Redes Neuronales para PLN

Parte 1

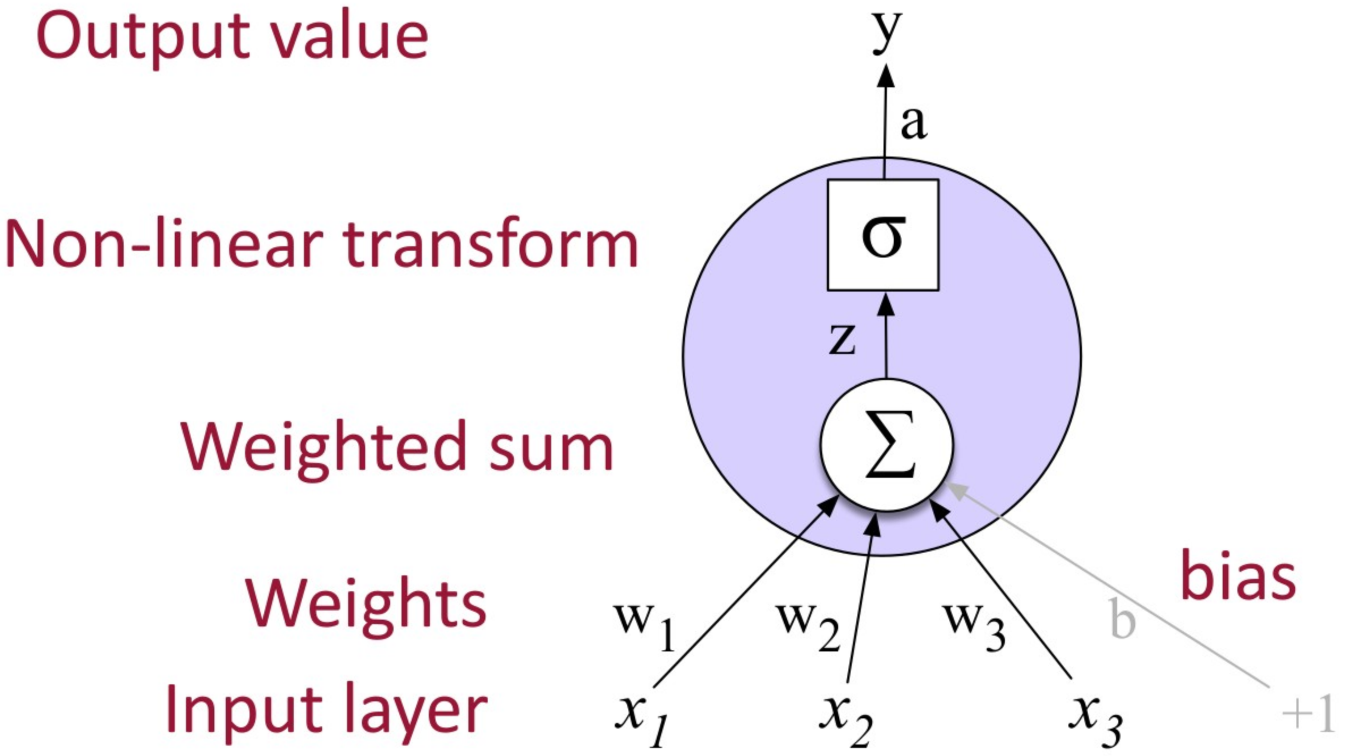
CDLN 2022

Jurafsky and Martin 3rd edition. Cap 7. https://web.stanford.edu/~jurafsky/slp3/

PyTorch https://pytorch.org

Artificial Neuron

Artificial Neuron



$$z = b + \sum_i w_i x_i$$

$$z = w \cdot x + b$$

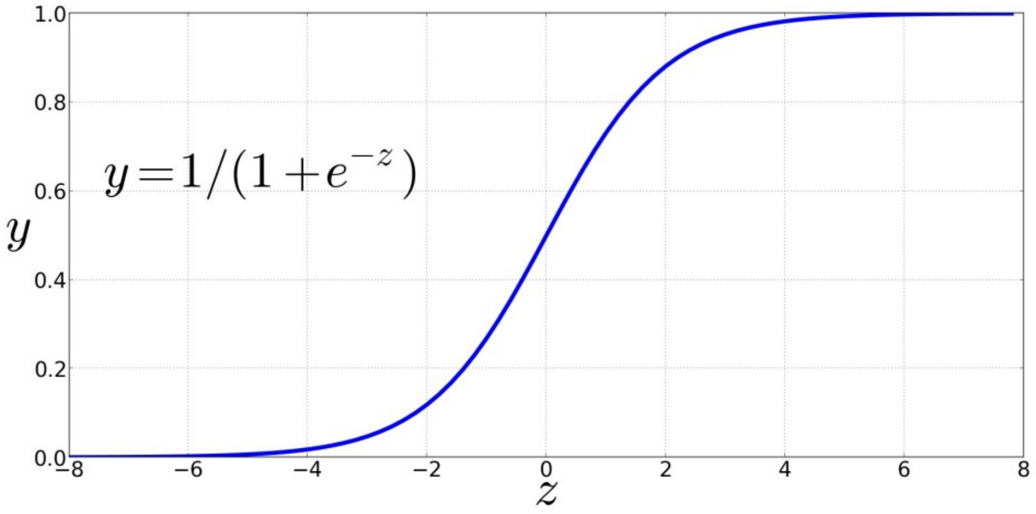
$$y = a = f(z)$$

Basado en https://web.stanford.edu/~jurafsky/slp3/

Activation Function

Sigmoid

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



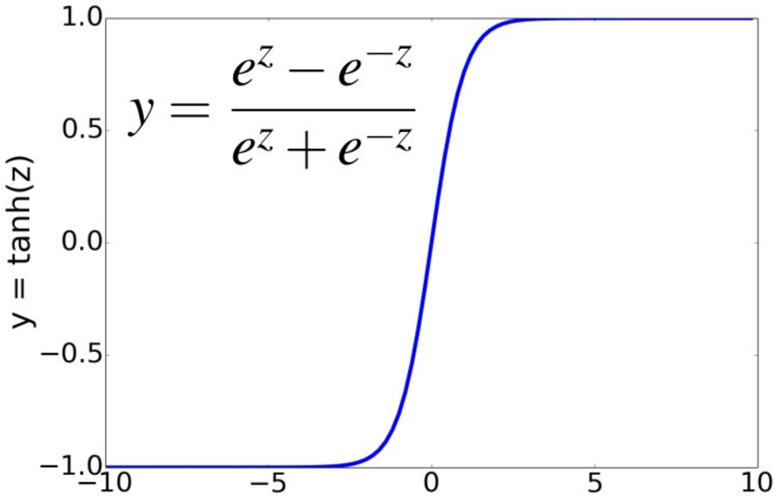
<u>Artificial Neuron using Sigmoid Activation function:</u>

