Foundations for Natural Language Processing

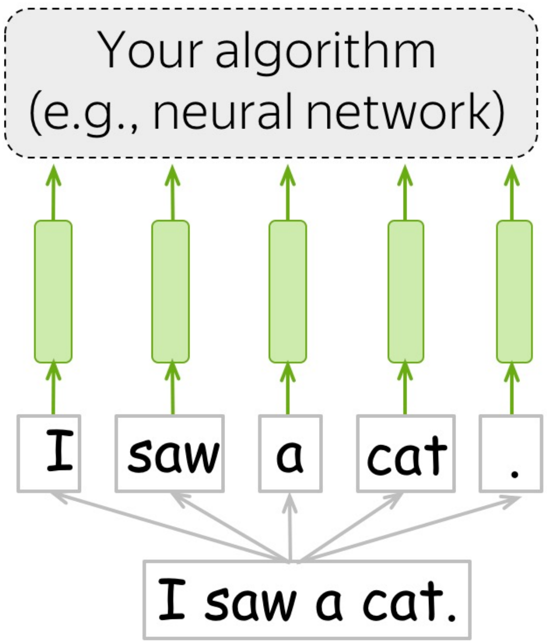
Neural Classifiers

Ivan Titov

with graphics / materials are from
Lena Voita and Edoardo Ponti

School of informatics

Neural models and word embeddings



Any algorithm for solving a task

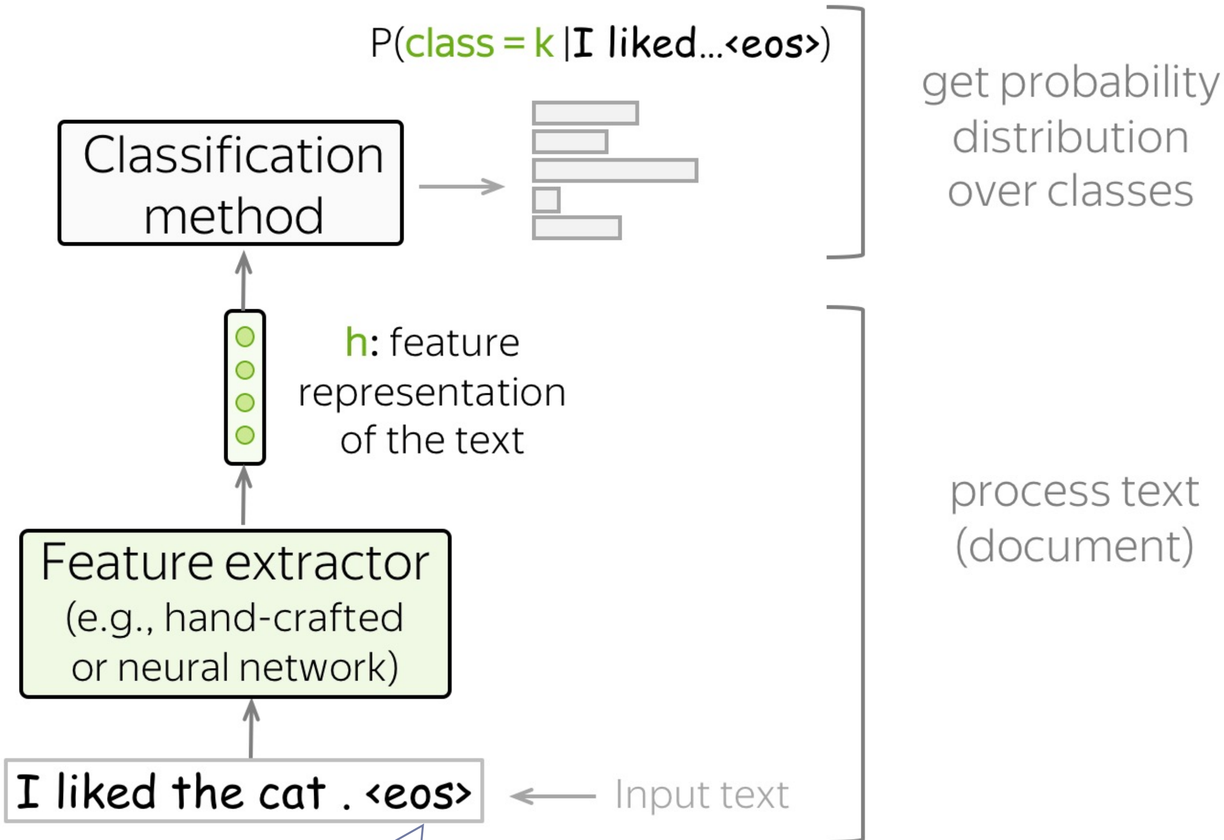
Word representation - vector
(input for your model/algorithm)

Sequence of tokens

Text (your input)

