# Sequence labelling

**IN4080**
**Natural Language Processing**

Yves Scherrer

# Different ways to look at the context

**Option 1:** Look at the previous tag decisions:

$$\hat{y}_i = \arg\max_{y_i \in Y} P(y_i | x_i, y_{i-1})$$

- HMM

**Option 2:** Look at the neighboring words:

$$\hat{y}_i = \arg\max_{y_i \in Y} P(y_i | x_{i-1}, x_i, x_{i+1})$$

- Logistic regression with context word features

**Option 3:** Combine the two:

$$\hat{y}_i = \arg\max_{y_i \in Y} P(y_i | x_{i-1}, x_i, x_{i+1}, y_{i-1})$$

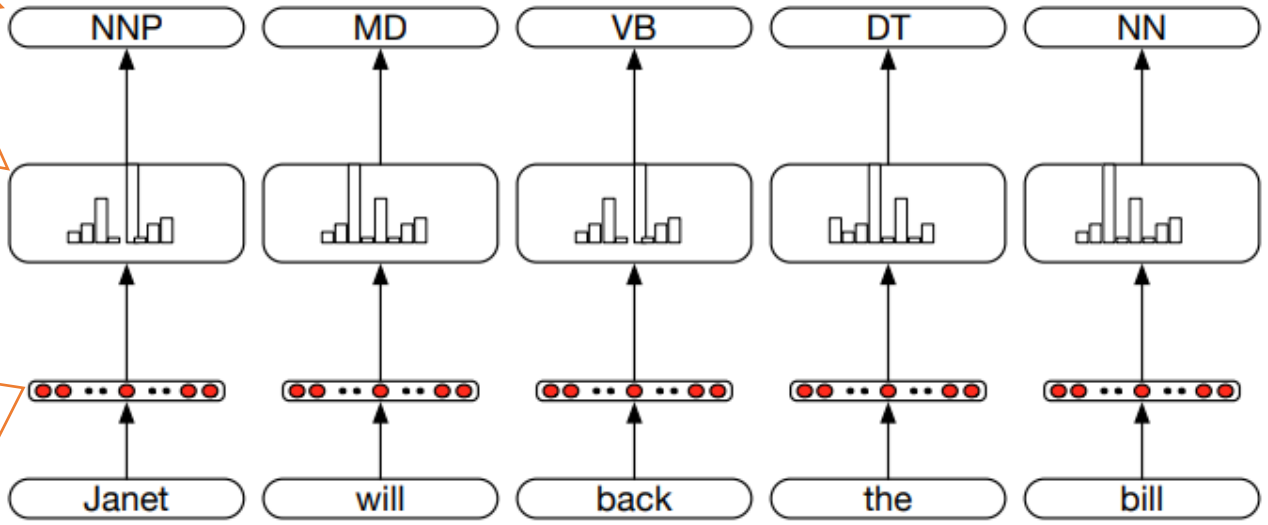
- Greedy LR with context word features and previous label features
- CRF, MEMM

# A logistic regression model for sequence labelling

Pick best (argmax)

Probability distribution over labels

Encode each word and its context as a feature vector of arbitrary (but fixed) length



# Training a LR sequence model

**Training involves prediction:**

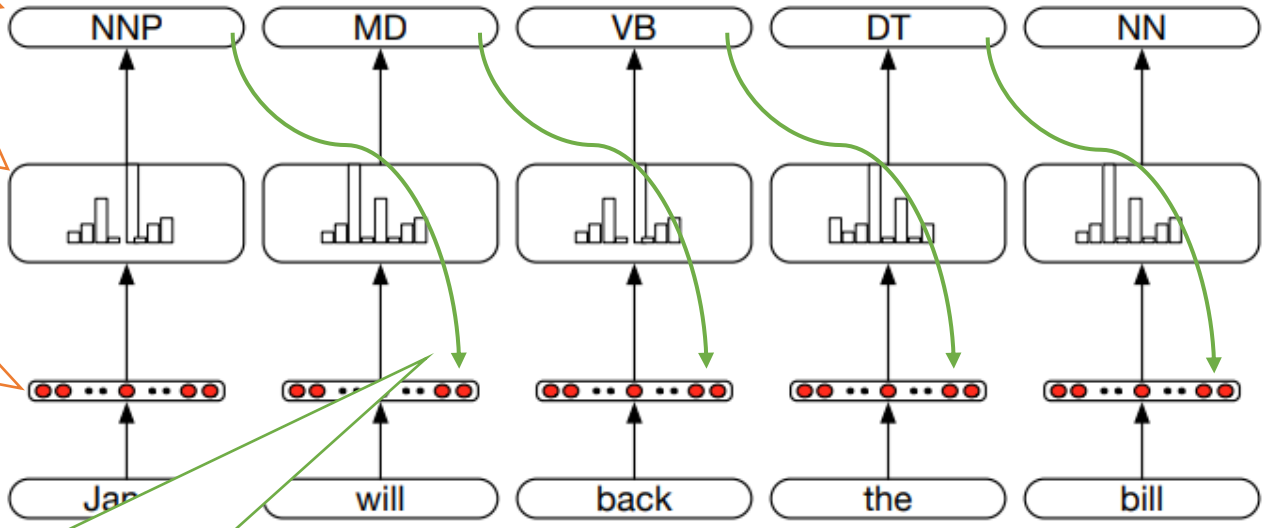
- Take one training sentence $\boldsymbol{x}_{1\ldots n}$
- Predict the label probabilities using the current weight vectors
- For each token $x_i$:
  - Update the weight vector of the correct label
  - Update the weight vectors of the incorrect labels
- Continue “until tired”

# A LR model with transition features and greedy decoding

Pick best (argmax)

Probability distribution over labels

Encode each word as a feature vector




**Greedy decoding:** The label predicted at the previous position is fed as a one-hot vector into the feature vector of the next word.

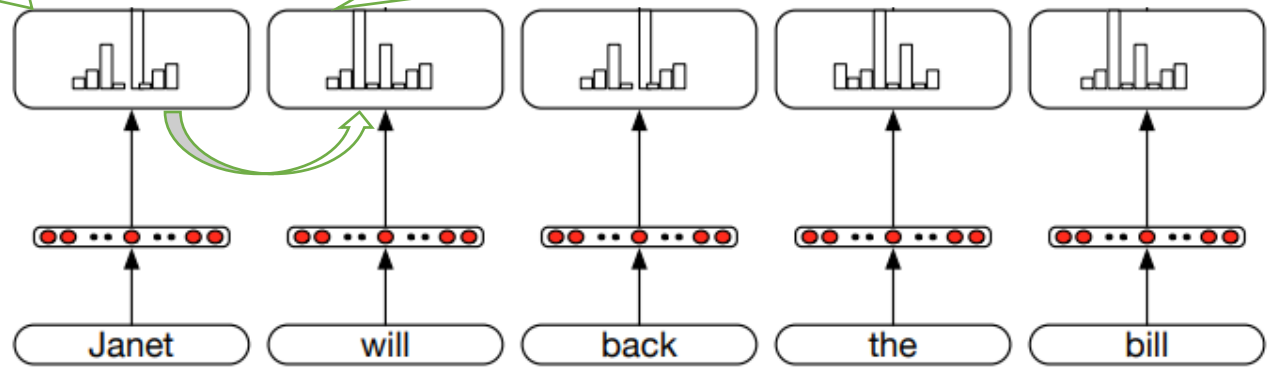
$$f^{(x)} = f^{(x_i)} \oplus f^{(y_{i-1})}$$



# Viterbi decoding for LR models: CRF, MEMM

**Viterbi decoding:** The labels are only predicted after reaching the end of the sentence. This is known as **CRF layer (conditional random field)** in discriminative models.

The model parameters consist of an **emission weights matrix** ($\boldsymbol{w}$, as before) and a **transition weight matrix** ($\boldsymbol{w_T}$).




The probability distribution over labels at the previous position is taken into account when computing current label probabilities.

# Reading

- Jurafsky & Martin, chapter 17
  - HMM, Viterbi algorithm, CRF

- Jurafsky & Martin, appendix A (not required)
  - Implementation details of the Viterbi algorithm

