2nd year project - Lecture 5

Introduction to Neural Networks for Text and Words as Embeddings

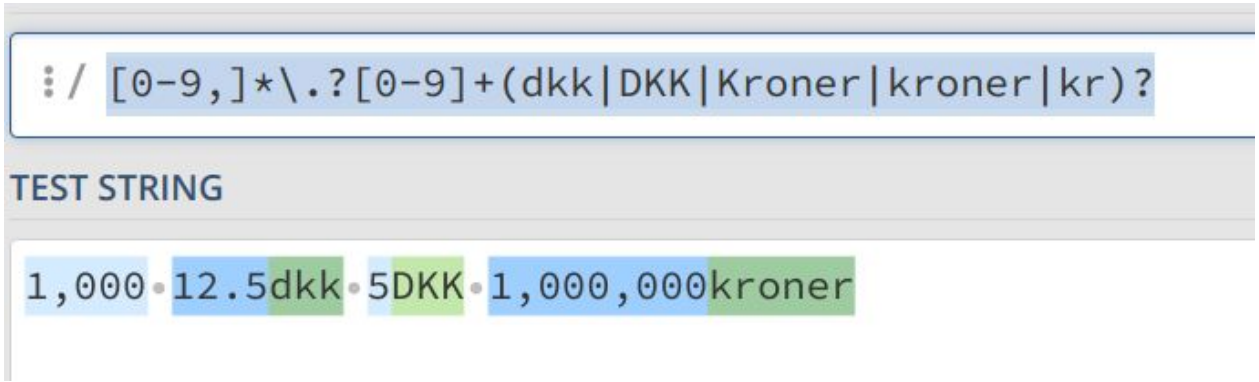
Slides based on Barbara Plank's slides

Until now

- RegEx, command line tools
- Experimental setup, annotation
- Tokenization, POS tagging, classification, language models

Common mistakes assignment 1

- 1b) Write a regular expression that matches numbers, e.g. 12, 1,000, 39.95
- 1c) Expand the previous solution to match Danish prices indications, e.g., ‘1,000 kr‘ or ‘39.95 DKK‘ or ‘19.95‘.



Common mistakes assignment 1

- How many of the names you found start with an uppercased character?

```
grep "B-PER" da_arto.conll grep "B-PER" | grep -c "^[A-Z]"
```

Mistake in assignment 3

I also made a mistake, the transition probabilities are incorrect (for <S>, which effects the others).
Apologies!

Today's overview

- Recap: feedforward neural network
- How to represent words in neural networks?
- CRF layer in neural networks

Learning outcomes

- Know what a feedforward neural network (FFNN) is
- Understand the benefits and limitations of a FFNN
- How to represent words in a neural net
- How does a CRF layer work, why can it be beneficial?

Ask questions anytime on menti.com: 2470 8779

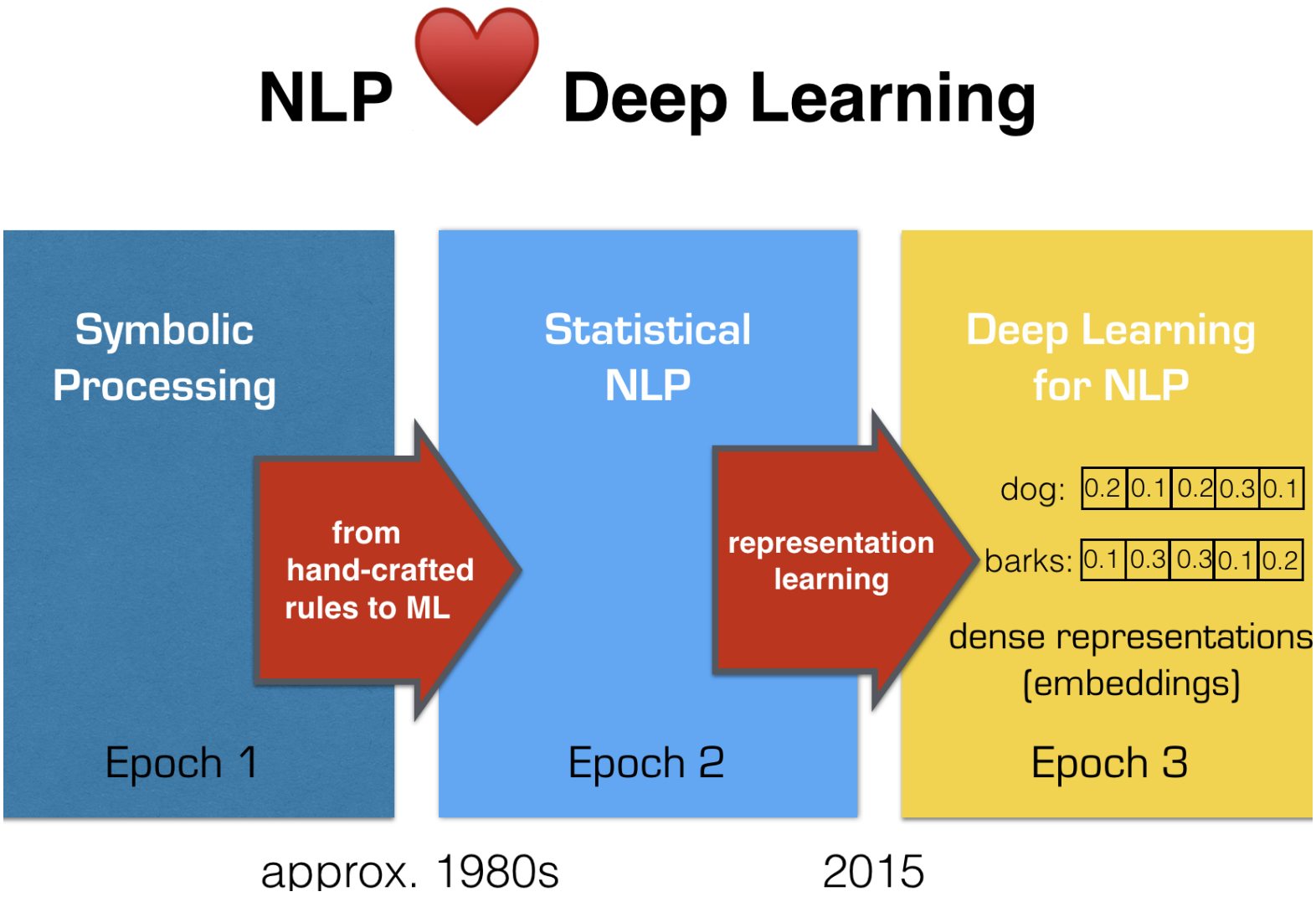
Neural Networks and Deep Learning: A note on terminology

- Deep Learning: is a modern way of calling neural networks, because modern neural nets are often deep (have many layers)

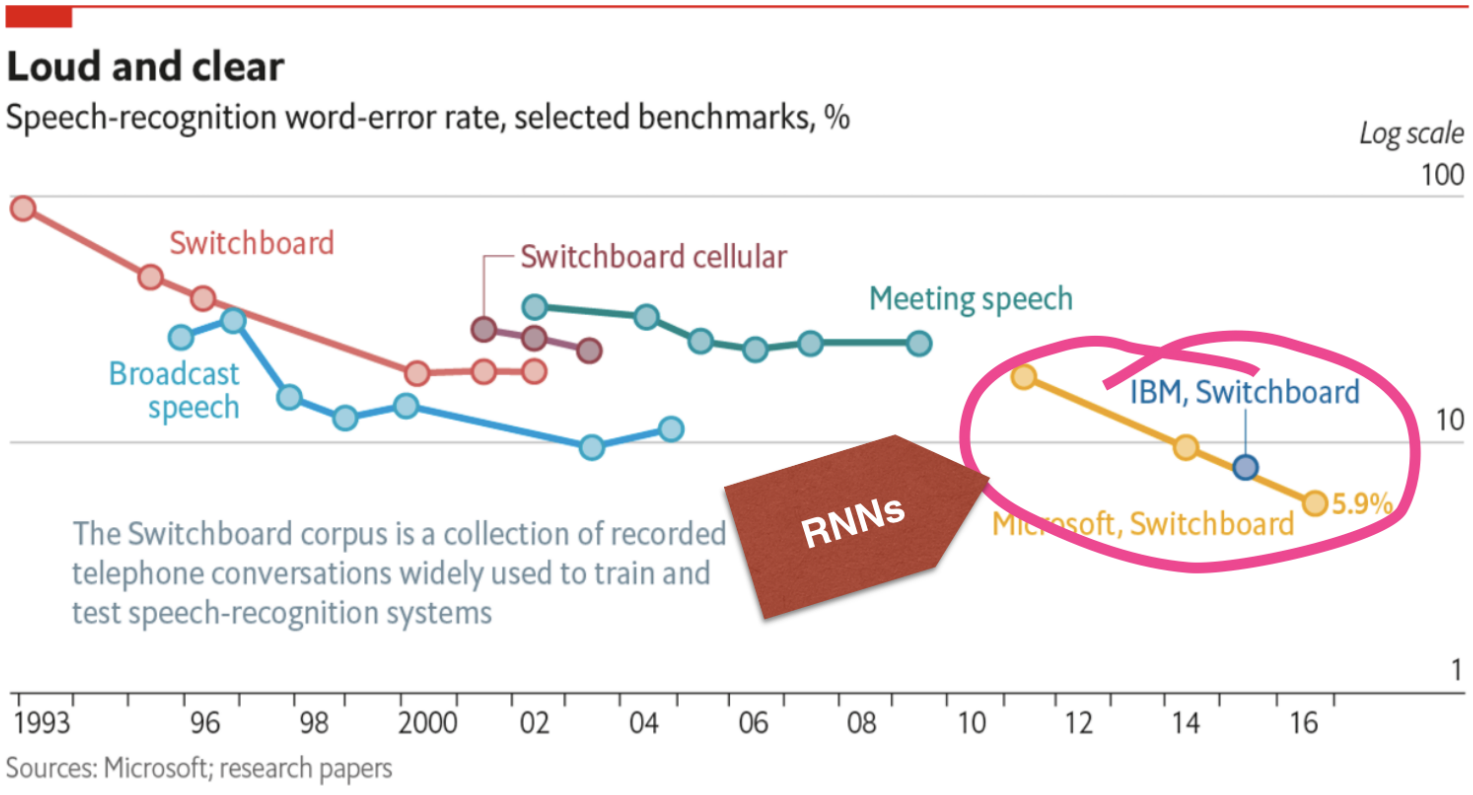
The Success Story of Deep Learning

- State of the art performance for countless real-world tasks (too much to list)
- Huge investements from industry (Google, Facebook, Apple etc.)
- Many new Deep Learning start-ups
- Very active and open research community
- Neural (dense) representations!

Barbara Plank's one slide history of the field (as you saw earlier):



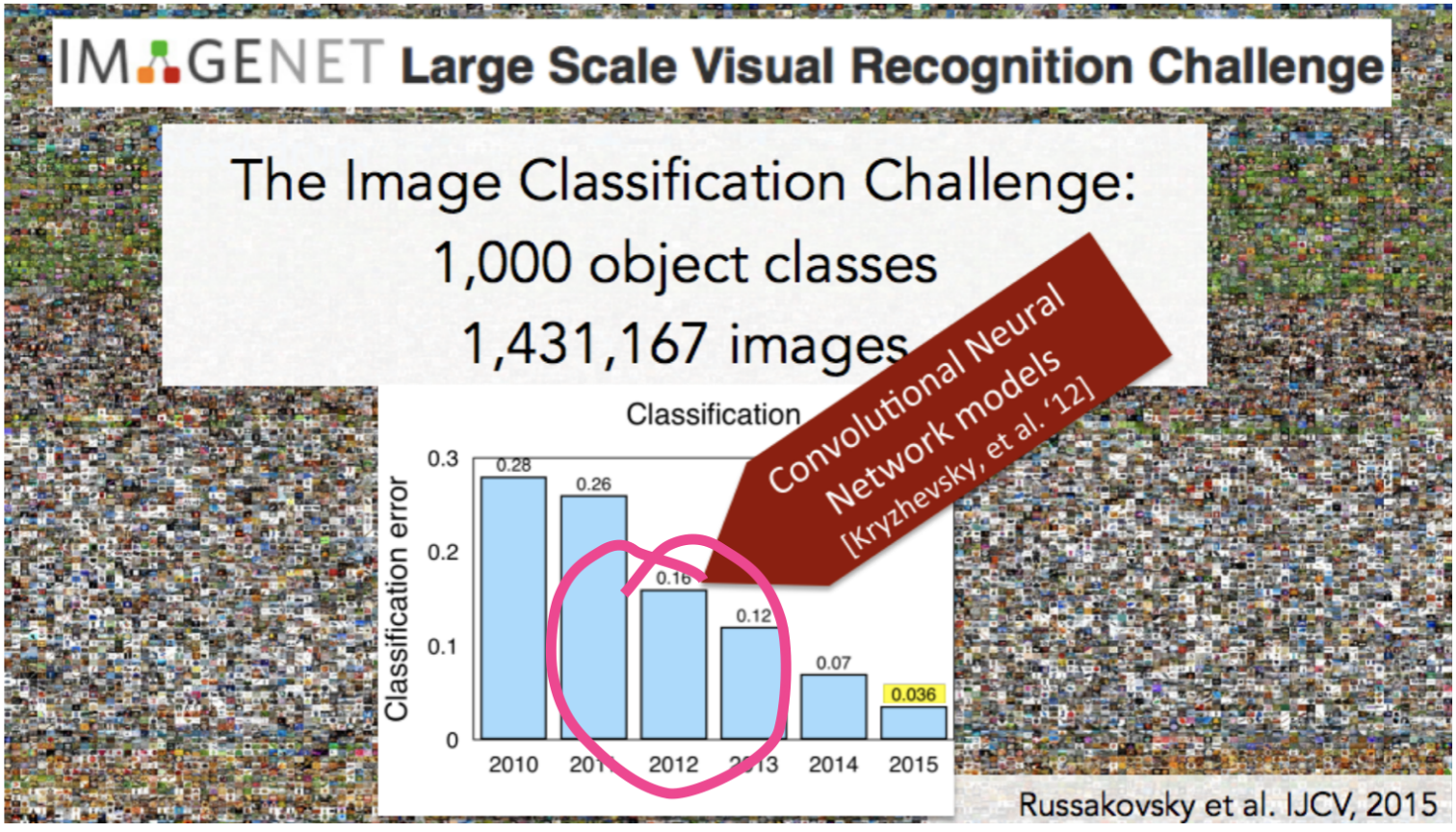
Speech processing



(Source: The Economist)