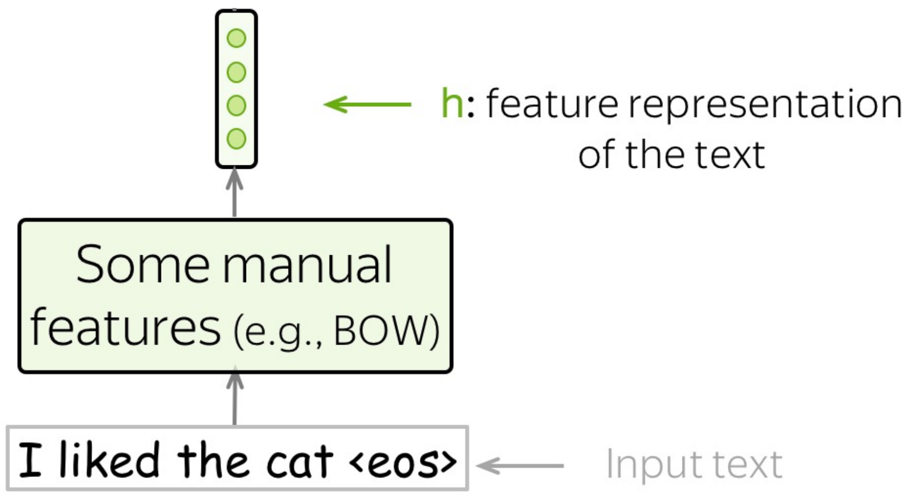
Classification

General Classification Pipeline



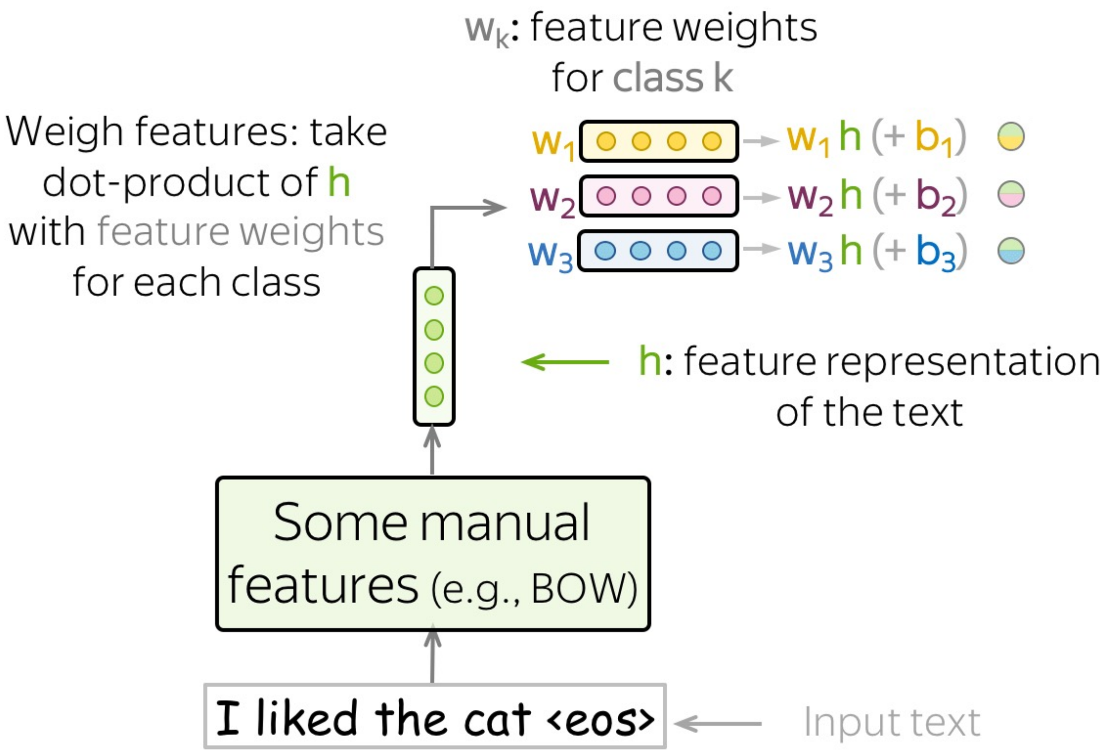
This is a "end-of-sentence" token, it will become soon clear why we often should use it

