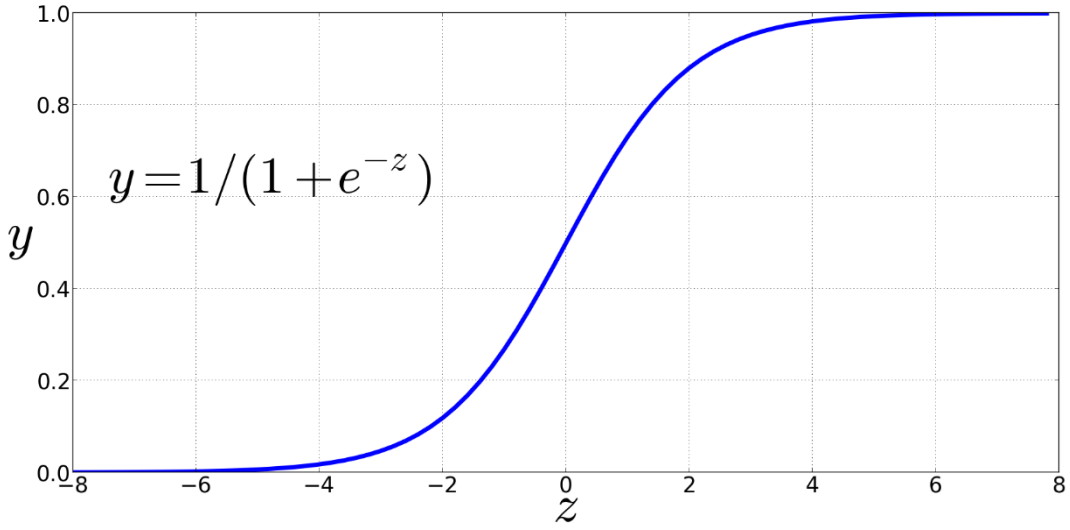
$$\mathbf{w} = [0.2, 0.3, 0.9] \qquad \mathbf{x} = [0.5, 0.6, 0.1]$$
$$b = 0.5$$

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} + b)}} = \frac{1}{1 + e^{-(.5*.2 + .6*.3 + .1*.9 + .5)}} = \frac{1}{1 + e^{-0.87}} = .70$$

Neural Networks – Units

Non-Linear Activation Functions - sigmoid

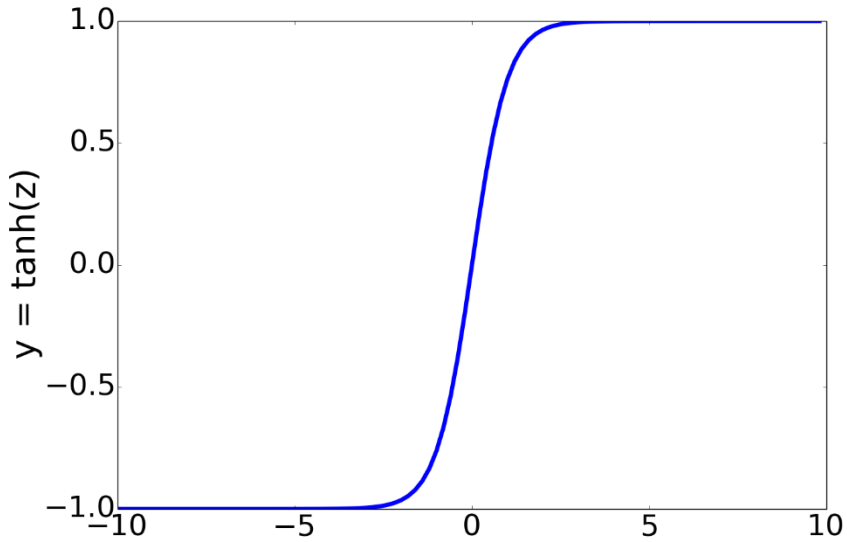
- The **sigmoid** has a number of advantages; it maps the output into the range [0,1], which is useful in squashing outliers toward 0 or 1.
- It's differentiable, which as we saw is handy for learning



Neural Networks – Units

Non-Linear Activation Functions - tanh

- **tanh** function that is very similar to sigmoid function but almost always better.
- **tanh** is a variant of the sigmoid that ranges from -1 to +1:
- It's differentiable, which as we saw is handy for learning

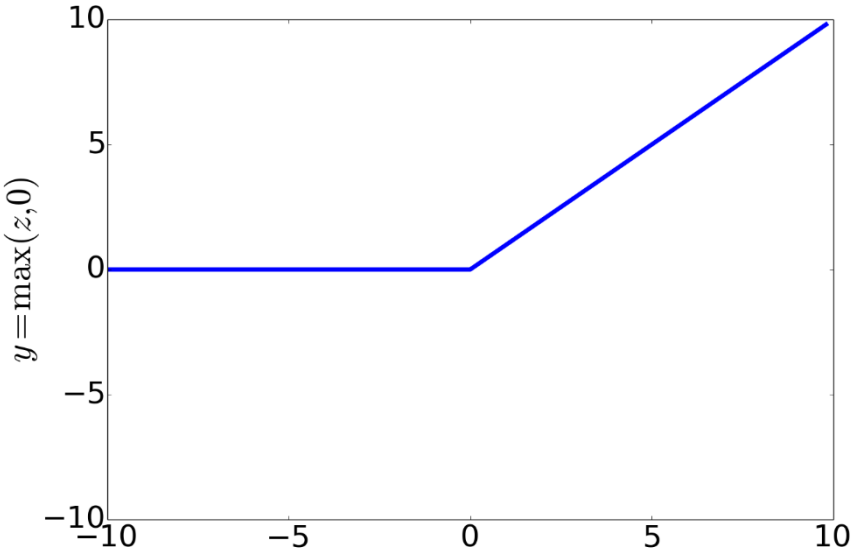


$$y = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

Neural Networks – Units

Non-Linear Activation Functions - ReLU

- The simplest activation function, and perhaps the most commonly used, is the **rectified linear unit**, called **ReLU**.
- It's just the same as z when z is positive, and 0 otherwise:

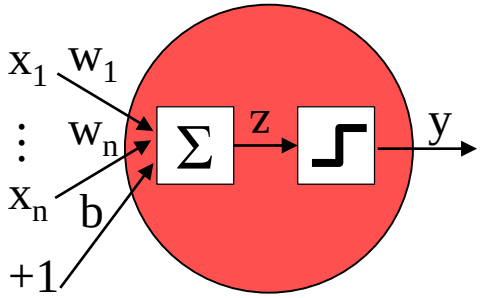


$$y = \text{ReLU}(z) = max(z, 0)$$

Neural Networks – Units

Perceptron

- The **perceptron**, is a very simple neural unit that has a binary output and it is not a non-linear activation function. The
- Output y of a **perceptron** is 0 or 1.



$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

Neural Networks – XOR Problem

- **The power of neural networks comes from combining units into larger networks.**
- A single neural unit cannot compute some very simple functions of its input.

- Can a perceptron represent each of the following boolean functions?

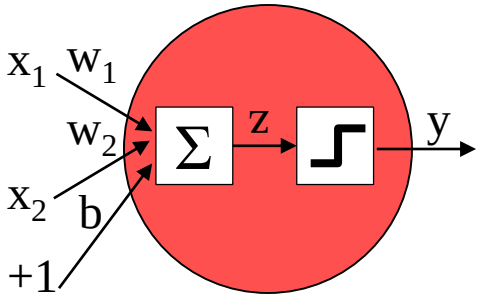
AND		
x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

OR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

Neural Networks – XOR Problem

- It's very easy to build a perceptron that can compute the logical AND and OR functions of its binary inputs

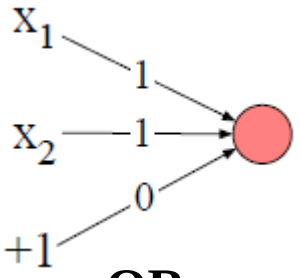


$$y = \begin{cases} 0, & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

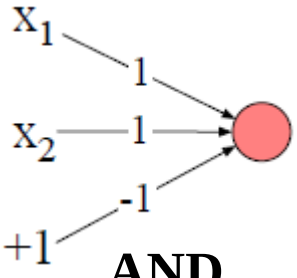
AND		
x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

OR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0



OR



AND

Neural Networks – XOR Problem

- Not possible to capture XOR with perceptrons.
- Perceptron equation given x_1 and x_2, is the equation of a line

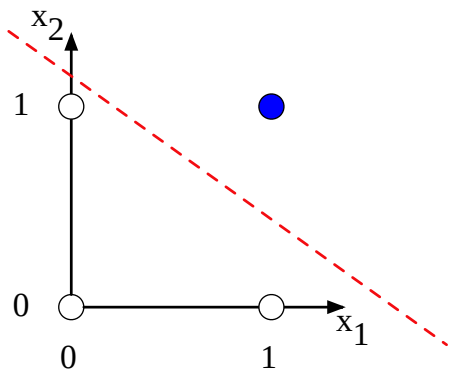
  $w_1x_1 + w_2x_2 + b = 0$

  in standard linear format: $x_2 = (-w_1/w_2)x_1 + (-b/w_2)$
- This line acts as a **decision boundary**
  - 0 if input is on one side of the line
  - 1 if on the other side of the line

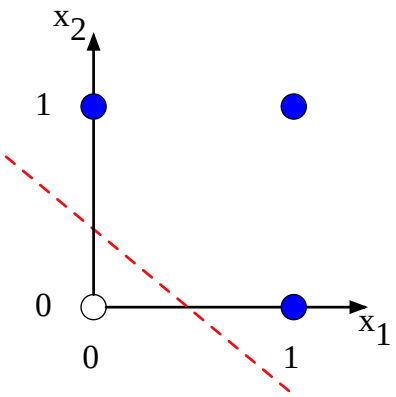
AND		
x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1

OR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

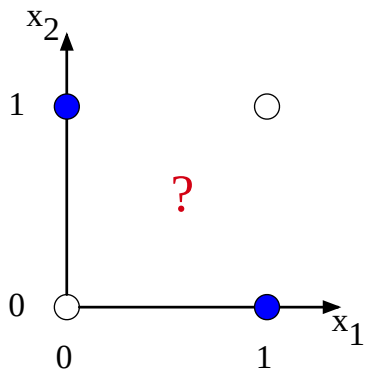
XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0



a) x_1 AND x_2



b) x_1 OR x_2



c) x_1 XOR x_2

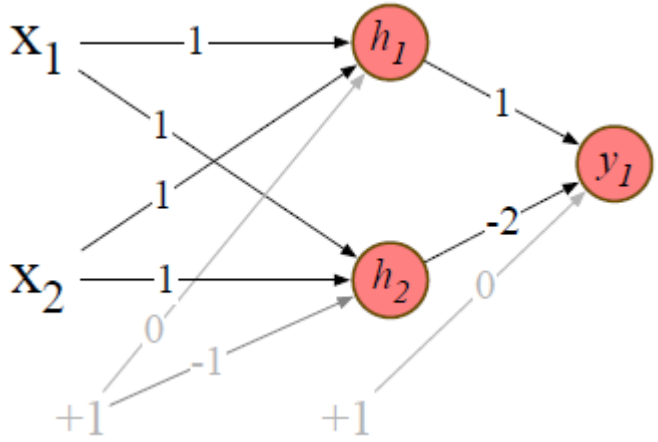
XOR is not a **linearly separable** function!

Neural Networks – XOR Problem

A Solution to XOR Problem

- XOR can't be calculated by a single perceptron
- XOR can be calculated by a layered network of units.

XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

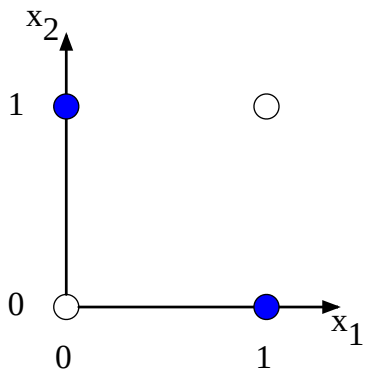


Neural Networks – XOR Problem

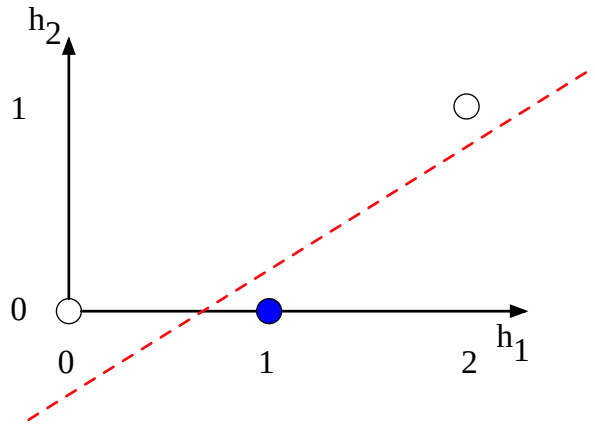
A Solution to XOR Problem

- XOR can be calculated by a layered network of units.