2010

Computer Vision



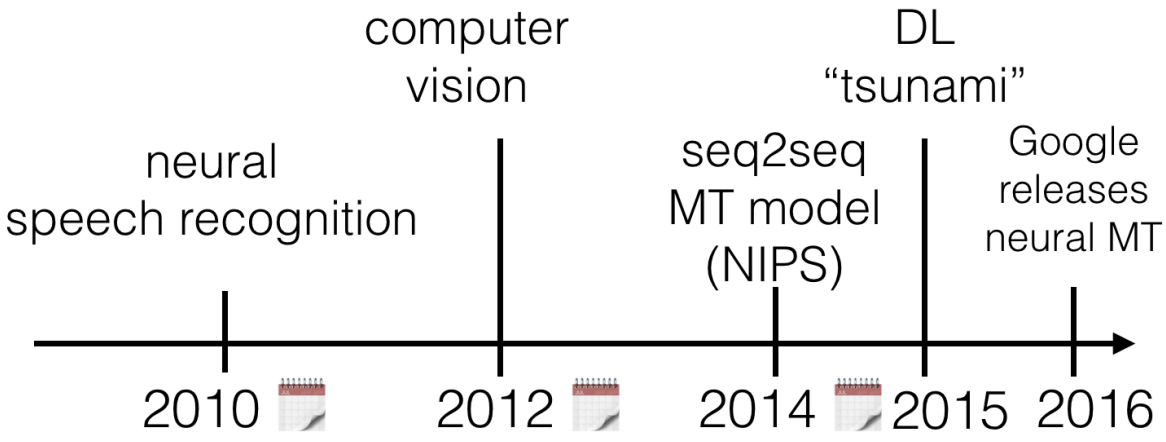
(src: slide by Fei-Fei Li)

2012

Natural Language Processing

The emergence of deep learning

"2015 seems like the year when the full force of the tsunami hit the major NLP conferences"
—Chris Manning (2015)



Recent advances: Elmo (2018), BERT (2019) and transformers (2017)

Traditional Machine Learning for NLP

- Focus on **feature engineering**:
  - What are good features to solve a task?
  - The focus is on developing many rich kinds of *hand-derived* feature templates based on domain knowledge)
  - Coupled with the use of traditional classifiers (e.g., Perceptron, Support Vector Machines, Logistic Regression)

Traditional feature engineering - Example

identity of w_i, identity of neighboring words
embeddings for w_i, embeddings for neighboring words
part of speech of w_i, part of speech of neighboring words
presence of w_i in a **gazetteer**
w_i contains a particular prefix (from all prefixes of length ≤ 4)
w_i contains a particular suffix (from all suffixes of length ≤ 4)
word shape of w_i, word shape of neighboring words
short word shape of w_i, short word shape of neighboring words
gazetteer features

Figure 8.15 Typical features for a feature-based NER system.

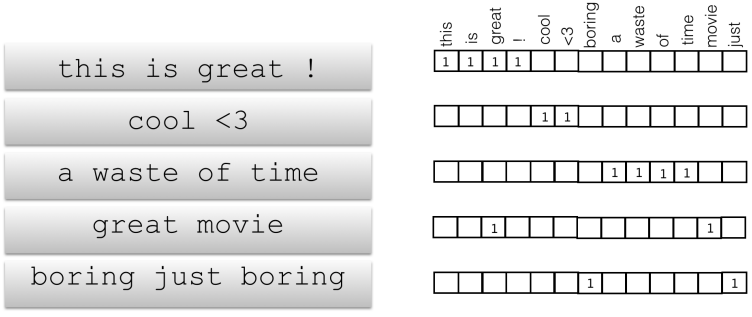
Modern features based on induced dense representations

- instead of using hand-derived features, build neural networks which take raw words as inputs and learn to induce features as part of the processing of learning to classify
- (**representation learning**)

From sparse binary to dense continuous feature representations

From sparse binary:

- $\mathbb{V}$ = vocabulary
- $f_{sb}(this) = [1, 0, .., 0]$ with $i = 1$ for word at index i and $len(f_{sb}) = |\mathbb{V}|$



"Bag of Words"

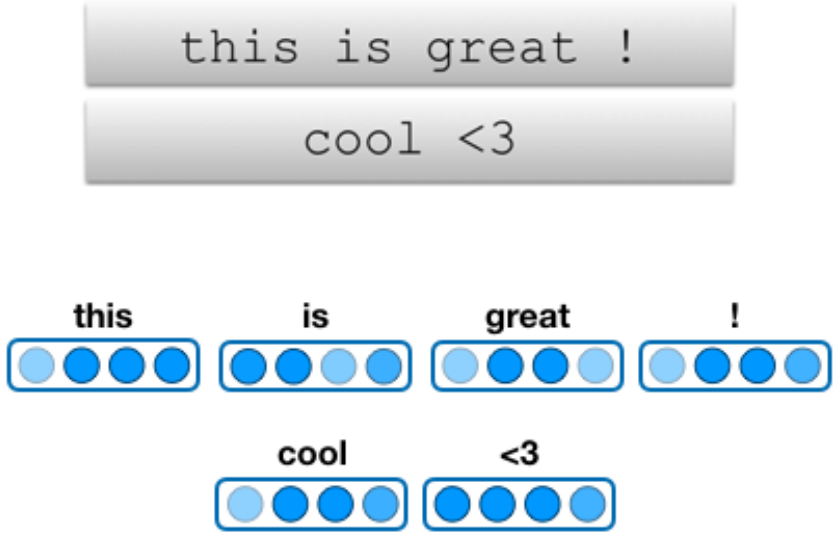
Note: not only words as features, see other examples in Figure 7.10 in J&M (word count, lexicon lookup...).

From sparse binary to dense continuous feature representations

To dense continuous (dc) features, e.g. word embeddings:

- $\mathbb{V} =$ vocabulary
- $d = 4$
- $f_{sb}(\text{this}) = [0.7, 0.8, 0.11, 0.3]$ with $len(f_{dc}) = d$

Dense continuous text encodings



How to combine word embeddings?

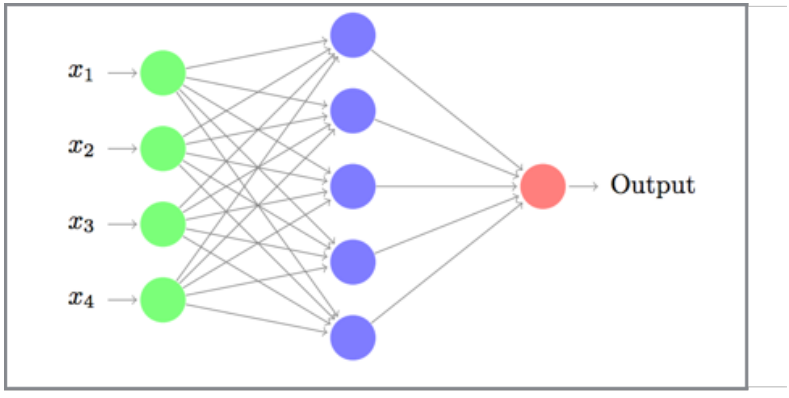
Modern features based on induced dense representations

- Word embeddings trained with the end task (lookup parameters) are one example of induced feature representations

  - We will get to word embeddings, what they are, and how we can train and use them from scratch, next week. To understand them, we need a profound knowledge of the basics of neural networks which we will see today
  - We will see the basic idea of a word embedding today (part 2)

A neural network

- A neural network is a **network** of neurons. A neuron is a basic computational unit. Each set of units (or neurons) at a certain depth represents a **layer** in the network.
- **Feedforward Neural Network (FFNN)** - it is called this way as its computation proceeds iteratively from one layer of neurons (or units) to the next.



- A FFNN has a set of *input* nodes, *output* node(s) and usually one or more layers with *hidden nodes*.

Different view of a Neural Network

From biological neurons to artificial neural networks

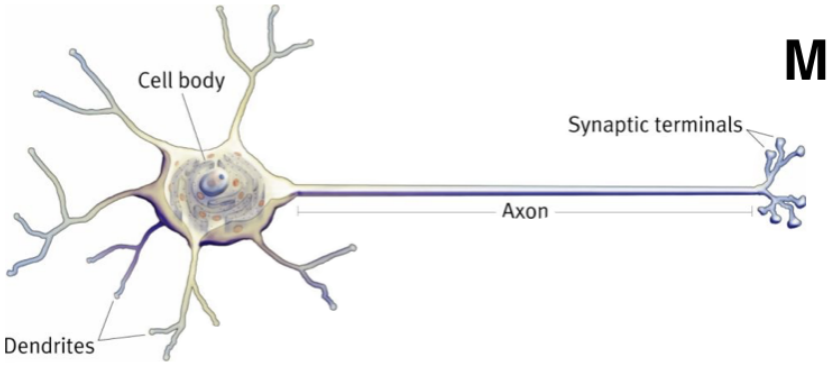
- **Cognitive science view**: a computational model inspired by the basic building blocks of the brain, the neurons (a brain-inspired metaphor)

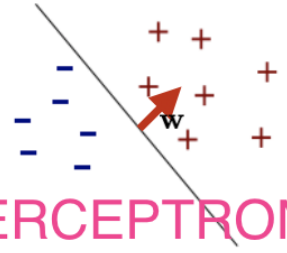
Different view of a Neural Network

From biological neurons to artificial neural networks

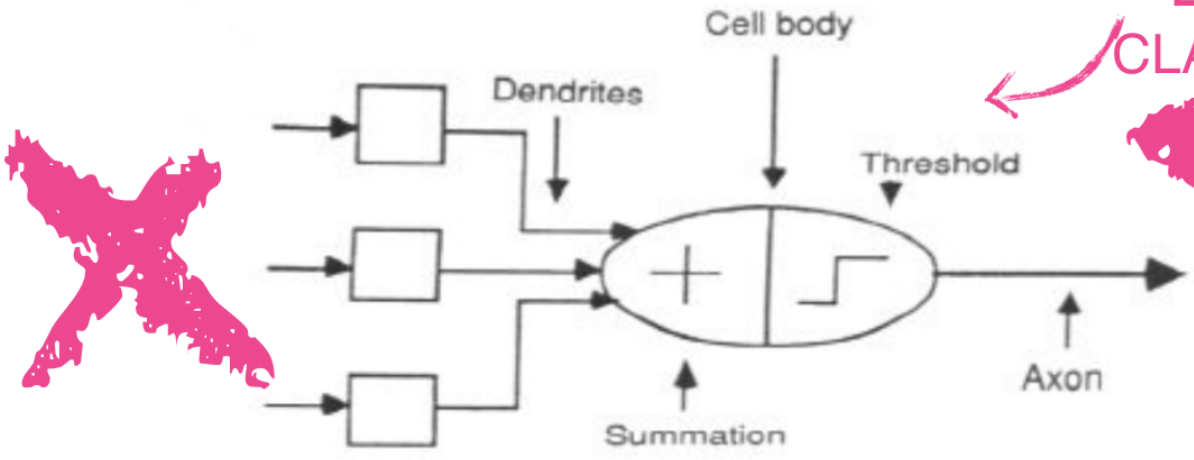
- **Cognitive science view**: a computational model inspired by the basic building blocks of the brain, the neurons (a brain-inspired metaphor)
- **Machine Learning view**:
  - **Linear algebra view**: function applications (algebraic operations)
  - **Two Graphical views**:
    - static view: a network of layers of neurons, math view as visualization
    - dynamic view: a computational graph (with automatic differentiation) for computation

McCulloch & Pitt (1943)



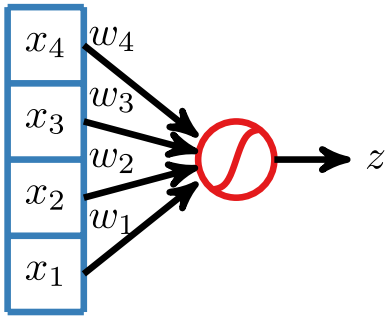


PERCEPTRON:
LINEAR
CLASSIFIER



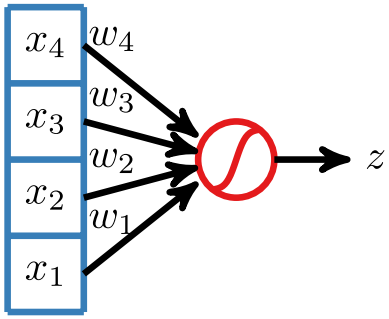
A Single Neuron (Unit) - a basic building block

A single neuron computes a number (a value) z (note: bias node b is commonly omitted in visualization, imagine an additional node with input value $+1$ and weight b):



A Single Neuron (Unit) - a basic building block

A single neuron computes a number (a value) z (note: bias node b is commonly omitted in visualization, imagine an additional node with input value $+1$ and weight b):



Mathematically:

$$z = \begin{cases} \sigma(w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + w_4 * x_4 + b) \\ = \sigma(\sum_{i=1}^{4} w_i x_i + b) \end{cases}$$

So at its heart, a neuron computes the value of z as the **weighted sum** of its inputs, plus the bias term. In other words, z is a **linear function** of x followed by a non-linear function application σ.

N.B. In the book (section 7.1) these two steps are explicitly teased apart, with a slighlty different notation: (weighted sum of inputs, with bias)

$$z = \sum_{i=1}^{4} w_i x_i + b$$

Applying σ:

$$y = a = \sigma(z)$$

What is σ?

In the case of the perceptron, σ is a **threshold** function. Intuitively, the perceptron only fires if the weighted sum is above some threshold. We can formulize this intuition as:

$$y = \begin{cases} 1 & \text{if}(\sum_j w_j x_j) > \textit{threshold} \\ 0 & \text{otherwise} \end{cases}$$

Hence, the perceptron is a **linear** classifier (no non-linearities are involved)

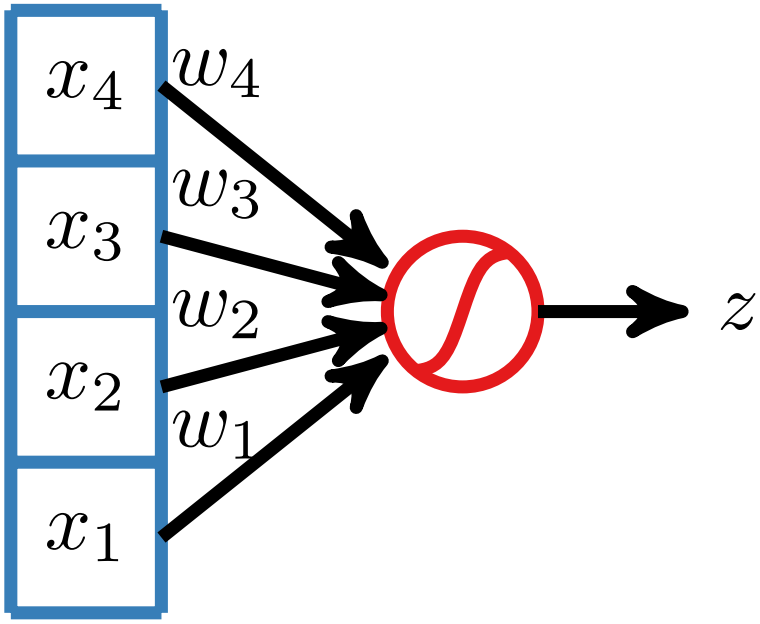
Let us rewrite the equation of the perceptron. First, notice that $\sum_{j=1} w_j x_j$ is the **dot product** of the weights and input, and can be written as:

$$\sum_{j=1} w_j x_j = \vec{w} \cdot \vec{x}$$

where $\vec{w}$ and $\vec{x}$ are now vectors. If it is clear from context we avoid the explicit vector notation and simply write: $w \cdot x$.