$$y = \sigma(w \cdot x + b) = \frac{1}{1+\exp(-(w \cdot x + b))}$$

Basado en https://web.stanford.edu/~jurafsky/slp3/

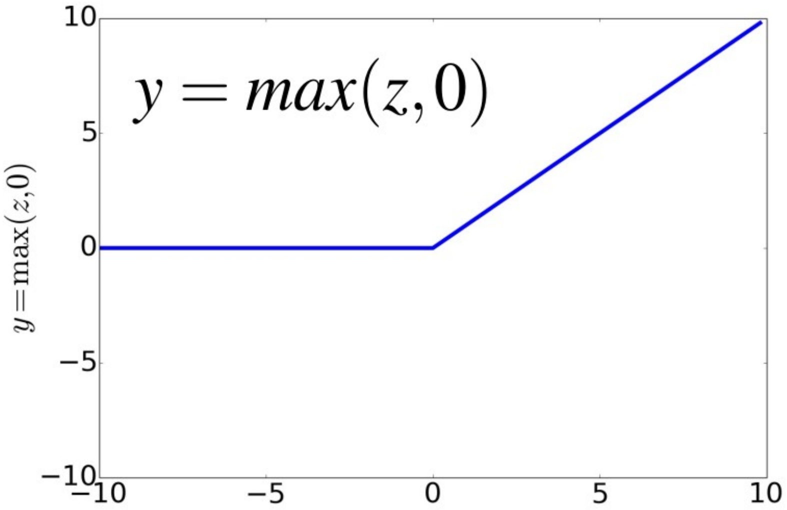
Activation Function

Non-Linear Activation Functions besides sigmoid



tanh

Most Common:



ReLU

Rectified Linear Unit

Basado en https://web.stanford.edu/~jurafsky/slp3/

The XOR problem

Logic Functions as Artificial Neurons

Minsky and Papert (1969)

Can neural units compute simple functions of input?

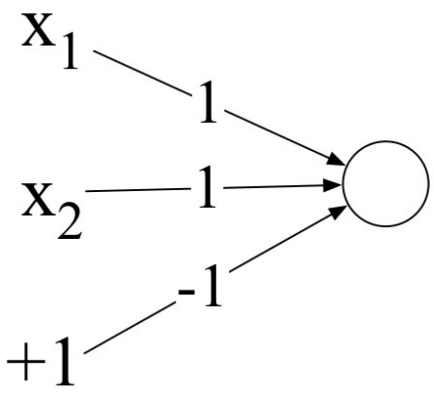
AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

Basado en https://web.stanford.edu/~jurafsky/slp3/

AND, OR, ¿XOR?

Perceptron

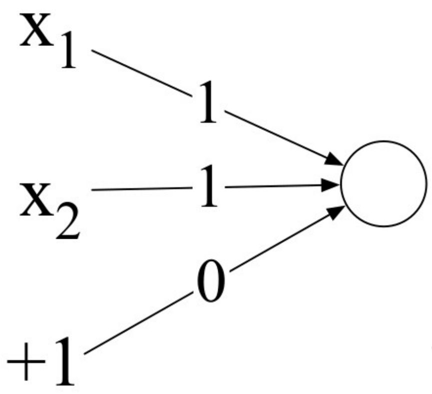
$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$



AND

AND

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



OR

OR

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

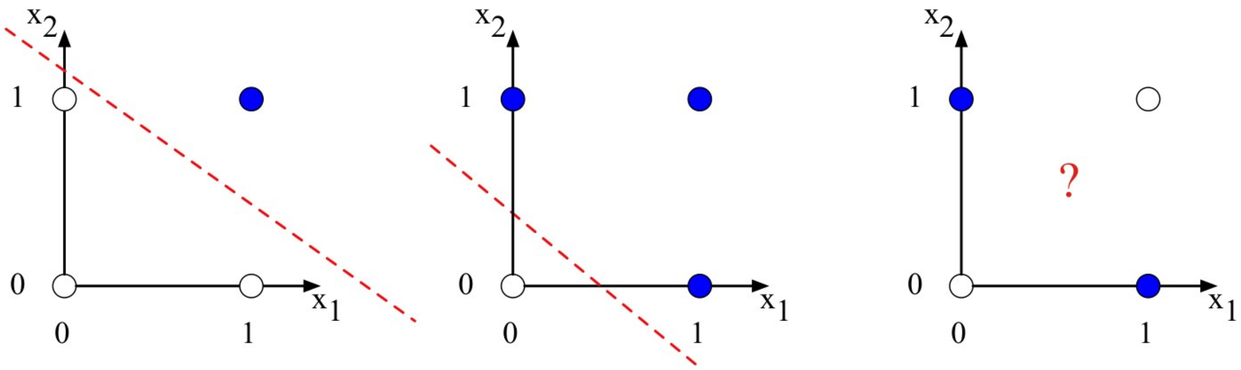
Basado en https://web.stanford.edu/~jurafsky/slp3/

XOR Problem

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: $x_2 = (-w_1/w_2)x_1 + (-b/w_2)$)



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!