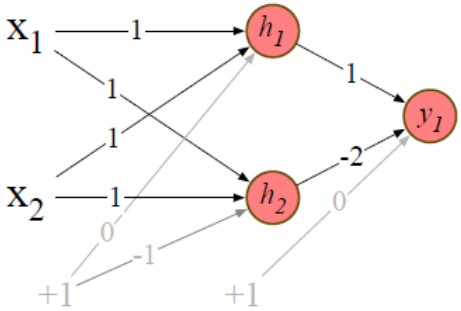
XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0



a) The original *x* space



b) The new (linearly separable) *h* space



- The hidden layer forming a new representation of the input.
- Notice that the input point [0, 1] has been collapsed with the input point [1, 0], making it possible to linearly separate the positive and negative cases of XOR
- In other words, we can view the hidden layer of the network as forming a useful representation for the input.

Neural Networks

- *Feedforward Neural Networks*

Feedforward Neural Networks

- A **feedforward neural network** is a multilayer network in which the units are connected with **no cycles**.
  - The outputs from units in each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers.
  - Later, we'll introduce networks with cycles, called **recurrent neural networks**.
- **Feedforward networks** have three kinds of nodes:
  - **input units, hidden units, and output units.**
- The core of the neural network is the hidden layer formed of **hidden units**.
  - Each unit is a neural unit.
- Each layer is **fully-connected**, meaning that each unit in each layer takes as input the outputs from all the units in the previous layer, and there is a link between every pair of units from two adjacent layers.
  - each hidden unit sums over all the input units.

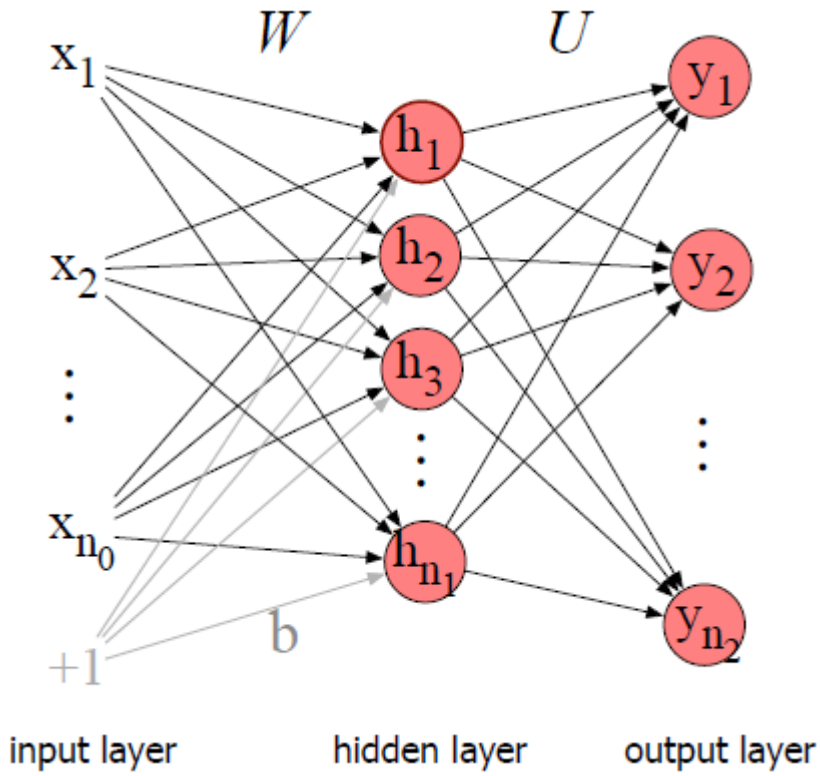
Feedforward Neural Networks

2-layer feedforward neural network

- A **simple 2-layer feedforward network**, with one hidden layer, one output layer, and one input layer.
  - input layer is not counted when enumerating layers
- A single hidden unit has as parameters an **input vector**, a **weight vector** and a **bias**.
- The parameters for the entire hidden layer:
  - **input vector x** $\mathbf{x_{n0}}$
  - **weight matrix W** $\mathbf{W_{n1 \times n0}}$
  - **bias vector b** $\mathbf{b_{n1}}$
- Each element W_{ji} of weight matrix W is weight of connection from input unit x_i to hidden unit h_j.
- Computation for hidden layer:

  h = activationfunc(Wx + b)

  $\mathbf{h_i = activationfunc(\sum_{i=1}^{n_0} W_{ji}\, xi + bj)}$



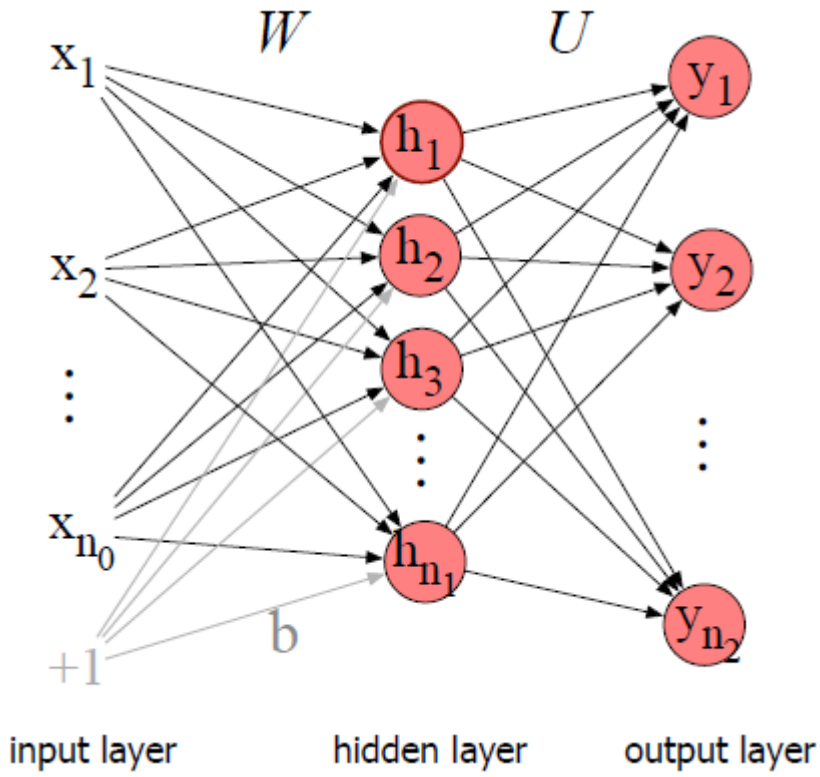
Feedforward Neural Networks

2-layer feedforward neural network

- A **single output unit** has:
  - Hidden layer outputs as an input vector
  - A weight vector
  - Some networks don't include a bias vector b in the output layer, so we'll not show it here.
- The parameters for the **entire output layer**:
  - **hidden unit outputs h** as input vector $\mathbf{h_{n1}}$
  - **weight matrix U** $\mathbf{U_{n2 \times n1}}$
- Computation for output layer:

  z = Uh

  y = activationfunc(z)



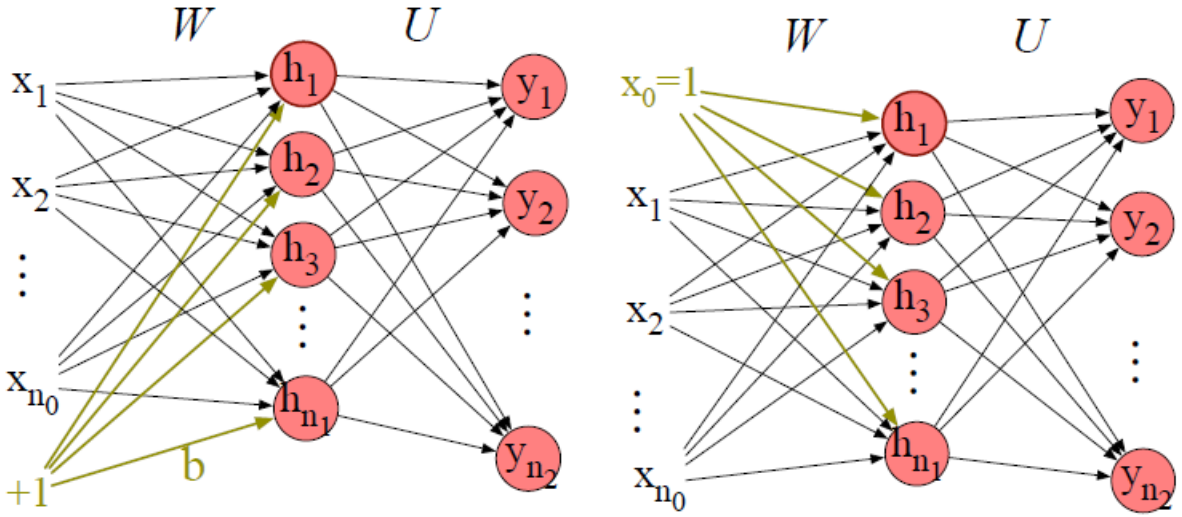
Feedforward Neural Networks

2-layer feedforward neural network - Replacing bias unit

- In order to simplfy our notation, we will see bias as a part of input.
  - In the input layer, will have a dummy node $x_0 = 1$
  - This dummy node still has an associated weight which represents the bias value b.
- If hidden units are sigmoid units:

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \quad \rightarrow \quad \mathbf{h} = \sigma(\mathbf{W}\mathbf{x})$$

$$\mathbf{h}_j = \sigma\left(\sum_{i=1}^{n_0} \mathbf{W}_{ji}\,\mathbf{x}_i + \mathbf{b}_j\right) \quad \rightarrow \quad \mathbf{h}_j = \sigma\left(\sum_{i=0}^{n_0} \mathbf{W}_{ji}\,\mathbf{x}_i\right)$$



Feedforward Neural Networks

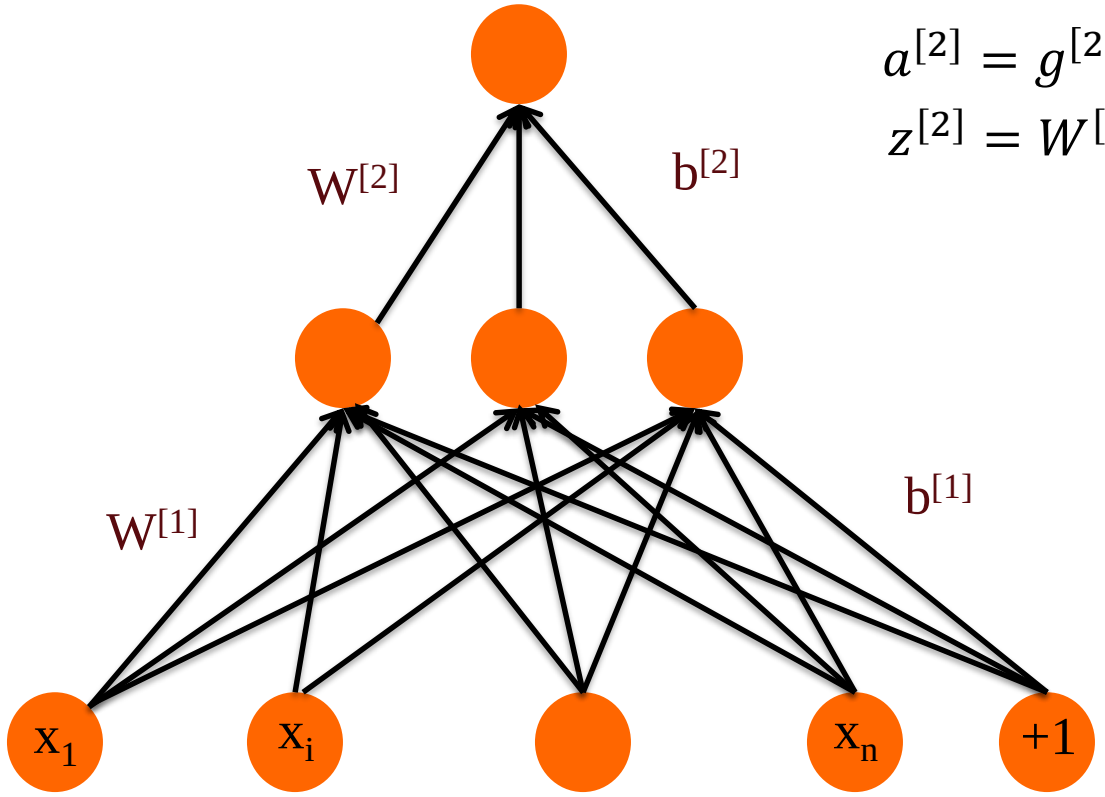
feedforward neural network – Multi-Layer Notation