# Neural networks for text classification

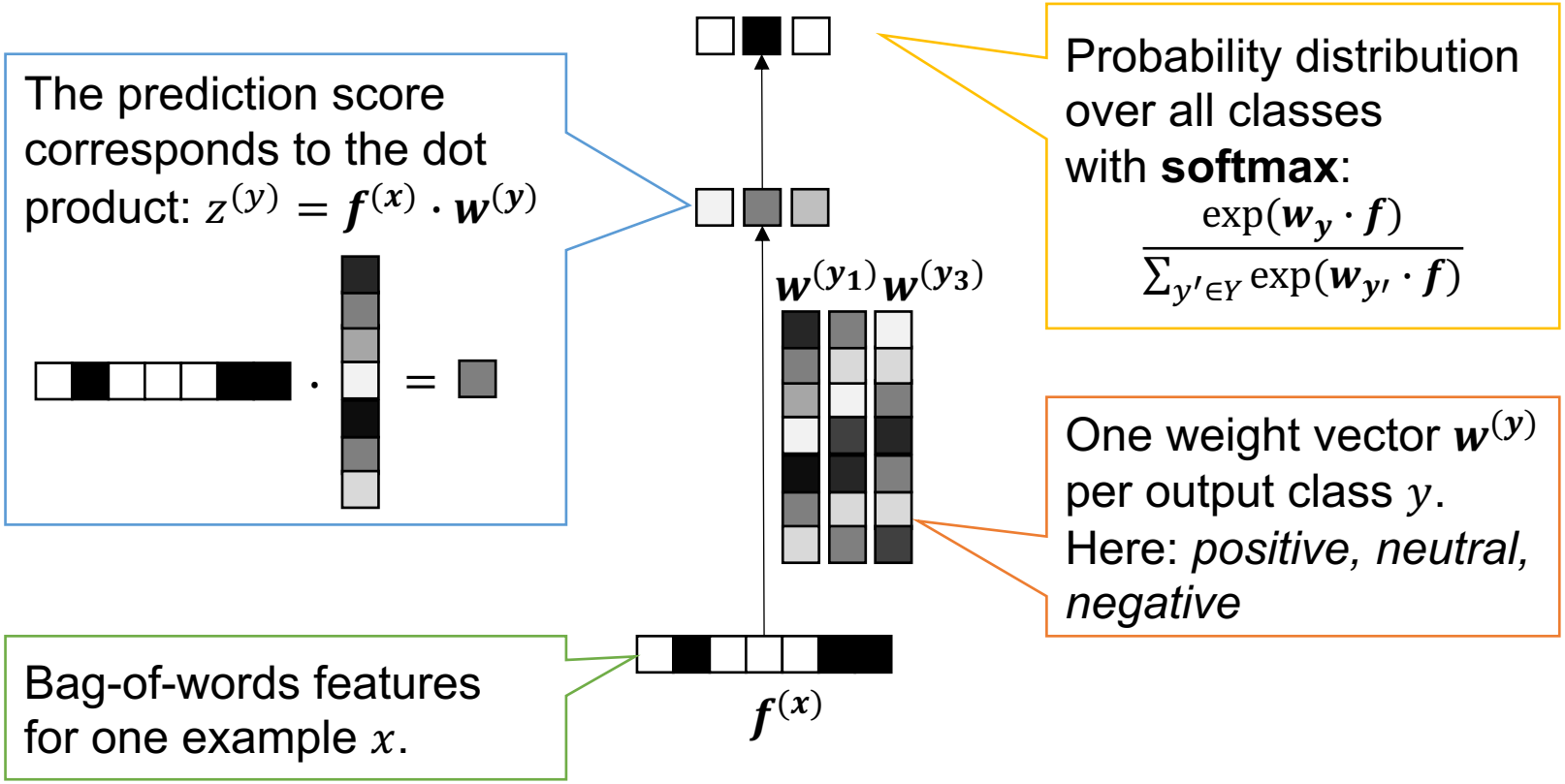
**IN4080**

**Natural Language Processing**

Yves Scherrer

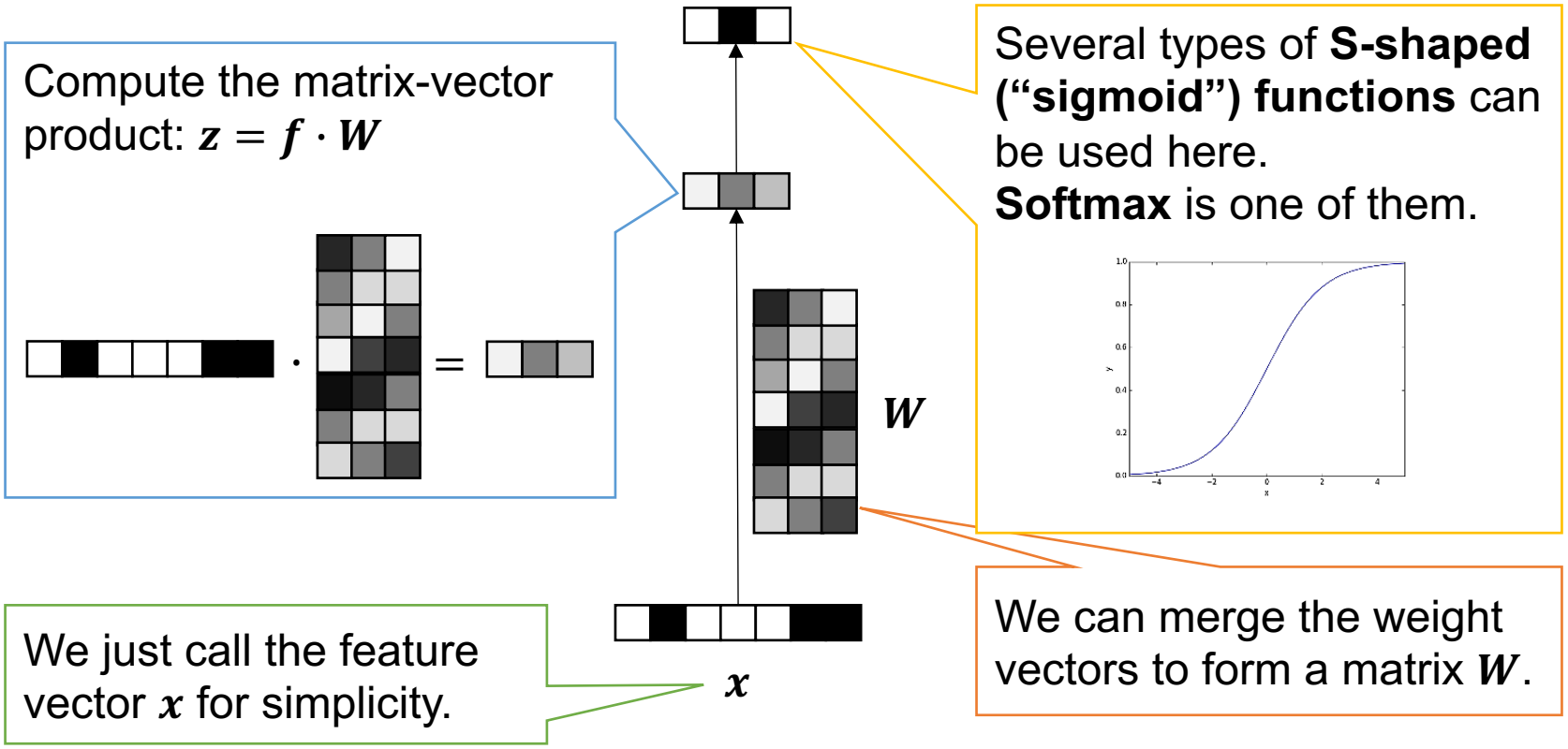
# Logistic regression models for text classification

Example: Determine the sentiment of a document



# Logistic regression models for text classification

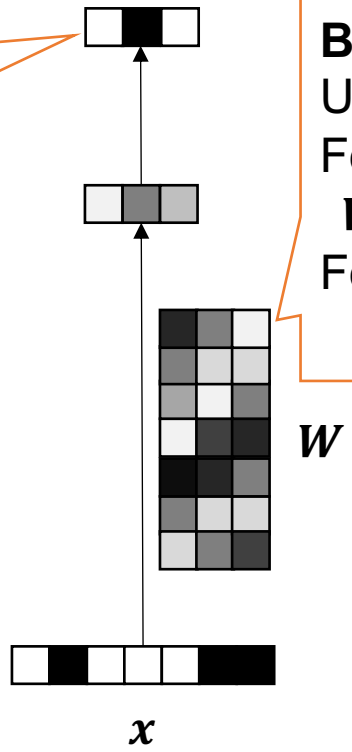
Example: Determine the sentiment of a document



# Logistic regression Training

Example: Determine the sentiment of a document

**Forward pass:**
Predict probability distribution using current weights




**Backward pass:**
Use **update rule** to modify weights:
For correct class $y$:

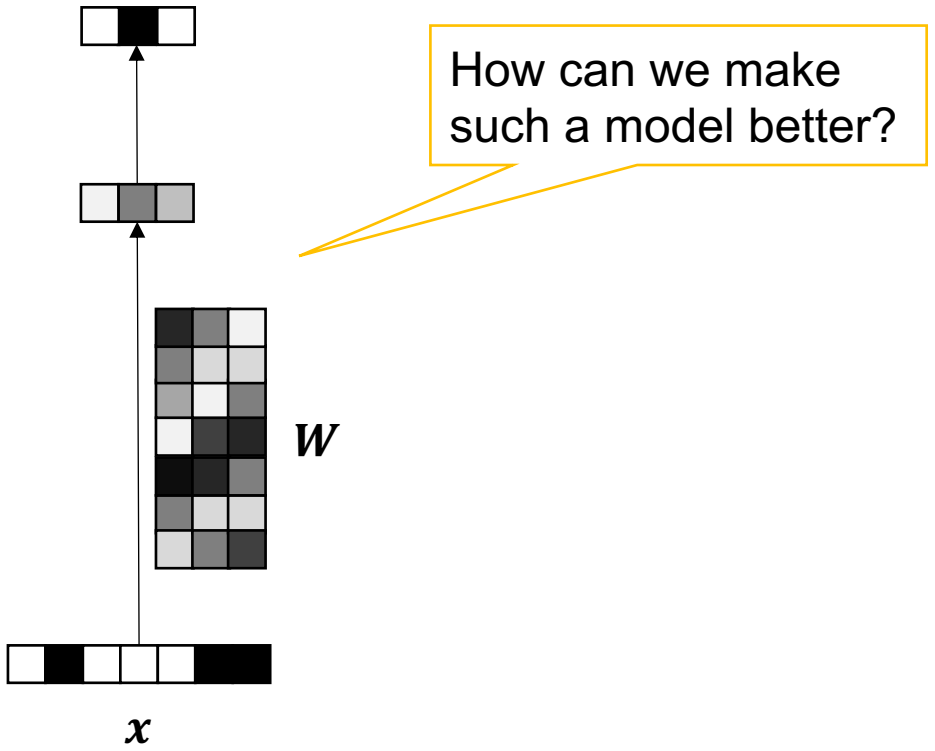
$$\boldsymbol{W}^{(y)} \leftarrow \boldsymbol{W}^{(y)} + \lambda \cdot (1 - P(y|x)) \cdot \boldsymbol{x}$$

For incorrect classes $z$:

$$\boldsymbol{W}^{(z)} \leftarrow \boldsymbol{W}^{(z)} - \lambda \cdot P(z|x) \cdot \boldsymbol{x}$$

# Logistic regression models for text classification

Example: Determine the sentiment of a document

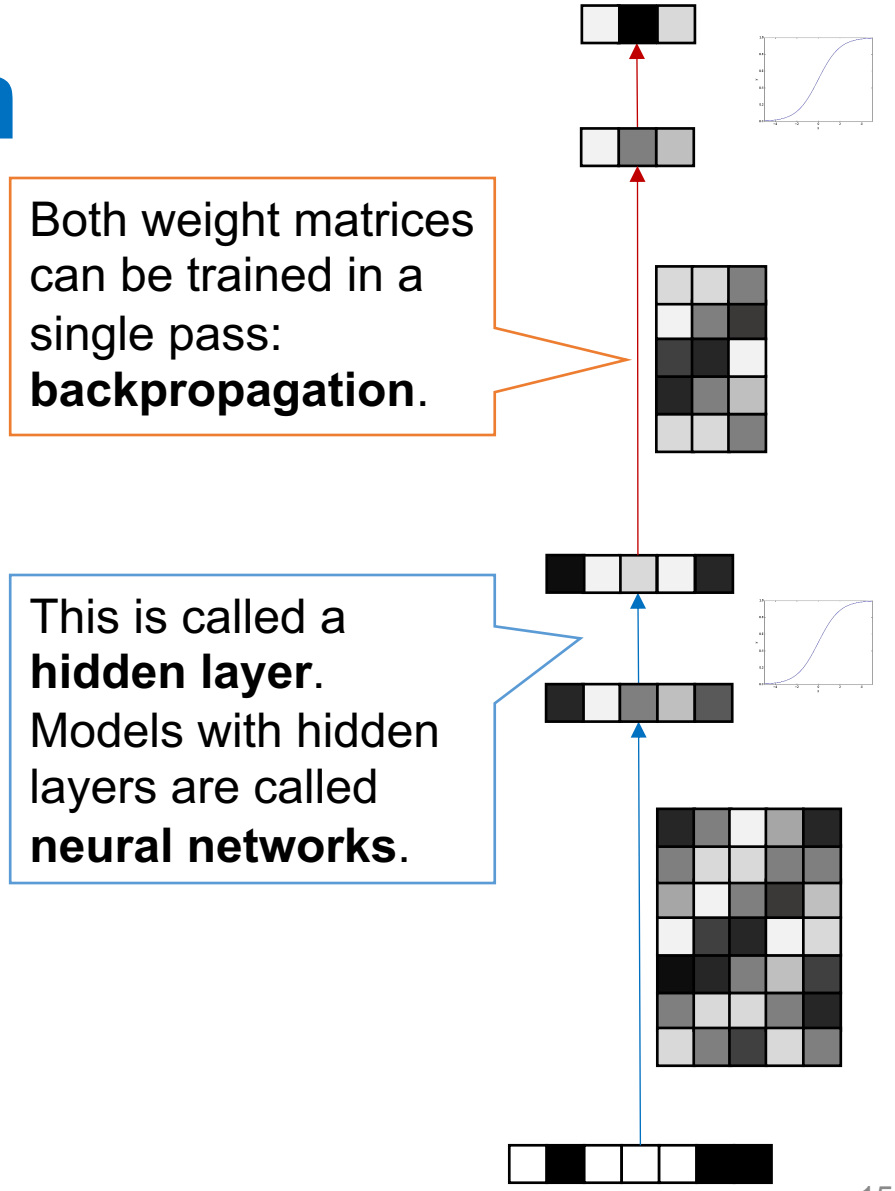


# Two-step classification

**Idea:**

- Partition the collection into e.g. 100 text classes
- Base the sentiment decision on the text class