So, for our example above we get:

$$z = \begin{cases} \sigma((\sum_{i=1}^{4} w_i x_i) + b) \\ = \sigma(w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + w_4 * x_4 + b) \\ = \sigma(\mathbf{w} \cdot \mathbf{x} + b) \quad \text{with } \mathbf{w}, \mathbf{x} \in \mathbb{R}^4 \end{cases}$$



We can move the threshold weight inside w (to simplify later the computation) by introducing b the bias term $b = -threshold$. Using these two changes, the equation changes from:

$$y = \begin{cases} 1 & \text{if}(\sum_j w_j x_j) > threshold \\ 0 & \text{otherwise} \end{cases}$$

to:

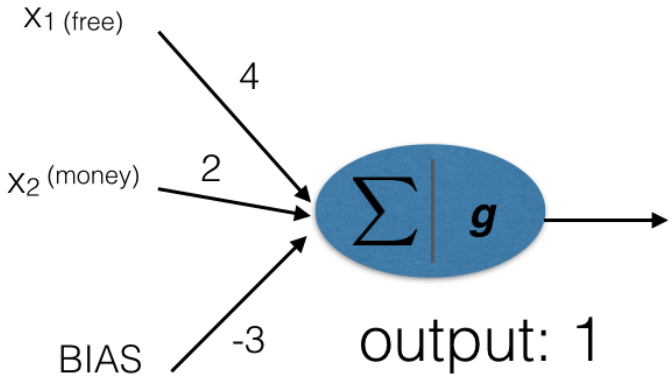
$$y = \begin{cases} 1 & \text{if}(\mathbf{w} \cdot \mathbf{x}) > 0) \\ 0 & \text{otherwise} \end{cases}$$

(where $\mathbf{w}$ has one more feature with constant input value 1 for the bias weight):

Example: Spam (spam=1)

Imagine we have the following features and weights:

free
money
BIAS (fixed value, always on)



$\mathbf{x}$

"free money"

FEATURE EXTRACTOR

BIAS	1
free	1
money	1
...	

$\phi(\mathbf{x})$

WEIGHTS

BIAS	-3
free	4
money	2
...	

$\mathbf{w}$

$(1)(-3) + (1)(4) + (1)(2) + \ldots = 3$

ACTIVATION

$$\sum_{j=1}^{m} \mathbf{w_j} \cdot \phi_\mathbf{j}(\mathbf{x})$$

Common activation functions (non-linearities)

- sigmoid $y = \sigma(z) = \frac{1}{1+e^{-z}}$, maps to [0,1]
- tanh, maps to (-1,+1)
- relu $y = max(x, 0)$, maps x to x if it is positive and 0 otherwise.

In practice, relu $>$ tanh $>$ sigmoid

Common activation functions (non-linearities)

- sigmoid $y = \sigma(z) = \frac{1}{1+e^{-z}}$, maps to [0,1]
- tanh, maps to (-1,+1)
- relu $y = max(x, 0)$, maps x to x if it is positive and 0 otherwise.

In practice, relu $>$ tanh $>$ sigmoid

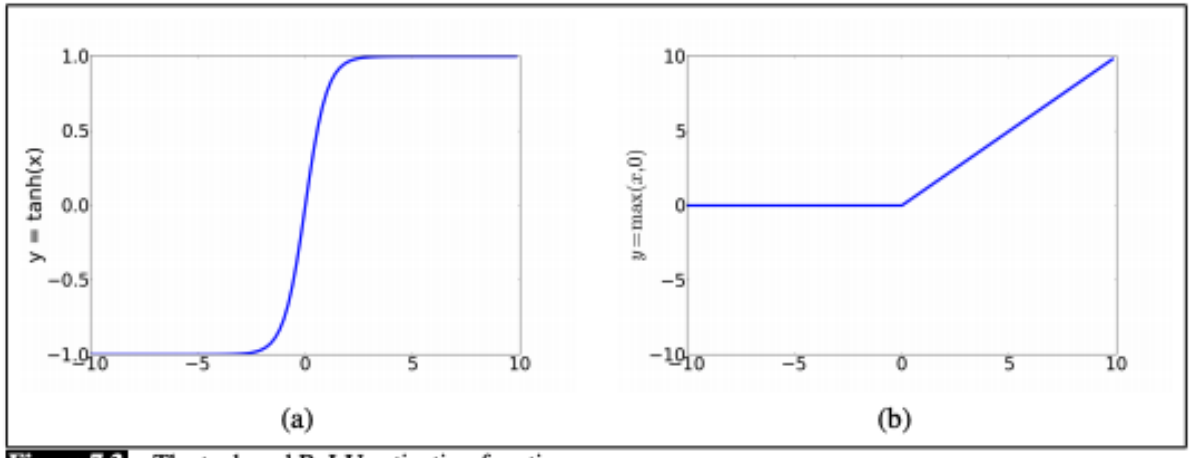


Figure 7.3 The tanh and ReLU activation functions.

Src: chapter 6 J&M

Vectorization

Note that

$$\sum_{j=1} w_j x_j = \vec{w} \cdot \vec{x}$$

is not only simpler in notation, calculating with vectors (/matrices) directly is much more efficient

```
import torch
inputs = torch.tensor([[0,0],[0,1],[1,0],[1,1]])

def compute(input_matrix):
    # The parameters of the model:
    W = torch.tensor([[-2],[-2]])
    b = 3
    # Matrix multiplication
    a = input_matrix.mm(W) + b
    # Sigmoid
    labels = [1 if elem > 0 else 0 for elem in a]
    return labels

labels = compute(inputs)
#[1, 1, 1, 0]
```

```
import torch
inputs = torch.tensor([[0,0],[0,1],[1,0],[1,1]])

def compute(input_matrix):
    # The parameters of the model:
    W = torch.tensor([[-2],[-2]])
    b = 3
    # Matrix multiplication
    a = input_matrix.mm(W) + b
    # Sigmoid
    labels = [1 if elem > 0 else 0 for elem in a]
    return labels

labels = compute(inputs)
#[1, 1, 1, 0]
```

Which logical function does this network implement?

```
import torch
inputs = torch.tensor([[0,0],[0,1],[1,0],[1,1]])

def compute(input_matrix):
    # The parameters of the model:
    W = torch.tensor([[-2],[-2]])
    b = 3
    # Matrix multiplication
    a = input_matrix.mm(W) + b
    # Sigmoid
    labels = [1 if elem > 0 else 0 for elem in a]
    return labels

labels = compute(inputs)
#[1, 1, 1, 0]
```

Which logical function does this network implement?

Beautiful! The NAND logical function (not AND).

Truth tables of logical 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

Src: chapter 6 J&M

Visualization

We can visualize the example by looking at where the input vectors are in the space and which label they get.


1
0
0
1

Remember, the perceptron is a **linear** classifier:

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

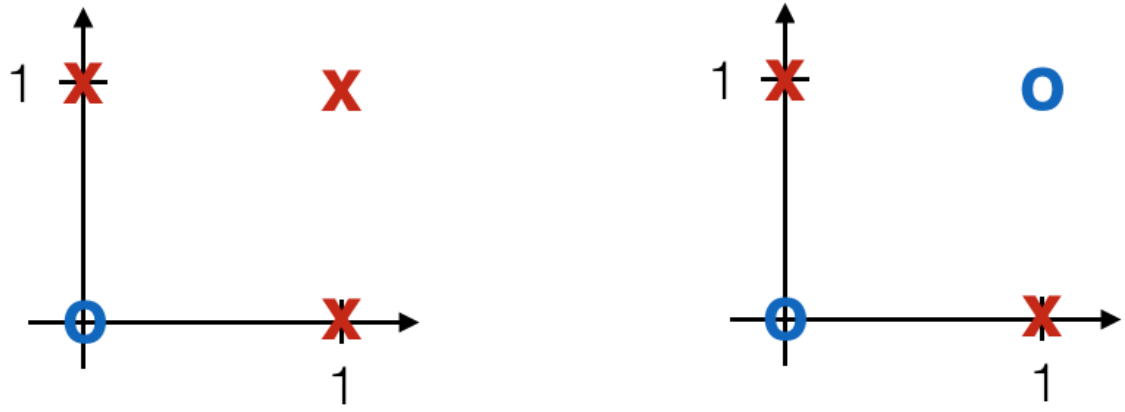
Remember, the perceptron is a **linear** classifier:

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

To be precise, since the inputs have usually more than 2 dimensions it is actually a **hyperplane**. Imagine the line in our NAND example.

Linearly separable?

Now have a look at the following two examples. Are they linearly separable? (hint: Which logical functions do they represent?)



left: OR, right: XOR

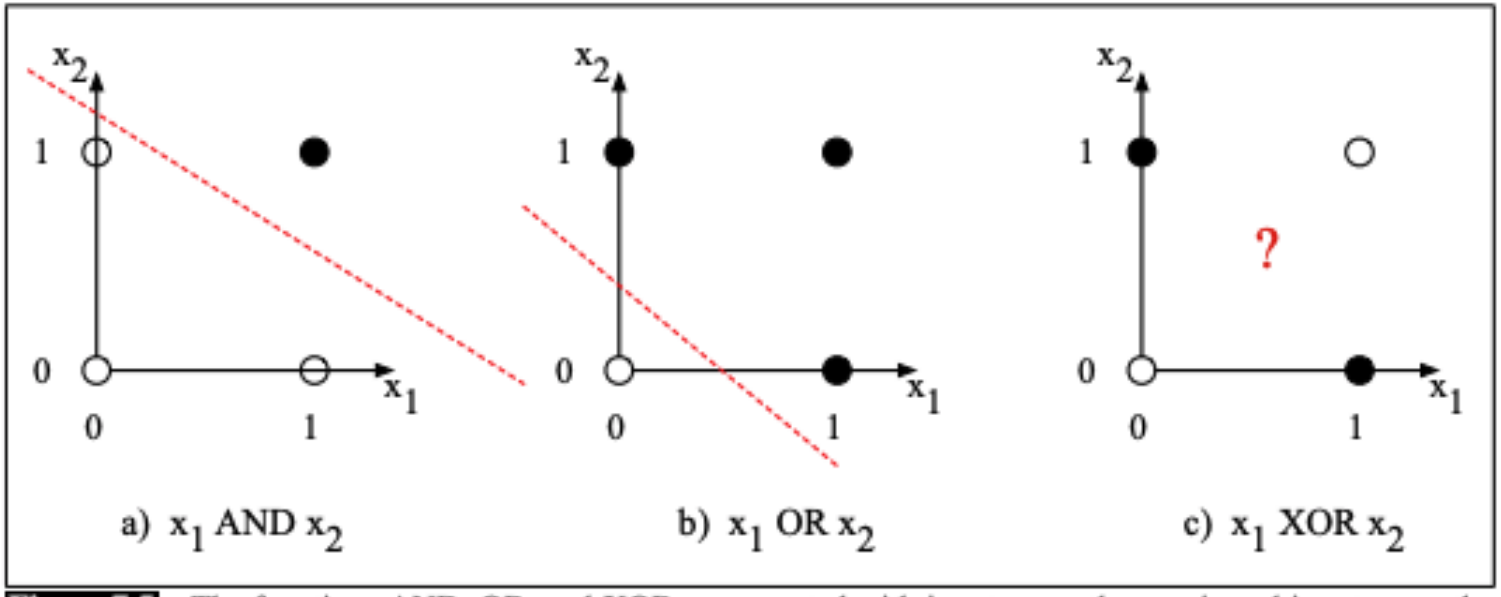


Figure 7.5 The functions AND, OR, and XOR, represented with input x_1 on the x-axis and input x_2 on the y axis. Filled circles represent perceptron outputs of 1, and white circles perceptron outputs of 0. There is no way to draw a line that correctly separates the two categories for XOR. Figure styled after Russell and Norvig (2002).

Src:

J&M, chp.7

A limitation of the perceptron. XOR is not a **linearly separable** function (Minsky and Papert, 1969). So what can we do about it?

A limitation of the perceptron. XOR is not a **linearly separable** function (Minsky and Papert, 1969). So what can we do about it?
The solution: neural networks For example to solve the XOR case, it can be done by using a layered **network of units**.Key idea: move to a model with higher **representational capacity**.

Why Neural Networks

- **non-linearity**
- **representational power**

Multiple Neurons

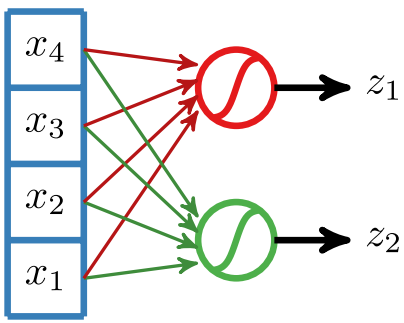
$$z_1 = \text{sigmoid}(\mathbf{w_1} \cdot \mathbf{x} + b_1)$$

$$z_2 = \text{sigmoid}(\mathbf{w_2} \cdot \mathbf{x} + b_2)$$

Multiple Neurons

$$z_1 = \text{sigmoid}(\mathbf{w_1} \cdot \mathbf{x} + b_1)$$

$$z_2 = \text{sigmoid}(\mathbf{w_2} \cdot \mathbf{x} + b_2)$$



What is $\mathbf{w_2}$?

Note: typically we refer to the set of weights from one layer to the next as $\mathbf{W}$. So here it comprises both $\mathbf{w_1}$ and $\mathbf{w_2}$.