- ***multi-layer notation*** to make easier to talk about deeper networks of depth more than 2.
  - superscripts in square brackets to mean layer numbers, starting at 0 for the input layer.
  - $W^{[1]}$ means the weight matrix for the (first) hidden layer, and $b^{[1]}$ means the bias vector for the (first) hidden layer.
  - $g(\cdot)$ to stand for the activation function, which will tend to be ReLU or tanh for intermediate layers and softmax for output layers.
  - $a^{[i]}$ to mean the output from layer i, and $z^{[i]}$ to mean the combination of weights and biases. The inputs x refer to more generally as $a^{[0]}$

Feedforward Neural Networks

feedforward neural network – Multi-Layer Notation

- ***multi-layer notation***



$$y = a^{[2]}$$

$$a^{[2]} = g^{[2]}(z^{[2]})$$ sigmoid or softmax

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[1]} = g^{[1]}(z^{[1]})$$ ReLU

$$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$$

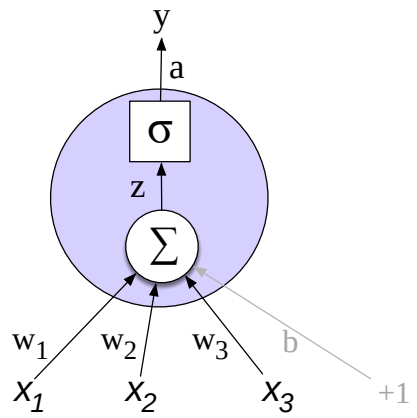
$a^{[0]}$

Feedforward Neural Networks

feedforward neural network – Multi-Layer Notation

- ***multi-layer notation***

$$
\begin{aligned}
z^{[1]} &= W^{[1]} a^{[0]} + b^{[1]} \\
a^{[1]} &= g^{[1]}(z^{[1]}) \\
z^{[2]} &= W^{[2]} a^{[1]} + b^{[2]} \\
a^{[2]} &= g^{[2]}(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$



for i **in** 1..n

$$
\begin{aligned}
z^{[i]} &= W^{[i]} a^{[i-1]} + b^{[i]} \\
a^{[i]} &= g^{[i]}(z^{[i]}) \\
\hat{y} &= a^{[n]}
\end{aligned}
$$

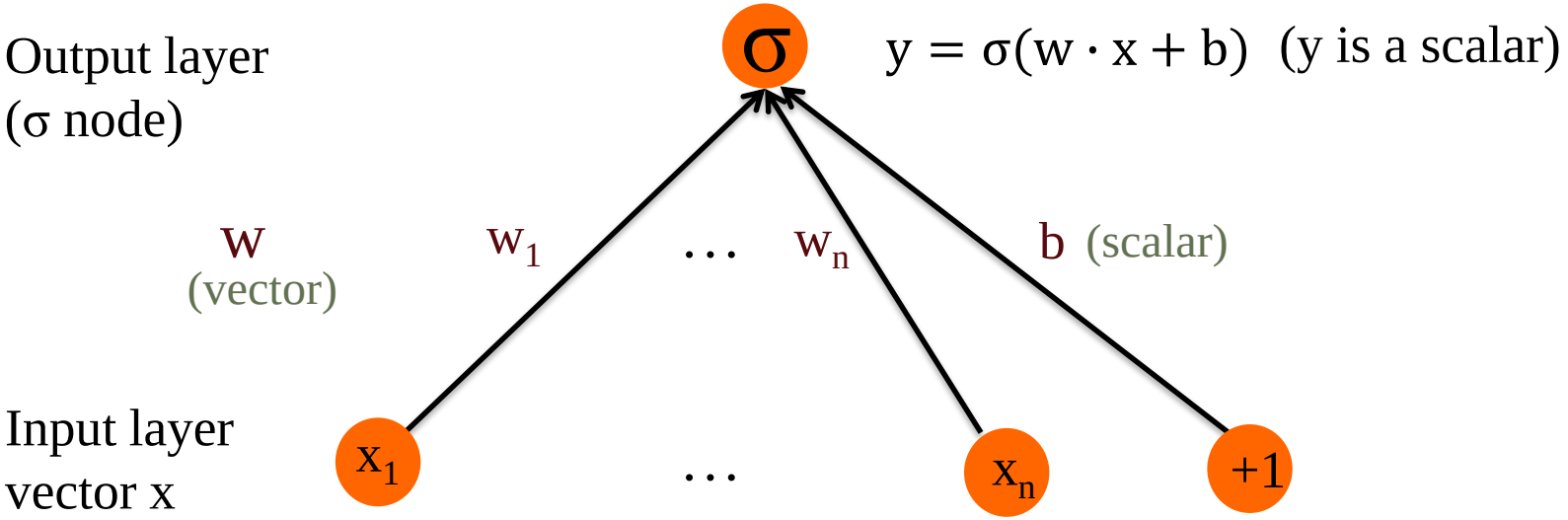
Neural Networks

- *Feedforward networks for NLP: Classification*

Feedforward Networks for NLP: Classification

Binary Logistic Regression as a 1-layer Network

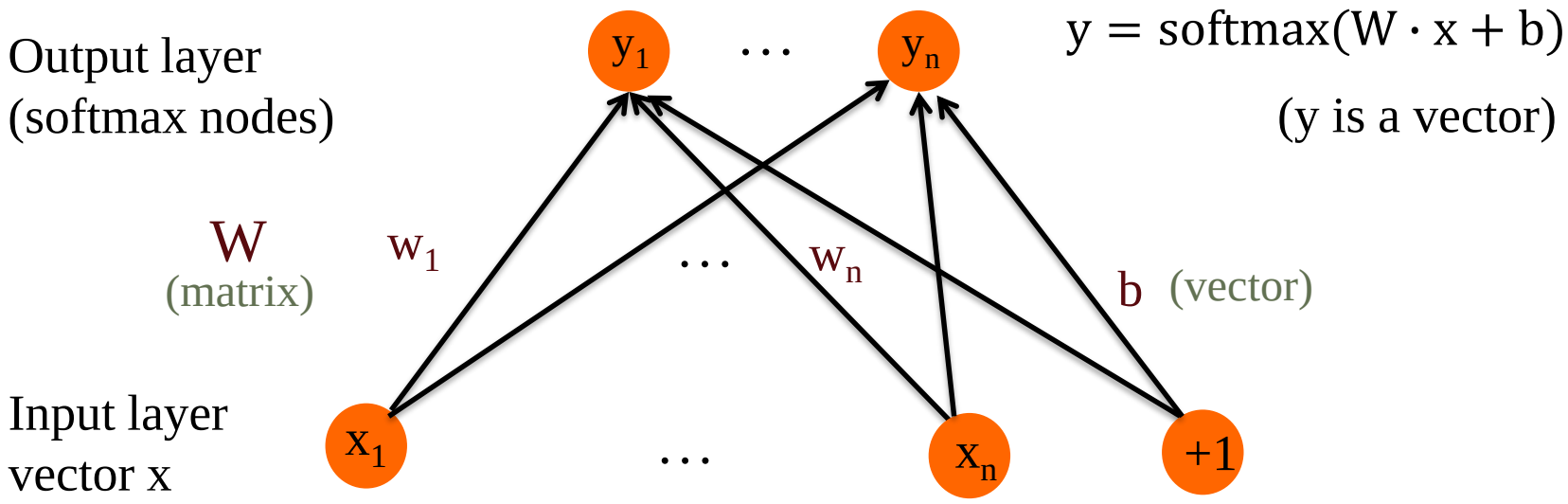
- Let's go back to logistic regression.
  - We can think of binary logistic regression as a 1-layer network (we don't count the input layer in counting layers!).
  - ***input layer*** (we could call layer 0) consisting of a **input vector x**, a **weight vector w**, and a **scalar bias b.**
  - **1st layer (the output layer)** computes a scalar **y** as **σ(w.x+b)** (sigmoid of weighted sum).



Feedforward Networks for NLP: Classification

Multinominal Logistic Regression as a 1-layer Network

- Multinomial Logistic Regression, where instead of a single sigmoid at the output, we have a softmax to turn the output values into probabilities.



Feedforward Networks for NLP: Classification

2-layer Network

- We will look at again sentiment analysis as a classification task.
- The **input element $\mathbf{x_i}$** could be scalar features: x_1=count(words$\in$doc), x_2=count(positive lexicon words$\in$doc), x_3=1 if "no" 2 doc, and so on.
- The **output layer $\hat{\mathbf{y}}$** could have
  - 2 nodes (one each for positive and negative) ➔ **sigmoid function**
  - 3 nodes (positive, negative, neutral), in which case $\hat{\mathbf{y}}_1$ would be the estimated probability of positive sentiment, $\hat{\mathbf{y}}_2$ the probability of negative and $\hat{\mathbf{y}}_3$ the probability of neutral.

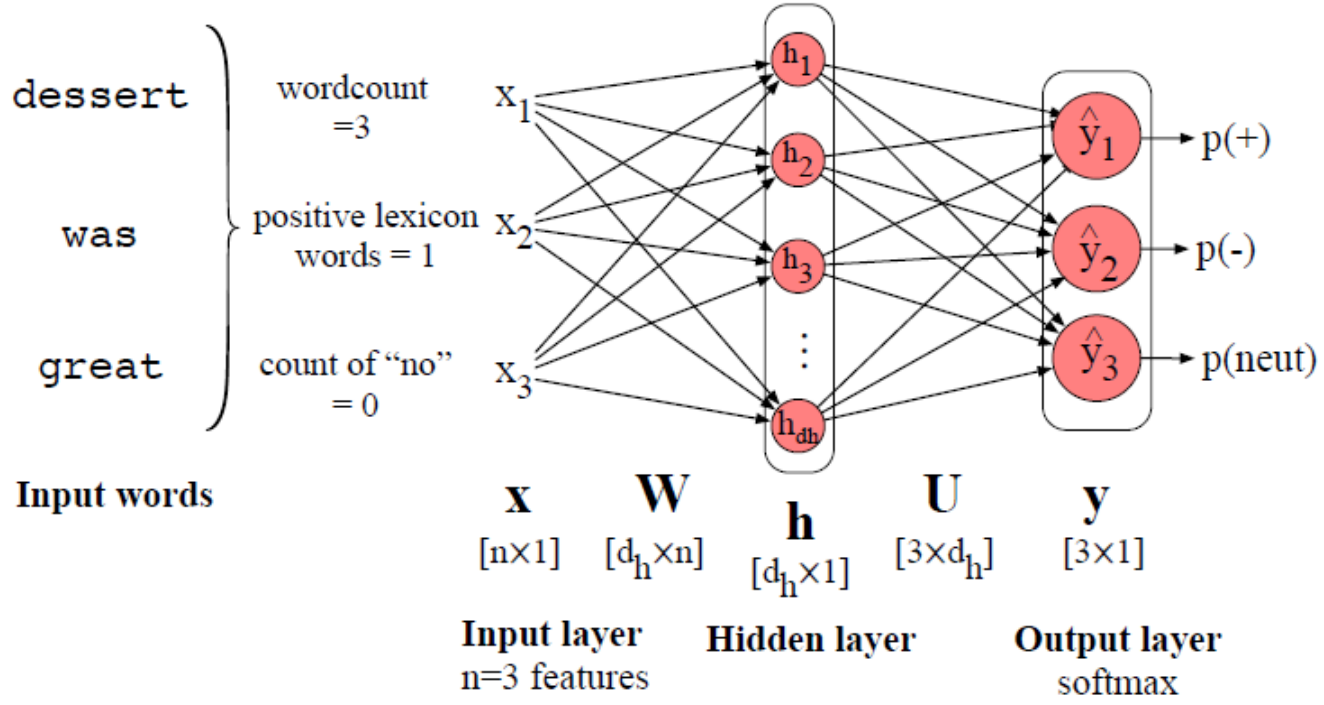
    ➔ **softmax function**
- In **hidden layer**, **sigmoid** (or tanh, ReLU,..) function can be used.