Basado en https://web.stanford.edu/~jurafsky/slp3/

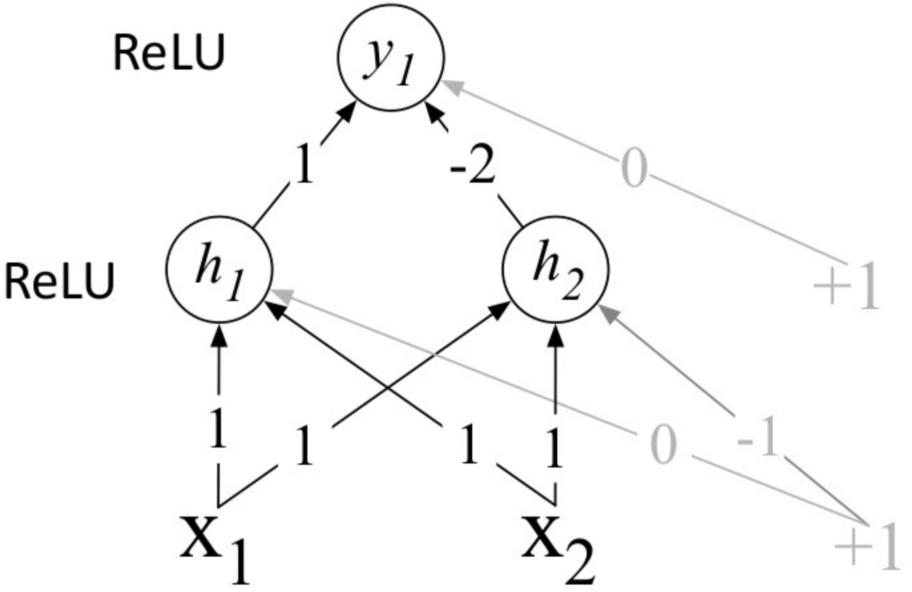
XOR Problem

XOR **can't** be calculated by a single perceptron

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

XOR

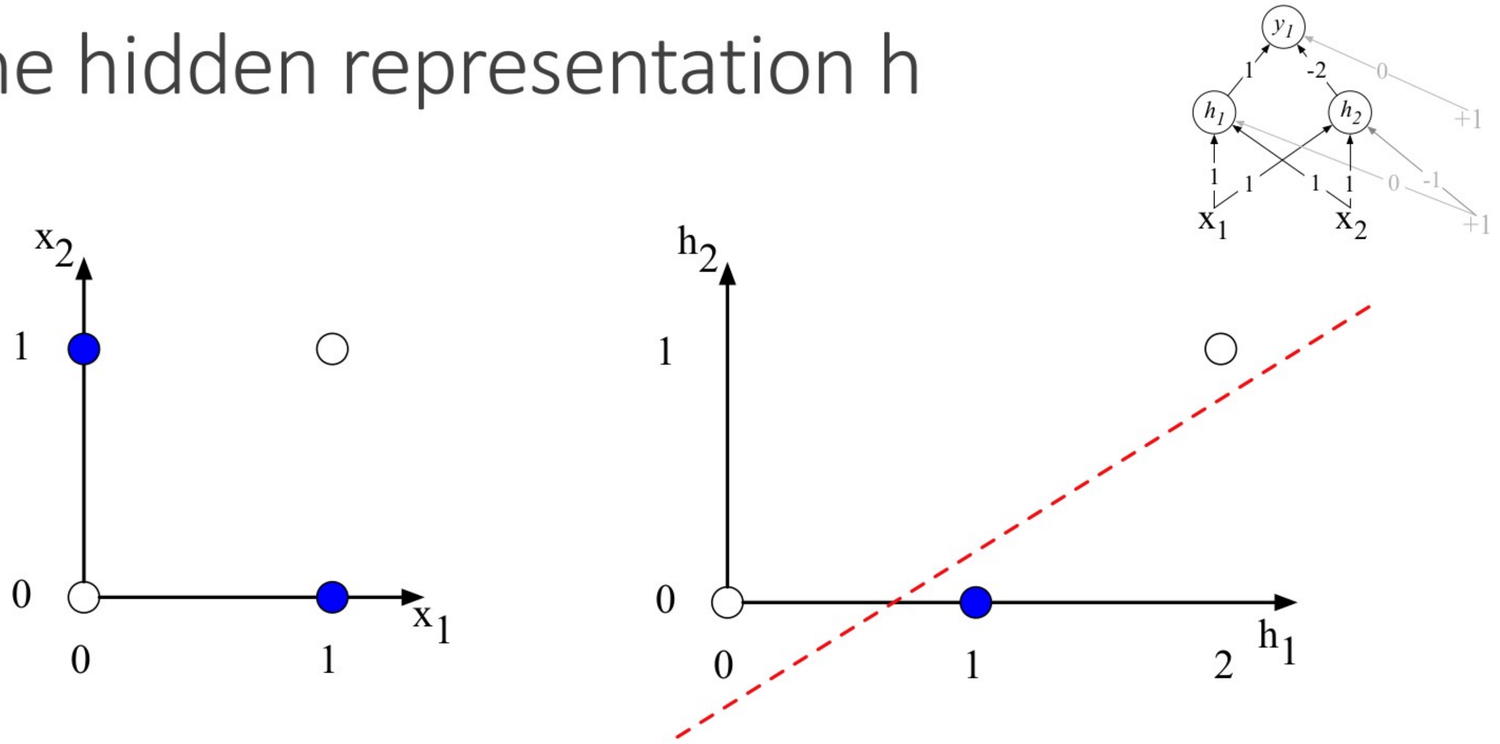
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0



Basado en https://web.stanford.edu/~jurafsky/slp3/

XOR Problem

The hidden representation h



a) The original *x* space

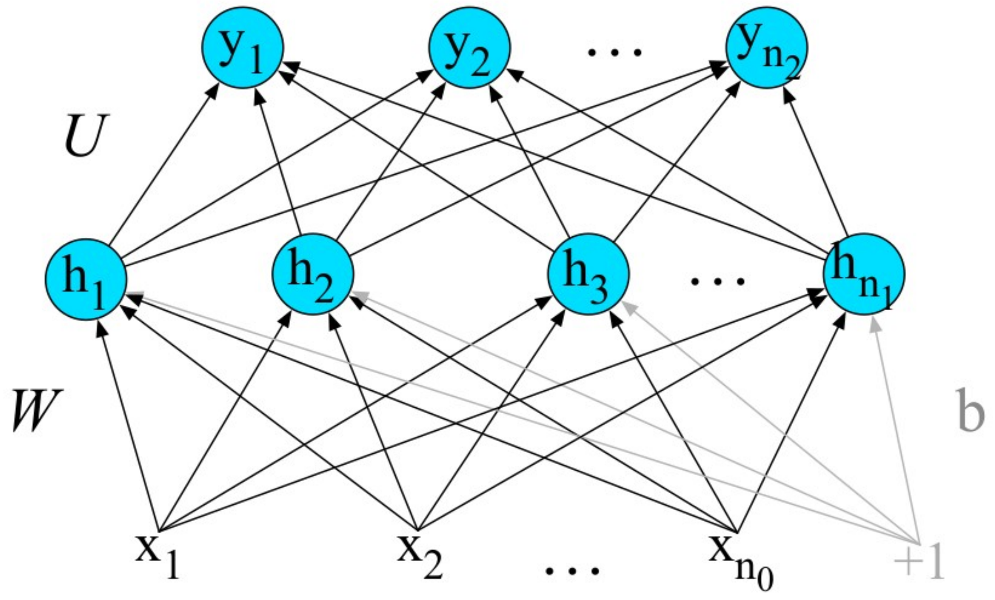
b) The new (linearly separable) *h* space

(With learning: hidden layers will learn to form useful representations)

Multi-Layer (Feedforward Fully Connected) Network

Multi-layer (FFNN, multilayer perceptron)

Can also be called **multi-layer perceptrons** (or **MLPs**) for historical reasons



