UNIVERSITÀ DEGLI STUDI DI MILANO

LA STATALE

Master Degree in Data Science and Economics

## Text Mining and Sentiment Analysis

# Introduction to Language Models

**Prof. Alfio Ferrara**

Department of Computer Science, Università degli Studi di Milano
Room 7012 via Celoria 18, 20133 Milano, Italia alfio.ferrara@unimi.it

sed noli modo

# Introduction



A **language model** in essentially a probability distribution over a sequence of words

$$P(w_1, w_2, \ldots, w_n)$$

which can be used for a surprisingly high number of tasks, including document search, document classification, text summarization, text generation, machine translation, and many others

**Note:** Instead of estimating the probability distribution of words, we can work at a finer granularity on the distribution of substrings of fixed length in words (e.g., characters, 2-chars blocks)

# Introduction

UNIVERSITÀ DEGLI STUDI DI MILANO
LA STATALE

**Example 1**

A LM may be used to guess the next word in a sequence

$$P(w_n \mid w_1, w_2, \ldots, w_{n-1})$$

**Example 2**

Or to guess the author (or any other categorical attribute) of as text

$$P(author \mid w_1, w_2, \ldots, w_n)$$

**Example 3**

Select the correct translation for a sentence

$$P(english \mid option1)\ vs$$
$$P(english \mid option2)$$

---

```
Yesterday > you >
studied, >
what > are > you >
going > to > do > … ?
                  today
```

```
"Twenty years from now
you will be more
disappointed
by the things that you
didn't do than by the
ones you did do"
                Mark Twain
```

"ci sono molti esempi"
> "there are many examples"
> ~~"are there many examples"~~

# Types of Language Models

**Statistical Language Models:** Estimate the probability distribution of words by enforcing statistical techniques such as n-grams maximum likelihood estimation (MLE) or Hidden Markov Models (HMM)

**Neural Language Models:** Popularized by Bengio et al. 2003, each word is associated with an embedding vector of fixed size and a Neural Network is used to estimate the next word given a sequence of preceeding words

We will see a general introduction to Neural Language Models

*Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. Journal of machine learning research, 3(Feb), 1137-1155*

# (Very fast) Introduction to neural networks

Neural Networks are composed by two or more **layers of nodes**, where each layer output provides an input for the subsequent layer. Such an input is combined with **weights** associated with the node connections before feeding the subsequent layer. In general terms, a NN can be seen as a tool for computing a combination of linear and non linear transformations of the form

$$\hat{\mathbf{y}} = \phi(W\mathbf{x} + b)$$

where $\hat{\mathbf{y}}$ is vector representing the **network prediction**, $\mathbf{x}$ is the **input vector** (the data), $W$ is a **matrix of weights** (that the network aims at learning), and $\phi(\cdot)$ is a (potentially) non linear transformation applied to data before the final prediction is returned, often called **activation function**

*For references on this lecture see Aggarwal, C. (2018). Machine learning for text (pp. 3121-3124). Cham: Springer International Publishing.*

# Learning

The NN main goal is to find (**learn**) the value of its parameters $\Theta$ (the weights *W* and the bias *b*) that make it possible to obtain a prediction $\hat{\mathbf{y}}$ as much close as possible to the real output ***y***, which is known from the examples available in the **training set** (**supervised learning**).