Feedforward Networks for NLP: Classification

2-layer Network for sentiment analysis

- Feedforward network sentiment analysis using hand-built features of the input text.

$$\mathbf{x} = [\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_N] \quad \text{(each } \mathbf{x}_i \text{ is a hand-designed feature)}$$
$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$
$$\mathbf{z} = \mathbf{U}\mathbf{h}$$
$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$$



Feedforward Networks for NLP: Classification

2-layer Network for sentiment analysis

- Instead of using *hand-built human-engineered features as the input* to our classifier, most applications of neural networks for NLP do something different.
- They use deep learning's ability to learn features from the **input data** by representing words as **embeddings** (like the word2vec or GloVe embeddings).
- There are various ways to represent an input for classification.
- One simple baseline **pooling** is to apply some sort of **pooling function** to the embeddings of all the words in the input.
  - For a text with n input words/tokens $w_1,\ldots,w_n$, we can turn the n embeddings $e(w_1),\ldots,e(w_n)$ (each of dimensionality d) into a single embedding also of dimensionality d by just summing the embeddings, or by taking their mean.

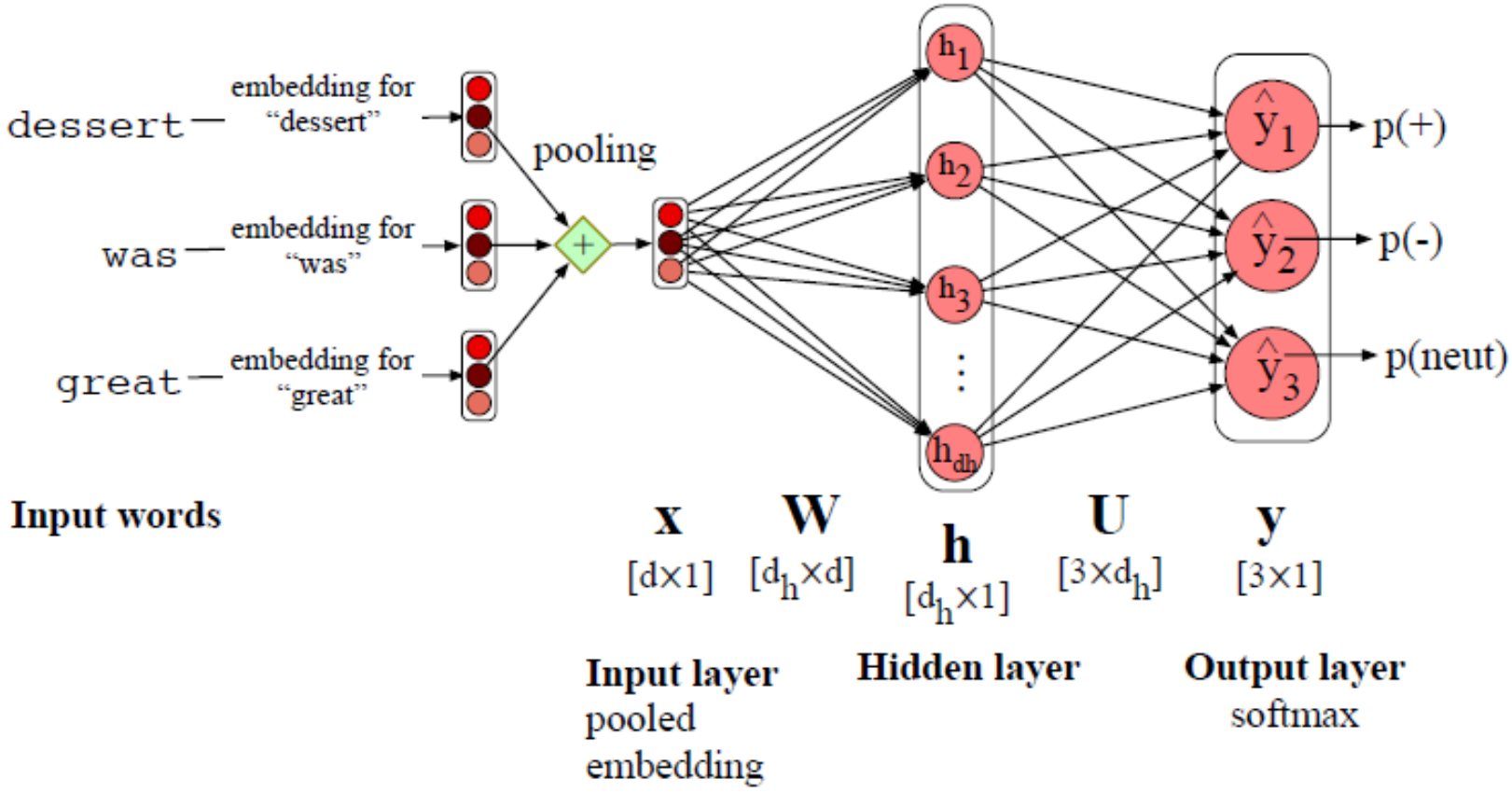
mean pooling: $$\mathbf{x}_{mean} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{e}(w_i)$$

  - There are many other options. The *element-wise max* of a set of n vectors is a new vector whose k^{th} element is the max of the k^{th} elements of all the n vectors.

Feedforward Networks for NLP: Classification

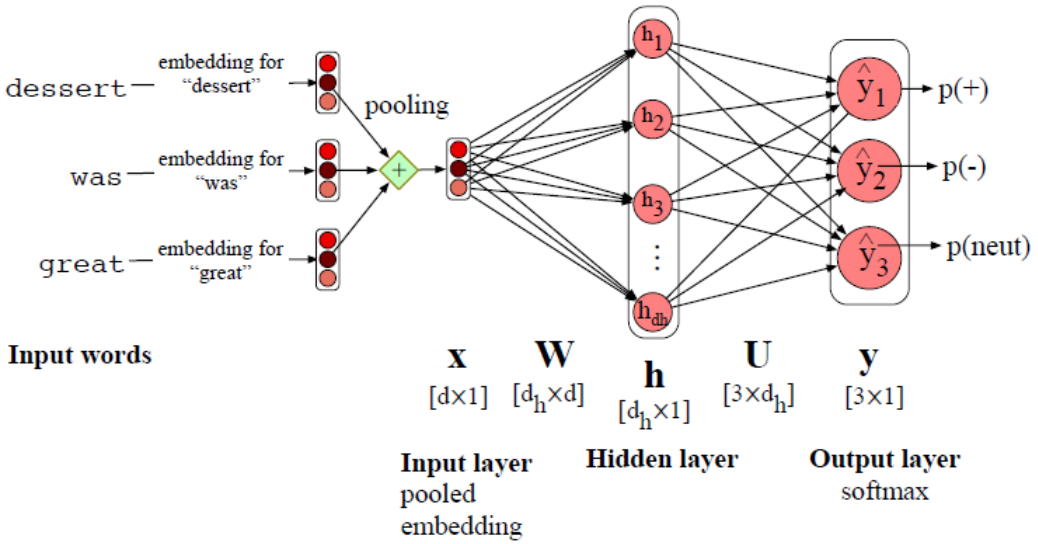
2-layer Network for sentiment analysis

- Feedforward network sentiment analysis using a **pooled embedding** of input words.
  - For *binary classification*, output layer is a ***single sigmoid unit*** instead of softmax units.



Feedforward Networks for NLP: Classification

2-layer Network for sentiment analysis



- The equations for this classifier assuming **mean pooling** (for a single example)

$$\mathbf{x} = \text{mean}(\mathbf{e}(w_1), \mathbf{e}(w_2), \ldots, \mathbf{e}(w_n))$$
$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$
$$\mathbf{z} = \mathbf{U}\mathbf{h}$$
$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$$

- The idea of using word2vec or GloVe embeddings as our input representation is called **pretraining**.
- Using pretrained embedding representations, whether ***simple static word embeddings*** like word2vec or the much more powerful ***contextual embeddings*** like BERT is one of the central ideas of *deep learning*.

Neural Networks

- ***Training Feedforward Neural Networks***

Training Feedforward Neural Networks

- A **feedforward neural net** is an instance of supervised machine learning in which we know the **correct output y** for each observation x, and the system produces **$\hat{y}$**, **the system's estimate of the true y**.
- The **goal of the training procedure** is to learn parameters $W^{[i]}$ and $b^{[i]}$ for each layer i that make **$\hat{y}$** for each training observation as close as possible to the true y.
- It's common to use the loss function used for logistic regression, **cross-entropy loss** function for neural networks too.
- To find the parameters that minimize this loss function, we can use the **gradient descent** optimization algorithm

Training Feedforward Neural Networks

- **Gradient descent** requires knowing the **gradient of the loss function**, the vector that contains the partial derivative of the loss function with respect to each of the parameters.
- In logistic regression, for each observation we could directly compute the derivative of the loss function with respect to an individual w or b.
- But for neural networks, with millions of parameters in many layers, it's much harder to see how to compute the partial derivative of some weight in layer 1 when the loss is attached to some much later layer.
- How do we partial out the loss over all those intermediate layers?

  ➔ The answer is the algorithm called **error backpropagation** or **backward differentiation**.

Training Feedforward Neural Networks

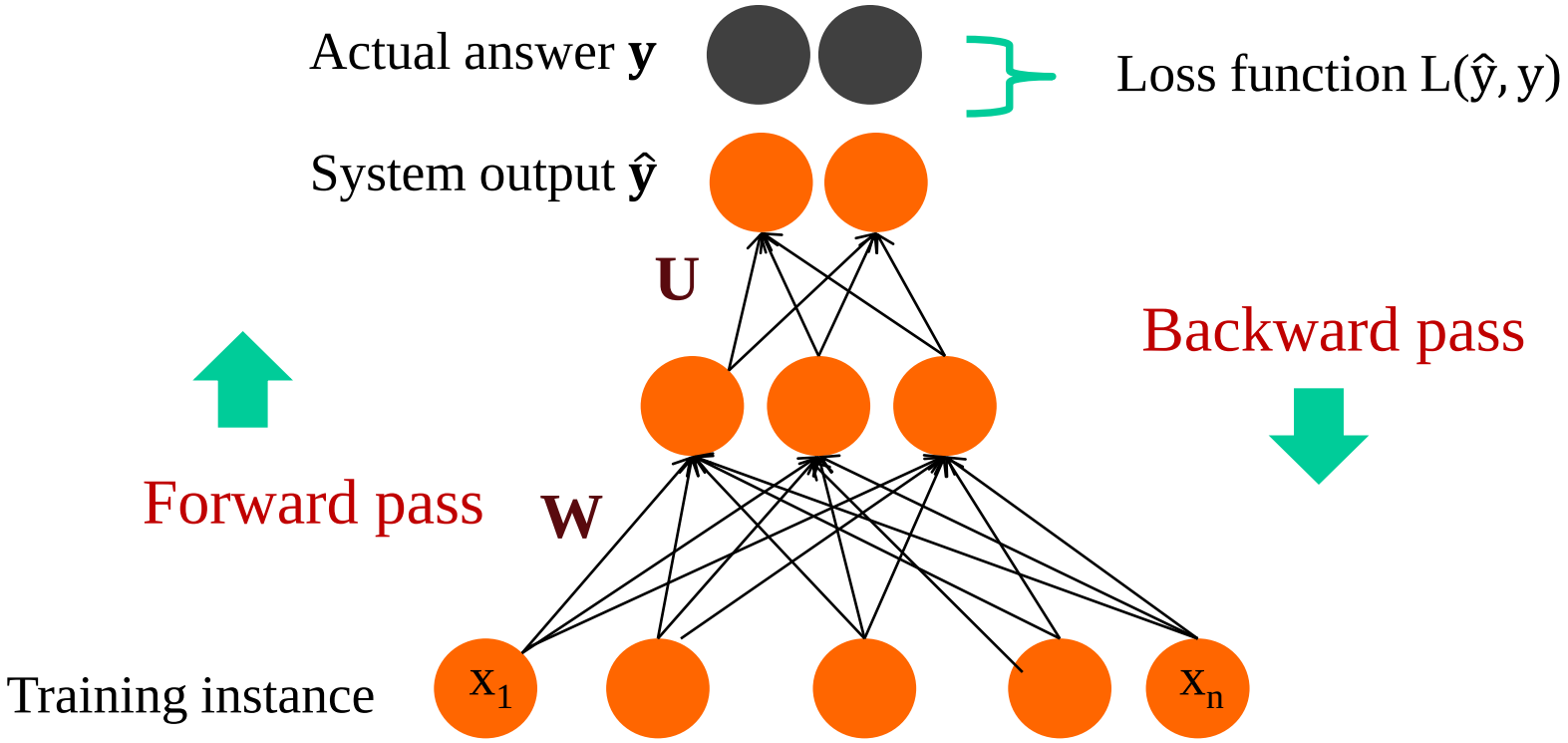
Intuition: training a 2-layer Network

- The **intuition of neural net training** is the **forward computation of the loss** and the **backward computation of the weight updates**.
- Given an input x, we run a forward pass through the network, computing the system output $\hat{\mathbf{y}}$.
- Then we compare $\hat{\mathbf{y}}$ to the true answer y, to get a loss for this example.
- Then we do a backward pass through the network, computing the gradients we need to update the weights.