Basado en https://web.stanford.edu/~jurafsky/slp3/

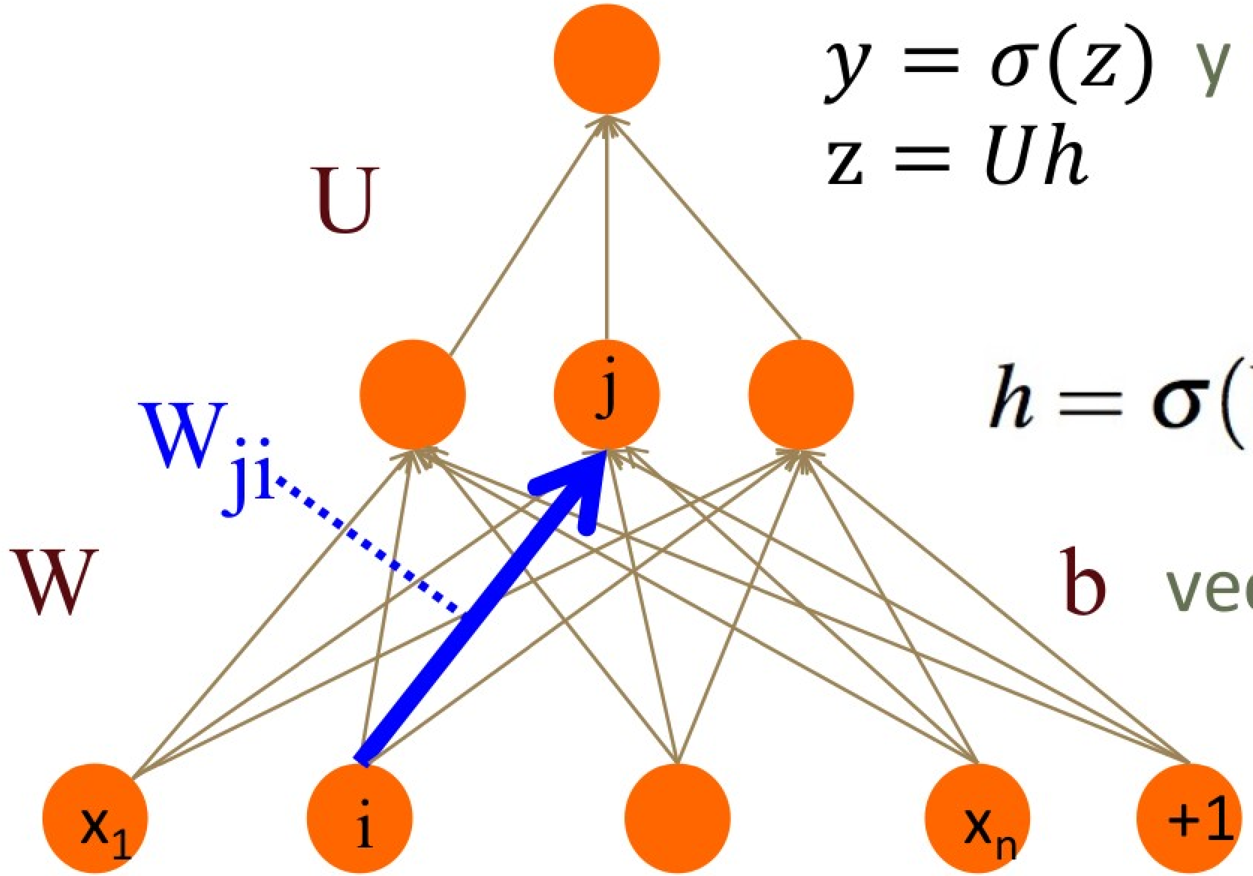
Two-layer Network Example (scalar output)

Output layer
(σ node)

$y = \sigma(z)$ y is a scalar
$z = Uh$

hidden units
(σ node)

$h = \sigma(Wx + b)$

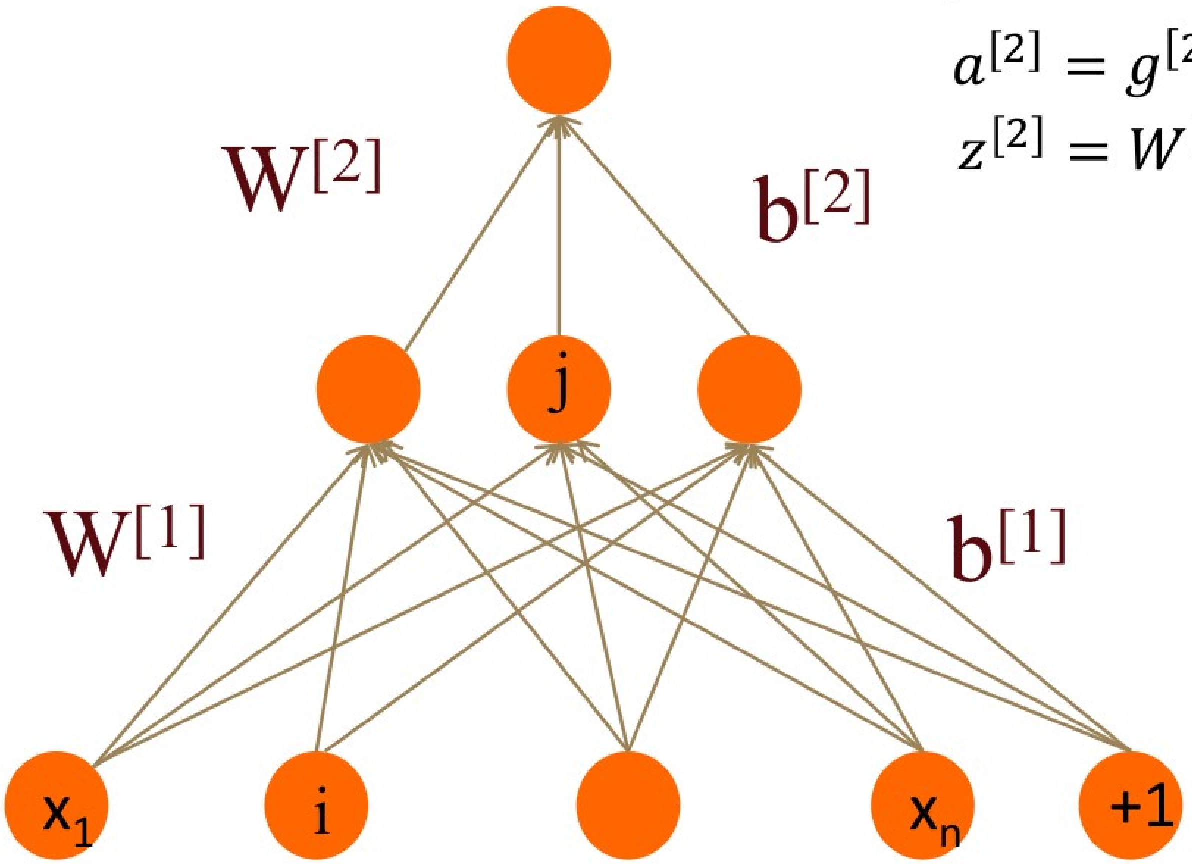


b vector

Input layer
(vector)