Let's use the output of the first model as the input features of the second model



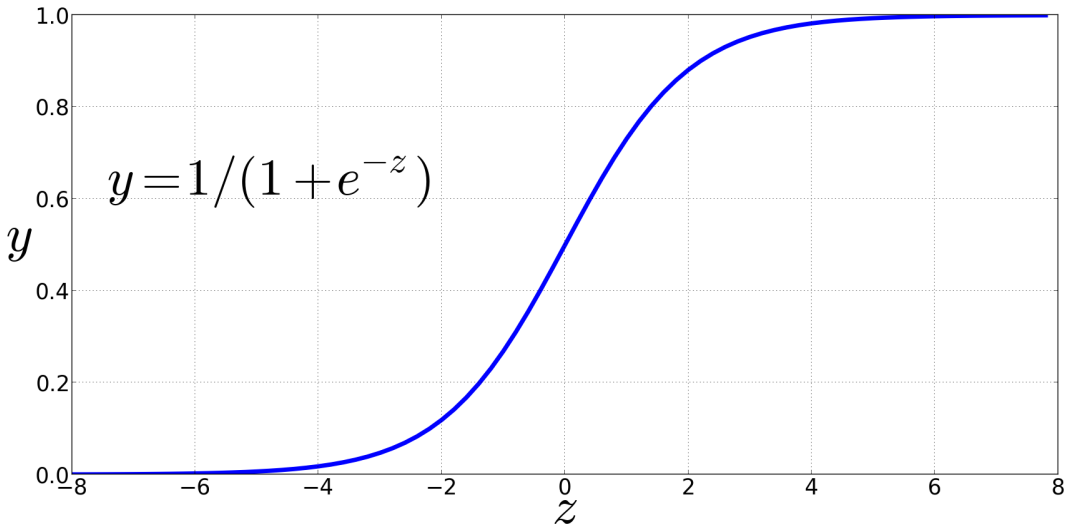
# Activation functions

**Softmax** is a so-called **activation function**

- Softmax is costly due to the normalization
  - In the hidden layers, we don't care about proper probability distributions and can use simpler activation functions (i.e. element-wise functions)

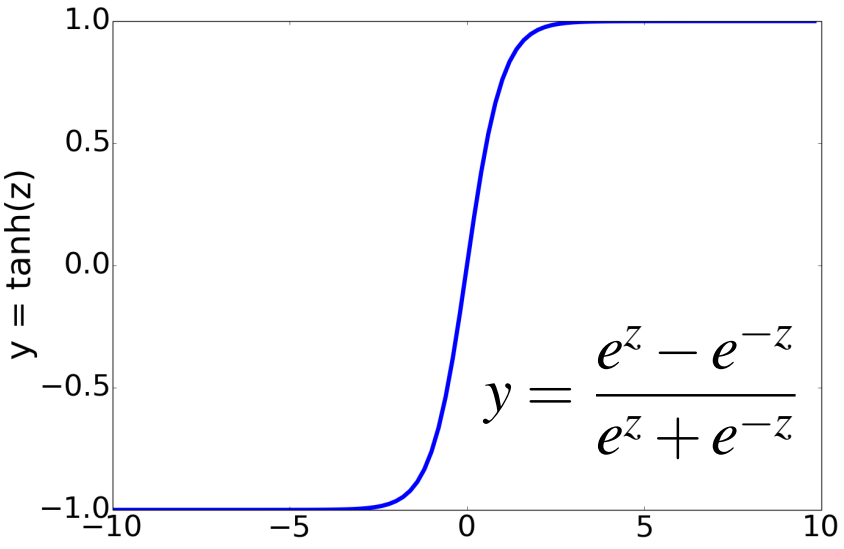
**Sigmoid:**

$$y = \sigma(z)$$



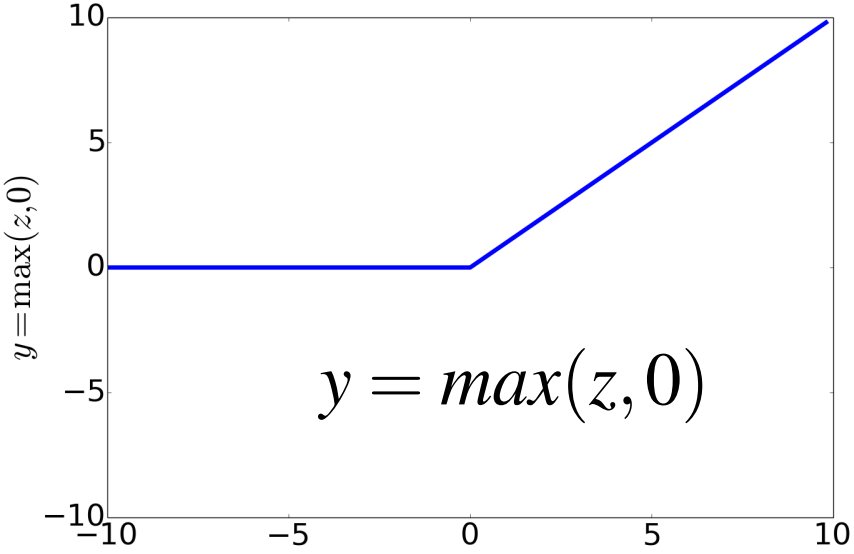
# Activation functions

## tanh



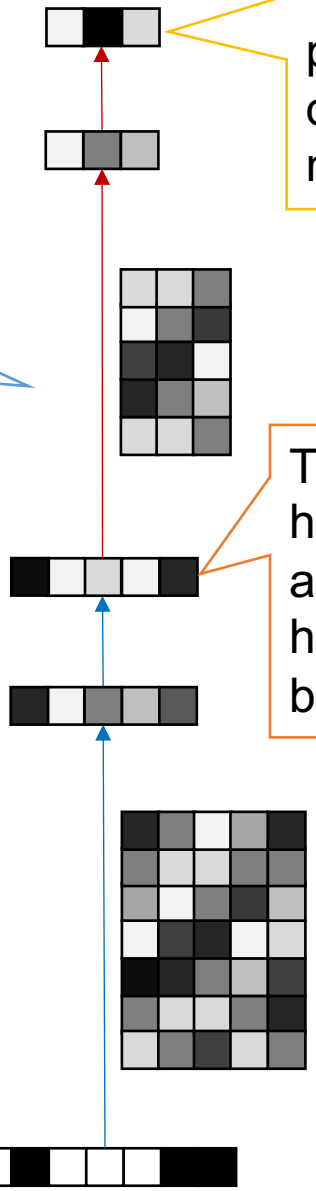

Similar to sigmoid, but often works better.

## ReLu (Rectified Linear Unit)




Very simple and often used.

# Neural networks



The size of the prediction vector is defined by the number of classes.

Several hidden layers can be added. Neural networks with more than one hidden layer are called **deep models**.

The size of the hidden layer vectors and the number of hidden layers can be freely chosen.

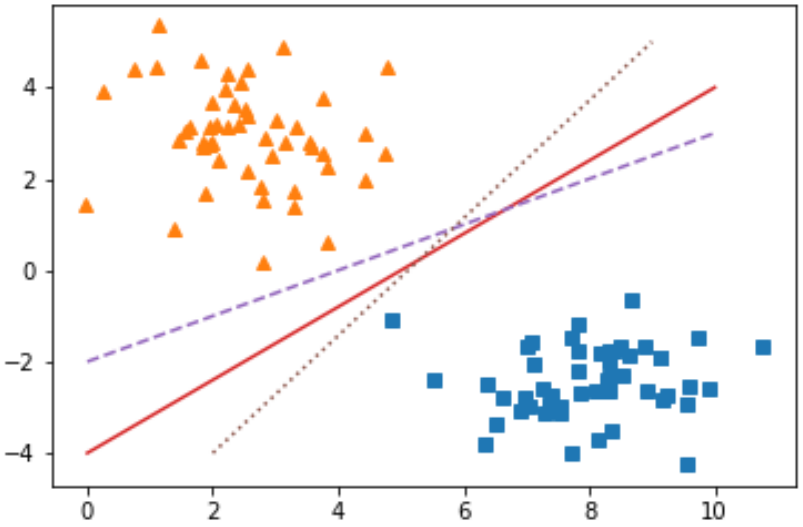
In a simple bag-of-words text classification model, the size of the input vector is defined by the number of distinct words in the training corpus.

# Why hidden layers?

Logistic regression is a **linear classifier**

Binary linear classifiers try to find a **straight line** that separates the instances of the two classes

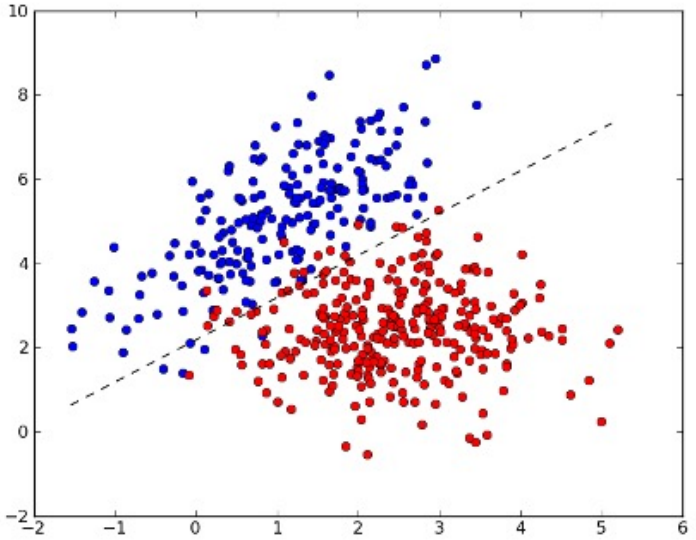
- This line is called **decision boundary**
- In higher-dimensional spaces, the line becomes a **hyper-plane**

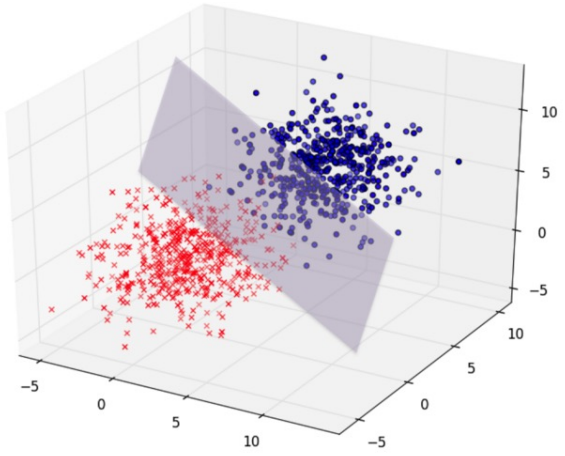


# Linear classifiers

A dataset is **linearly separable** if the classes can be separated by a straight line (or a hyper-plane)

- There are relatively simple problems that cannot be solved with a linear classifier
  - For example, the XOR problem