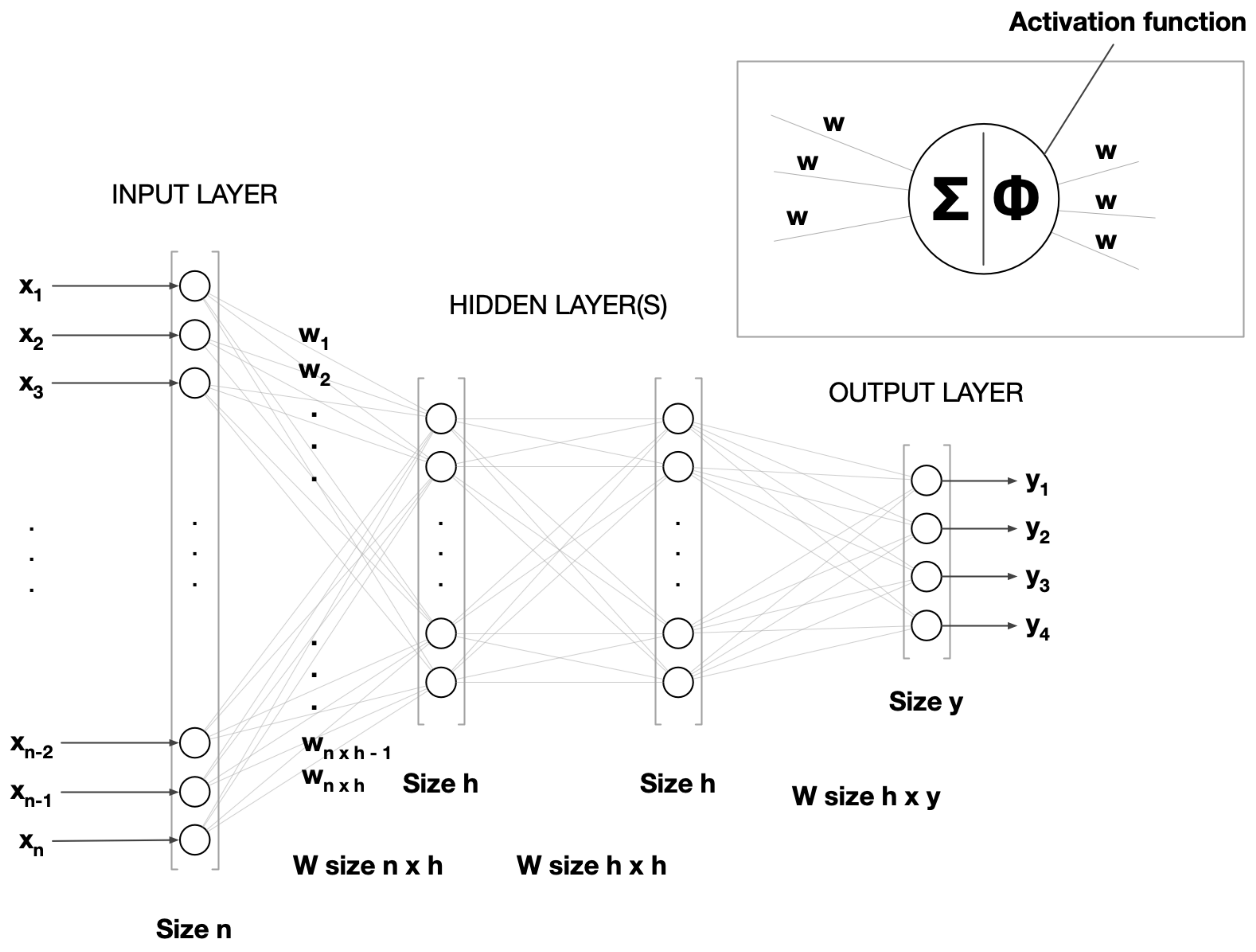
To perform learning, several **iterations** are tried starting with random values for the weigths. For each iteration we update weights and we base the subsequent iteration on the feedback provided by the **error in the prediction**, with the goal of **minimizing the error**. A crucial component of NN is thus to have a function that measures the error, called **loss function**. Given *t* as one of the iterations, $\eta$ being the learning rate, $L(\cdot)$ the loss function, and *X* represents input data. The general form of the learning step is

$$W^{t+1} \leftarrow W^t + \eta L(X)X$$

# Overview

When neural networks are used for **textual data**, usually we have:
**Input** is **word vectors**, either sparse or dense
Dense vectors for words can be given as input (i.e., used a pre-trained models, such as **word embeddings**, **LDA**) or can be calculated by the network itself as part of the training process, starting from a sparse word representation (such as **one-hot encoding**, **co-occurrence count**, **n-grams matrix**)