Logistic regression



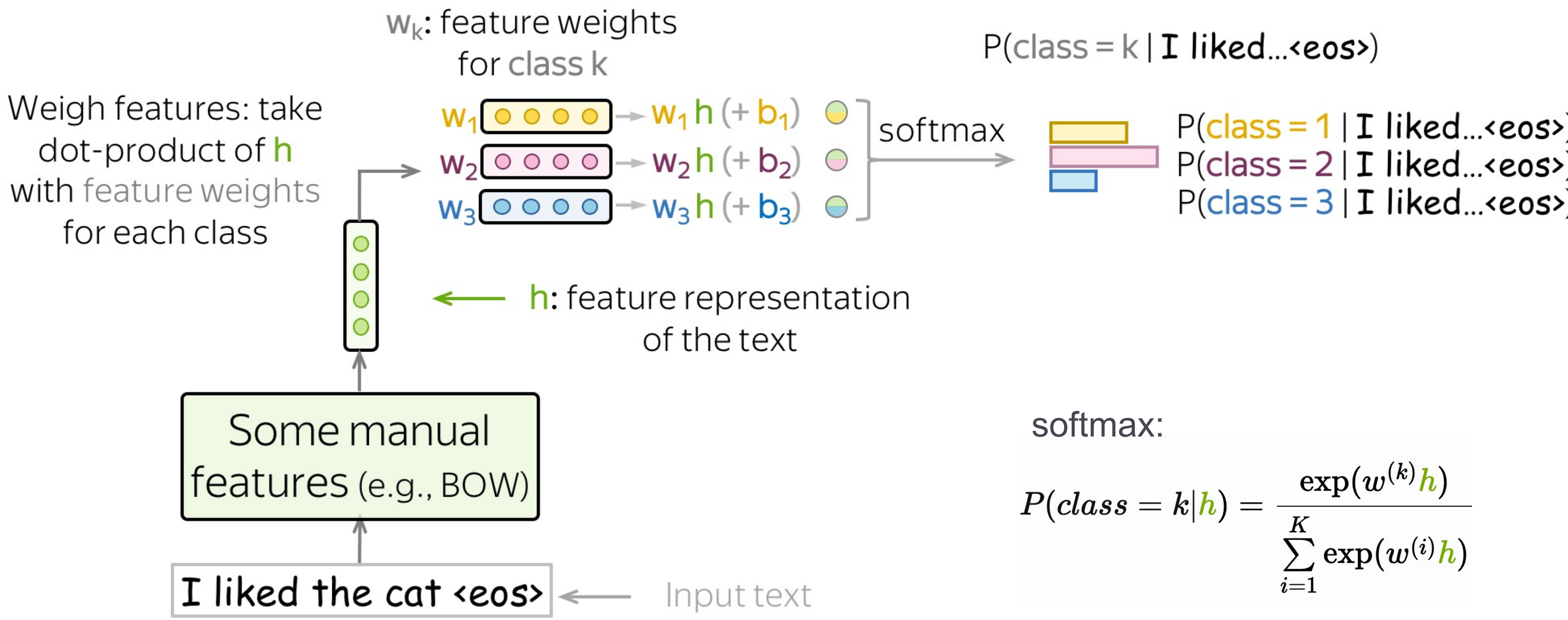
Note a slight change to notation from the previous lecture