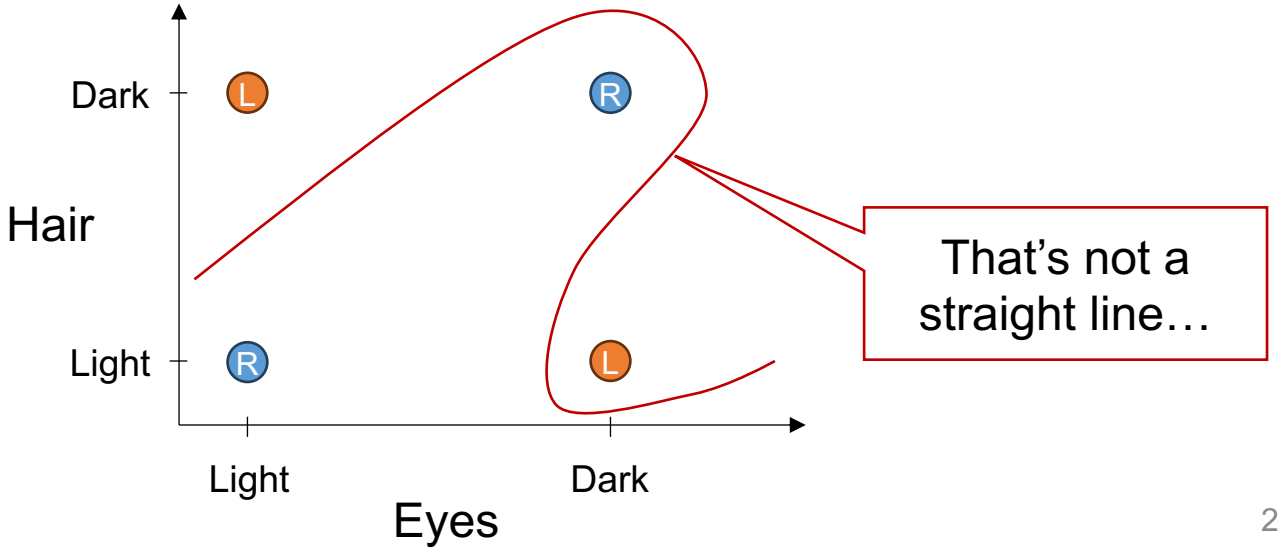


# The XOR problem

Let's imagine a world where:

- People have either light or dark hair
- People have either light or dark eyes
- People are right-handed whenever their hair and eye color matches, and left-handed when it doesn't match



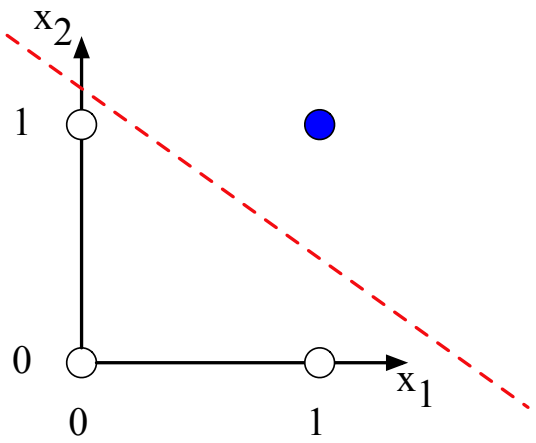
# The XOR problem

0 = right-handed
1 = left-handed

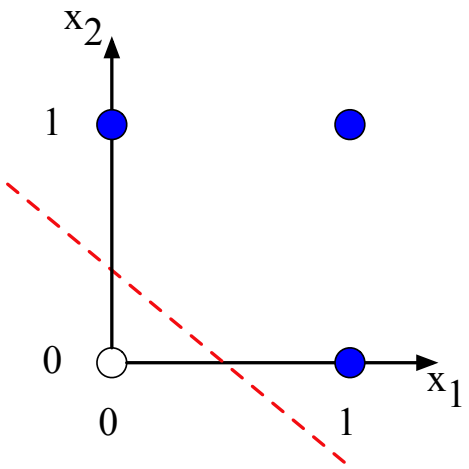
0 = light, 1 = dark
$x_1$ = eyes, $x_2$ = hair

$$x = [0, 0] \Rightarrow y = 0$$
$$x = [0, 1] \Rightarrow y = 1$$
$$x = [1, 0] \Rightarrow y = 1$$
$$x = [1, 1] \Rightarrow y = 0$$

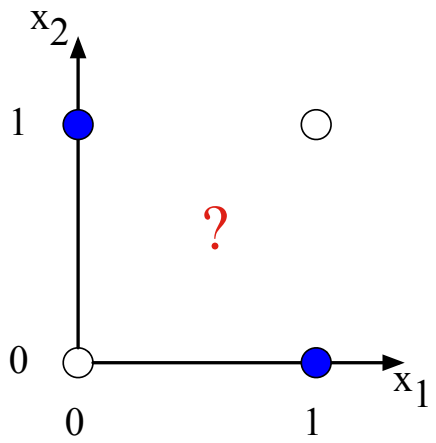
It is impossible to define a weight vector $\boldsymbol{w}$ that produces the correct predictions (using the dot product).




a) $x_1$ AND $x_2$




b) $x_1$ OR $x_2$




c) $x_1$ XOR $x_2$

# Extended features

We can extend the feature vector with combinations of existing features:

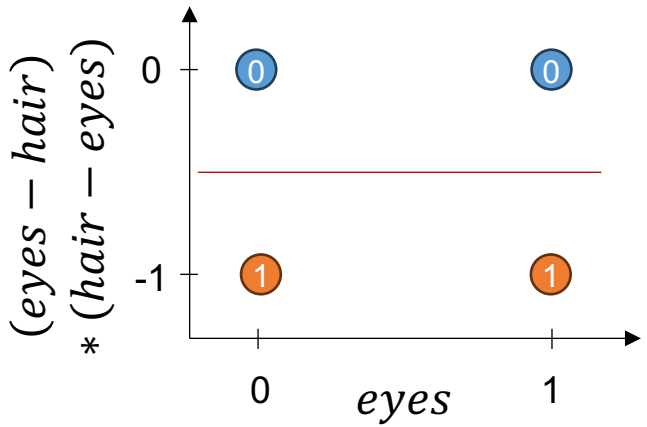
$$[x_1, x_2] \implies [x_1, x_2, x_1 * x_2, x_2 - x_1, \dots]$$

- Datasets which are not linearly separable may become separable when using extended features.

Let's try:

$$x = [eyes, hair, (eyes - hair) * (hair - eyes)]$$

- $x = [0, 0, 0]$ $y = 0$
- $x = [0, 1, -1]$ $y = 1$
- $x = [1, 0, -1]$ $y = 1$
- $x = [1, 1, 0]$ $y = 0$



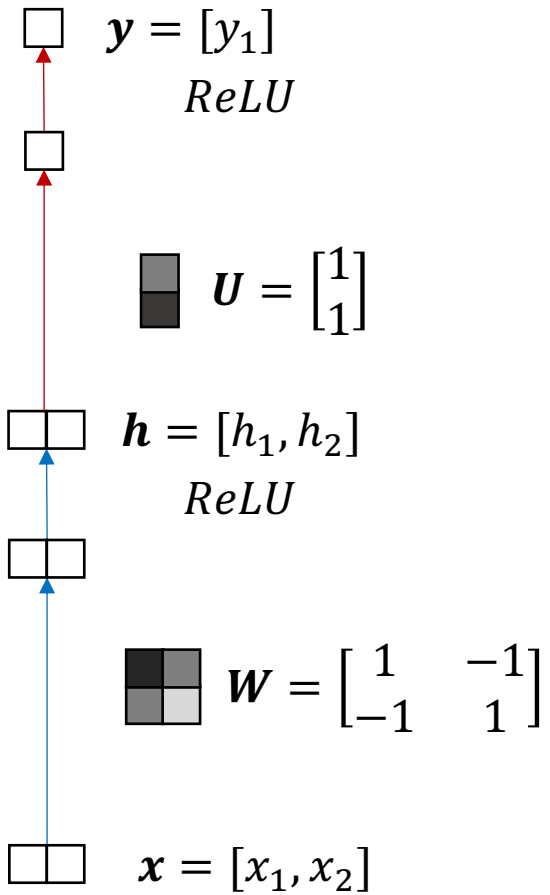
# Extended features

$$x = [eyes, hair, (eyes - hair) * (hair - eyes)]$$

- How did we figure that out???
  - XOR can be defined as $(A \vee \neg B) \wedge (B \vee \neg A)$
- Features have to be defined manually
- Real-life features generally don't correspond to propositional logic functions…

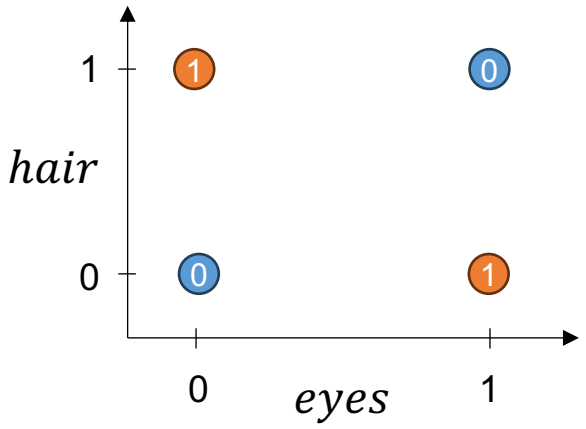
# The XOR problem in a neural network

| | | | | |
|---|---|---|---|---|
| $y$ | 0 | 1 | 1 | 0 |
| $h \cdot U$ | 0 | 1 | 1 | 0 |
| $h$ | $[0\ \ 0]$ | $[0\ \ 1]$ | $[1\ \ 0]$ | $[0\ \ 0]$ |
| $x \cdot W$ | $[0\ \ 0]$ | $[-1\ \ 1]$ | $[1\ \ -1]$ | $[0\ \ 0]$ |
| $x$ | $[0\ \ 0]$ | $[0\ \ 1]$ | $[1\ \ 0]$ | $[1\ \ 1]$ |

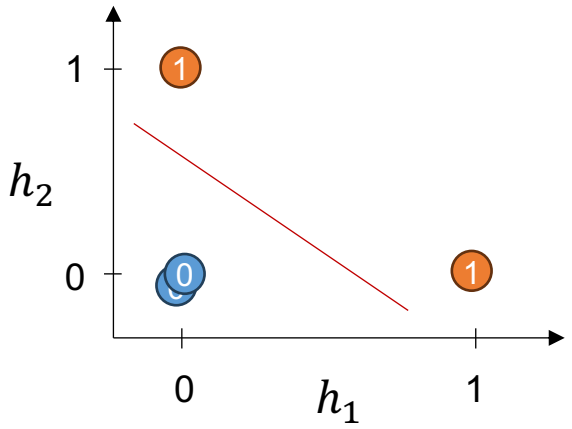


# Why does this work?

The original space:




The new hidden space:

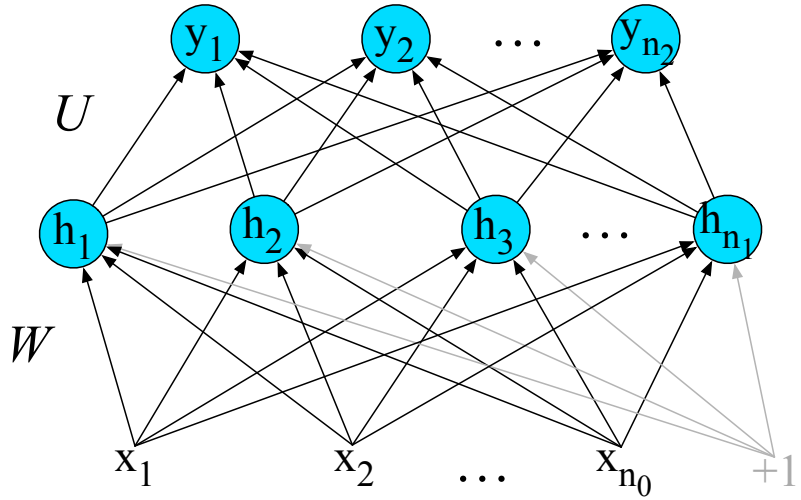


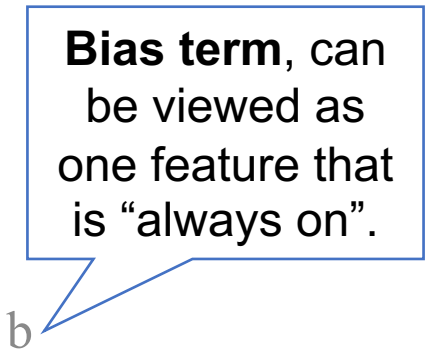

- This only works for a particular set of weight values in $W$ and $U$. But these can be learned automatically.