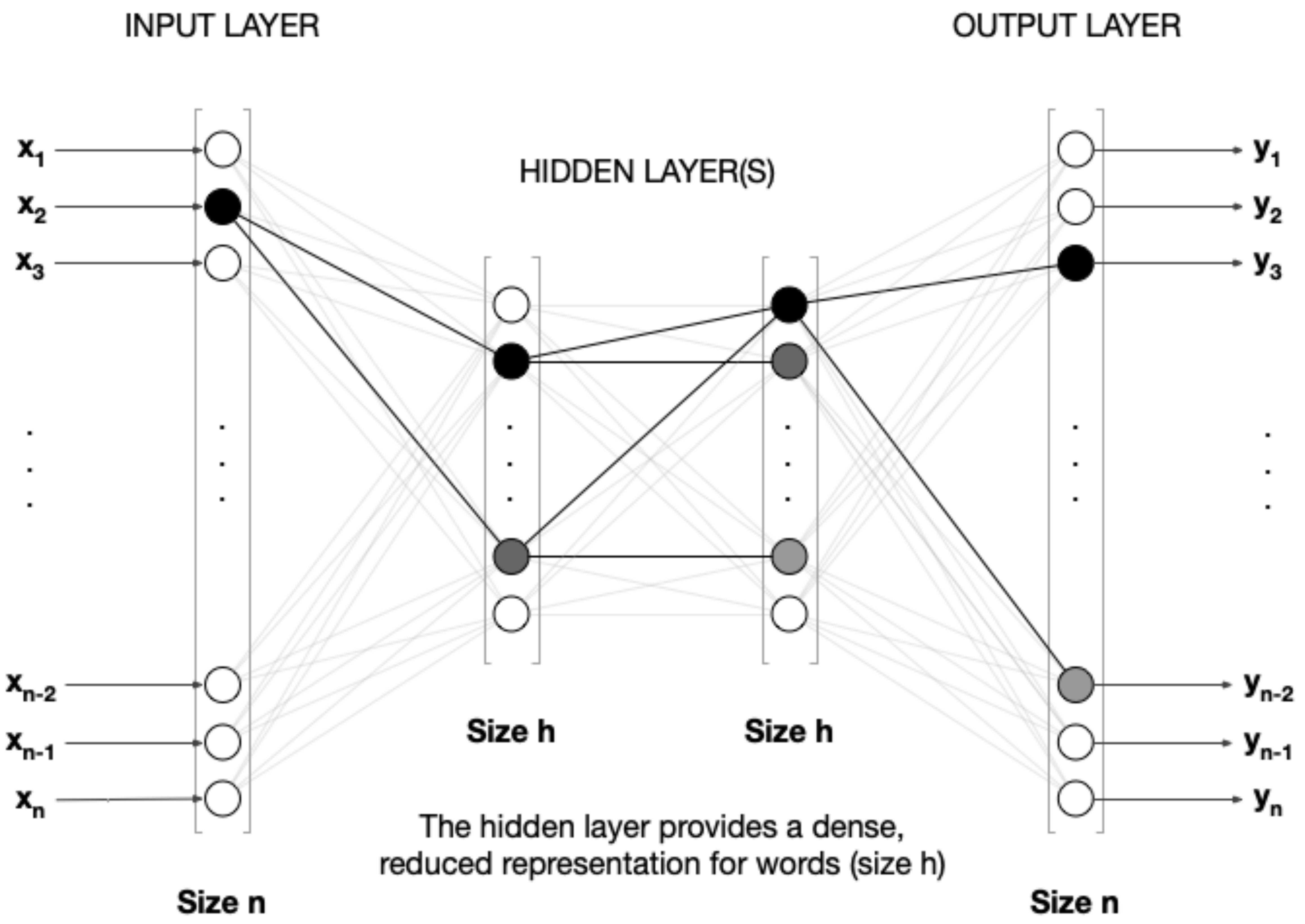
The **output** depends on the task (**multi-class** or **multi-label classification**, **autoencoders**)



# Autoencoders

Autoencoders are special network architectures where an input (represented by an input layer of size *n*) is mapped onto itself (an output layer of size *n*), such as when a network is used to map words on other words.