Training Feedforward Neural Networks

Intuition: training a 2-layer Network

- The **intuition of neural net training** is the **forward computation of the loss** and the **backward computation of the weight updates**.



Training Feedforward Neural Networks

Intuition: training a 2-layer Network

For every training tuple (x, y)

- Run *forward* computation to find our estimate $\hat{y}$
- Run *backward* computation to update weights:
  - For every output node
    - Compute loss L between true y and the estimated $\hat{y}$
    - For every weight w from hidden layer to the output layer
      - Update the weight
  - For every hidden node
    - Assess how much blame it deserves for the current answer
    - For every weight w from input layer to the hidden layer
      - Update the weight

Training Feedforward Neural Networks

Intuition: training a 2-layer Network

- To **adjust weights**
  - Find the **gradients**, the derivative of the loss function with respect to weights
  - Move weights in the opposite direction of the gradient

$$w^{t+1} = w^t - \eta \frac{d}{dw} L(f(x;w),y)$$

- **Cross-entropy loss** for **multinomial logistic regression**

$$\begin{aligned} L_{\text{CE}}(\hat{\mathbf{y}},\mathbf{y}) &= -\sum_{k=1}^{K} y_k \log \hat{y}_k \qquad \text{(where } c \text{ is the correct class)} \\ &= -\log \hat{y}_c = -\log \frac{\exp(\mathbf{z}_c)}{\sum_{j=1}^{K} \exp(\mathbf{z}_j)} \end{aligned}$$

- **Gradient** (for a single example) in **multinomial logistic regression**

$$\begin{aligned} \frac{\partial L_{\text{CE}}}{\partial w_{k,i}} &= -(y_k - \hat{y}_k) x_i \\ &= -(y_k - p(y_k = 1|x)) x_i \\ &= -\left(y_k - \frac{\exp(\mathbf{w_k} \cdot \mathbf{x} + b_k)}{\sum_{j=1}^{K} \exp(\mathbf{w_j} \cdot \mathbf{x} + b_j)} \right) x_i \end{aligned}$$

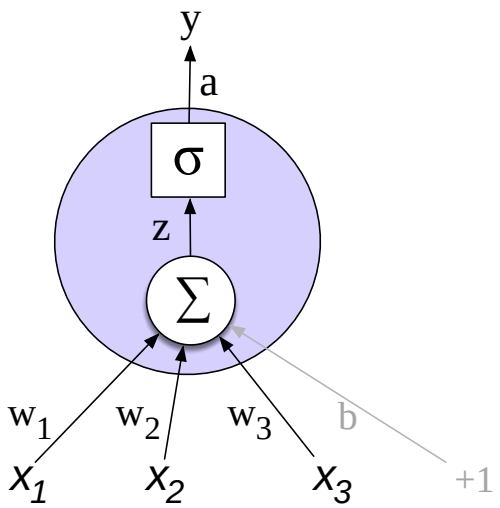
Training Feedforward Neural Networks

Intuition: training a 2-layer Network

- Where did that derivative come from?

Using the chain rule! f (x) = u(v(x))

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$$



Derivative of the weighted sum

Derivative of the Activation

Derivative of the Loss

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_i}$$

Training Feedforward Neural Networks

Intuition: training a 2-layer Network

- **Gradients:**

$$\frac{\partial L_{\text{CE}}(\hat{\mathbf{y}}, \mathbf{y})}{\partial \mathbf{w}_{k,i}} = -\left(\mathbf{y}_k - \frac{\exp(\mathbf{w_k} \cdot \mathbf{x} + b_k)}{\sum_{j=1}^{K} \exp(\mathbf{w_j} \cdot \mathbf{x} + b_j)}\right)\mathbf{x}_i$$

- But **these derivatives** only give correct updates for one weight layer
- For deep networks, computing the gradients for each weight is much more complex, since we are computing the derivative with respect to weight parameters that appear all the way back in the very early layers of the network, even though the loss is computed only at the very end of the network.
- **For training, we need the derivative of the loss with respect to each weight in every layer of the network**
  - But the loss is computed only at the very end of the network!
- **Solution: error backpropagation**
  - **backpropagation** is a special case of **backward differentiation** which relies on **computation graphs**.

Neural Networks

- *Training Feedforward Neural Networks: Computation Graphs*

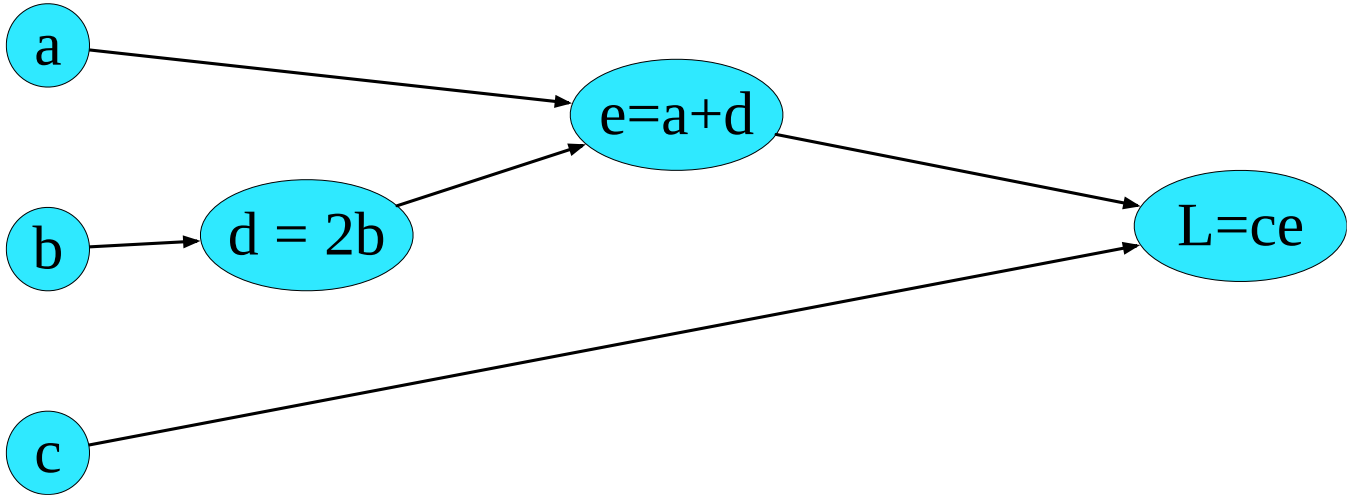
Computation Graphs

- For training, we need the derivative of the loss with respect to each weight in every layer of the network
  - But the loss is computed only at the very end of the network!
- Solution: error backpropagation
  - backpropagation is a backward differentiation which relies on c**omputation graphs**.
- A **computation graph** is a representation of the process of computing a mathematical expression, in which the computation is broken down into separate operations, each of which is modeled as a node in a graph.

Computation Graphs

Example

- Consider computing the function **L(a, b, c) = c(a + 2b)**
  - If we make each of the component addition and multiplication operations explicit, and add names (d and e) for the intermediate outputs, the resulting series of computations is:
- **Computations: d = 2 * b e = a + d L = c * e**
- **Its computation graph**:

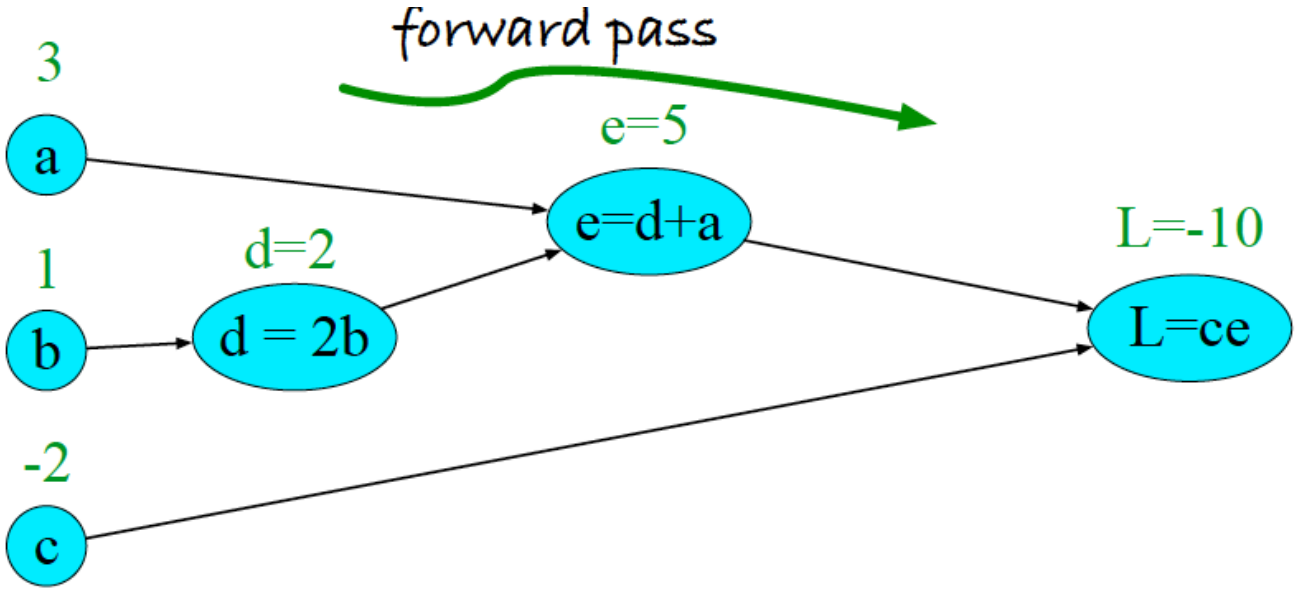


Computation Graphs

Example – forward pass

- **Function:** **L(a, b, c) = c(a + 2b)**
- **Computations:** **d = 2 * b** **e = a + d** **L = c * e**

- Forward computation for inputs a=3, b=1, c=-2



Computation Graphs

Example – backward pass

- The importance of the computation graph comes from the **backward pass**
- This is used to **compute the derivatives that we'll need for the weight update.**
- In this example our goal is to compute the derivative of the output function L with respect to each of the input variables,

  $\partial \mathbf{L}/\partial \mathbf{a}$ $\partial \mathbf{L}/\partial \mathbf{b}$ $\partial \mathbf{L}/\partial \mathbf{c}$

- The derivative $\partial \mathbf{L}/\partial \mathbf{a}$ tells us how much a small change in **a** affects L, while holding the others constant.

Computation Graphs

Example – backward pass – chain rule

- **Chain Rule:** Computing the **derivative of a composite function**:

f (x) = u(v(x)) $$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$$

f (x) = u(v(w(x))) $$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx}$$

- **Chain rule** applies to our example:

L(a,b,c) = c(a+2b) **d = 2*b** **e = a+d** **L = c*e**

$$\frac{\partial L}{\partial c} = e \qquad \frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