Logistic regression



Note a slight change to notation from the previous lecture

Logistic regression

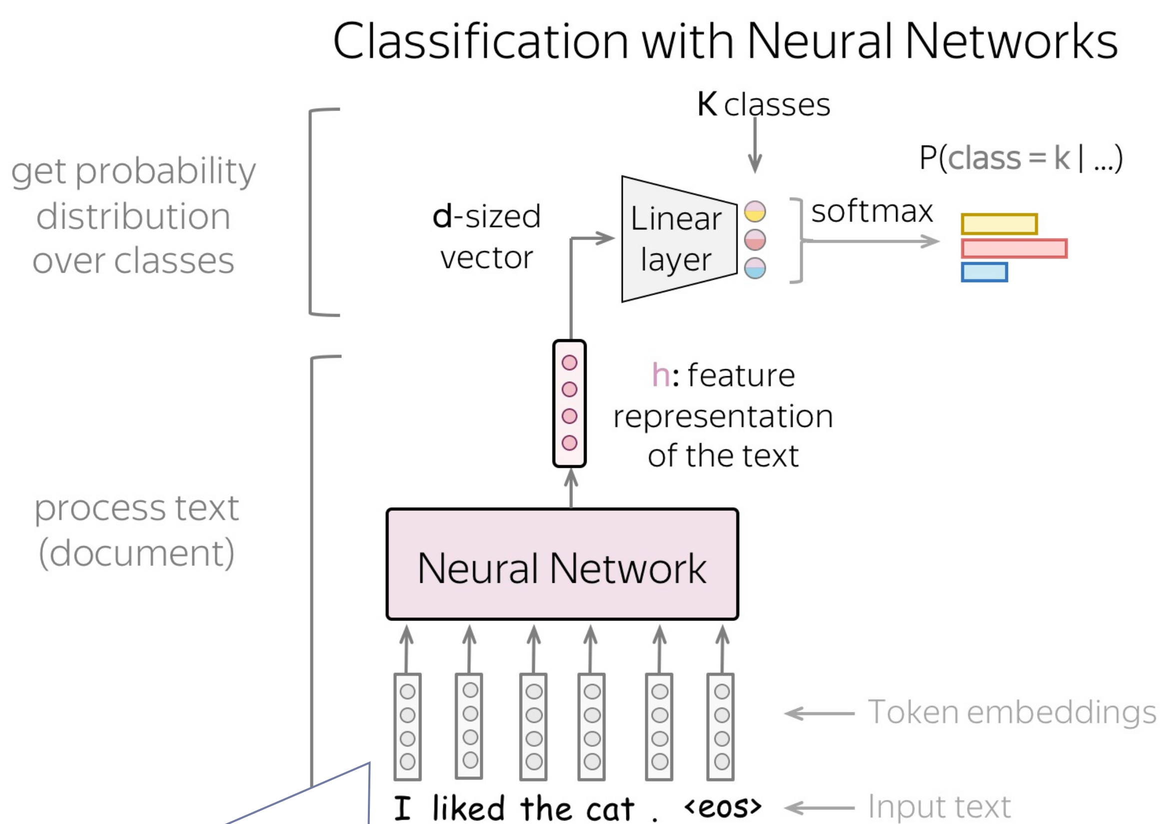


softmax:

$$P(class = k|h) = \frac{\exp(w^{(k)}h)}{\sum_{i=1}^{K}\exp(w^{(i)}h)}$$

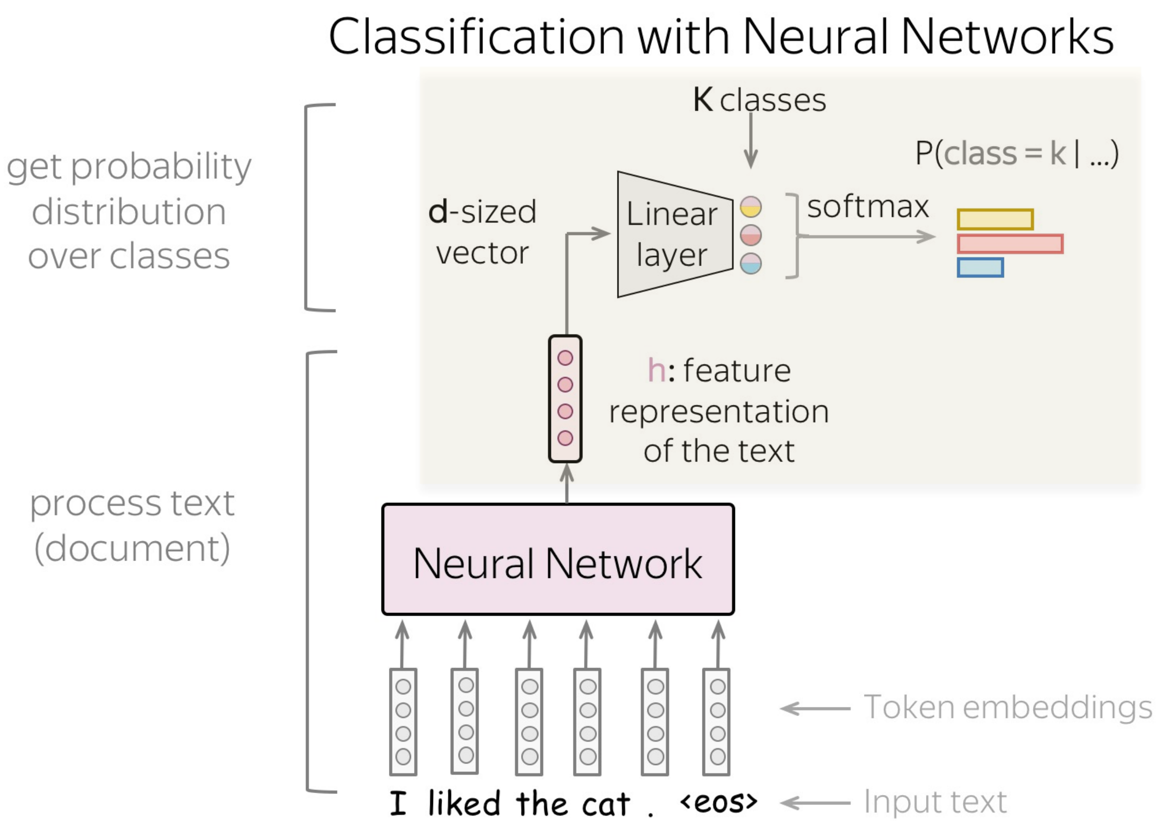
Note a slight change to notation from the previous lecture