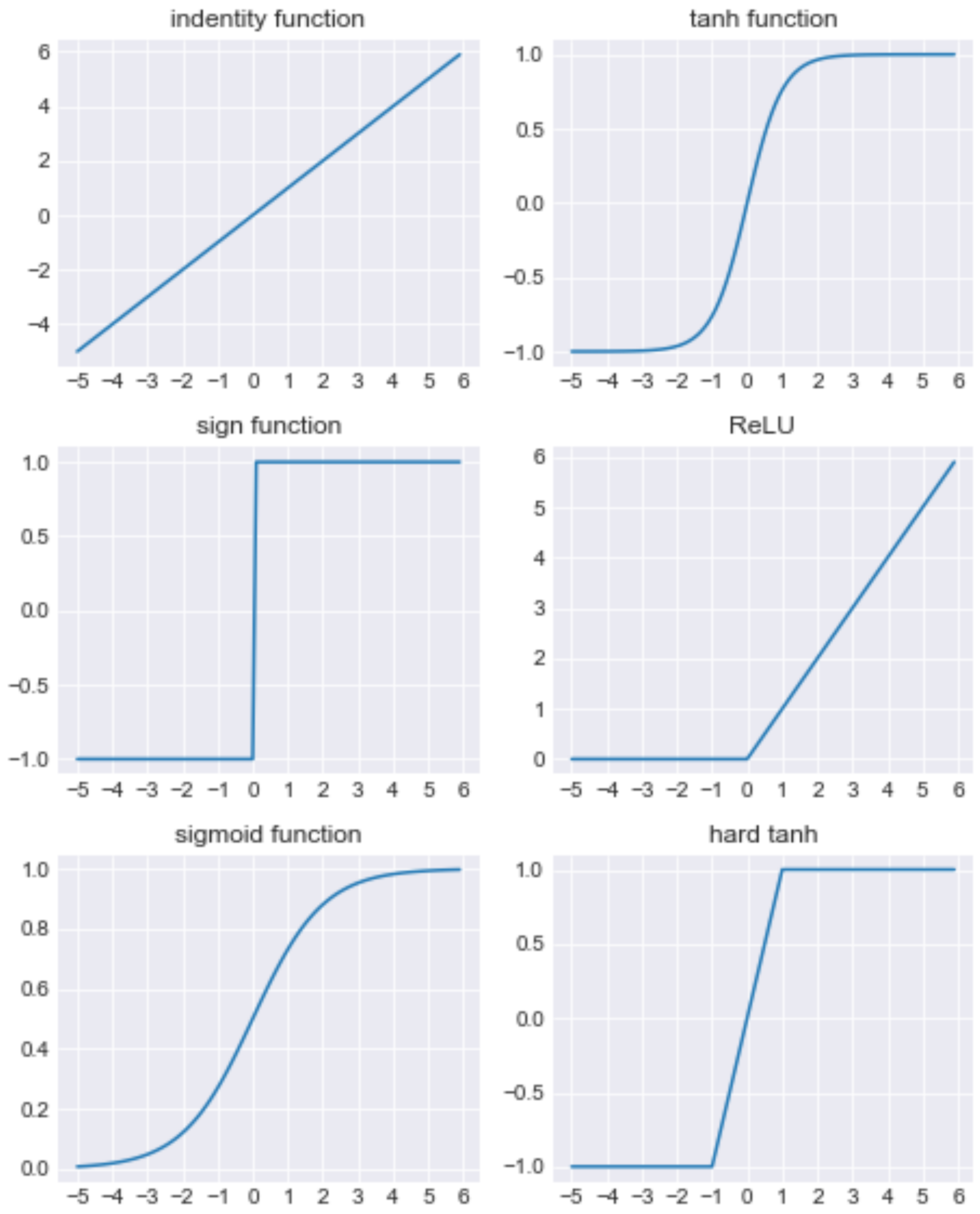
An example that we have seen is Word2vec. An interesting side-effect of this architecture is that we can take the hidden layer as a reduced dense represetation of the input (as in word embeddings)



# Activation function

There are many options for the choice of the activation function $\phi(\cdot)$ having that $\hat{y} = \phi(Wx)$

- $\phi(x) = x$ (indentity)
- $\phi(x) = x^+$ (sign)
- $\phi(x) = \dfrac{1}{1+\epsilon^{-x}}$ (sigmoid)
- $\phi(x) = \max\{x, 0\}$ (**Re**ctified **L**inear **U**nit)
- $\phi(x) = \dfrac{\epsilon^{2x}-1}{\epsilon^{2x}+1}$ (tanh)
- $\phi(x) = \max\{\min\{x, 1\}, -1\}$ (hard tanh)



# Output nodes

When we need to predict one (or more) outputs among multiple options, such as in case of multi-class (or multi-label) classification or with autoencoders, a very common choice is to model the output layer as a **softmax** layer, in order to enforce a probabilistic interpretation of the results.