Note: Different naming than in the book (J&M) where ‘z‘ is **before** the activation is applied, and ‘a‘ is the output of the activation (which here we denote ‘z‘):

$$z_1 = \text{sigmoid}(\mathbf{w_1} \cdot \mathbf{x} + b_1)$$

In the book, it is a instead (**the activation value**):

$$z_1 = \mathbf{w_1} \cdot \mathbf{x} + b_1$$

$$a = \text{sigmoid}(z_1)$$

Solution to the XOR problem after Goodfellow et al. (2016)

A solution for XOR with 2 units/neurons

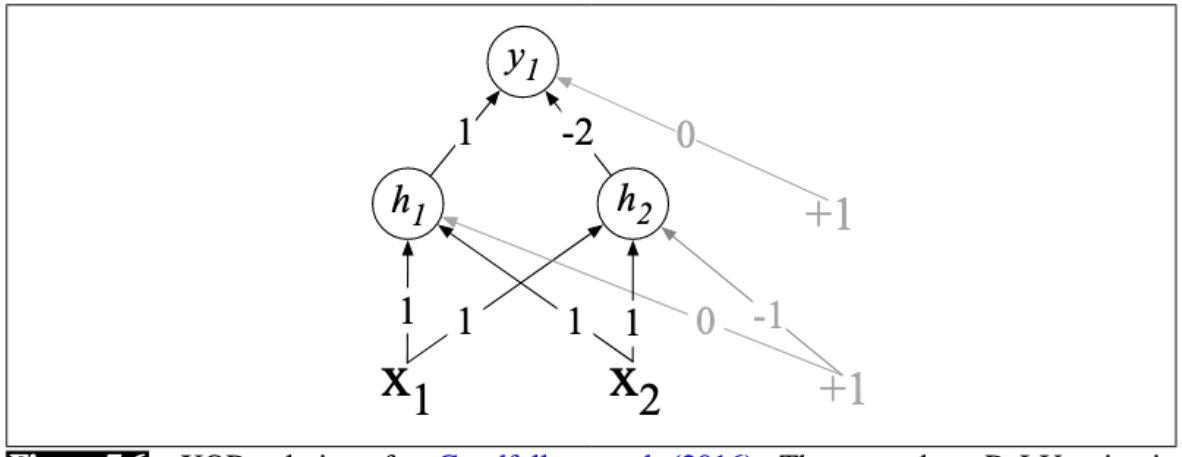


Figure 7.6 XOR solution after Goodfellow et al. (2016). There are three ReLU units, in two layers; we've called them h_1, h_2 (h for "hidden layer") and y_1. As before, the numbers on the arrows represent the weights w for each unit, and we represent the bias b as a weight on a unit clamped to +1, with the bias weights/units in gray.

Src: J&M, chp.7

A new representation for the input

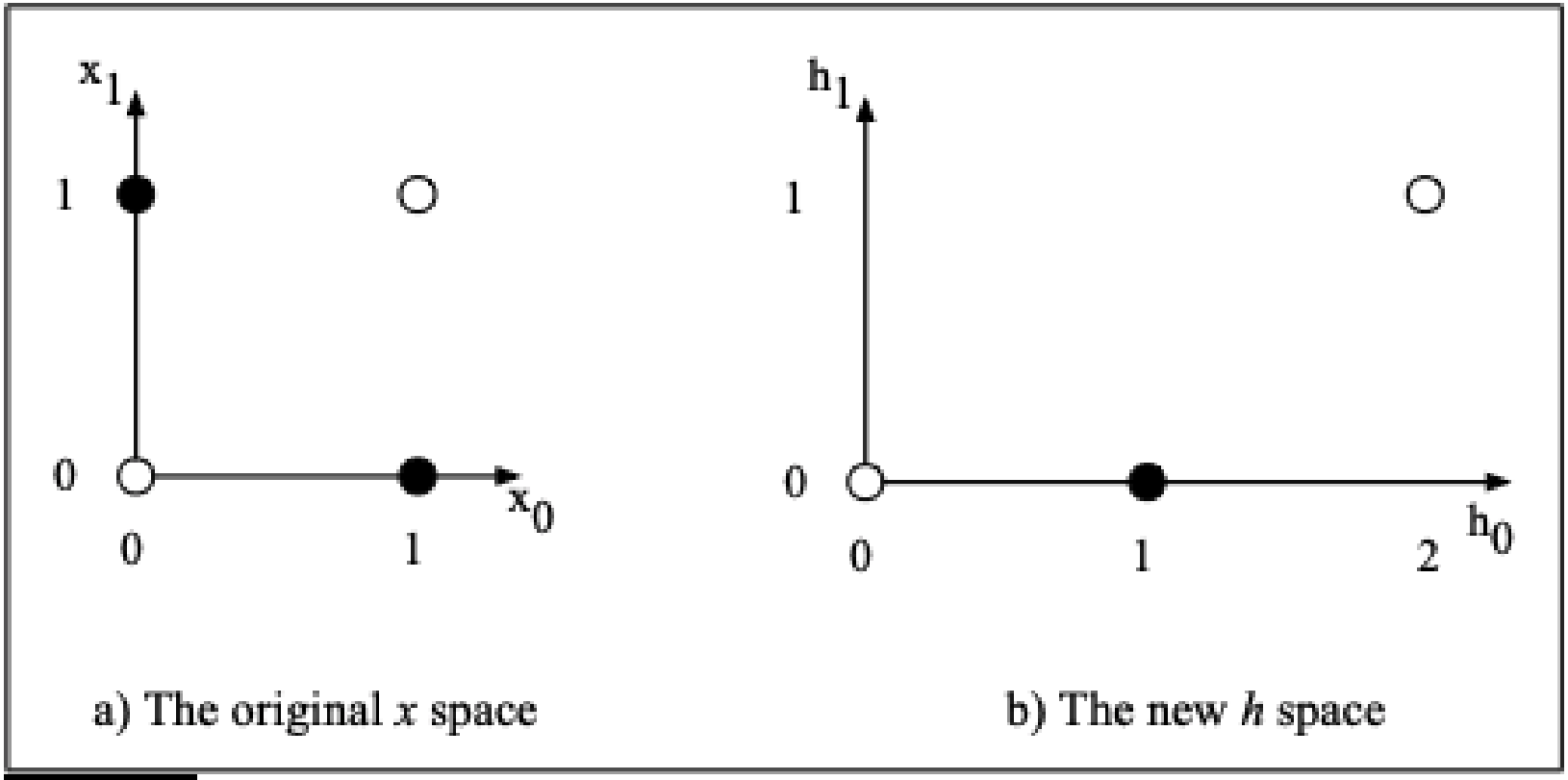


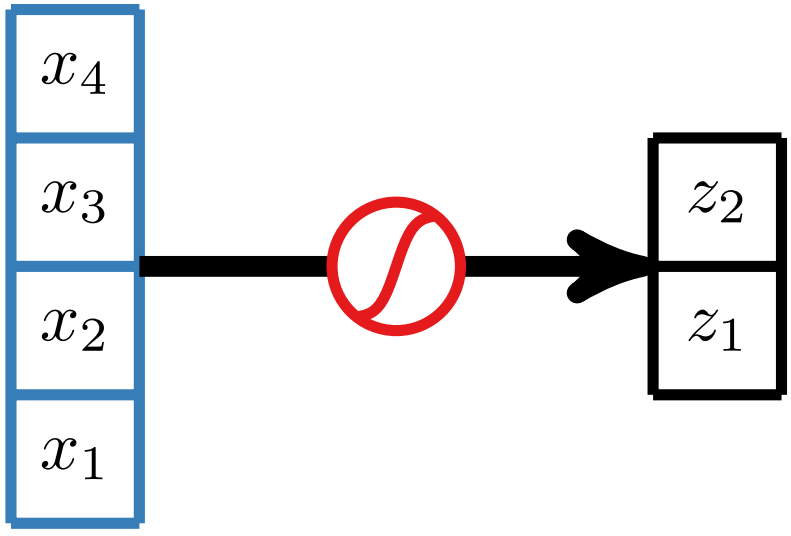
Figure 7.7 The hidden layer forming a new representation of the input. Here is the representation of the hidden layer, h, compared to the original input representation x. Notice that the input point [0 1] has been collapsed with the input point [1 0], making it possible to linearly separate the positive and negative cases of XOR. After Goodfellow et al. (2016).

Src: J&M, chp.7

Multiple Neurons - Another view: Projections

$f_\theta : \mathbb{R}^4 \rightarrow \mathbb{R}^2$

Multiple Neurons - Another view: Projections

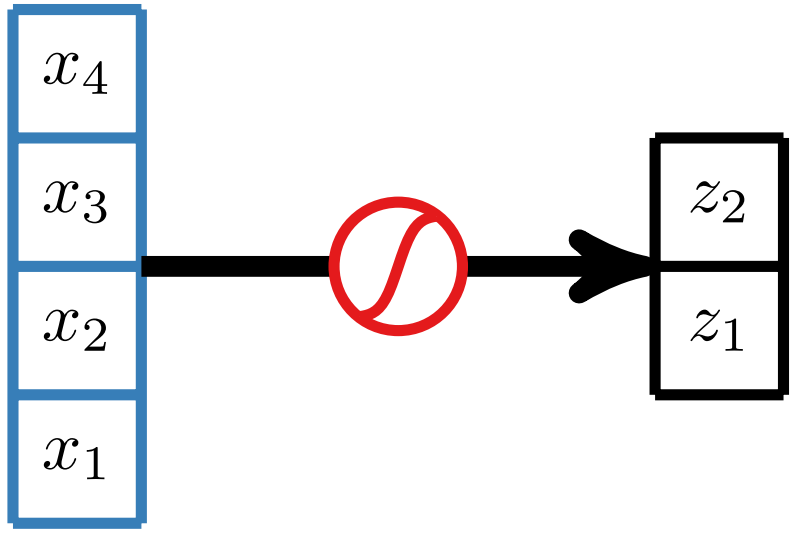


$f_\theta : \mathbb{R}^4 \rightarrow \mathbb{R}^2$

$$\mathbf{z} = \text{sigmoid}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

What is the size of **W**? .. and **b**?

Multiple Neurons - Another view: Projections



$f_\theta : \mathbb{R}^4 \rightarrow \mathbb{R}^2$

$$\mathbf{z} = \text{sigmoid}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

What is the size of $\mathbf{W}$? .. and $\mathbf{b}$?

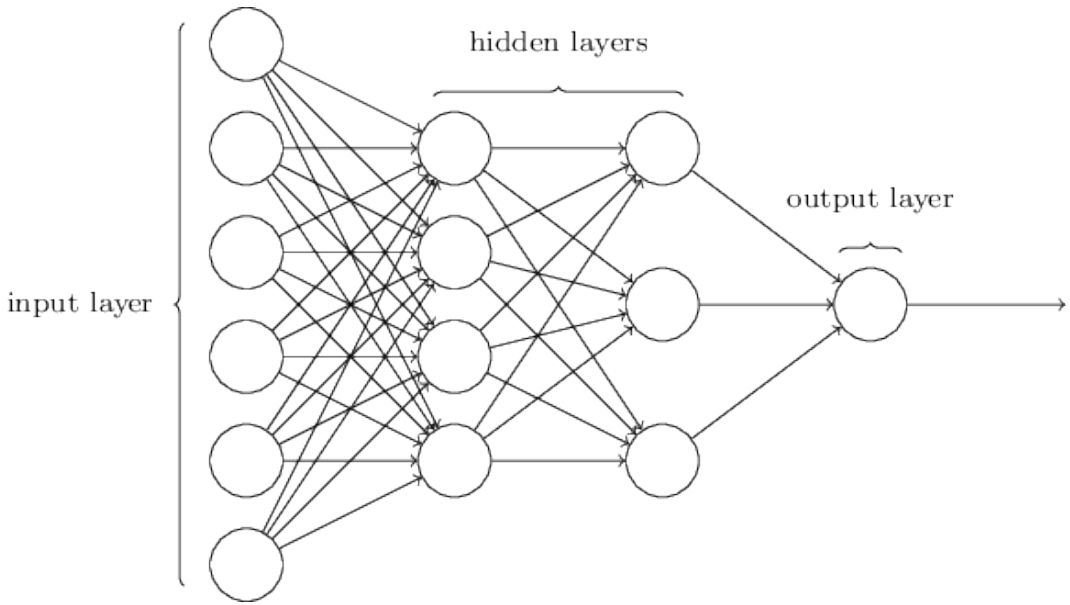
$$\mathbf{z} = \text{sigmoid}(\mathbf{W}\mathbf{x} + \mathbf{b}) \quad \text{with } \mathbf{W} \in \mathbb{R}^{4\times 2}, \mathbf{b} \in \mathbb{R}^2$$

Modularity: Multi-layer Perceptron

- Note: MLP is a bit of a historical misnomer, as technically these are no longer perceptrons (a perceptron is purely linear)
- feedforward neural network: a multilayer neural network (with no cycles): the outputs from one unit are passed on to units in the next higher layer

Modularity: Multi-layer Perceptron

- **fully connected**: each unit in each layer takes as input the outputs from all the units in the previous layer function application



Note: bias term is often ignored in figures for simplicity

A visualization which includes the parameters (weights) explicitly:

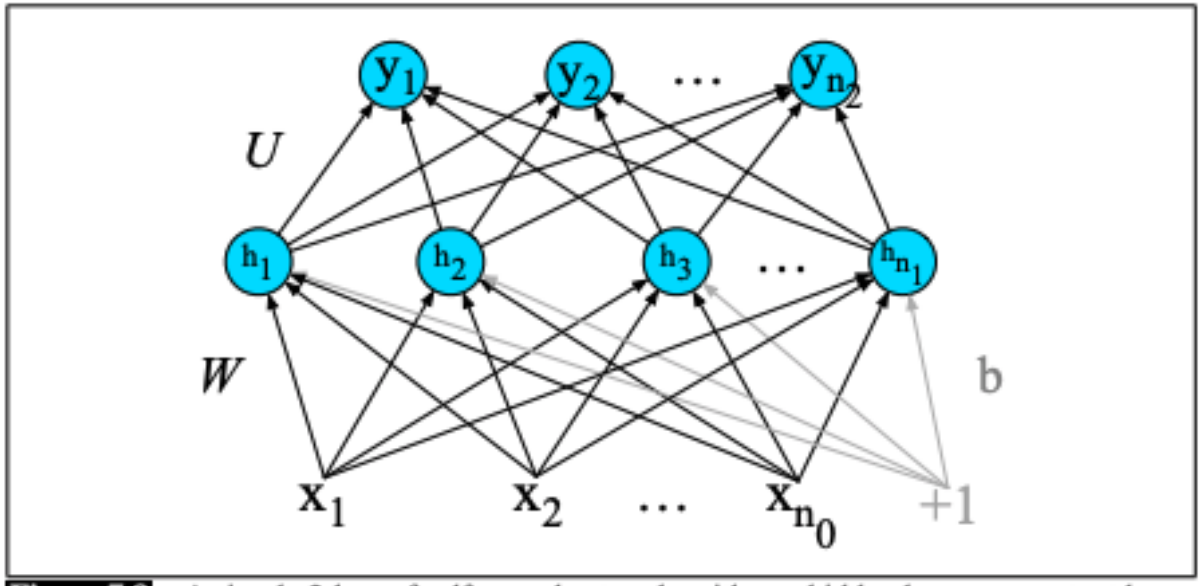
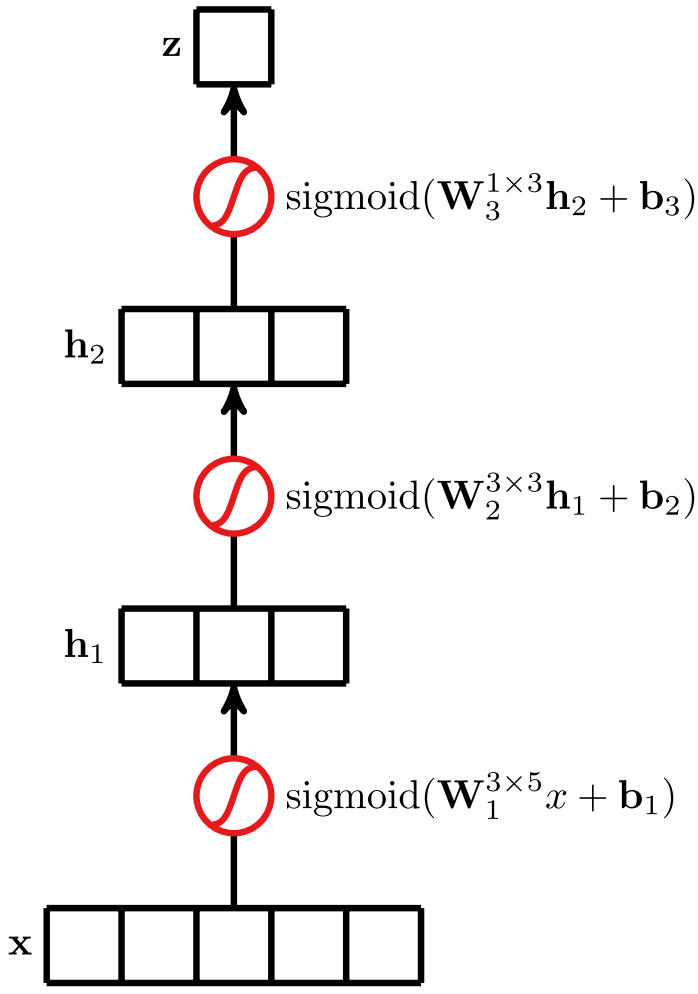


Figure 7.8 A simple 2-layer feedforward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

In grey: the bias node.