Basado en https://web.stanford.edu/~jurafsky/slp3/

Two-layer Network Example



$y = a^{[2]}$

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

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

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

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

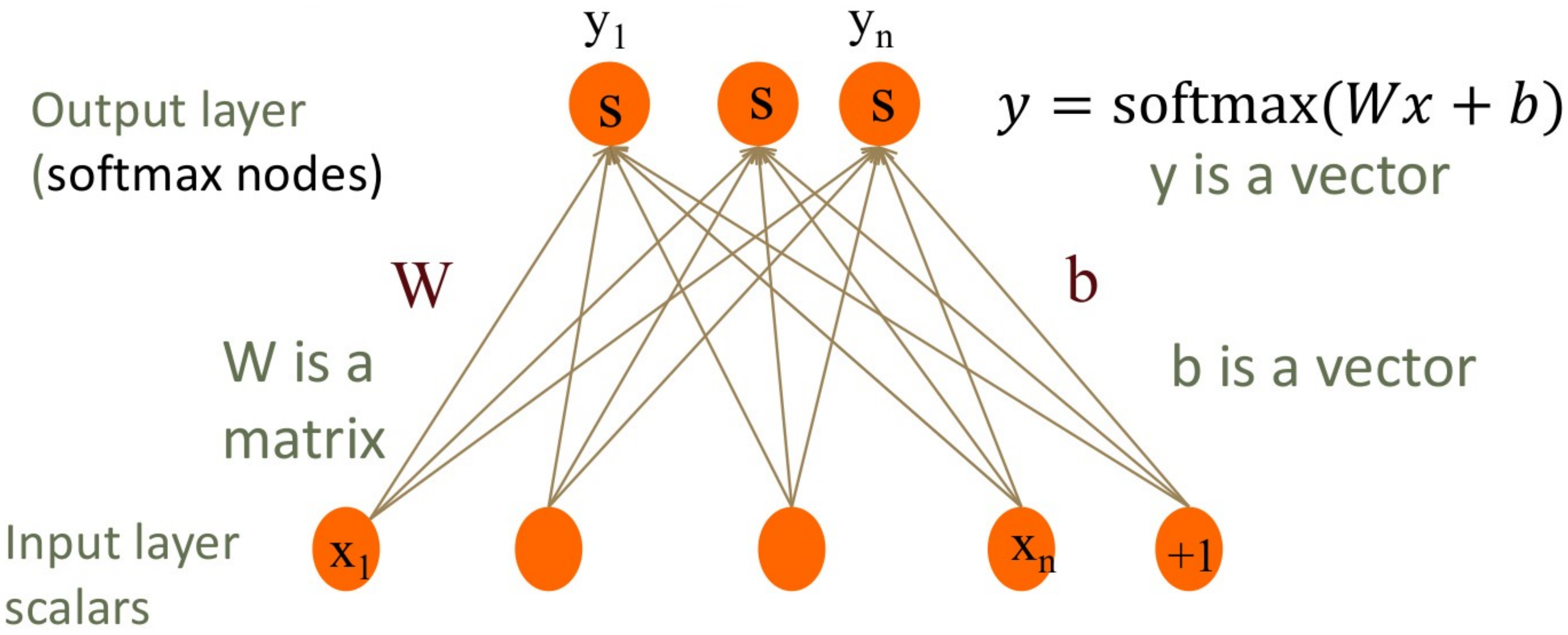
$a^{[0]}$

Basado en https://web.stanford.edu/~jurafsky/slp3/

Multinomial classification (softmax)

Sigmoid – Binary Classification (1 output)

Softmax – Multinomial Classification (k outputs)



Basado en https://web.stanford.edu/~jurafsky/slp3/

Multinomial classification (softmax)

For a vector z of dimensionality k, the softmax is:

$$\text{softmax}(z) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{k} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{k} \exp(z_i)}, \ldots, \frac{\exp(z_k)}{\sum_{i=1}^{k} \exp(z_i)} \right]$$

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{k} \exp(z_j)} \quad 1 \leq i \leq k$$

Example:

$$z = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

$$\text{softmax}(z) = [0.055, 0.090, 0.006, 0.099, 0.74, 0.010]$$

Basado en https://web.stanford.edu/~jurafsky/slp3/

Two-layer Network Example (softmax)

Output layer
(σ node)

$$y = \text{softmax}(z)$$
$$z = Uh$$

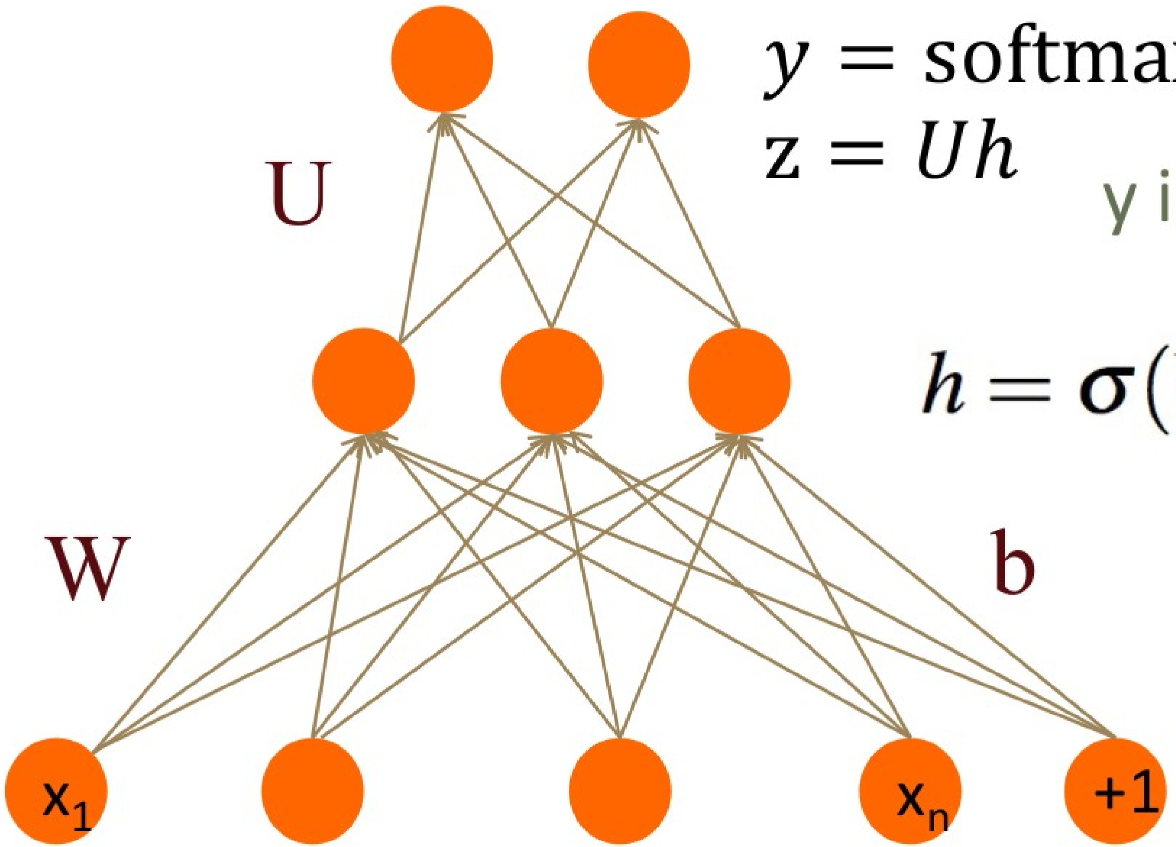
y is a vector

hidden units
(σ node)

$$h = \sigma(Wx + b)$$

Could be ReLU
Or tanh

Input layer
(vector)



Basado en https://web.stanford.edu/~jurafsky/slp3/

Backpropagation (NN training)

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** (Rumelhart, Hinton, Williams, 1986)

- **Backprop** is a special case of **backward differentiation**
- Which relies on **computation graphs**.