Having an output with *n* dimensions:

$$\phi(x)_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp x_j} \; \forall i \in \{1, \ldots, n\}$$

# Loss functions

The use of *softmax* produces a probabilistic output, which requires also a some constraints on the choice of the loss function

**Binary target (logistic regression)**

$$L = \log(1 + \exp(-y \cdot \hat{y}))$$

**Categorical targets (cross-entropy loss)**

given $\hat{y}_1, \ldots, \hat{y}_k$ as the probabilities predicted for each of the *k* targets, assume that $r \in \{1, \ldots, k\}$ is the correct target according to the training set for the instance under evaluation. Then the cross-entropy loss is

$$L = -\log(\hat{y}_r)$$

# Multilayer networks

When a network has multiple hidden layers, you can see it as a composition of functions of the form $h_n(\ldots h_1(f(x)))$, where each function $h_i$ corresponds to the $i$th hidden layer.

However, this makes training more difficult, because the error gradient has to be back-propagated through the layers.

The algorithm performing training is articulated in two phases, a **forward phase** and a **backward phase**.

# Multilayer networks

**Forward**: The input produces a forward cascade of computation across the layers. The final output is compared with the training set expected output and the **derivative of the loss function is computed**.

**Backward**: Denote $w_{h_{r-1},h_r}$ as the weights of the transition between layer $h_{r-1}$ and $h_r$ and consider to have $h_1, \ldots, h_k$ hidden layers before the output layer *o*. The Loss function derivative is decomposed along the path from $h_1$ to $h_k$ as follows

$$\frac{\partial L}{\partial w_{h_{r-1},h_r}} = \frac{\partial L}{\partial o} \left[ \frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right] \frac{\partial h_r}{\partial w_{h_{r-1},h_r}} \ \forall r \in 1 \ldots k,$$

Where the component $\frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i}$ has to be aggregated (summed) for each pattern of nodes connecting $h_r$ to *o*.

# Recurrent Neural Networks (RNN)

When dealing with **sequence-to-sequence** learning, we aim at predicting the value $s_i$ in a sequence (e.g., the *i*th word in a text) given the previous $s_{i-n+1}, \ldots, s_{i-1}$ sequence elements (e.g., the previous words).

The issue here is that with a feedforward network, each prediction $\hat{s}_i$ may be based on data about the previous elements in the sequence (such as for n-gram models), but it is **independent** from the previous predictions of the network itself.

The idea of **Recurrent Neural Networks (RNN)** is instead to use the output $\hat{s}_i$ of the network on the instance $s_i$ as an input (together with data) for the subsequent prediction(s) $\hat{s}_{i+j}$