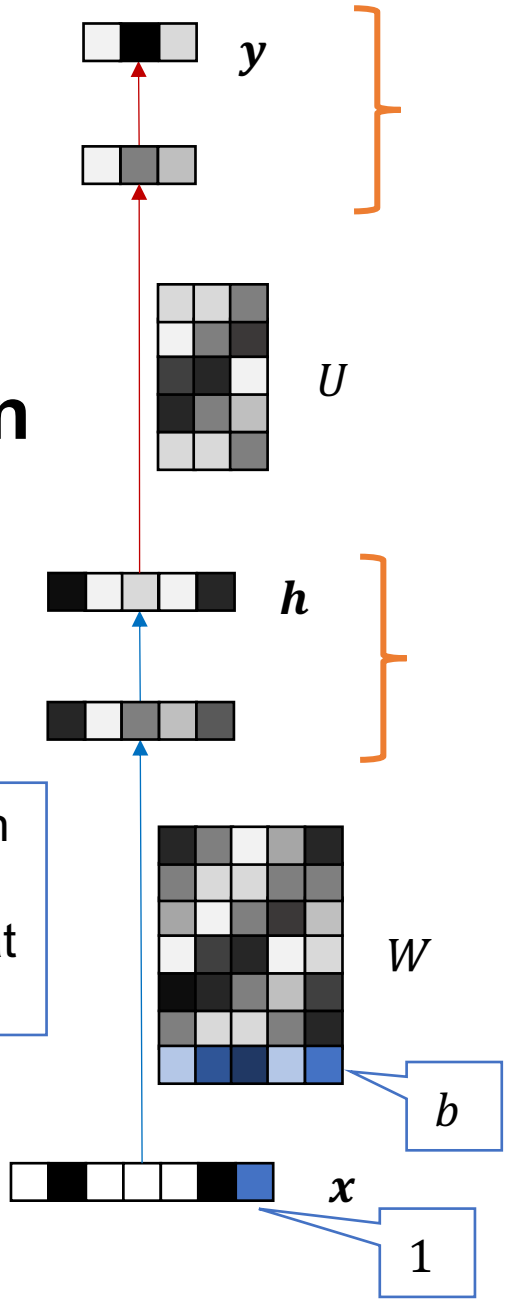
# Feed-forward neural networks

# Feed-forward neural networks

- All arrows go “upwards”
- Also called **multi-layer perceptron** (for historical reasons)
- Input units $x_i$, hidden units $h_i$, output units $y_i$

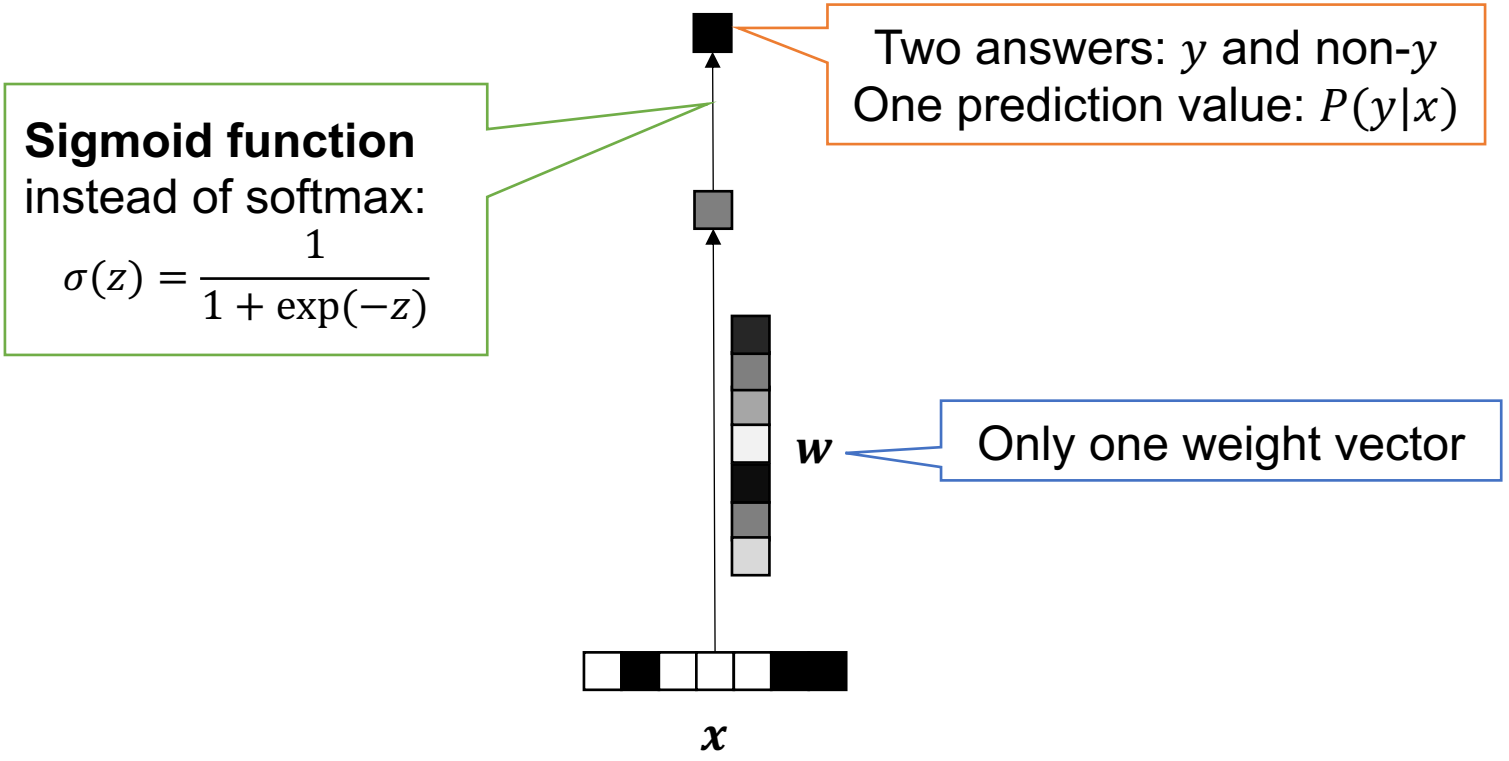


# Binary logistic regression

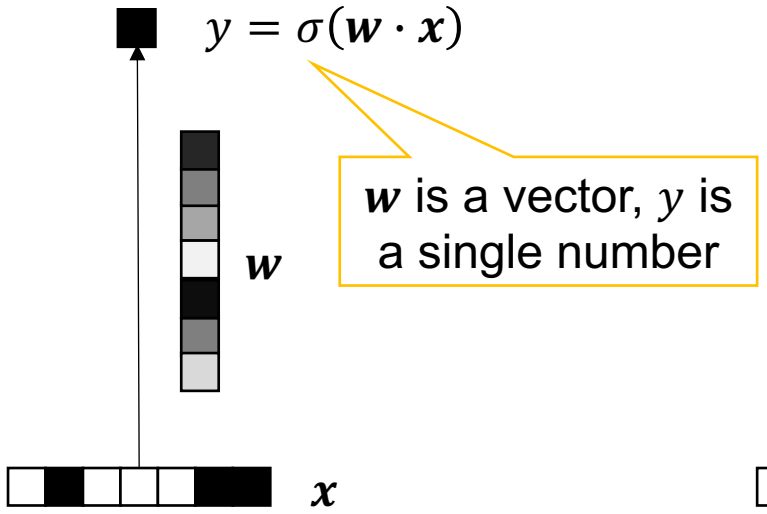
Example: Determine if a text has positive sentiment.