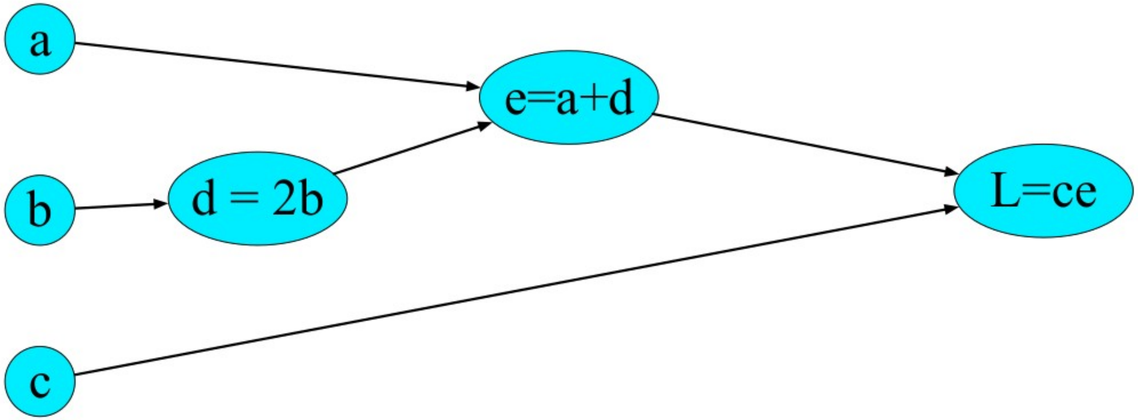
Basado en https://web.stanford.edu/~jurafsky/slp3/

Computation Graph

Example: $L(a,b,c) = c(a+2b)$

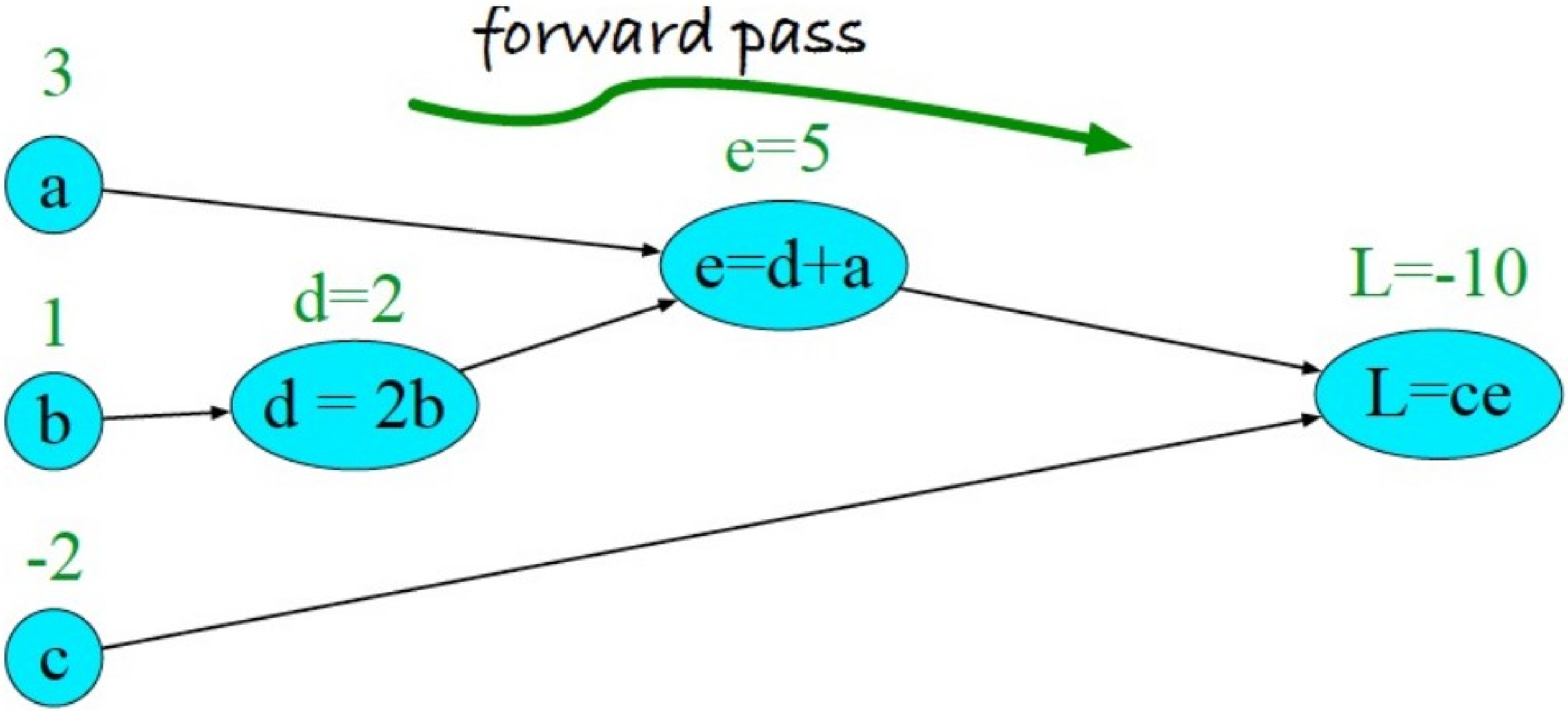
Computations:

$$d = 2*b$$
$$e = a+d$$
$$L = c*e$$



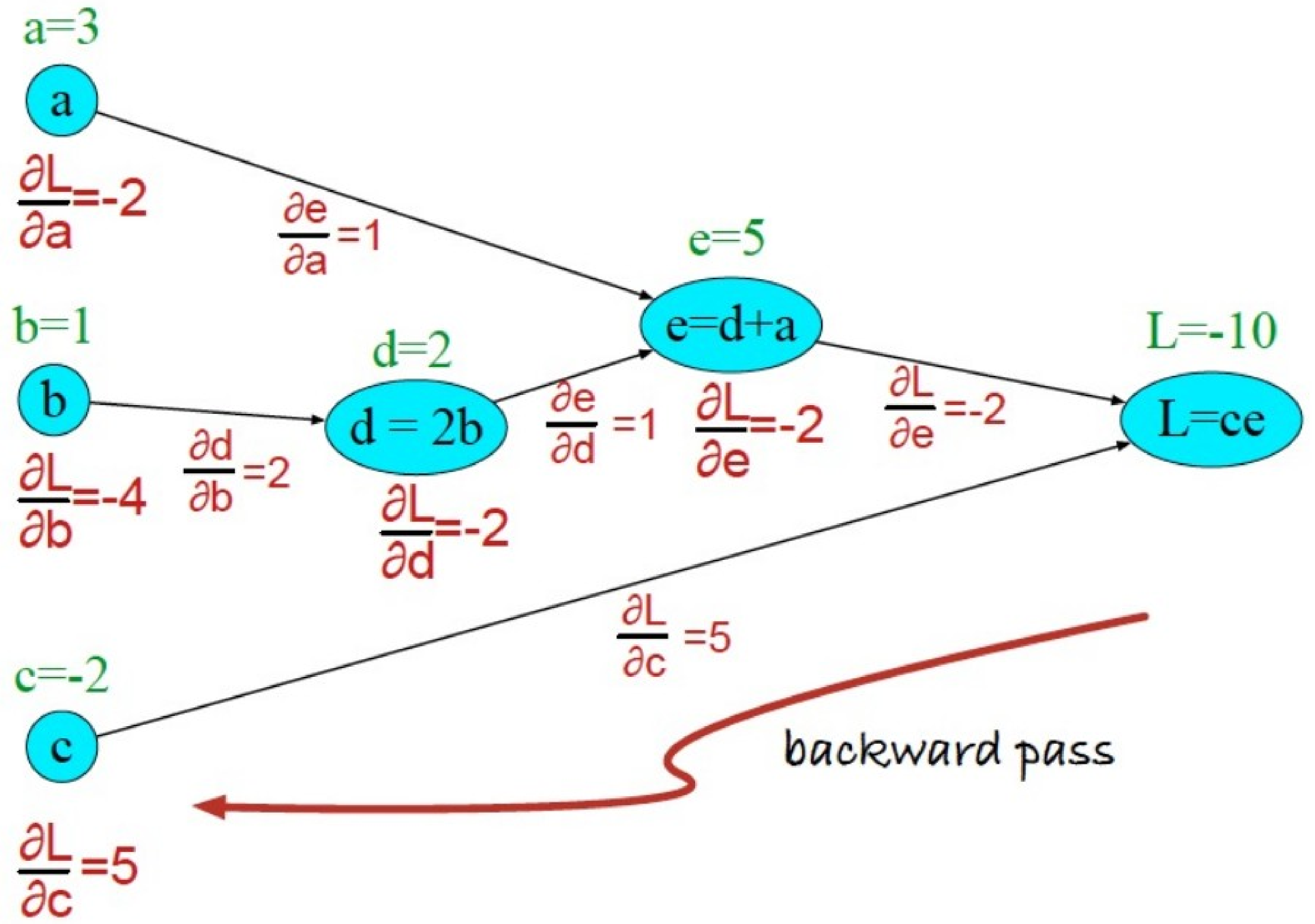
Basado en https://web.stanford.edu/~jurafsky/slp3/

Computation Graph



Basado en https://web.stanford.edu/~jurafsky/slp3/

Computation Graph

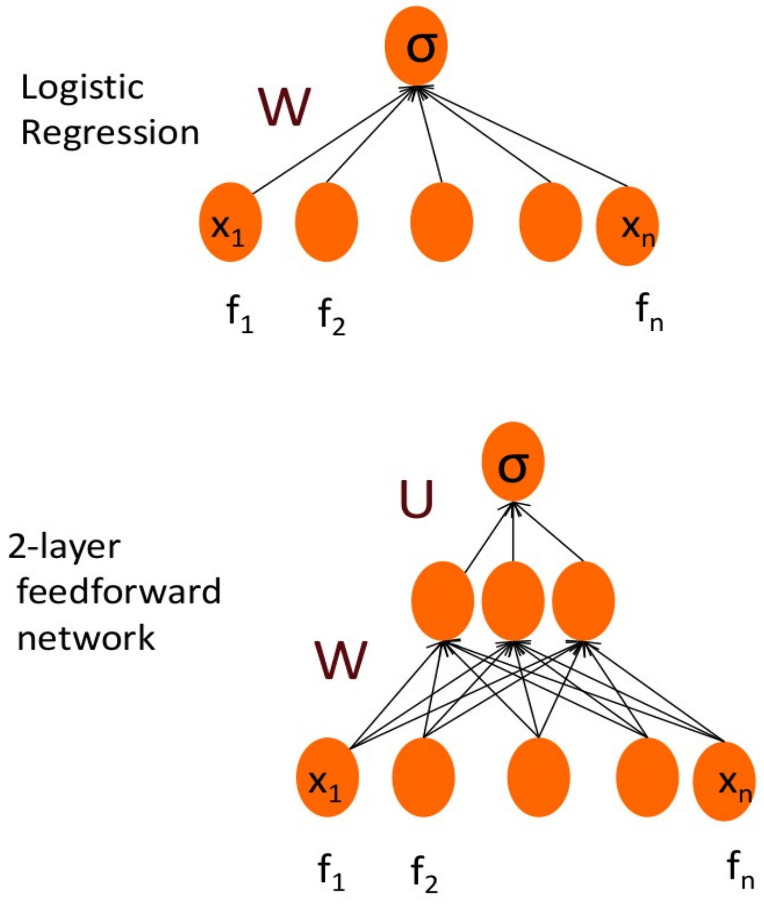


Basado en https://web.stanford.edu/~jurafsky/slp3/

Feedforward Networks for NLP

Text Classification

Use FFNN like a Logistic Regression



Var	Definition
x_1	count(positive lexicon) $\in$ doc)
x_2	count(negative lexicon) $\in$ doc)
x_3	$\begin{cases} 1 & \text{if "no"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$
x_4	count(1st and 2nd pronouns $\in$ doc)
x_5	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$
x_6	log(word count of doc)

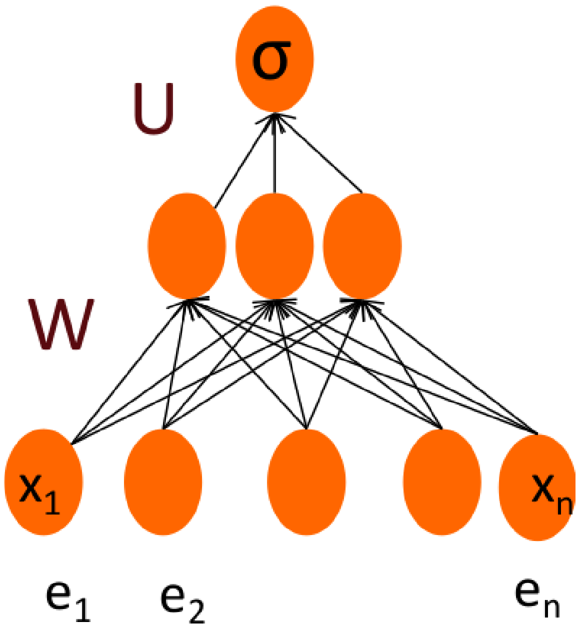
Basado en https://web.stanford.edu/~jurafsky/slp3/

Representation Learning

The real power of deep learning comes from the ability to **learn** features from the data

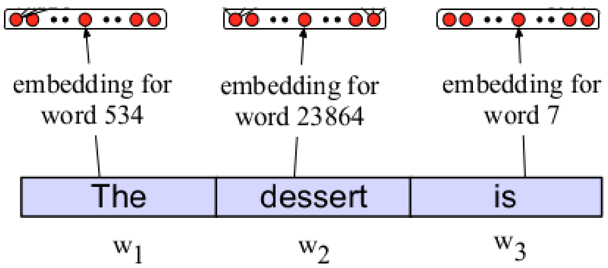
Instead of using hand-built human-engineered features for classification

Use learned representations like embeddings!