NN Classifier



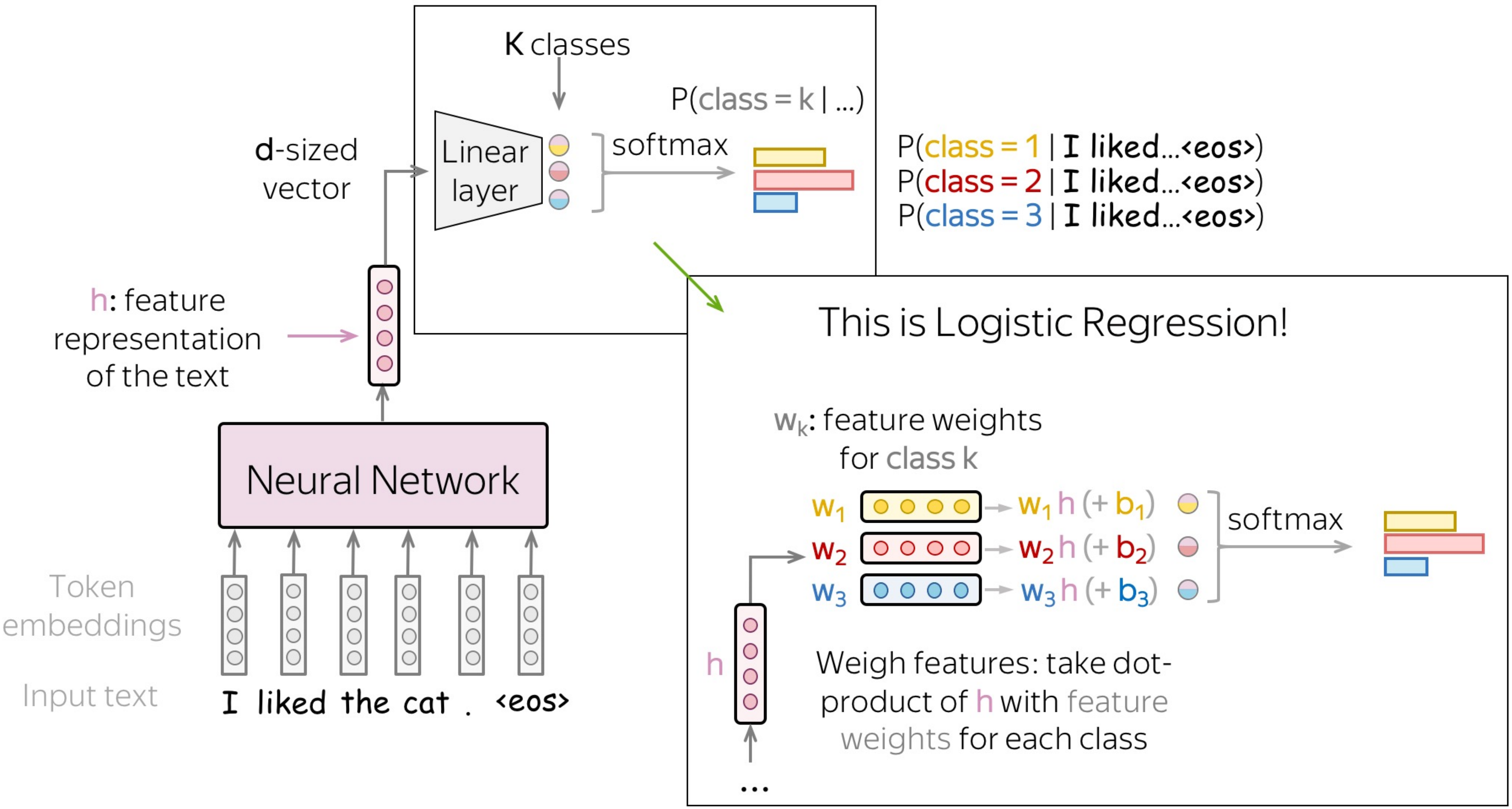
We will spend a lot of time discussing embeddings in this lecture and also next week