Computation Graphs

Example – backward pass

L(a,b,c) = c(a+2b) **d = 2*b** **e = a+d** **L = c*e**

$$\frac{\partial L}{\partial c} = e \qquad \frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

$$L = ce : \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

$$e = a + d : \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

$$d = 2b : \quad \frac{\partial d}{\partial b} = 2$$

Computation Graphs

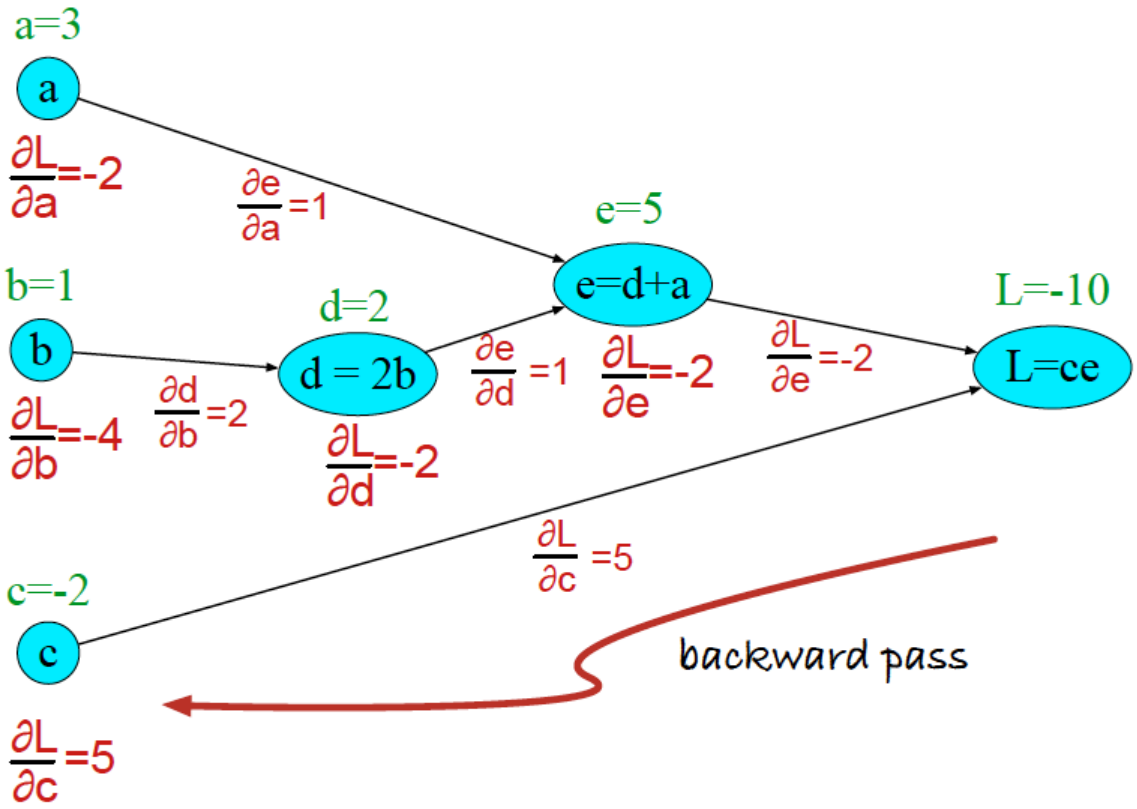
Example – backward pass

$$\frac{\partial L}{\partial c} = e \qquad \frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

$$L = ce : \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

$$e = a + d : \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

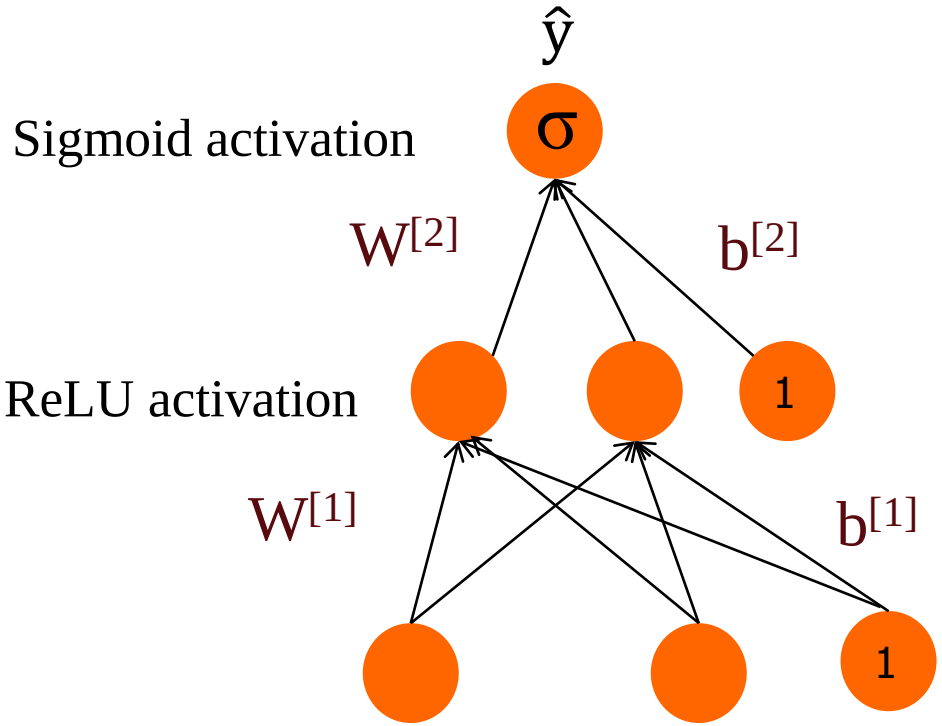
$$d = 2b : \quad \frac{\partial d}{\partial b} = 2$$



Computation Graphs

Backward differentiation on a two layer network

- Of course computation graphs for real neural networks are much more complex.
- **A sample computation graph** for a 2-layer neural network with n_0=2, n_1=2, n_2=1



The function that the computation graph is computing is:

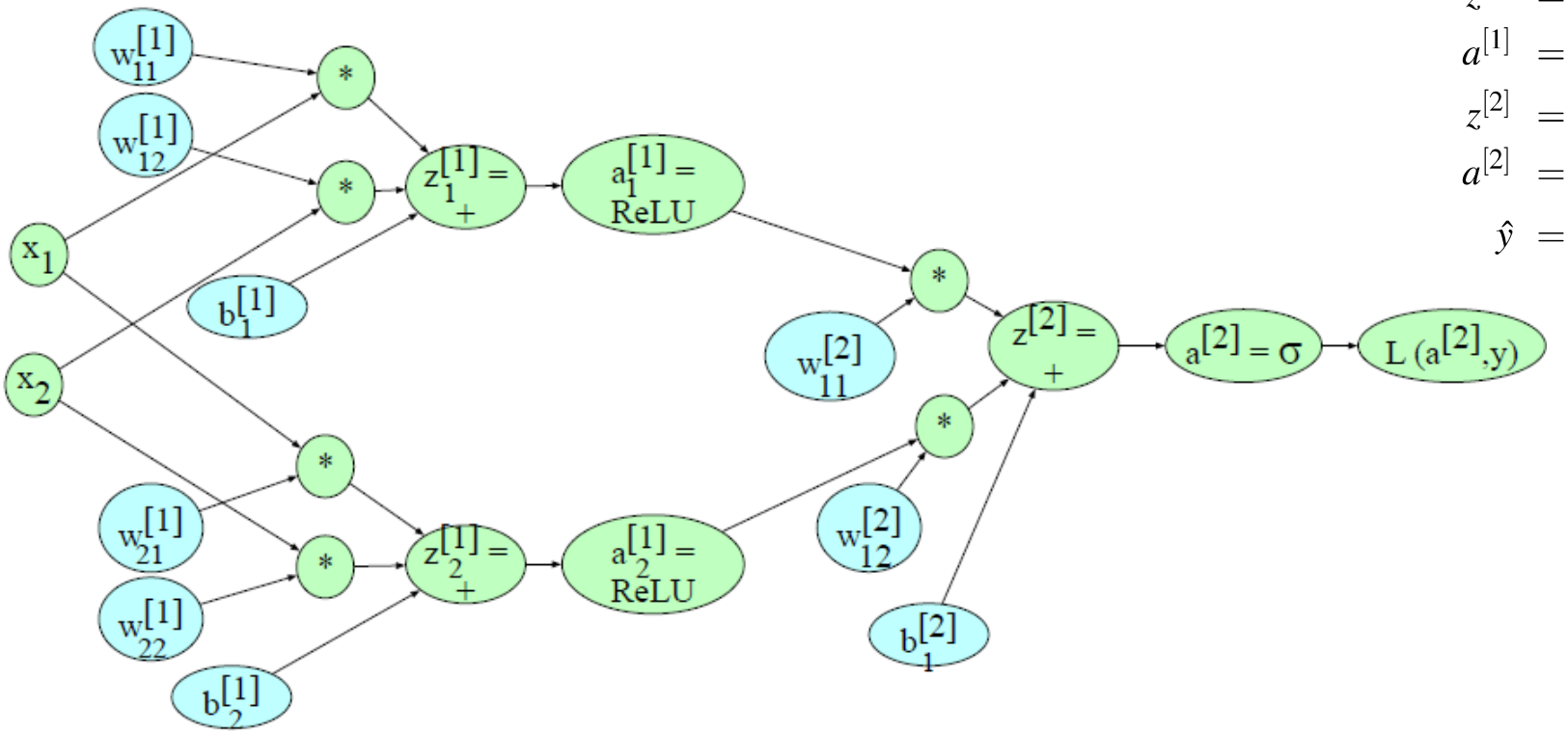
$$
\begin{aligned}
z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\
a^{[1]} &= \text{ReLU}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

Computation Graphs

Backward differentiation on a two layer network – computation graph

- The weights that need updating (those for which we need to know the partial derivative of the loss function) are shown in orange .
  - So for a particular example x1,x2, we would run the forward pass, assign values to each of our nodes, and then starting with this last node, run backwards pass

$$
\begin{aligned}
z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\
a^{[1]} &= \text{ReLU}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

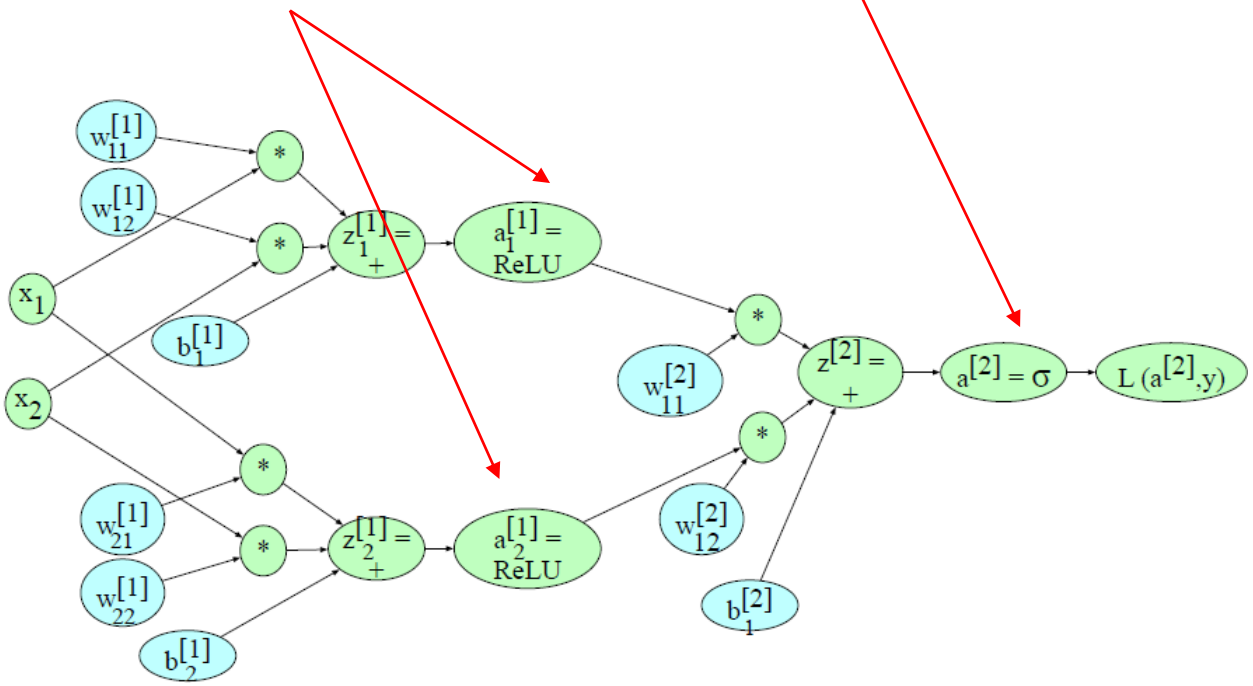


Computation Graphs

Backward differentiation on a two layer network – backward pass

- The weights that need updating (those for which we need to know the partial derivative of the loss function) are shown in **light blue**.
  - In order to do the backward pass, we need the derivatives of all the functions in the graph.