Src: J&M, chp.7

Neural Network as Projection



Function applications!

$$f_{1,\theta} : \mathbb{R}^5 \to \mathbb{R}^3$$

$$f_{2,\theta} : \mathbb{R}^3 \to \mathbb{R}^3$$

$$f_{3,\theta} : \mathbb{R}^3 \to \mathbb{R}^1$$

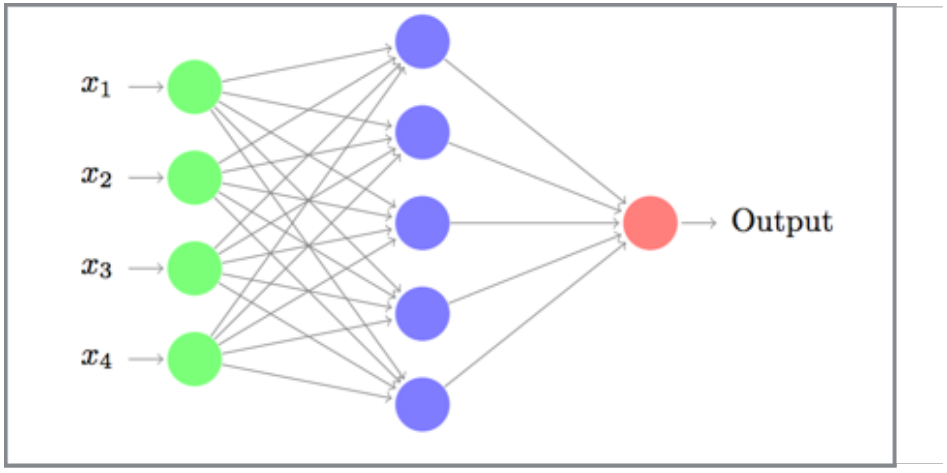
$$g_\theta = f_{3,\theta} \circ f_{2,\theta} \circ f_{1,\theta}$$

$$g_\theta(\mathbf{x}) = f_{3,\theta}(f_{2,\theta}(f_{1,\theta}(\mathbf{x})))$$

$$g_\theta : \mathbb{R}^5 \to \mathbb{R}^1$$

Connecting the different views

Lets look at a more detailed example. The network can be formulized as:



$$NN_{MLP1}(\mathbf{x}) = \mathbf{W^2}(\mathbf{W^1}\mathbf{x} + \mathbf{b^1}) + \mathbf{b^2}$$

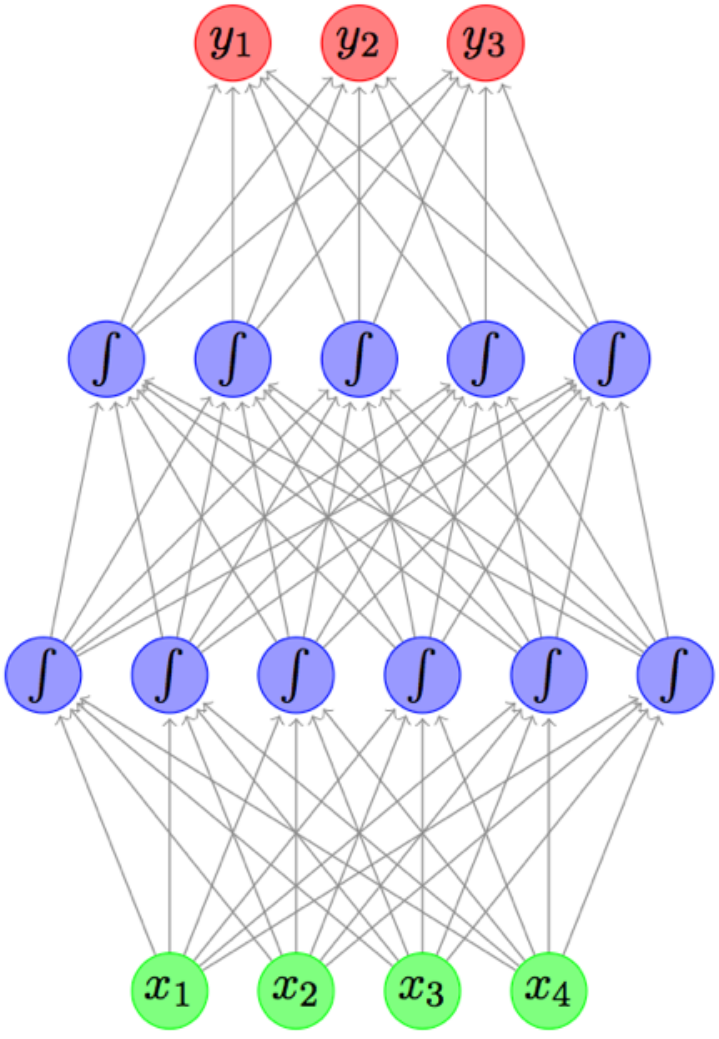
Question: what are all the terms in the formula above, and how can you connect them to the picture above? What is the size of the matrix $\mathbf{W^1}$?

Time for a few questions
www.menti.com 2470 8779

A feedforward neural network with 2 hidden layers:

$$NN_{MLP2}(\mathbf{x}) = \sigma^2(\sigma^1(\mathbf{x}\mathbf{W^1} + \mathbf{b^1})\mathbf{W^2} + \mathbf{b^2})\mathbf{W^3} + \mathbf{b^3}$$

The MLP2 is illustrated here (vertically).



It is a bit cumbersome to see, so lets break the formula

$$NN_{MLP2}(\mathbf{x}) = \sigma^2(\sigma^1(\mathbf{x}\mathbf{W^1} + \mathbf{b^1})\mathbf{W^2} + \mathbf{b^2})\mathbf{W^3} + \mathbf{b^3}$$

down into parts:

$$\mathbf{h^1} = \sigma^1(\mathbf{x}\mathbf{W^1} + \mathbf{b^1})$$

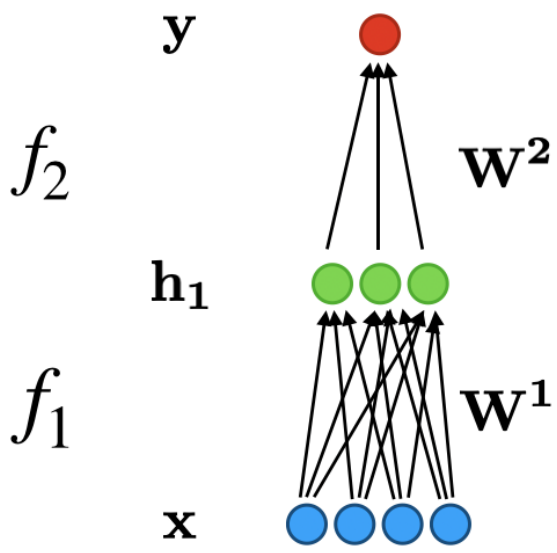
$$\mathbf{h^2} = \sigma^2(\mathbf{h_1}\mathbf{W^2} + \mathbf{b^2})$$

$$NN_{MLP2}(\mathbf{x}) = \mathbf{h^2}\mathbf{W^3} + \mathbf{b^3}$$

Voila! Now we have a wonderful description of a neural network, both graphically and algebraically.

From the graphical view to the computational graph abstraction

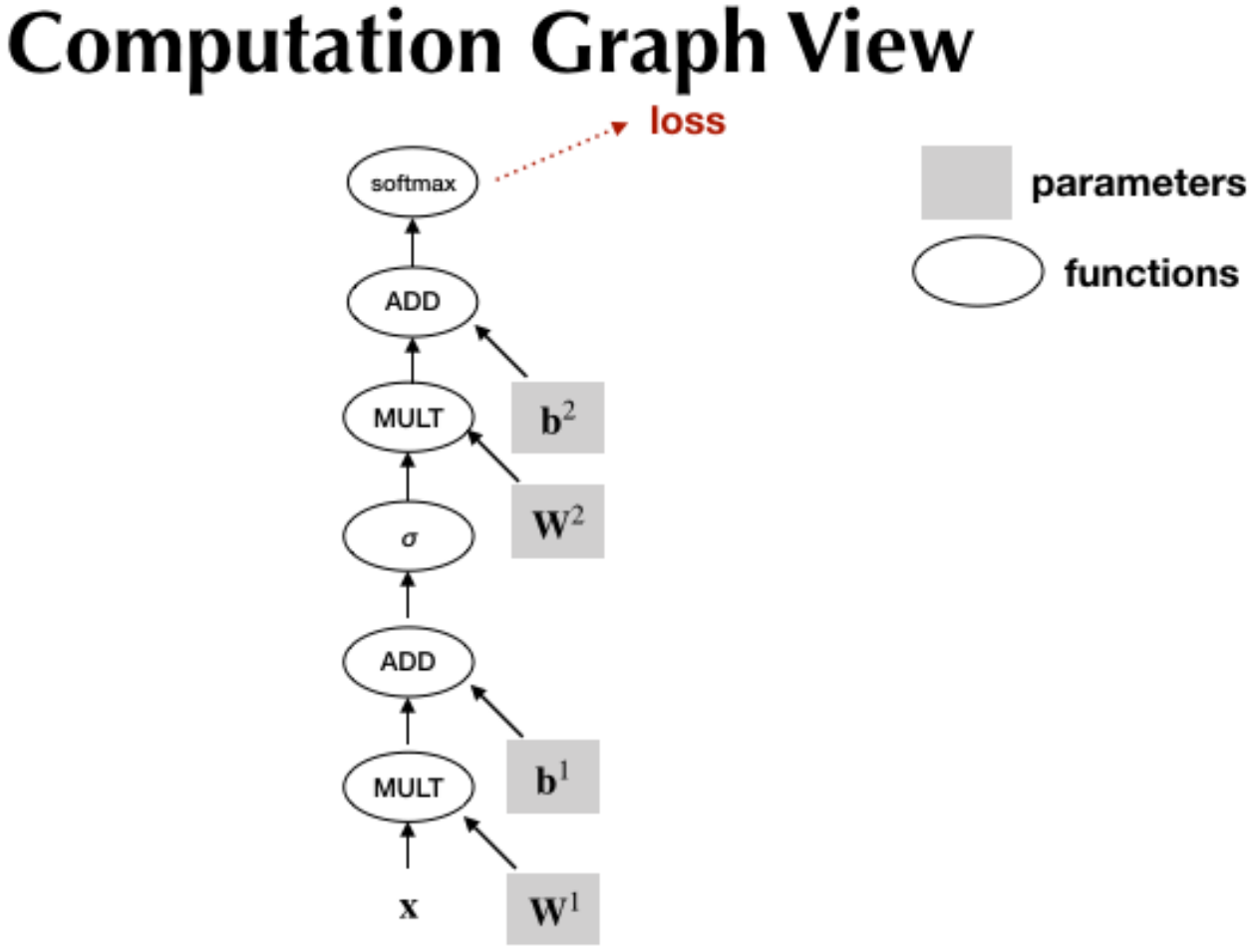
$$\hat{y} = \textbf{softmax}(f_2(f_1(x)))$$



fully-connected layers

Computational graph view

Sequence of operations/functions, inputs or parameters:



$$NN_{MLP1}(\mathbf{x}) = \sigma(\mathbf{x}\mathbf{W}^{\mathbf{1}} + \mathbf{b}^{\mathbf{1}})\mathbf{W}^{\mathbf{2}} + \mathbf{b}^{\mathbf{2}}$$

Where do the weights come from?

In the logical computation examples, we made up the weights. In the real world, we learn them automatically using the error backpropagation algorithm (starting from random weights).