# Recurrent Neural Networks (RNN)

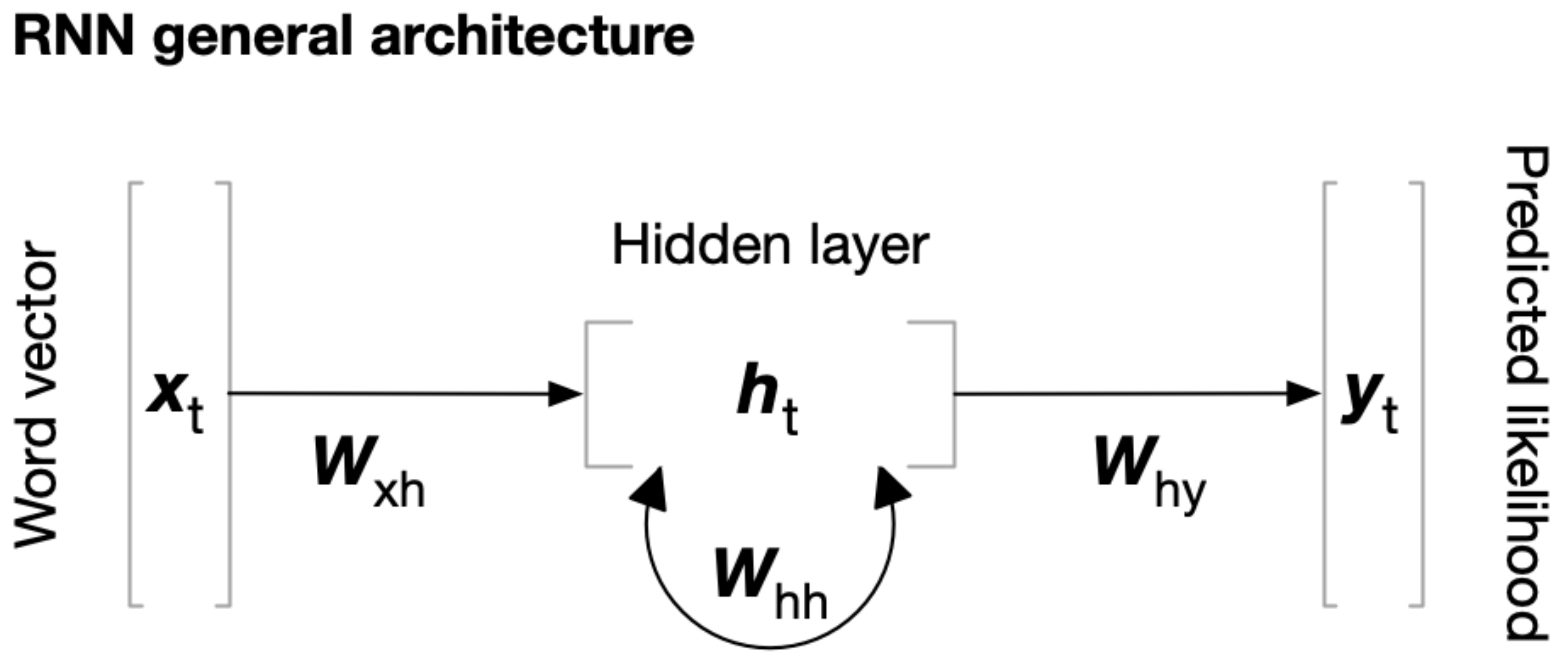
The basic idea is that the state of the hidden layer $h_t$ at time *t* if a function of the form