NN Classifier

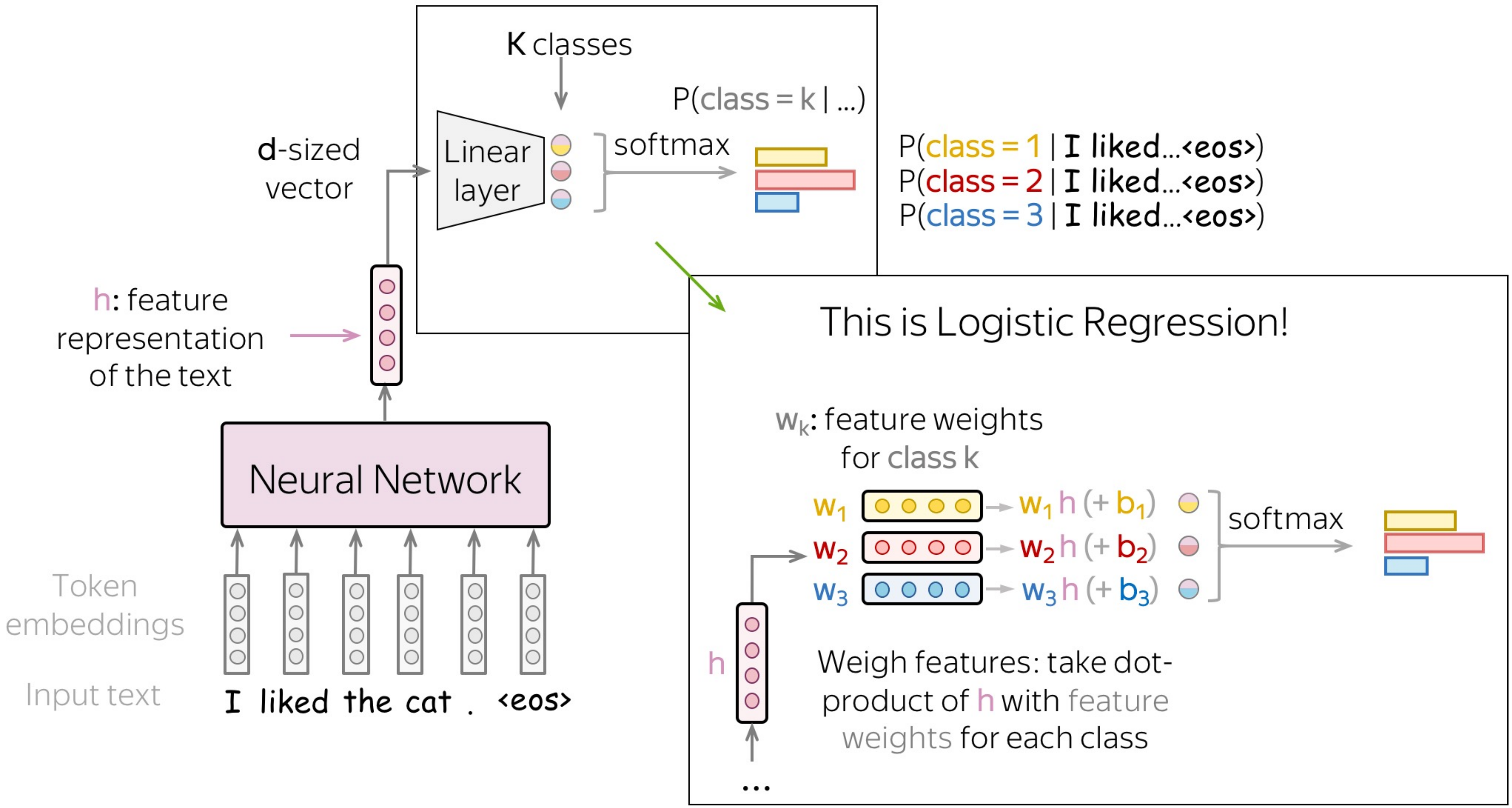


The highlighted part is the logistic regression!

NN Classifier

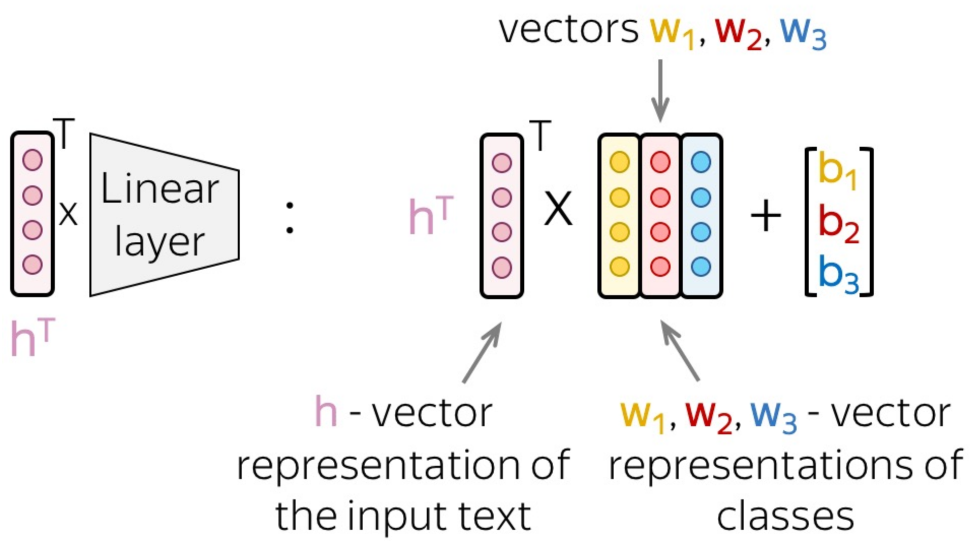


NN Classifier

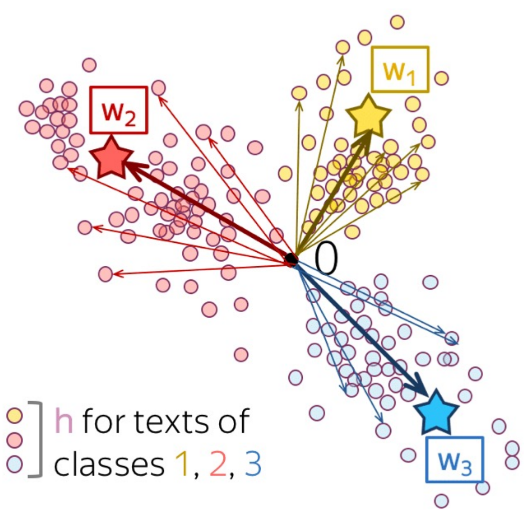


Intuition: the representation of the document points in the direction of the class representation