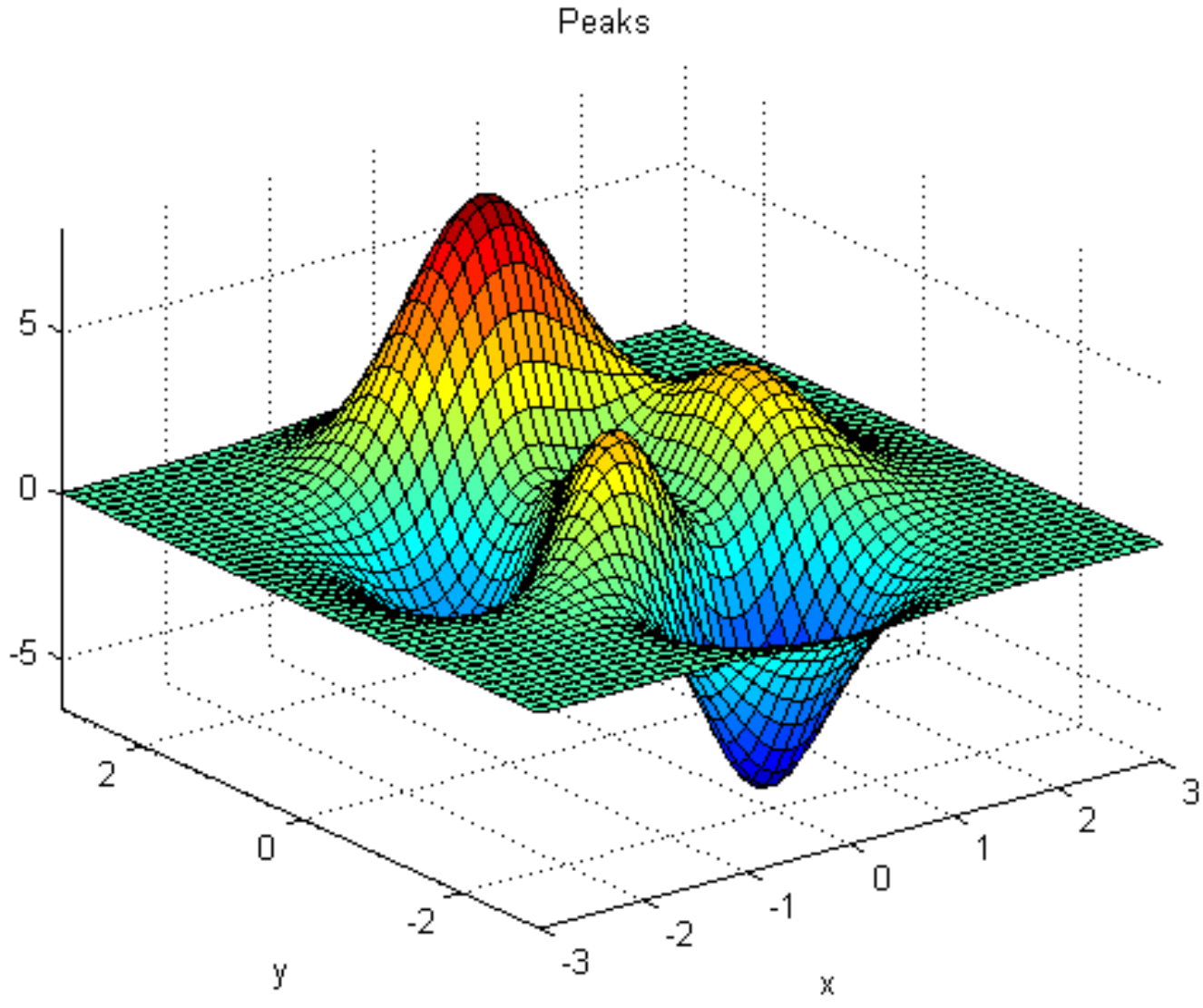
Model

- Model: some function f_θ parameterized by θ that we want to learn from data $\mathcal{D} = \{(x_i, y_i)\}$
- θ is the collection of all the weights of the network

Where do these weights come from?

Training a NN:

- training data
- model
- loss function
- training algorithm


Peaks
5
0
-5
2
0
-2
y
-3
-2
-1
0
1
2
3
x

Stochastic Gradient Descent

Goal: find parameters θ of model f_θ that minimize loss function $\mathcal{L}$:

1. Initialize parameters θ
2. Shuffle training data $\mathcal{D}$
   - For every example $(x_i, y_i) \in \mathcal{T}$
     1. Find direction that improves loss
        - Calculate gradient of parameters w.r.t. loss $\frac{\partial \mathcal{L}(f_\theta, x_i, y_i)}{\partial \theta}$
     2. Update parameters with learning rate α
        - $\theta := \theta - \alpha * \frac{\partial \mathcal{L}(f_\theta, x_i, y_i)}{\partial \theta}$
   - Go to 2.

Loss Functions

A function $\mathcal{L}$ that given a model f_θ, output $\hat{y}$ and gold output y measures how far we are away from the truth, for example

- Squared distance

$$\mathcal{L}(f_\theta, \hat{y}, y) = ||f_\theta(\hat{y}) - y||^2$$

- Cross-entropy (CE) loss (C = classes):

$$\mathcal{L}(f_\theta, \hat{y}, y) = -\sum_{i=1}^{C} y_i log \hat{y}_i$$

- If only 1 gold label:

$$\mathcal{L}(f_\theta, \hat{y}, y) = -log \hat{y}_i$$

Forward pass

Consider:

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

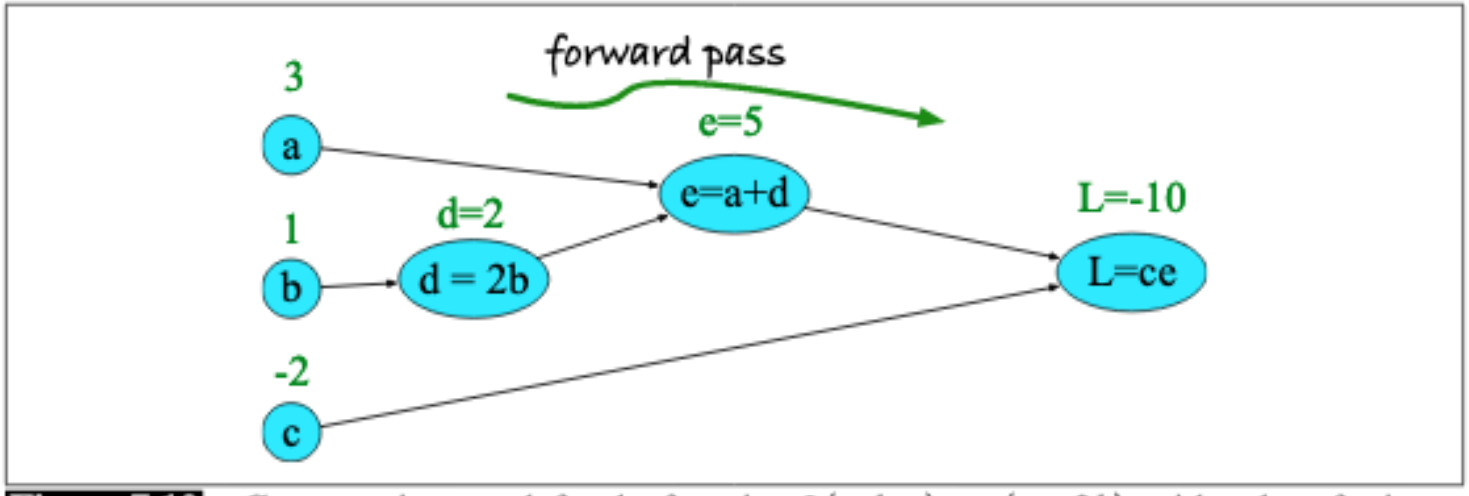


Figure 7.10 Computation graph for the function $L(a,b,c) = c(a+2b)$, with values for input nodes $a = 3$, $b = 1$, $c = -2$, showing the forward pass computation of L.

Chain rule

Backwards differentiation makes use of the chain rule. Suppose we want to calculate the derivate of $f(x) = u(v(x))$, we can use the chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial u}{\partial v}\frac{\partial v}{\partial x}$$

So for our example, the derivatives we need are (see Chapter 7):

Eq. 7.26 thus requires five intermediate derivatives: $\frac{\partial L}{\partial e}$, $\frac{\partial L}{\partial c}$, $\frac{\partial e}{\partial a}$, $\frac{\partial e}{\partial d}$, and $\frac{\partial d}{\partial b}$, which are as follows (making use of the fact that the derivative of a sum is the sum of the derivatives):

$$\begin{aligned} L = ce &: \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e \\ e = a + d &: \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1 \\ d = 2b &: \quad \frac{\partial d}{\partial b} = 2 \end{aligned}$$

Backward pass

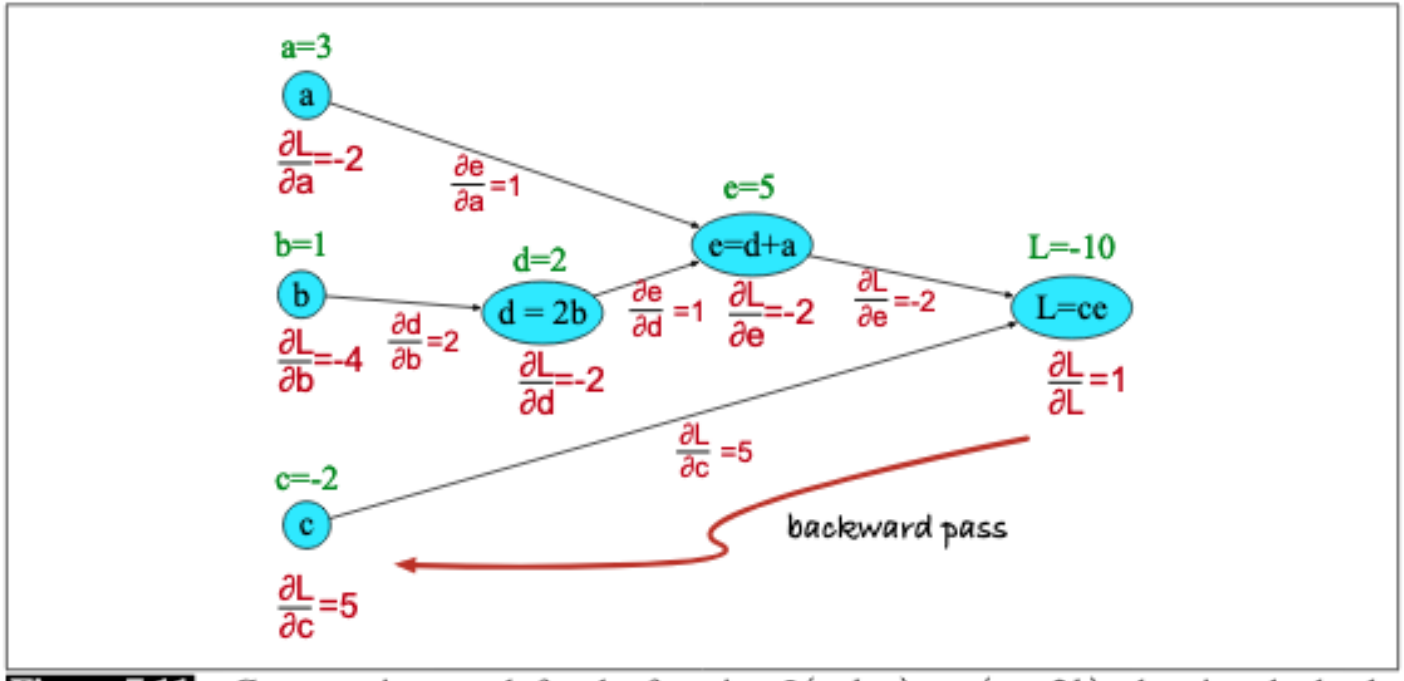
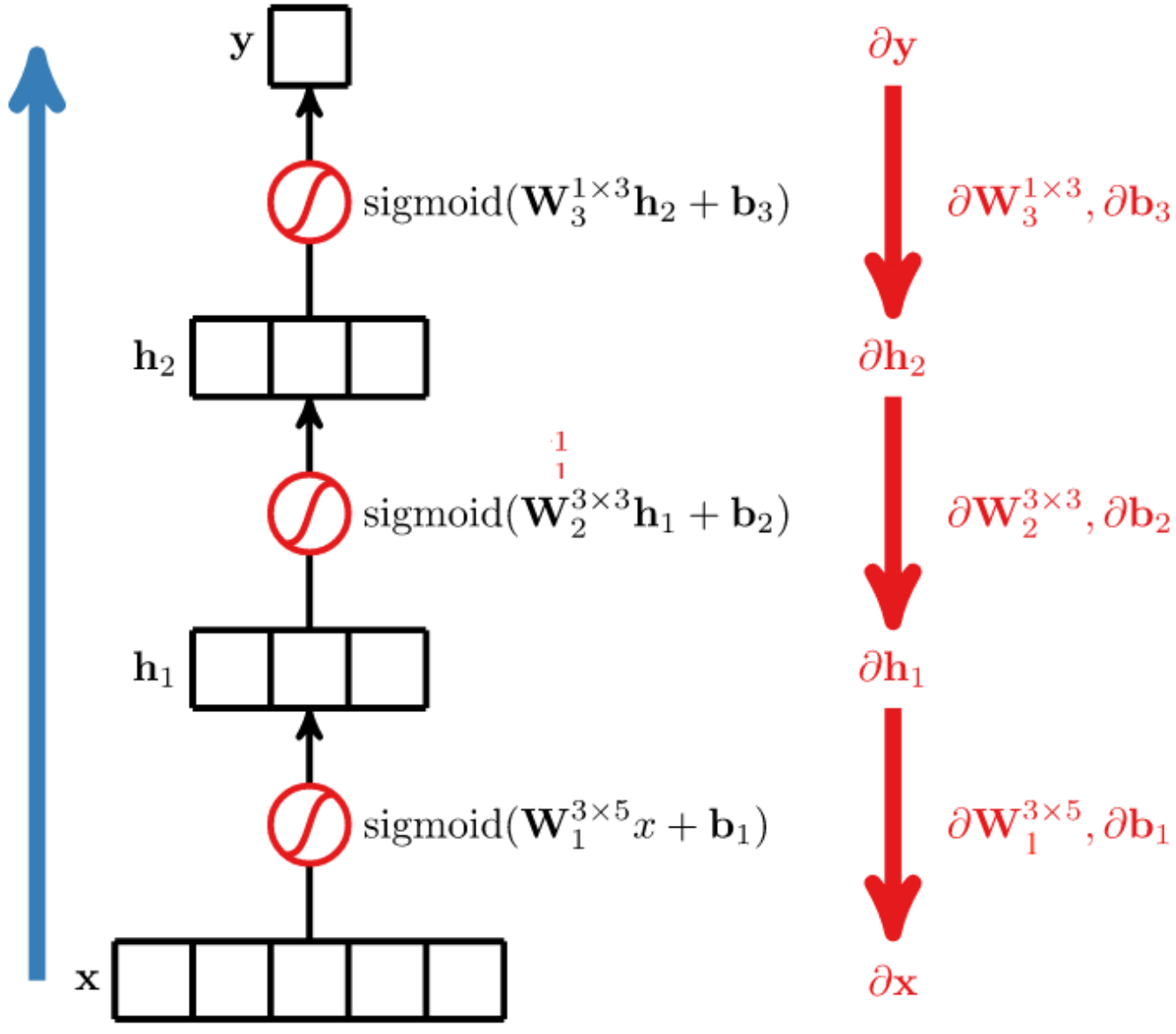


Figure 7.11 Computation graph for the function $L(a,b,c) = c(a+2b)$, showing the backward pass computation of $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$.