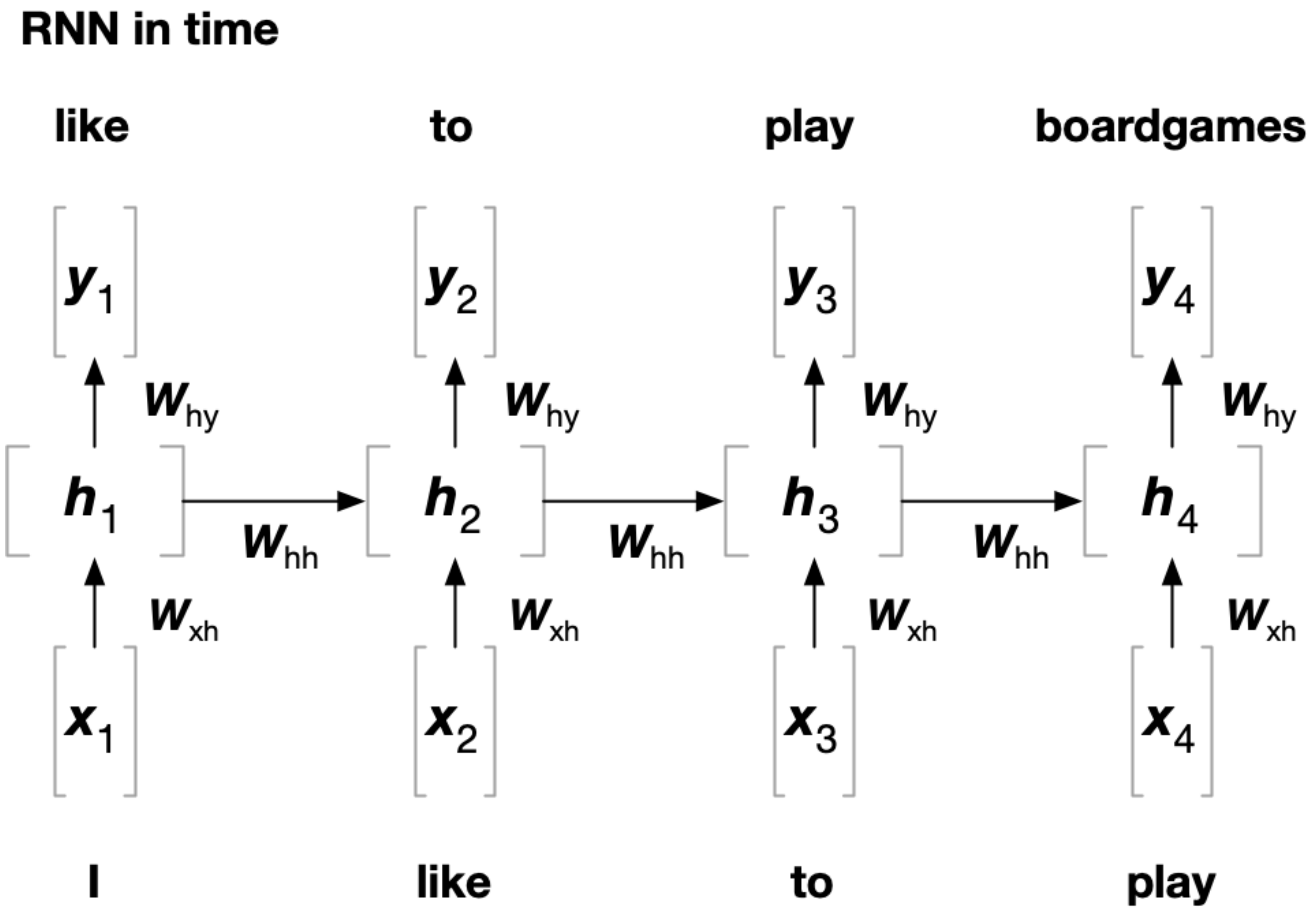
$h_t = f(h_{t-1}, x_t)$

Given *d* as the data dimensions (e.g., the vocabulary for language models) and *p* as the dimension of the hidden layers, we will have to work with three matrices of weights: $W_{xh} \in \mathbb{R}^{p \times d}$, $W_{hh} \in \mathbb{R}^{p \times p}$, and $W_{hy} \in \mathbb{R}^{d \times p}$ for input to hidden layer, hidden layer to hidden layer, and hidden layer to output. Thus we will have:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1}), \quad y_t = W_{hy}h_t$$

# Recurrent Neural Networks (RNN)



# RNN Training

The softmax probabilities of the correct words at various time-stamps are aggregated to create the loss function.

The back-propagation algorithm is updated to **backpropagation through time (BPTT)**.

1. running the input sequentially in the forward direction through time and computing the error/loss at each time-stamp (same as BP)
2. computing the changes in edge weights in the backwards direction on the network without any regard for the fact that weights in different time layers are shared (same as BP)
3. adding all the changes in the (shared) weights corresponding to different instantiations of an edge in time (BPTT specific)

# Intuition of Long Short Term Memory networks (LSTM)



In RNN a common problem is that successive multiplication by the weight matrix is highly unstable (*vanishing/exploding gradients* problem).

This problem is an issue especially for long sequences, requiring the network to have a **long memory**.

To address the problem, in LSTM we introduce a new hidden vector of $p$ dimensions, referred to as the *cell state*. The cell state is a kind of long-term memory that retains at least a part of the information in earlier hidden states by using a combination of partial *forgetting* and *increment* operations on previous cell states.

See further details in Section 10.7.7.1 of Aggarwal, C. (2018). Machine learning for text (pp. 3121-3124). Cham: Springer International Publishing.