Representation of the document



Intuition: the representation of the document points in the direction of the class representation



What do we optimize? (recap)

Optimize conditional log-likelihood, as with logistic regression, which is equivalent to using cross-entropy loss

Training example: **I liked the cat on the mat <eos>**

Label: k

↑ target

Model prediction:

P(class = i | I liked...<eos>)

Target:

p^*

Cross-entropy loss:

$$-\sum_{i=1}^{K} p_i^* \cdot \log P(y = i|x) \rightarrow min \quad (p_k^* = 1, p_i^* = 0, i \neq k)$$

For one-hot targets, this is equivalent to

$$-\log P(y = k|x) \rightarrow min$$

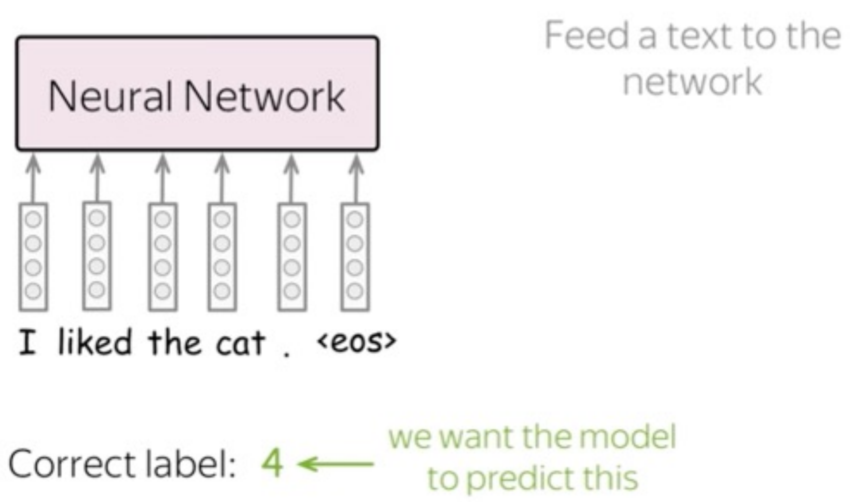
The target distribution is one-hot:

$$p^* = (0, \ldots, 0, 1, 0, \ldots)$$

Recall: we derived the gradient in the previous lecture

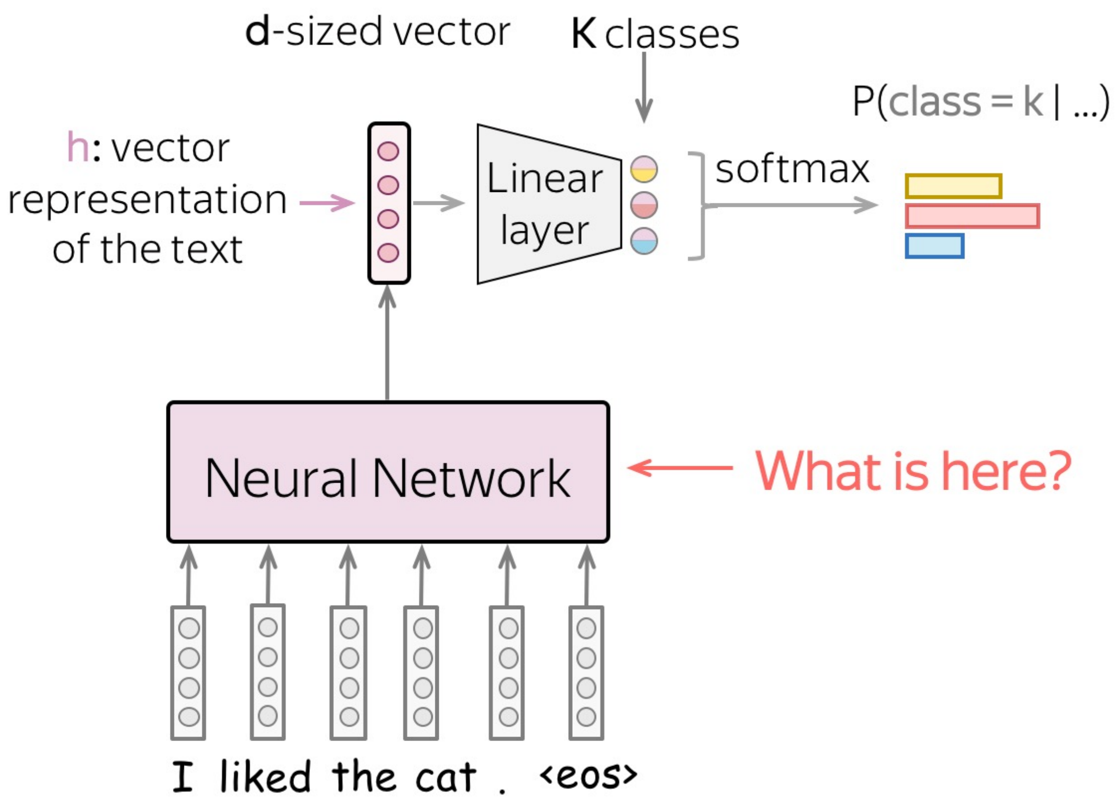
What do we optimize?