$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for\ \ z < 0 \\ 1 & for\ \ z \geq 0 \end{cases} \qquad \frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z)) \qquad \frac{d\tanh(z)}{dz} = 1-\tanh^2(z)$$



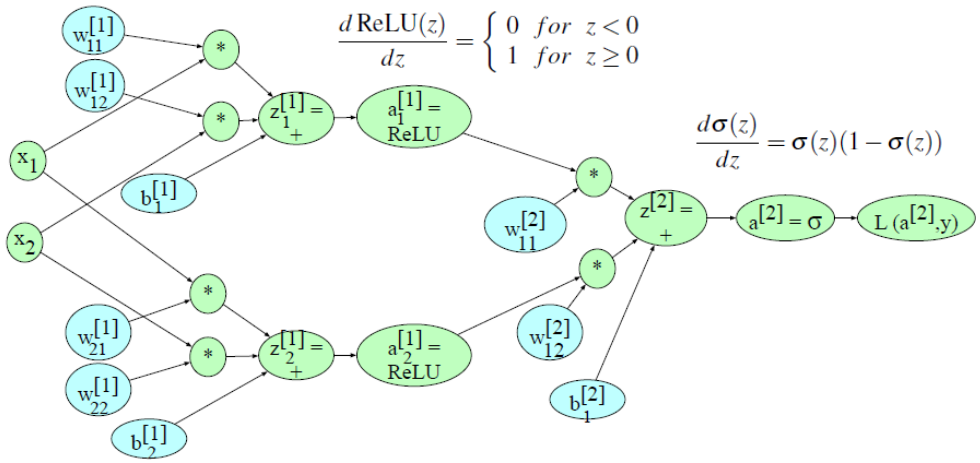
$$\begin{aligned} z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\ a^{[1]} &= \mathrm{ReLU}(z^{[1]}) \\ z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} &= \sigma(z^{[2]}) \\ \hat{y} &= a^{[2]} \end{aligned}$$

Computation Graphs

Backward differentiation on a two layer network – backward pass

- Derivative of the loss function L with respect to $z^{[2]}$,

$$\frac{\partial L}{\partial z^{[2]}} = \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}}$$



$$L_{CE}(a^{[2]}, y) = -\left[y \log a^{[2]} + (1-y)\log(1-a^{[2]})\right]$$

$$\frac{\partial L}{\partial a^{[2]}} = -\left(\left(y\frac{\partial \log(a^{[2]})}{\partial a^{[2]}}\right) + (1-y)\frac{\partial \log(1-a^{[2]})}{\partial a^{[2]}}\right) = -\left(\left(y\frac{1}{a^{[2]}}\right) + (1-y)\frac{1}{1-a^{[2]}}(-1)\right) = -\left(\frac{y}{a^{[2]}} + \frac{y-1}{1-a^{[2]}}\right)$$

$$\frac{\partial a^{[2]}}{\partial z^{[2]}} = a^{[2]}(1-a^{[2]})$$

$$\frac{\partial L}{\partial z^{[2]}} = \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} = -\left(\frac{y}{a^{[2]}} + \frac{y-1}{1-a^{[2]}}\right) a^{[2]}(1-a^{[2]}) = a^{[2]} - y$$

Let log=ln

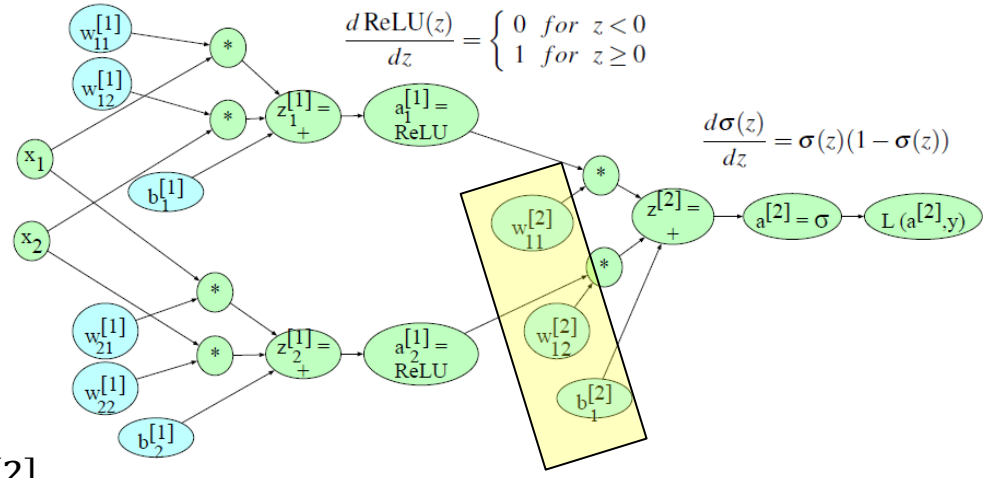
$$\frac{d\ln(x)}{dx} = \frac{1}{x}$$

$$\frac{d\ln(f(x))}{dx} = \frac{1}{f(x)}\frac{df(x)}{dx}$$

Computation Graphs

Backward differentiation on a two layer network – backward pass

- Derivative of the loss function L with respect to output layer weights



$$\frac{\partial L}{\partial w_{11}^{[2]}} = \frac{\partial L}{\partial z^{[2]}} \frac{\partial z^2}{\partial w_{11}^{[2]}}$$

$$z^{[2]} = w_{11}^{[2]} * a_1^{[1]} + w_{12}^{[2]} * a_2^{[1]} + b_1^{[2]}$$

$$\frac{\partial L}{\partial z^{[2]}} = a^{[2]} - y \qquad \frac{\partial z^2}{\partial w_{11}^{[2]}} = a_1^{[1]} \qquad \frac{\partial z^2}{\partial w_{12}^{[2]}} = a_2^{[1]} \qquad \frac{\partial z^2}{\partial b_1^{[2]}} = 1$$

$$\frac{\partial L}{\partial w_{11}^{[2]}} = \left(a^{[2]} - y\right) a_1^{[1]} \qquad \frac{\partial L}{\partial w_{12}^{[2]}} = \left(a^{[2]} - y\right) a_2^{[1]} \qquad \frac{\partial L}{\partial b_1^{[2]}} = \left(a^{[2]} - y\right)$$

Computation Graphs

More details on learning

- **Overfitting**: Various forms of **regularization** are used to prevent **overfitting**.
  - One of the most important is **dropout**:
  - Randomly dropping some units and their connections from the network during training

- Tuning of **hyperparameters** is also important.
  - The parameters of a neural network are weights W and biases b; those are learned by gradient descent.
  - The **hyperparameters** are things that are chosen by the algorithm designer; optimal values are tuned on a ***devset*** rather than by gradient descent learning on the training set.
  - Hyperparameters include the **learning rate**, the **mini-batch size**, the **model architecture** (the number of layers, the number of hidden nodes per layer, the choice of activation functions), **how to regularize**, and so on.

Neural Networks

- *Feedforward Neural Language Modeling*

Feedforward Neural Language Modeling

- **Language Modeling**: Calculating the probability of the next word in a sequence given some history.
- We've seen N-gram based LMs
- But neural network LMs far outperform n-gram language models
- **State-of-the-art neural LMs** are based on more powerful neural network technology like ***Transformers***
- But simple **feedforward LMs** can do almost as well!

Feedforward Neural Language Modeling

- **Neural language models** have many advantages over the ***n-gram language models.***

Advantages: Compared to n-gram models, neural language models

- can handle much longer histories,
- can generalize better over contexts of similar words, and
- are more accurate at word-prediction.

Disadvantages. On the other hand, neural net language models

- are much more complex,
- are slower and need more energy to train,
- and are less interpretable than n-gram models, so for some smaller tasks an n-gram language model is still the right tool.

Feedforward Neural Language Modeling

Why Neural LMs work better than N-gram LMs

- A **feedforward neural language model (LM)** is a feedforward network that takes a representation of some number of previous words (w_{t-1},w_{t-2},..) as input and outputs a probability distribution over possible next words.
  - Like a N-gram LM, a feedforward neural LM approximates the probability of a word:

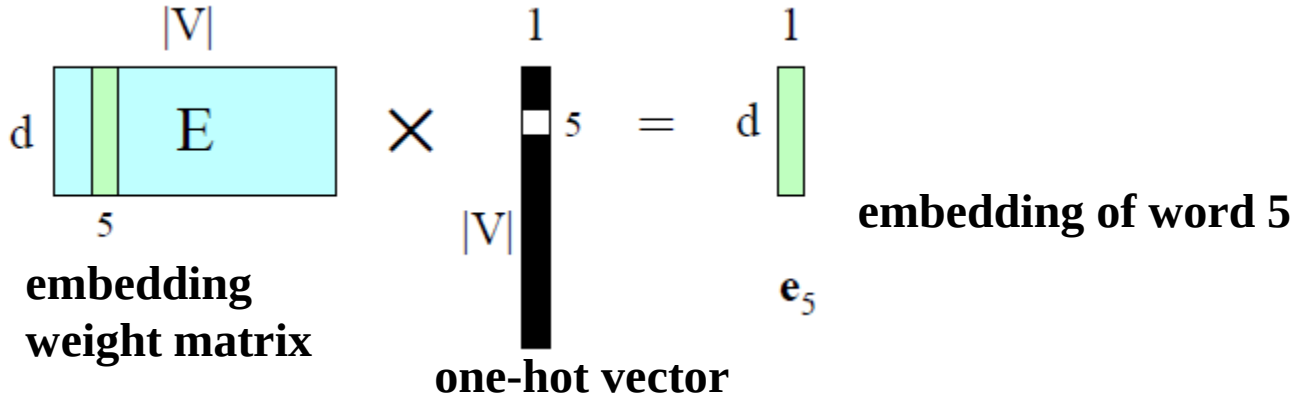
$$P(w_t|w_1,\ldots,w_{t-1}) \approx P(w_t|w_{t-N+1},\ldots,w_{t-1})$$

- **Neural language models** represent words in this prior context by their **embeddings**, rather than just by their word identity as used in n-gram language models.
- Using embeddings allows neural language models to generalize better to unseen data.
  - Training data**: I have to make sure that the cat gets fed**
  - Never seen in training: **dog gets fed**
  - Test data**:** **I forgot to make sure that the dog gets ___**
  - N-gram LM can NOT predict '**fed**' in this test data
  - Neural LM can use similarity '**cat**' and '**dog**' embeddings to generalize and predict '**fed**' after '**dog**'

Feedforward Neural Language Modeling

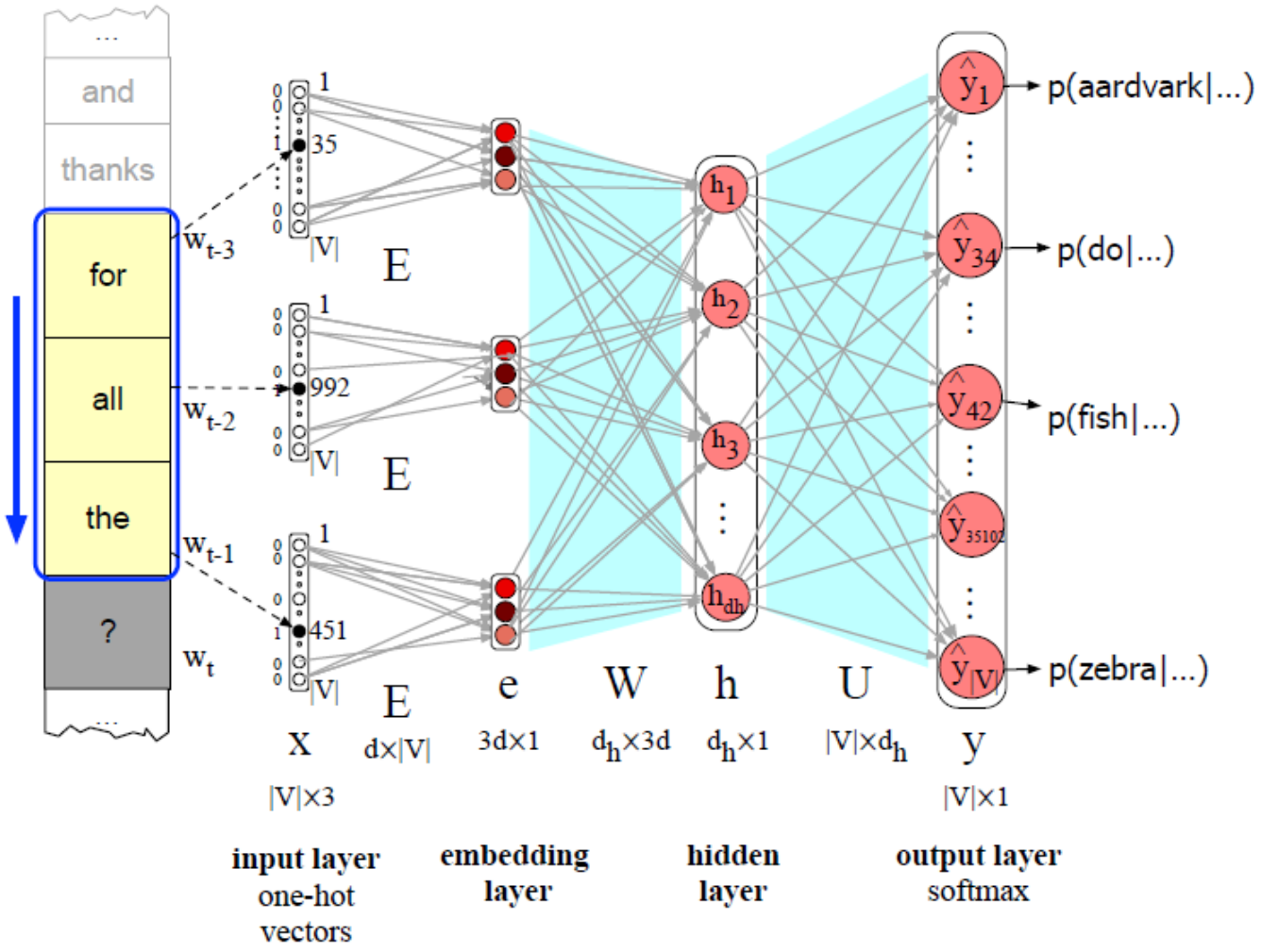
Forward inference in a feedforward neural language model

- **Forward inference** is the task, given an input, of running a forward pass on the network to produce a probability distribution over possible outputs (next words).
- First represent each of the N previous words as a **one-hot vector** of length |V|
- A **one-hot vector** is a vector that has one element equal to 1 corresponding to word's index in vocabulary, while all other elements are set to zero.
- Multiplying **embedding weight matrix E** by **a one-hot vector** that has only one non-zero element x_i=1 simply selects out the relevant column vector for word i, resulting in the embedding for word i.