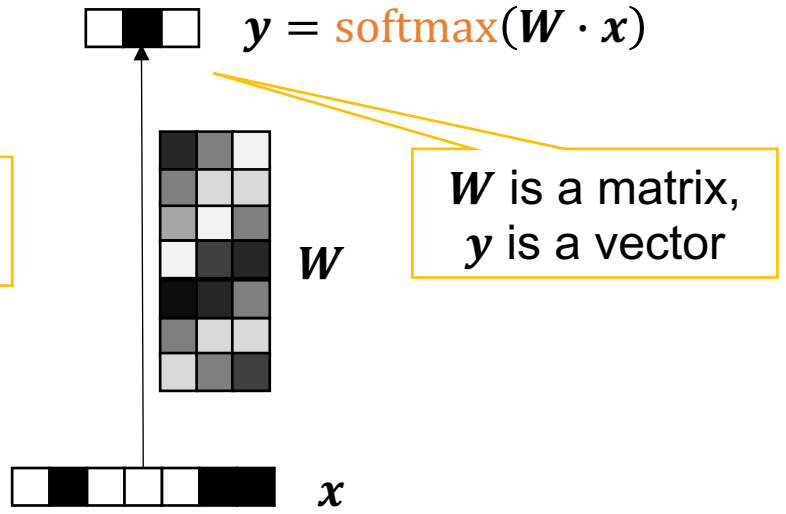
# Feed-forward neural networks

FFNN with 1 layer = logistic regression

**Binary LR:**




**Multinomial LR:**

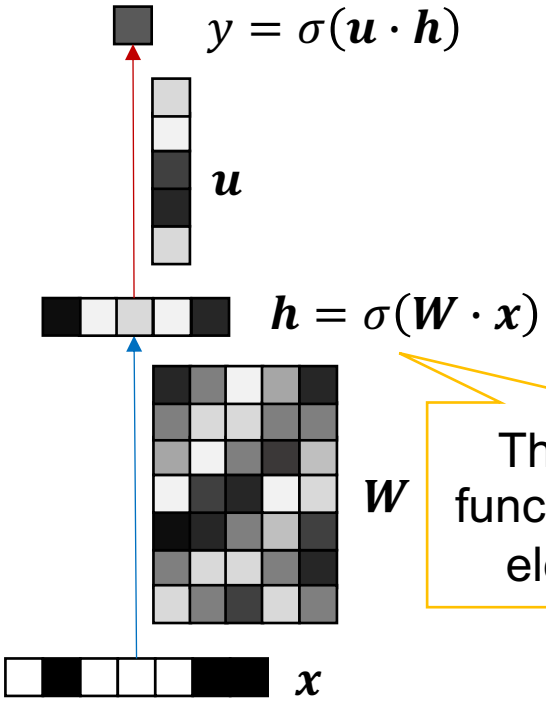


# Feed-forward neural networks

Two-layer networks

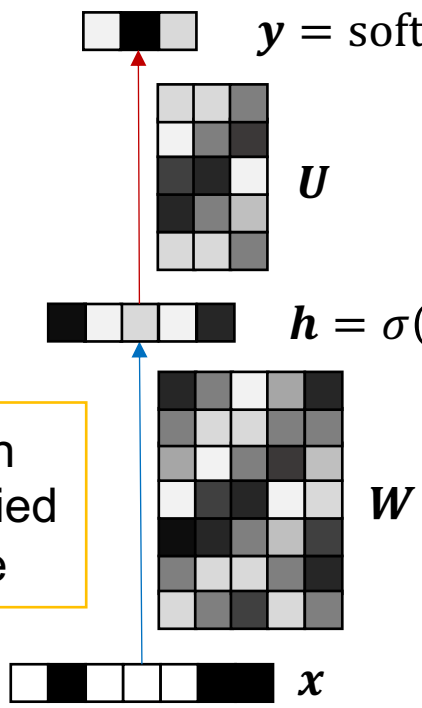
More layers can be added analogously

**Binary:**




The activation function is applied element-wise

**Multinomial:**



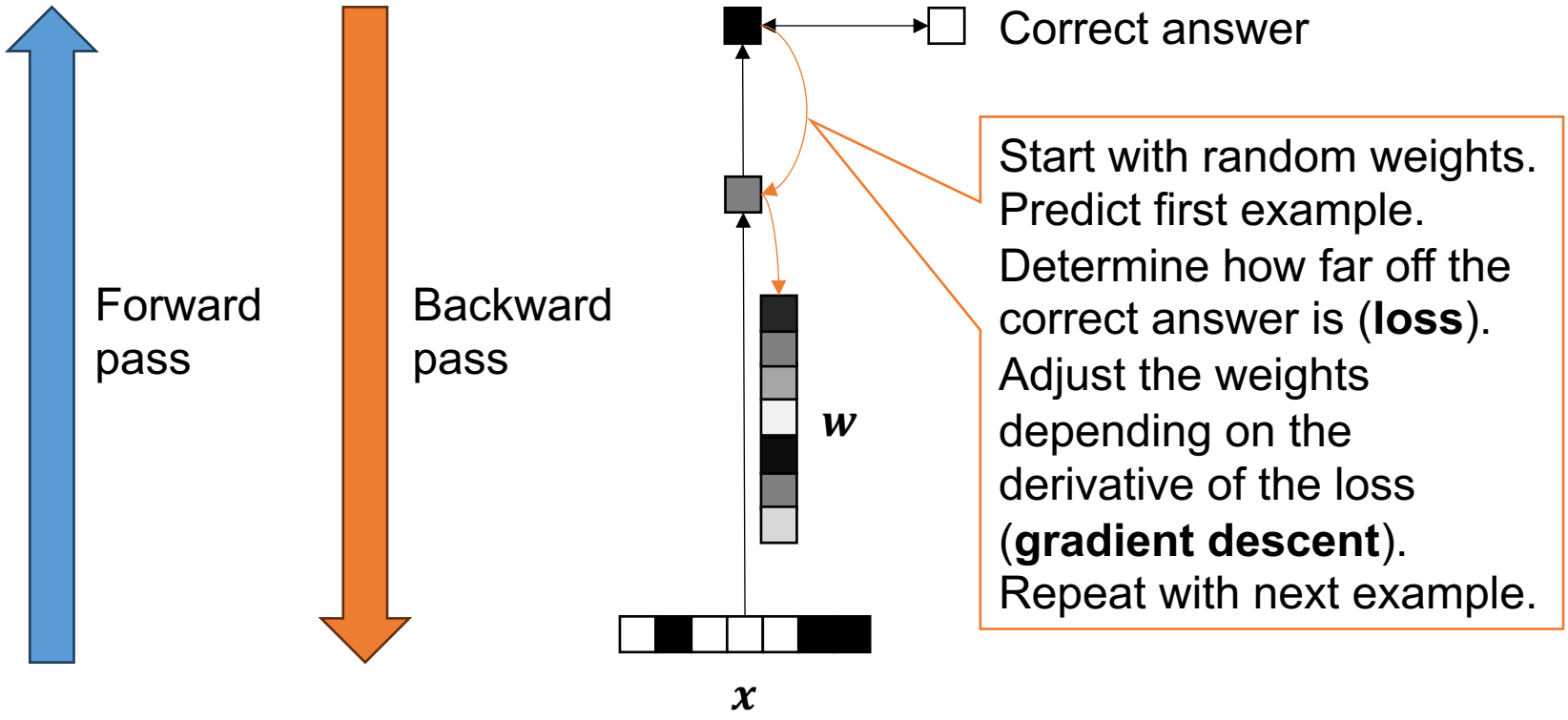

This needs to be a softmax to get proper probabilities

The activation function is applied element-wise

# Training neural networks

# Binary logistic regression Training

Example: Determine if a text has positive sentiment.



# Binary logistic regression Training

For binary logistic regression, we typically use the **cross-entropy loss**:

Goal: maximize probability of the correct label

Binary classification: only two options $\hat{y}$ and $1 - \hat{y}$

$$L_{CE}(\boldsymbol{w}) = -\log P(y|x;\boldsymbol{w})$$
$$= -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$
$$= -[y \log \sigma(\boldsymbol{w}\cdot\boldsymbol{x}) + (1-y)\log(1-\sigma(\boldsymbol{w}\cdot\boldsymbol{x}))]$$

**True label:** $y = 0$ or $y = 1$
**Predicted label:** $\hat{y} = \sigma(\boldsymbol{w}\cdot\boldsymbol{x})$

# Gradient descent for LR

Use the **derivative** of the loss function with respect to weights

$$\frac{d}{dw}L(\hat{y},y) = \frac{d}{dw}L(f(x;w),y)$$

to tell us how to adjust weights for each training item:

$$w_{i,j} \leftarrow w_{i,j} - \lambda \frac{d}{dw_{i,j}}L(\hat{y},y)$$

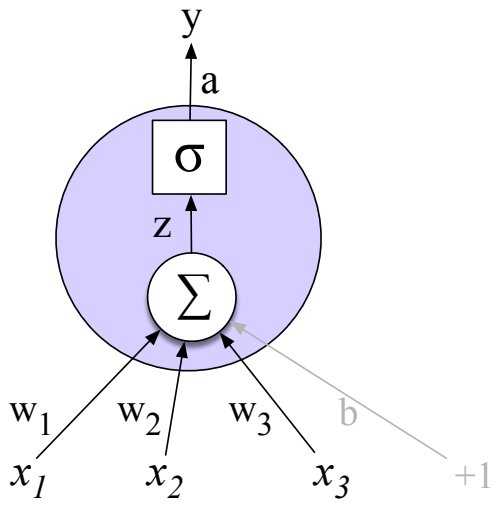
For logistic regression:

$$\frac{d}{dw_j}L_{CE}(\hat{y},y) = (\sigma(\boldsymbol{w}\cdot\boldsymbol{f}) - y)\cdot x_j$$

# Gradient descent for NN

A neural network layer is essentially the same as a logistic regression classifier.

- **Chain rule:** if $f(x) = u(v(x))$ then $\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$




- Idea:
  1. Compute the derivative of the loss
  2. Compute the derivative of the activation function
  3. Compute the derivative of the dot product

# 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 $u$ 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

# Computation graphs

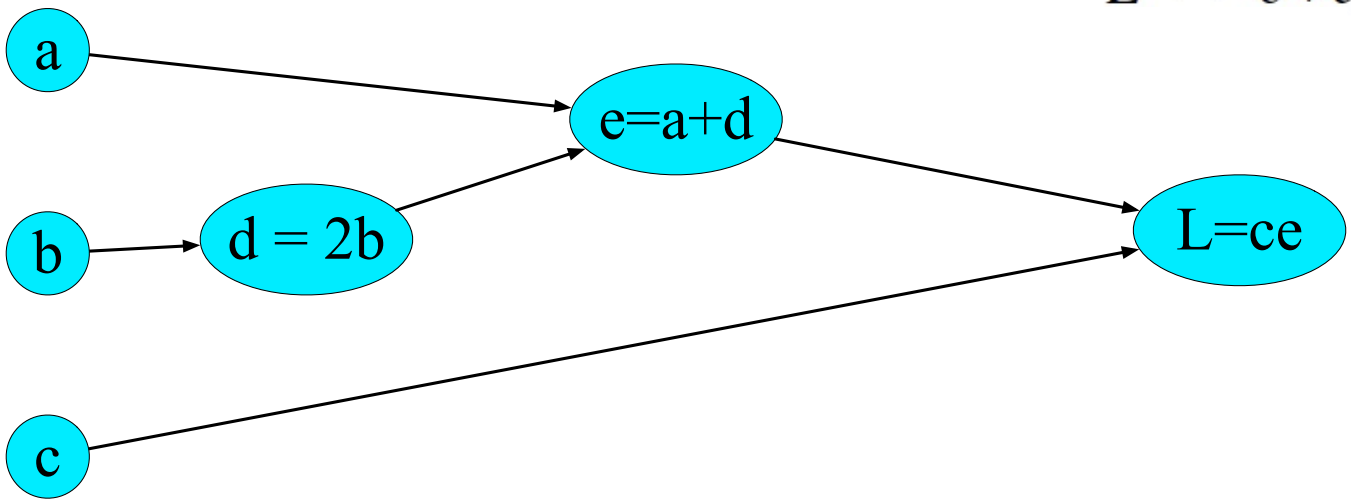
- What if we have a neural network with several layers?
  - We need the derivative of the loss with respect to each weight in **every layer** of the network.
  - But the loss can only be computed at the end of the network.
- Solution: **backpropagation**
  - “Distributes” the loss gradient over all the layers.
  - Relies on **computation graphs**.
  - A computation graph represents the process of computing a mathematical expression.

# Example

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

Computations:

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

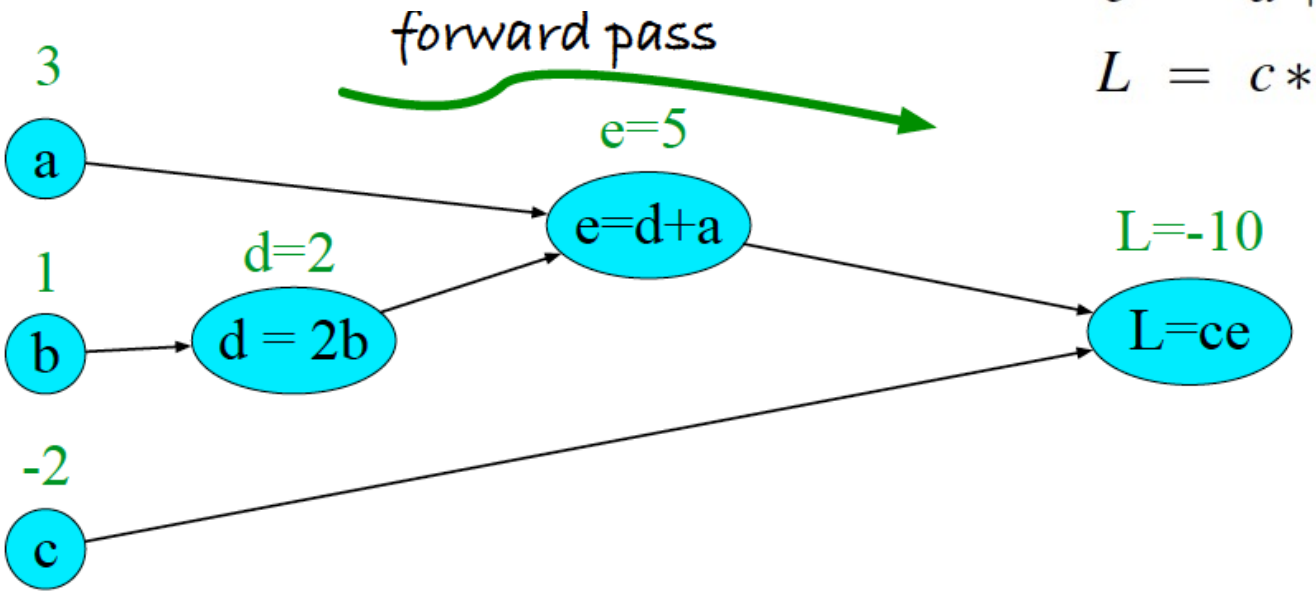


# Example

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

Computations:

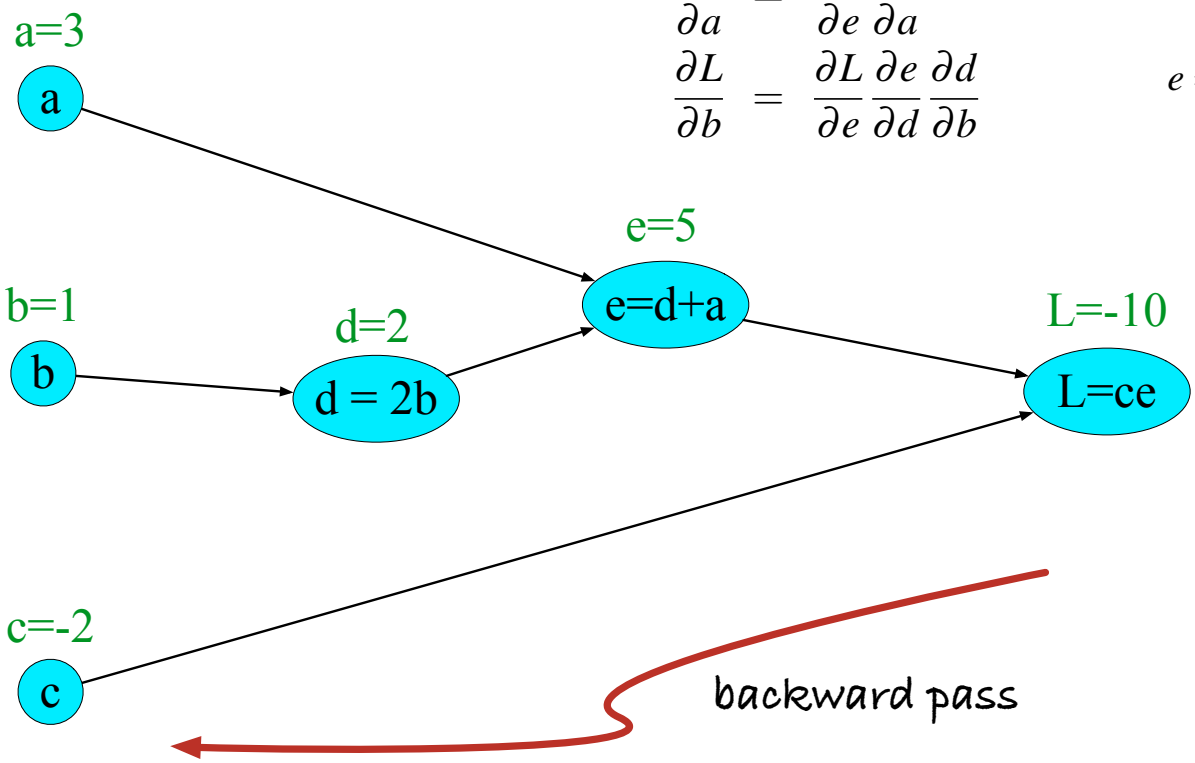
$$d = 2*b$$
$$e = a+d$$
$$L = c*e$$



# Example

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

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



# Example

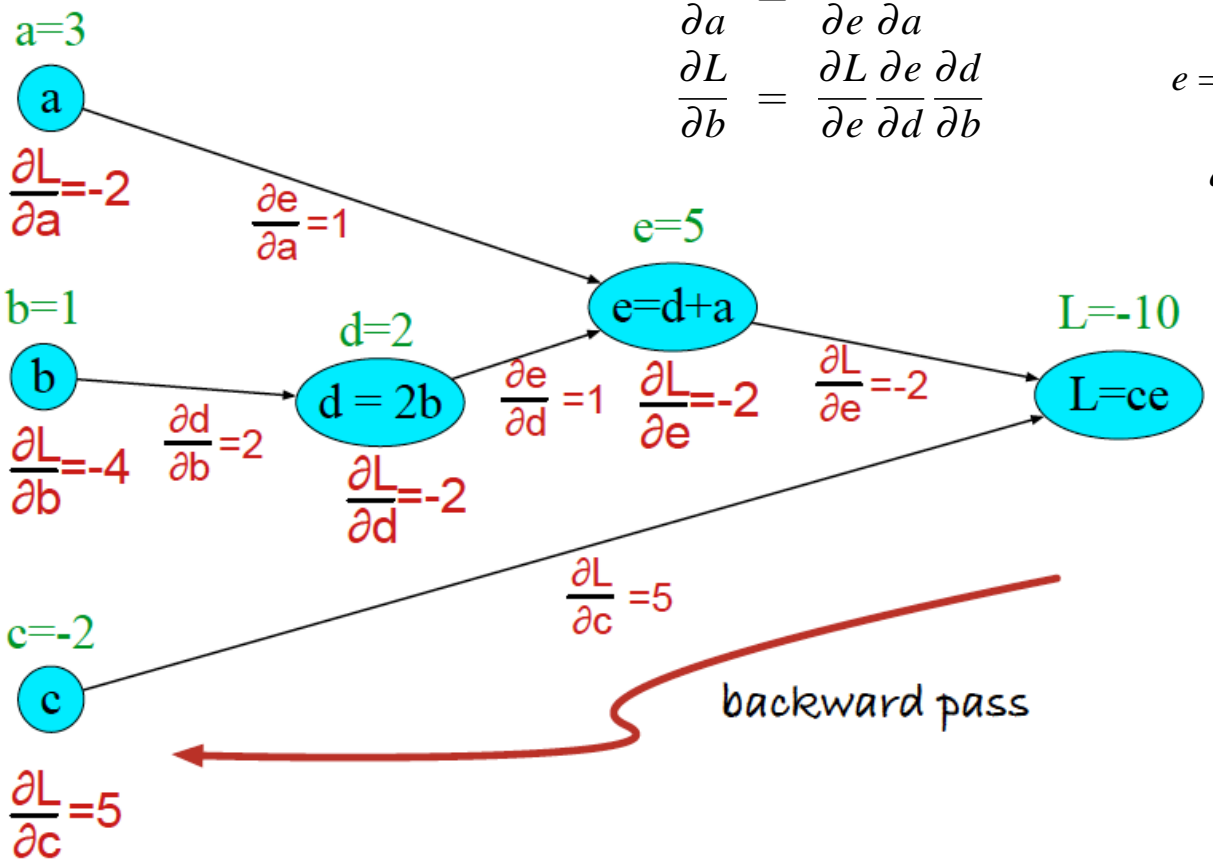
$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

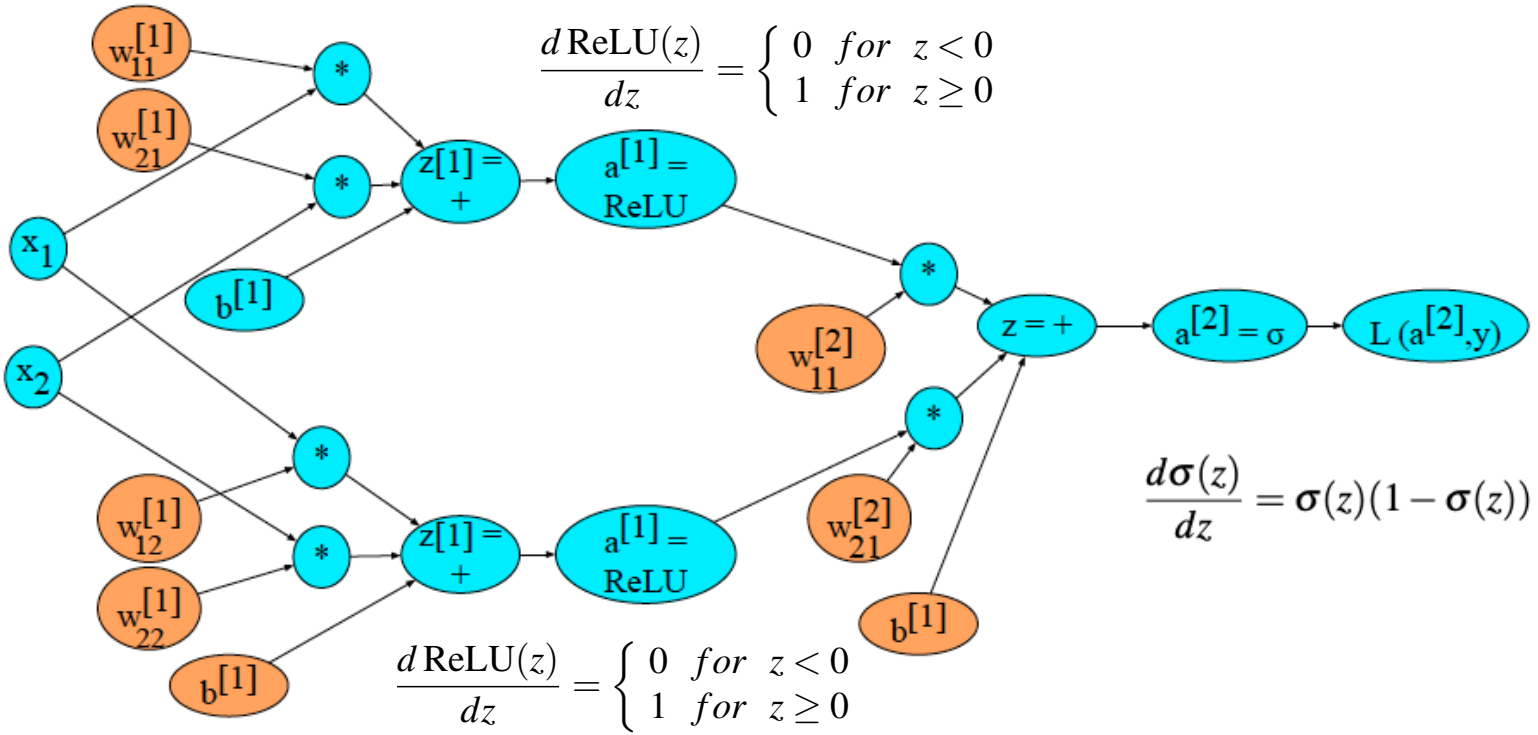
$$L = ce : \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

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

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



# Backpropagation in a two-layer network



# Summary

To train a neural network, we need to:

- Be able to represent the network as a computation graph
- Use only differentiable operations
  - Dot product is ok
  - Sigmoid, tanh and ReLU are also ok
  - Cross-entropy loss is usually fine
- Use a toolkit that knows how to do the complicated differentiation stuff automatically ☺
  - Write the forward pass function and let it determine the backward pass function