Backpropagation: Calculating gradients in Neural Networks



A summary:

Feed forward neural networks

A feed forward network defines a mapping

$$y = f_\theta(\mathbf{x}) = f^{(n)}(f^{(n-1)}(\dots f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x}))) \dots))$$

A fully connected layer is given by

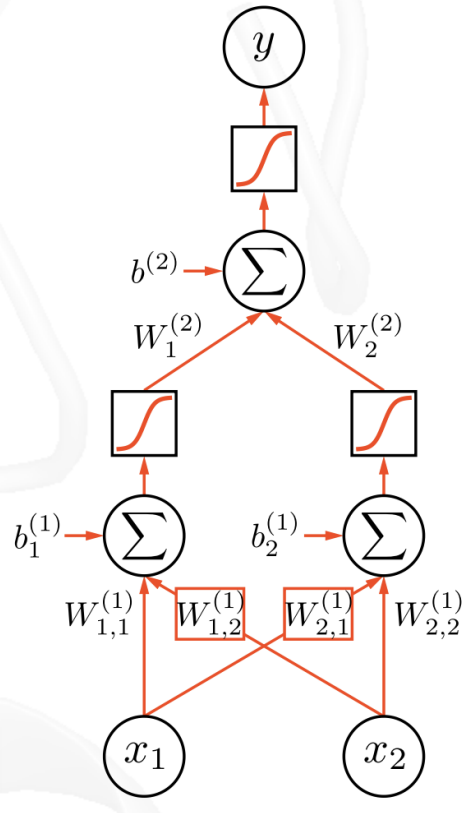
$$f^{(i)}(\mathbf{x}) = \sigma(W^{(i)}\mathbf{x} + \mathbf{b}^{(i)}) \,,$$

where σ is the activation function,
e.g. $\sigma(z) = \max(0, z)$.

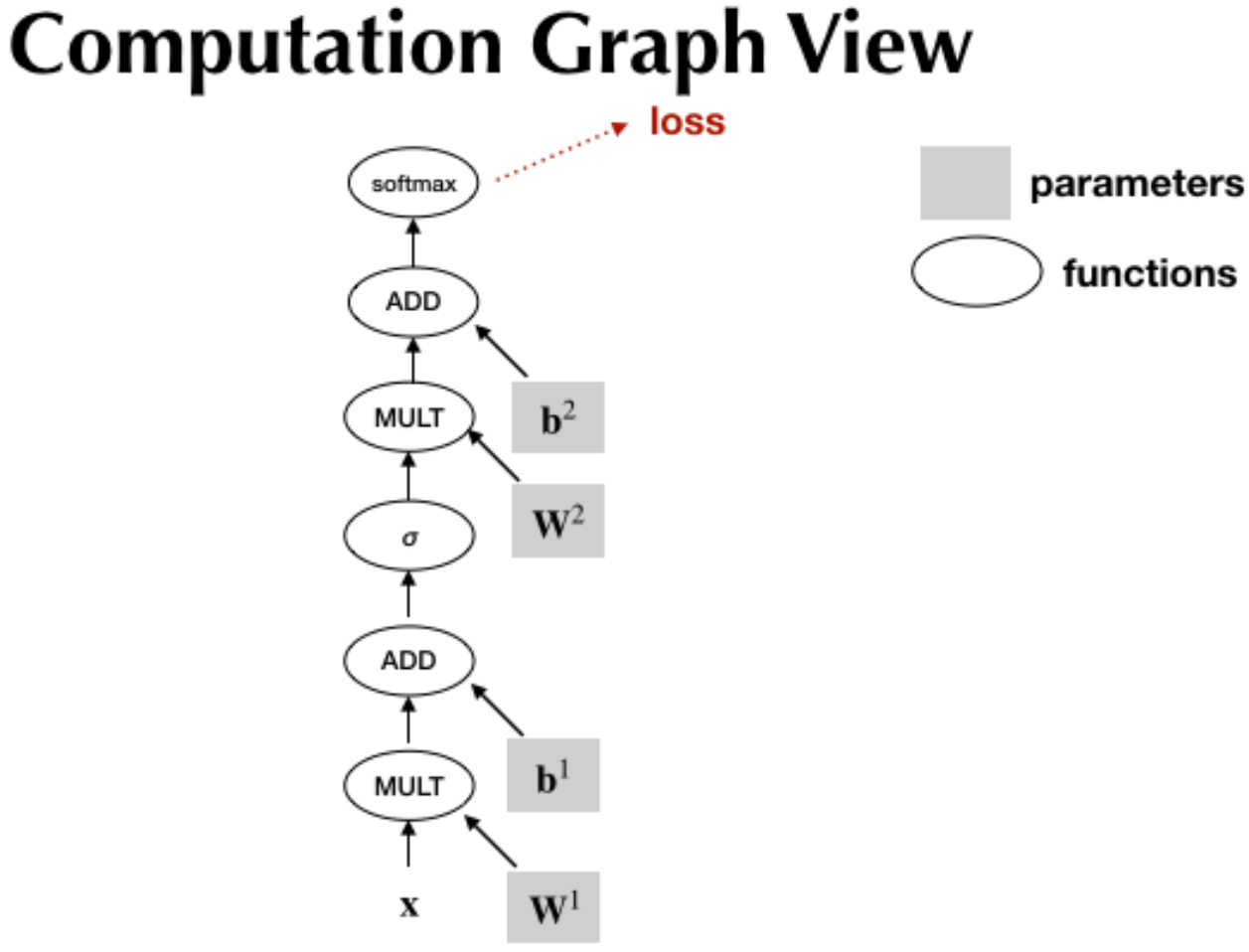
Learning θ is done by minimizing a loss function

$$\hat{\theta} = \arg\min_\theta \sum_i (y_i - f_\theta(\mathbf{x}))^2$$

for a dataset $\{(\mathbf{x}_i, y_i)\}_i$.



Text encoders: What's the input to a FFNN?



$$NN_{MLP1}(\mathbf{x}) = \sigma(\mathbf{x}\mathbf{W}^1 + \mathbf{b}^1)\mathbf{W}^2 + \mathbf{b}^2$$

FNN requires fixed-length input!

How can we represent text as input to a FFNN (=a variable-length input)?

- Central theme in NLP: Sequences are everywhere!

Possibilities

- option 1: n-hot vector
- option 2: word embeddings

Recap Traditional way: Representing text as BOW (n-hot)

The **traditional** approach for extracting features for an NLP model is the **n-hot** representation:

- extract a set of core linguistic features $f_1, ..f_n$
- define a vector whose length is the total number of features; (n-hot): 1 at position k if the k-th feature is active; this feature vector represents the **instance x** (**sparse representation**, n-hot encoding)
- use **x** as representation for an instance (=input), train the model with n-hot features (e.g. Bag-of-words)

1-hot

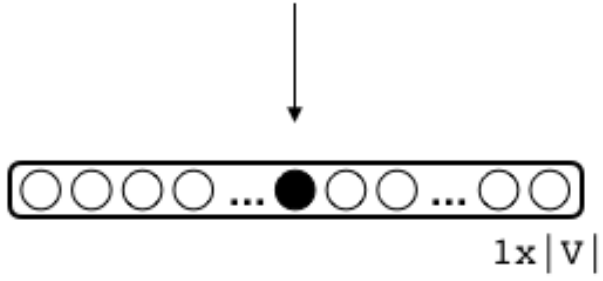
- single word

One-hot encoding

- Sparse high-dimensional vector of dimension |V| (=size of vocabulary)

Symbol (word, char,..)

yellow

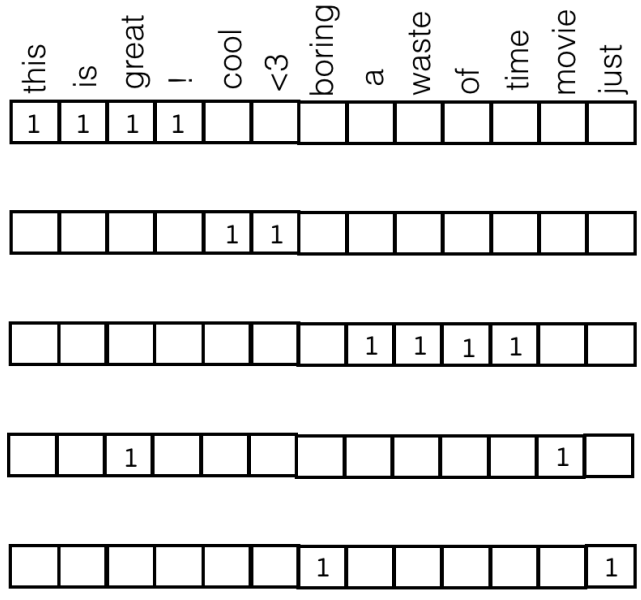


one-hot vector
(length V, one entry is 1)

n-hot

- document

```
this is great !
cool <3
a waste of time
great movie
boring just boring
```

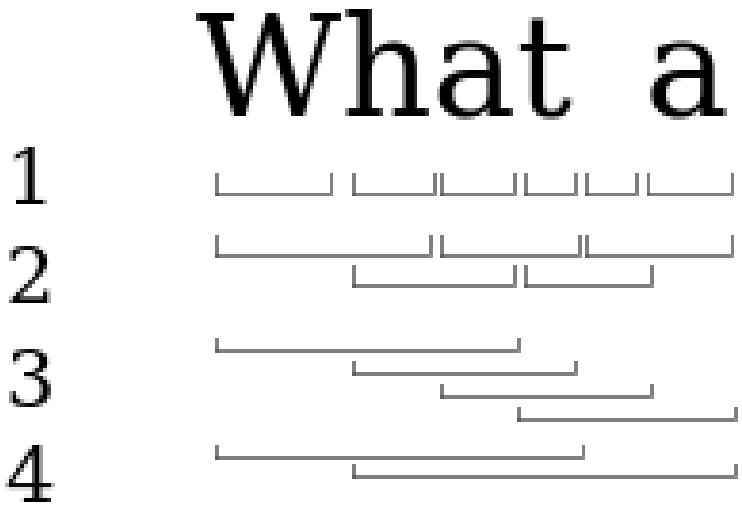


Bag of n-grams

- instead of representing single words (as in BOW), you can use bag of n-grams
  - bigrams as features: "this is", "is great", "great !"

Bag of n-grams

- instead of representing single words (as in BOW), you can use bag of n-grams
  - bigrams as features: "this is", "is great", "great !"
- Bag of character n-grams: "th","hi","is", "s ", " g", "gr",...



- Combine orders of n-grams: uni-, bi-, and trigrams

Word embeddings

i.e., word embeddings are a **dense continuous** representations of a word. Typically when talking about word embeddings we mean:

- A **vector** for each word.
- Entire vocabulary = **matrix E** which holds all word vectors, i.e. stores $|V|$ * d values (where d is the word embedding dimensionality)

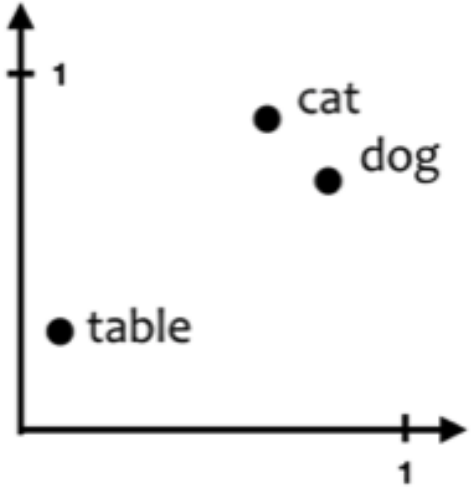
Embeddings

Dense continuous: Embeddings

- "Embed" symbol $f_{dc}(w) \mapsto \mathbb{R}^d$
  in dense low-dimensional space (d << |V|)
- Dimensionality d (hyperparameter)

$\mathbb{V} = \{\textbf{cat}, \textbf{dog}, \textbf{table}\}$ $d = 2$

$E \in \mathbb{R}^{3\times2}$



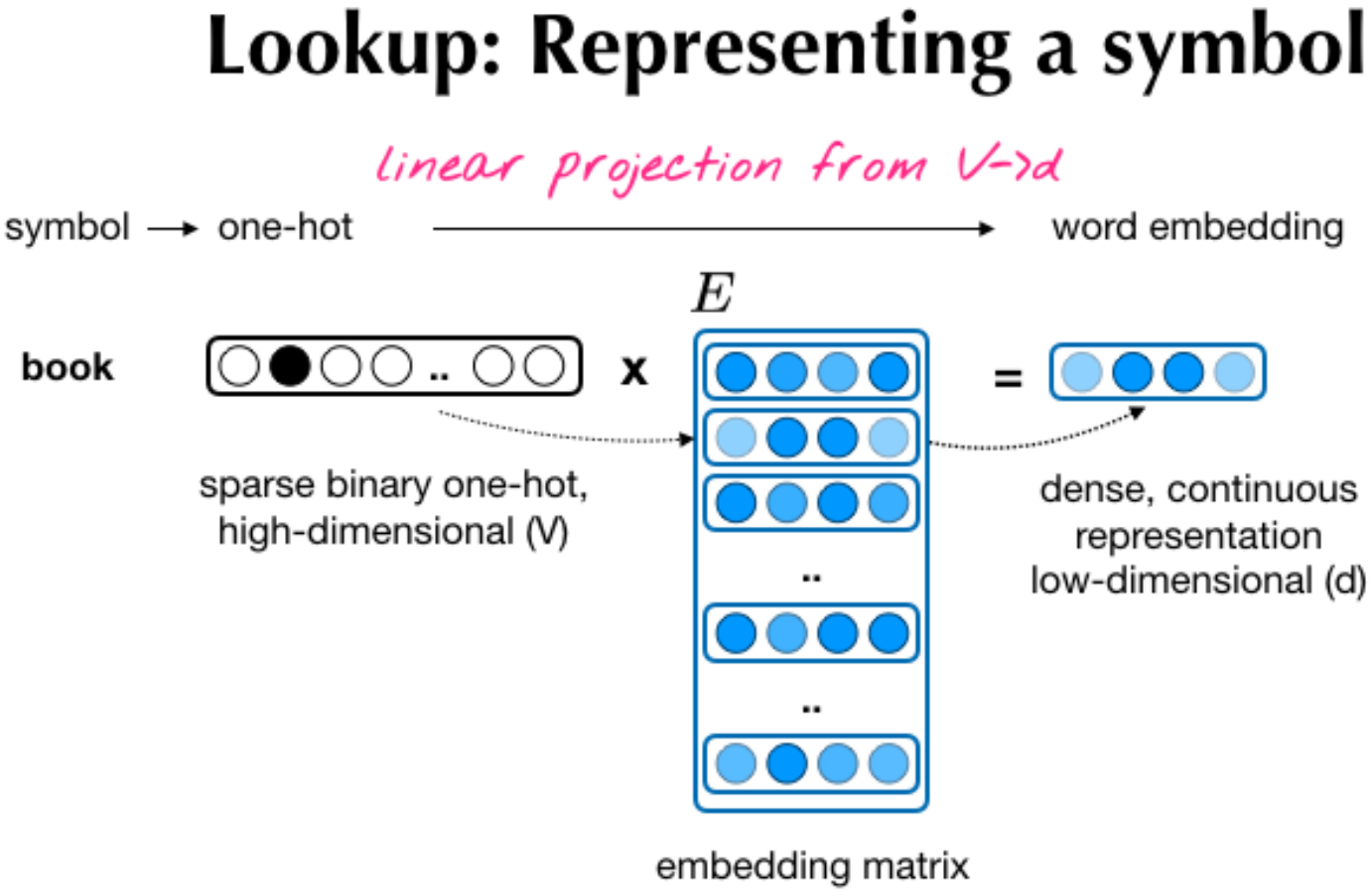
$f_{sb}(\textbf{cat}) = [0.7, 0.8]$
$f_{sb}(\textbf{dog}) = [0.75, 0.6]$
$f_{sb}(\textbf{table}) = [0.1, 0.15]$