Feedforward Neural Language Modeling

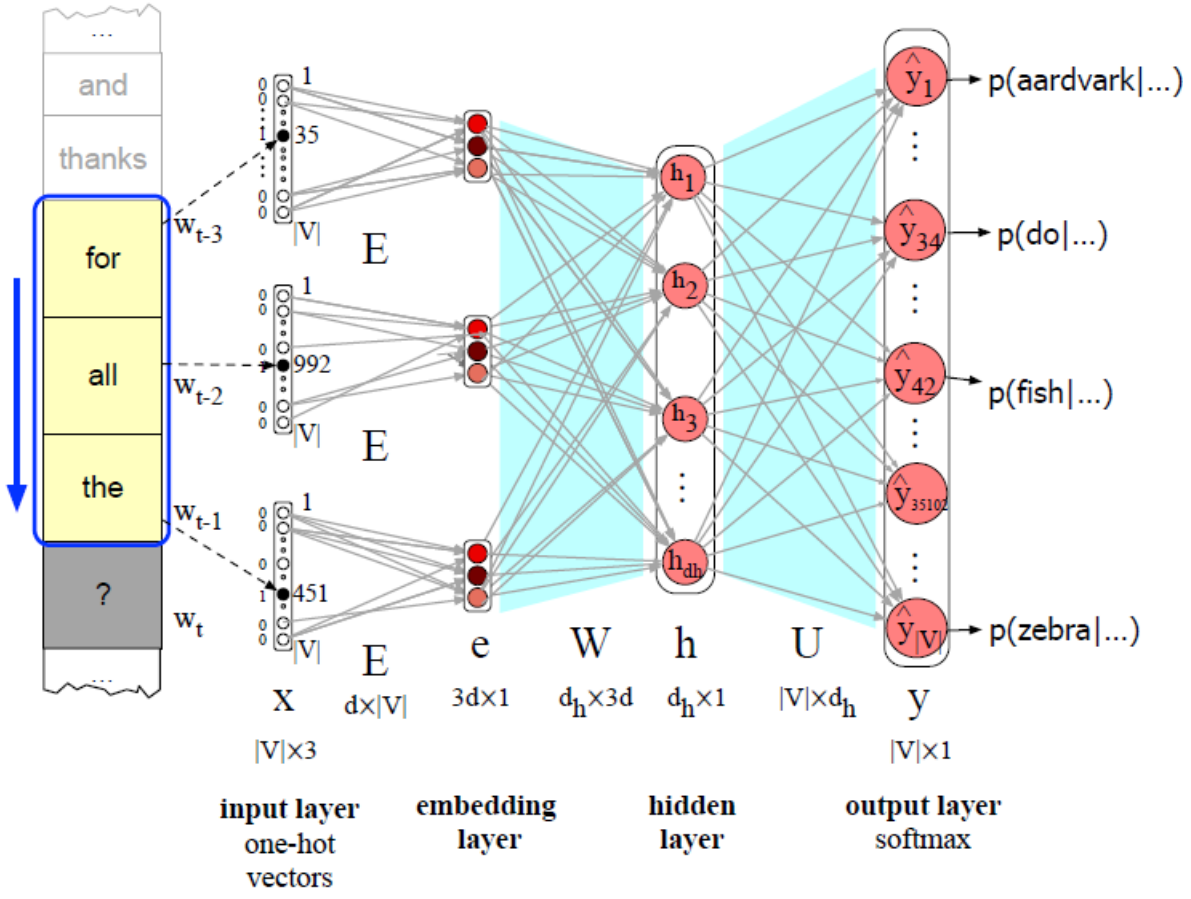
a feedforward neural language model with a window size of 3



- At each timestep t the network computes a d-dimensional embedding for each context word
- Concatenates 3 resulting embeddings to get embedding layer e.
- Embedding vector e is multiplied by a weight matrix W and then an activation function is applied element-wise to produce the hidden layer h.
- then h is multiplied by another weight matrix U.
- Finally, a softmax output layer predicts at each node i the probability that the next word w_t will be vocabulary word V_i.

Feedforward Neural Language Modeling

a feedforward neural language model with a window size of 3



- Equations for a neural language model with a window size of 3, given one-hot input vectors for each input context word

$$\mathbf{e} = [\mathbf{E}\mathbf{x}_{t-3}; \mathbf{E}\mathbf{x}_{t-2}; \mathbf{E}\mathbf{x}_{t-1}]$$
$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{e} + \mathbf{b})$$
$$\mathbf{z} = \mathbf{U}\mathbf{h}$$
$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$$

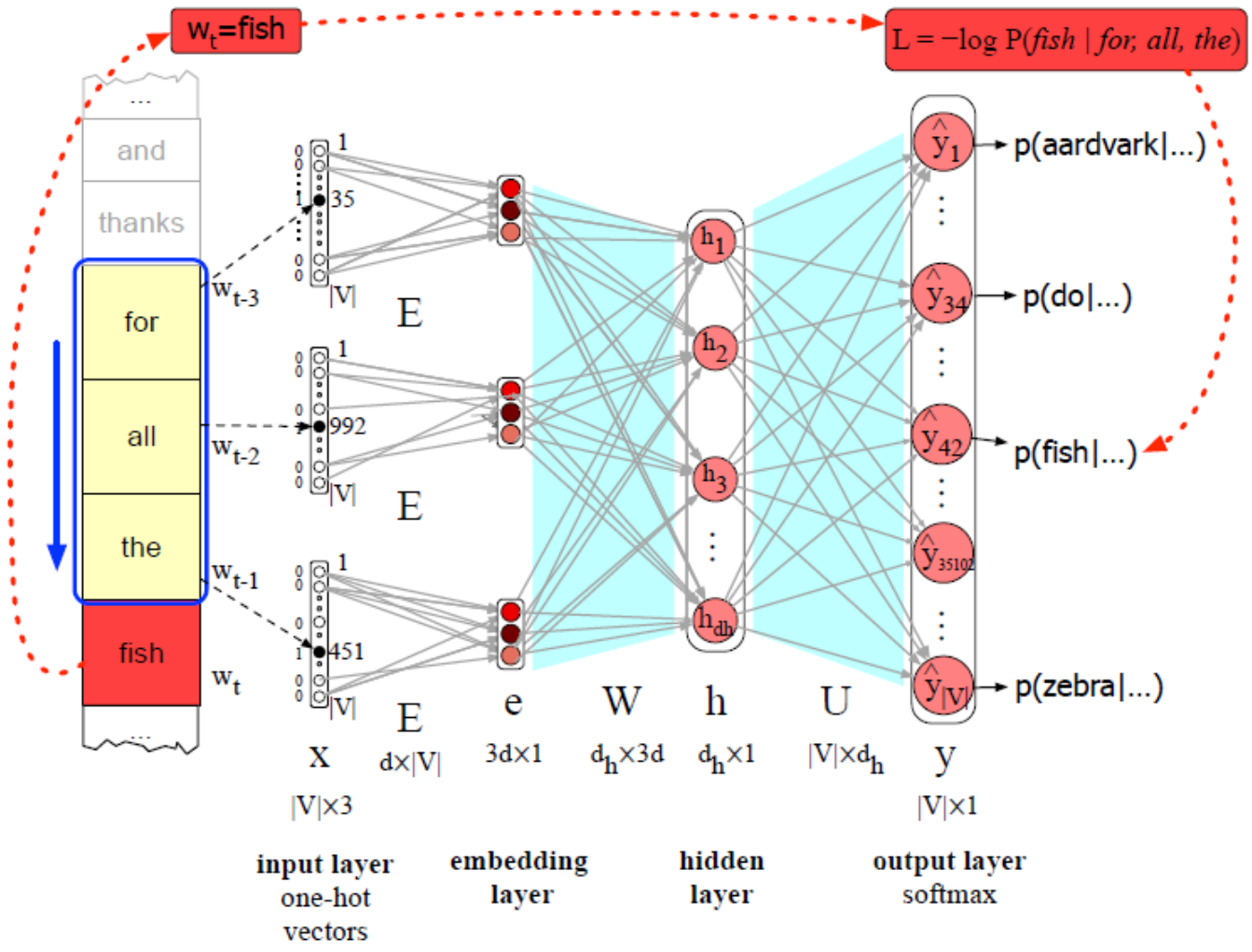
Feedforward Neural Language Modeling

Training Feedforward Language Model

- In **self-training for language modeling**, we take a corpus of text as training material and at each time step t ask the model to predict the next word.
  - We simply train the model to minimize the error in predicting the true next word in the training sequence.
- In practice, training the model means setting the parameters **θ = E,W,U,b**.
- It's ok to **freeze** the embedding layer E with initial word2vec values.
  - Freezing means we use word2vec or some other pretraining algorithm to compute the initial embedding matrix E, and then hold it constant while we only modify W, U, and b.
  - Often we'd like to learn the embeddings simultaneously with training the network.
- To train the entire model including E, i.e. to set all the parameters **θ = E,W,U,b**.
  - We can do this via gradient descent using error backpropagation on the computation graph to compute the gradient.
  - Training thus not only sets the weights W and U of the network, but also as we're predicting upcoming words, we're learning the embeddings E for each word that best predict upcoming words.

Feedforward Neural Language Modeling

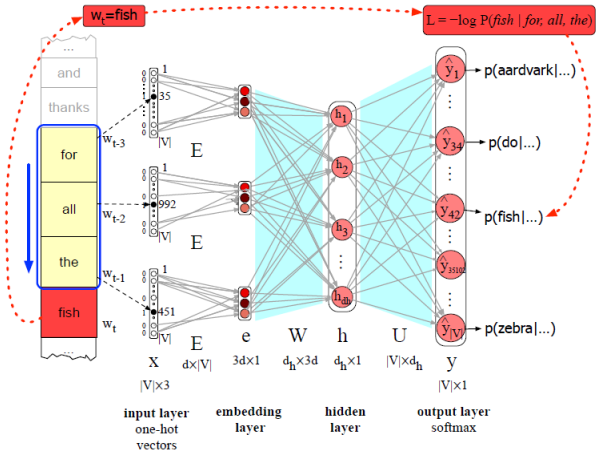
Training Feedforward Language Model



Feedforward Neural Language Modeling

Training Feedforward Language Model

- Training proceeds by taking as input a very long text, concatenating all the sentences, starting with random weights, and then iteratively moving through the text predicting each word w_t .



- At each word w_t, we use the cross-entropy loss

$$L_{CE}(\hat{y}, y) = -\log \hat{y}_i, \quad \text{(where } i \text{ is the correct class)}$$

$$L_{\text{CE}} = -\log p(w_t | w_{t-1}, \ldots, w_{t-n+1})$$

- The parameter update for stochastic gradient descent for this loss from step s to s+1:

$$\theta^{s+1} = \theta^s - \eta \frac{\partial \left[-\log p(w_t | w_{t-1}, \ldots, w_{t-n+1}) \right]}{\partial \theta}$$

- This gradient can be computed in any standard neural network framework which will then backpropagate through **θ = E,W,U,b**.

Nueral Networks and Neural Language Models
Summary

- **Neural networks** are built out of **neural units**, originally inspired by biological neurons but now simply an abstract computational device.
- Each **neural unit** multiplies input values by a **weight vector**, adds a **bias**, and then applies a **non-linear activation function** like ***sigmoid***, ***tanh***, or ***rectified linear unit***.
- In a **fully-connected, feedforward network**, each unit in layer i is connected to each unit in layer i+1, and there are no cycles.
- The **power of neural networks** comes from the ability of early layers to learn representations that can be utilized by later layers in the network.
- Neural networks are trained by optimization algorithms like **gradient descent**.
- **Error backpropagation**, backward differentiation on a computation graph, is used to compute the gradients of the loss function for a network.
- **Neural language models** use a neural network as a probabilistic classifier, to compute the probability of the next word given the previous n words.
- Neural language models can use **pretrained embeddings**, or can learn embeddings from scratch in the process of language modeling.