Basado en https://web.stanford.edu/~jurafsky/slp3/

Text Classification



This assumes a fixed size length (3)!

Kind of unrealistic.

Some simple solutions (more sophisticated solutions later)

1. Make the input the length of the longest review
   - If shorter then pad with zero embeddings
   - Truncate if you get longer reviews at test time
2. Create a single "sentence embedding" (the same dimensionality as a word) to represent all the words
   - Take the mean of all the word embeddings
   - Take the element-wise max of all the word embeddings
     - For each dimension, pick the max value from all words

Neural Language Model

Neural Language Model

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!

Neural Language Model

Task: predict next word w_t

given prior words w_{t-1}, w_{t-2}, w_{t-3}, …

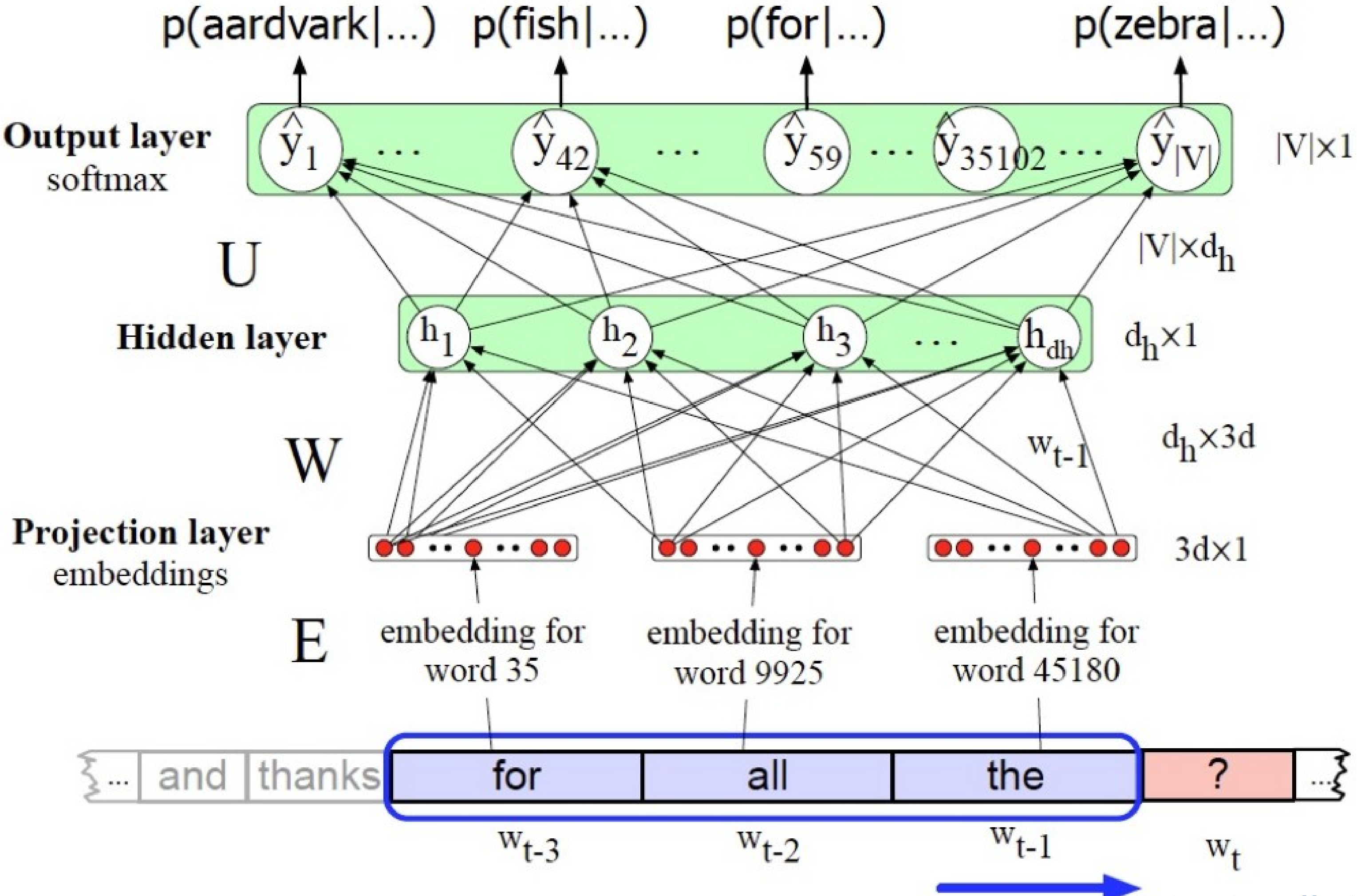
Problem: Now we're dealing with sequences of arbitrary length.

Solution: Sliding windows (of fixed length)

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

Basado en https://web.stanford.edu/~jurafsky/slp3/

Neural Language Model



Basado en https://web.stanford.edu/~jurafsky/slp3/

Why Neural LMs perform better than n-gram LMs?

Training data:

We've seen: I have to make sure that the cat gets fed.

Never seen: dog gets fed

Test data:

I forgot to make sure that the dog gets ___

N-gram LM can't predict "fed"!

Neural LM can use similarity of "cat" and "dog" embeddings to generalize and predict “fed” after dog

Basado en https://web.stanford.edu/~jurafsky/slp3/

PyTorch

- Framework de Machine Learning para python
- Grafo de computación (diferenciación automática)
- Suporte de GPU
- Funcionalidades para Redes Neuronales

Instalación: pip install torch

Ejemplo Simple: Definir FFNN en PyTorch

Usando el módulo nn :

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
      super(Net, self).__init__()
      self.fc1 = nn.Linear(300, 128)
      self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
      x = self.fc1(x)
      x = F.relu(x)
      x = self.fc2(x)
      output = F.softmax(x, dim=1)
      return output
```

Instanciar y ejecutar:

```
# Network execution
my_nn = Net()
result = my_nn(some_input)
print (result)
```

Ejemplo Simple: Definir FFNN en PyTorch

Loss Function:

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(),
                      lr=0.001, momentum=0.9)
```

Training:

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0
```

En la que viene seguimos con NN