=

|V| xd
embedding matrix

Note: d < |V|

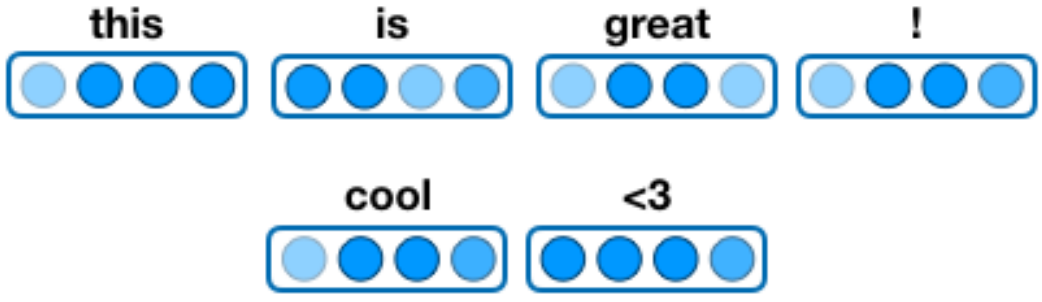
Embeddings as lookup parameters



Adapted from Lecture 5 (Riedel) http://web.stanford.edu/class/cs224n/slides/cs224n-2019-lecture06-rnnlm.pdf

Dense continuous text encodings

```
this is great !
cool <3
```



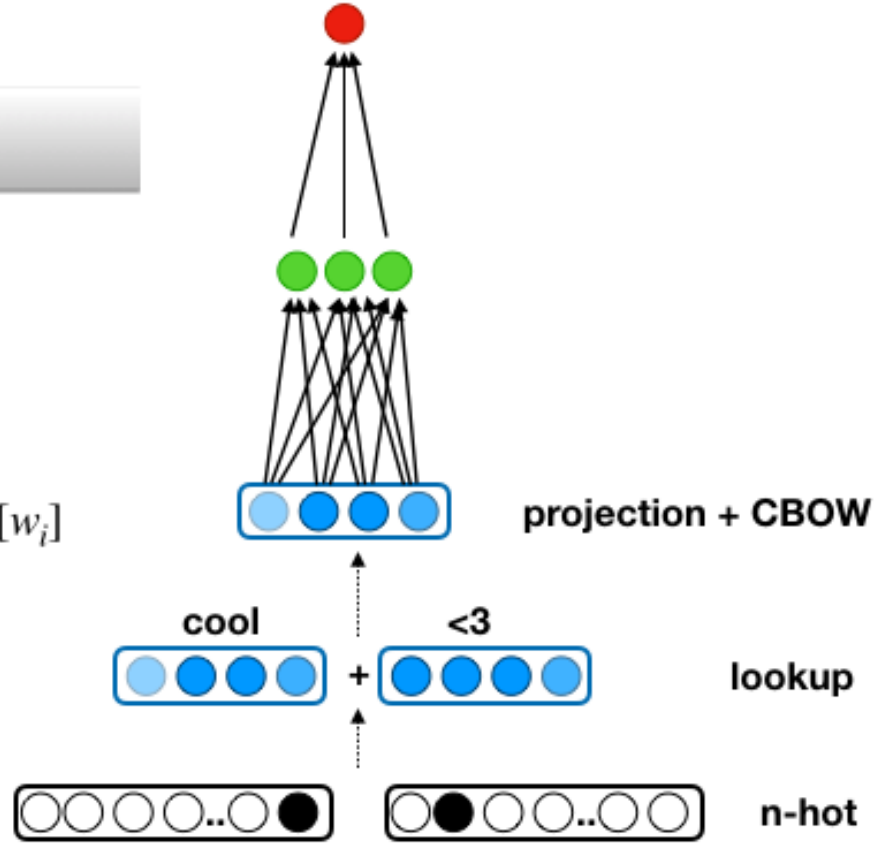
How to combine word embeddings?

Dense continuous text encoding: e.g. continuous BOW (CBOW)

Example input document 1:

```
cool <3
```

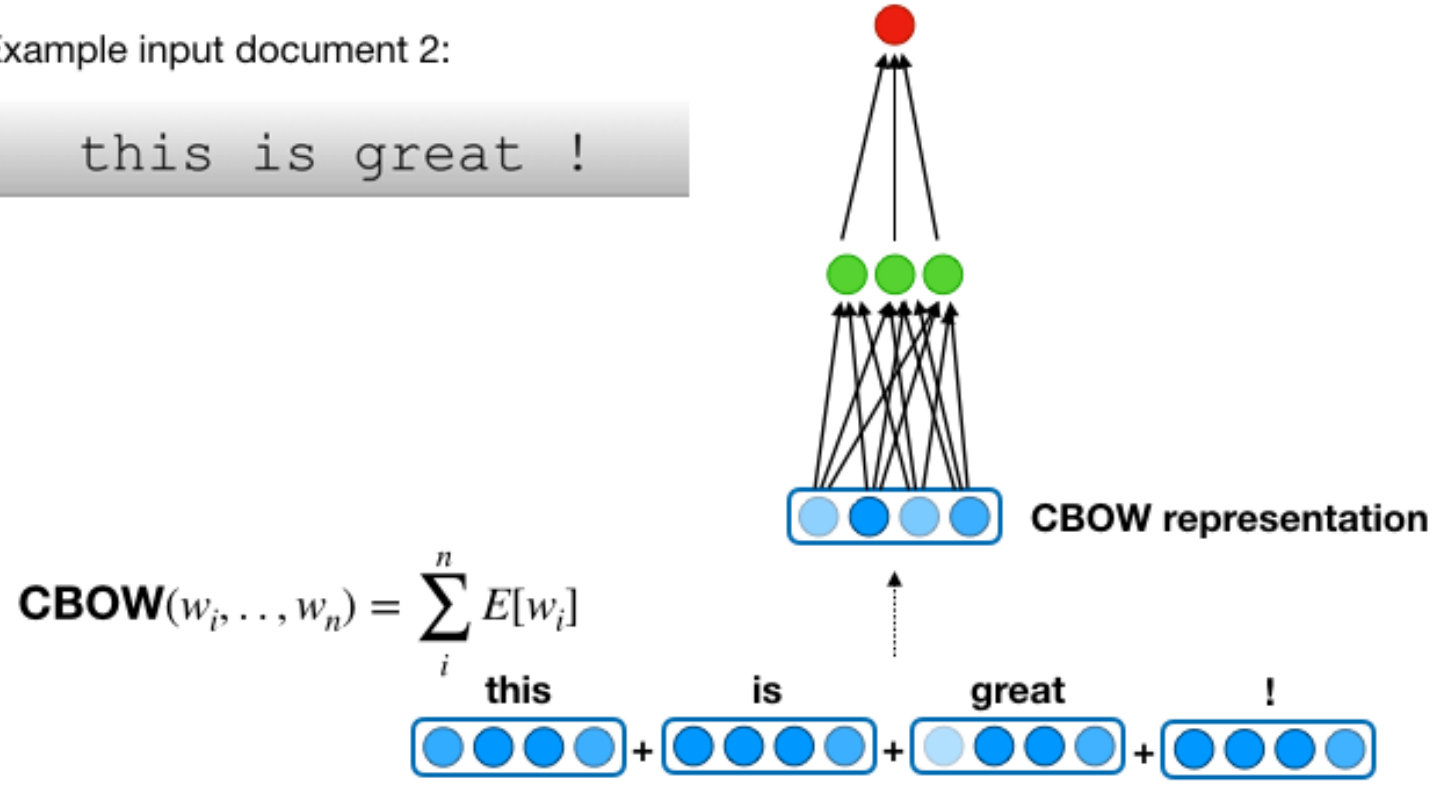
$$\textbf{CBOW}(w_i, \ldots, w_n) = \sum_i^n E[w_i]$$



Dense continuous text encoding: e.g. continuous BOW (CBOW)

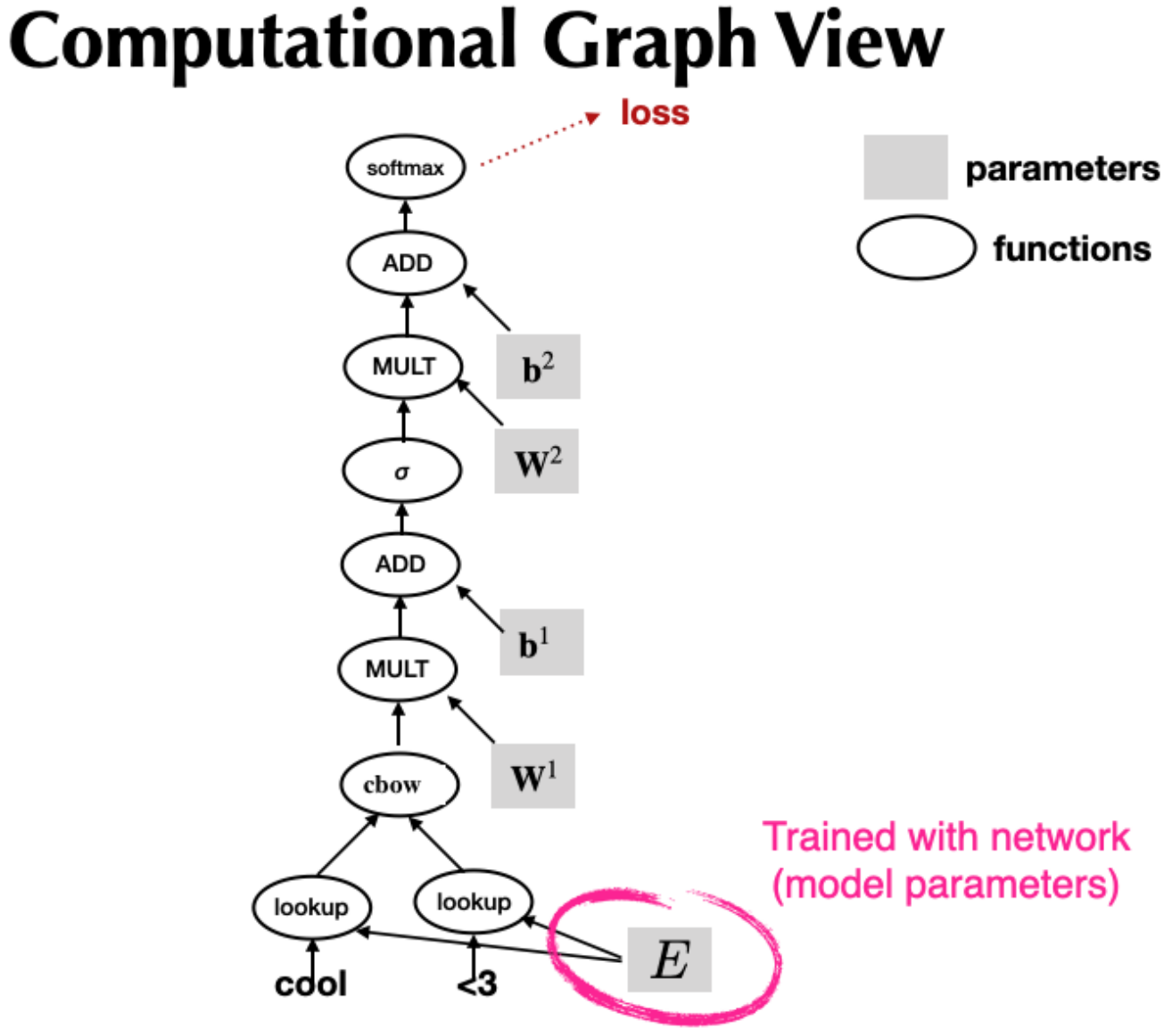
Example input document 2:

```
this is great !
```



$$\textbf{CBOW}(w_i, \ldots, w_n) = \sum_i^n E[w_i]$$

Lookup parameters: can be trained with the network



The values of the **embedding vectors** are treated as model parameters and can be **trained together** with the other

Remember limitations n-gram LM

- purchased vs. bought
- long distance dependencies
- but they take order into account (depending on n)!

Where to get word embeddings from?

- Today: we can train them with the task at hand (randomly initialize E, learn it with e.g. sentiment analysis model)
- Next lecture: can we get word embeddings from plain text?

To summarize

We can represent text as:

- n-hot (over chars/words/n-grams)
- sequences of 1-hot encodings (over chars/words/n-grams)
- sequeunces of embeddings (of chars/words/n-grams)
- Single embeddings (average over sequence of embeddings)

Conditional Random Fields

Proposed by Lafferty et al. (2001):

- A complex version of MMEM
- Avoids "label bias"
- Not used in our course, we will instead focus on the CRF-layer

CRF-layer

The FFNN does not take the surrounding predictions into account (but Viterbi did!)

- The CRF layer is basically a viterbi trelis **on top of** the network (i.e. before extracting labels)
- It has no emission probabilities (those come from the network)
- It has a matrix of #labels * #labels, defining the transition probabilities
- Weights are updated during backprop

CRF-layer

Benefits:

- Take context of labels into account
- We can also set constraints!, if we set the transition of O $\mapsto$ I-PER to 0.0 for example
- Cheap, not so many weights, inference is fast

CRF-layer

Benefits:

- Take context of labels into account
- We can also set constraints!, if we set the transition of O $\mapsto$ I-PER to 0.0 for example
- Cheap, not so many weights, inference is fast
- In practice, mostly used for NER

CRF-layer

Downsides:

- We would hope the network already takes context into account by itself
- A CRF layer might discourage the network from doing this
- Not available in PyTorch, but you can use other existing implementations in the project (I use an adapted version of AllenNLP: `https://github.com/machamp-nlp/machamp/blob/master/machamp/modules/allennlp/conditional_random_field.py`

CRF-layer

Downsides:

- We would hope the network already takes context into account by itself
- A CRF layer might discourage the network from doing this
- Not available in PyTorch, but you can use other existing implementations in the project (I use an adapted version of AllenNLP: `https://github.com/machamp-nlp/machamp/blob/master/machamp/modules/allennlp/conditional_random_field.py`
  - How to use: `https://github.com/machamp-nlp/machamp/blob/master/machamp/model/crf_label_decoder.py`

Time for a few questions
www.menti.com 2470 8779

Summary of today's lecture - key concepts:

Part 1

- What is Deep Learning, NLP: A Short History
- A single neuron, multiple neurons, MLP/FFNN, hidden units as another representation (higher capacity models)
- Different views of a Neural Network (graphical, mathematical/linear algebra or function application, computational graph abstraction)

Part 2

- words as vectors
- word embedding lookup (matrix) parameters, task-specific word-embedding training
- averaging word embeddings (CBOW as text encoder) to get fixed-size input for a FFNN

Part 3

- CRF layer