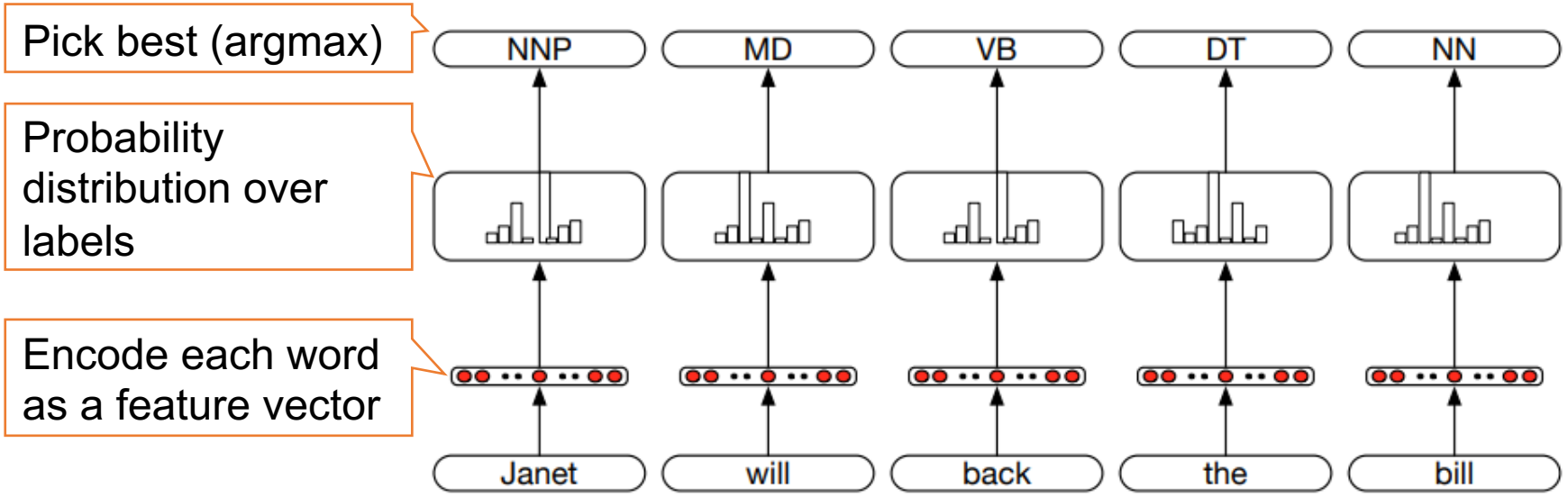
# Word vectors and embeddings

**IN4080**

**Natural Language Processing**

Yves Scherrer

# Sequence labelling with logistic regression




Feature vector:
Concatenation of word embeddings of current and context words



# Vector semantics

**How do we get vectors that represent the meaning of words?**

- Word-document (or term-document) matrices based on co-occurrence counts
  - Count weighting with tf-idf
- Word-context matrices based on co-occurrence counts
  - Dimensionality reduction (SVD, LSA)
    - Makes the vectors short and dense
- An alternative approach: word2vec
    - Creates short and dense vectors directly

# Vector semantics

**Why short and dense vectors?**

- Easier to use as feature vectors in machine learning (fewer weights to tune)
- Generalize better than explicit counts
- Capture semantic relations (synonymy, …) better

**Word embeddings:** Another name for short and dense word vectors

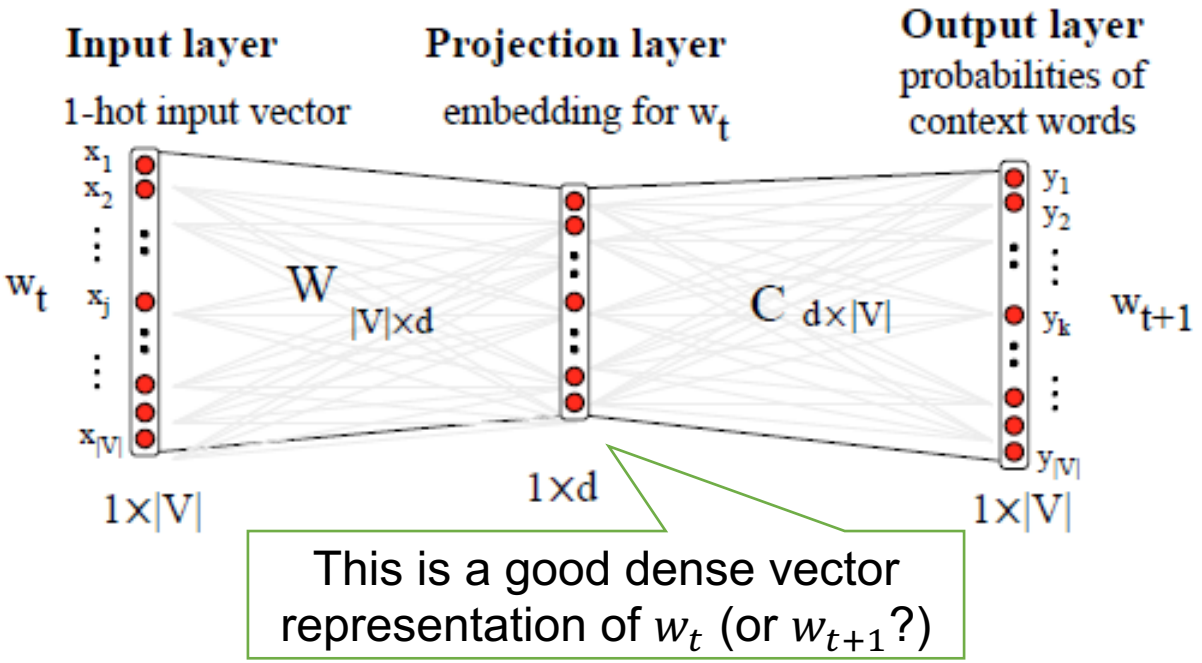
# Word embeddings

- Count-based methods:
  - Count how often each word $w$ occurs near "apricot"
- Alternative: **predict rather than count**
  - How likely is word $w$ to show up near "apricot"?
- We don't actually care about this task, but we take the hidden representations of the network as the word embeddings
- This is an instance of **self-supervision**:
  - Words that occur near "apricot" in the corpus can act as likely answers
  - Words that never occur near "apricot" in the corpus can act as unlikely answers
  - No need for human annotation

# Word embeddings

**First idea: a neural bigram language model**

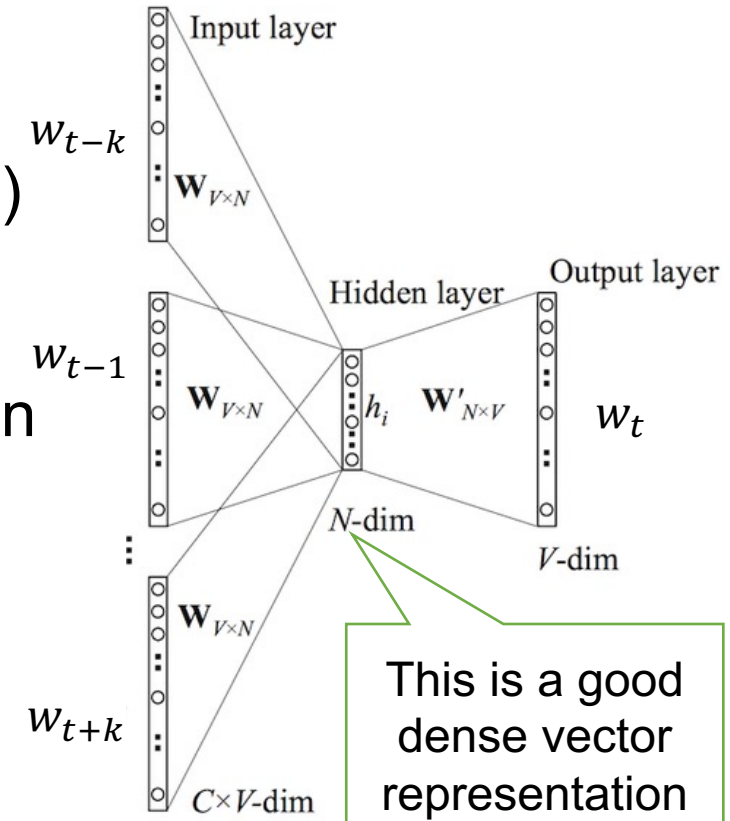
- Input: one-hot vector of word $w_t$
- One hidden layer
- Output: probability distribution over $w_{t+1}$



# Word embeddings

**Continuous bag-of-words (CBOW):**

- Input: concatenation (or sum) of context word vectors $w_{t-k}, \dots, w_{t-1}, w_{t+1}, \dots, w_{t+k}$
- Output: probability distribution for target word $w_t$

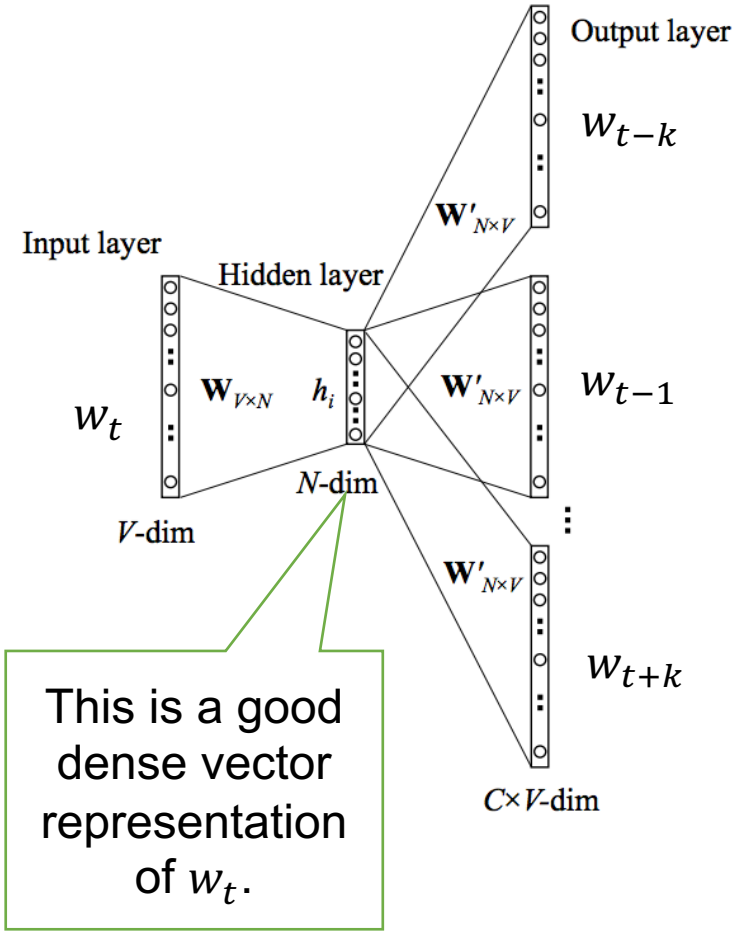


# Word embeddings

**Skip-gram:**

- The opposite of CBOW
- Target word as input
- Context words as output

- This requires $2k$ softmax calculations
  - Complicated
  - Expensive (vocabulary vectors are very large)



# Skip-gram with negative sampling (SGNS)

- Reformulate model as a binary classification task:
  - Input: one-hot vectors of two words $w_t$ and $w_c$
  - Output: probability that $w_c$ occurs in the context of $w_t$
- We need two types of training examples:
  - Word pairs with high probability (positive examples)
  - Word pairs with low probability (negative examples)
- Train a binary classifier on this data
  - Feed-forward neural network
  - A logistic regression classifier works just fine…