Optimize conditional log-likelihood, as with logistic regression



(video, not visible in pdf)

Neural models for text classification



The neuron

Most basic computational unit.

The input $\mathbf{x} \in \mathbb{R}^d$ is a vector with d dimensions.

The output $z \in \mathbb{R}$. This means that we have d inputs and 1 output.

The output is obtained as:

$$z = \sum_{i=1}^{d} w_i x_i + b = \mathbf{w}^\top \mathbf{x} + b$$

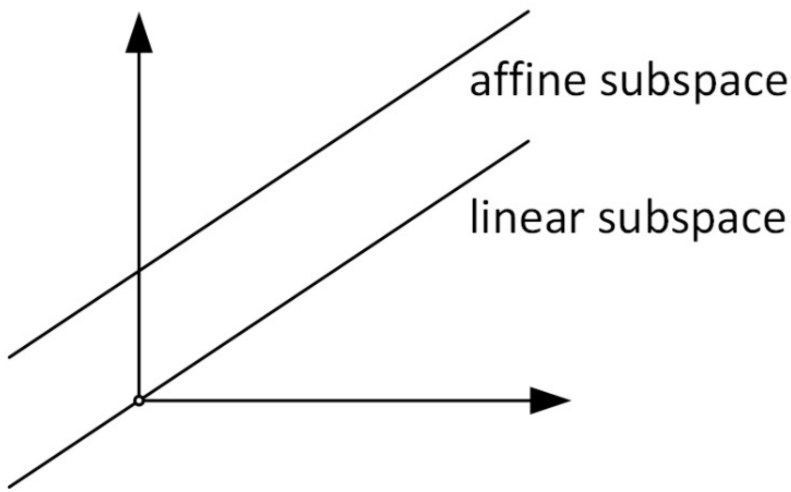
$\mathbf{w} \in \mathbb{R}^d$ are called the **weights**: they multiply each dimension of the input by its 'importance'.

$b \in \mathbb{R}$ is called the **bias** and provides an additive shift.

Why the bias?

Neurons with weights only index functions passing via the origin.

The bias allows for modelling the set of **affine** functions, which is a superset of **linear** functions.



Activation functions *a(z)*

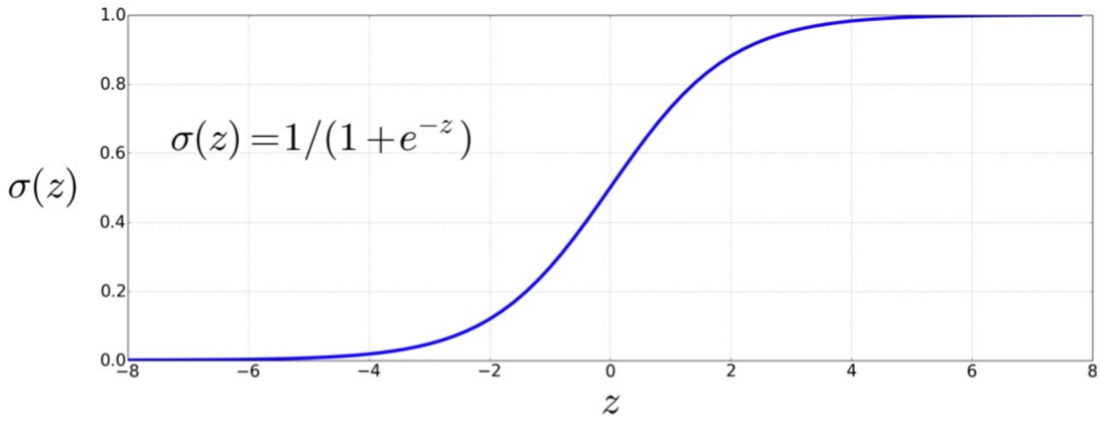
Identity (results in a **linear** model, can perform **regression**):

$$y = f(\mathbf{x}) = a(z) = z$$

Sigmoid (results in a **log-linear** model, can perform **logistic regression** / **binary classification**):

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

Sigmoid function

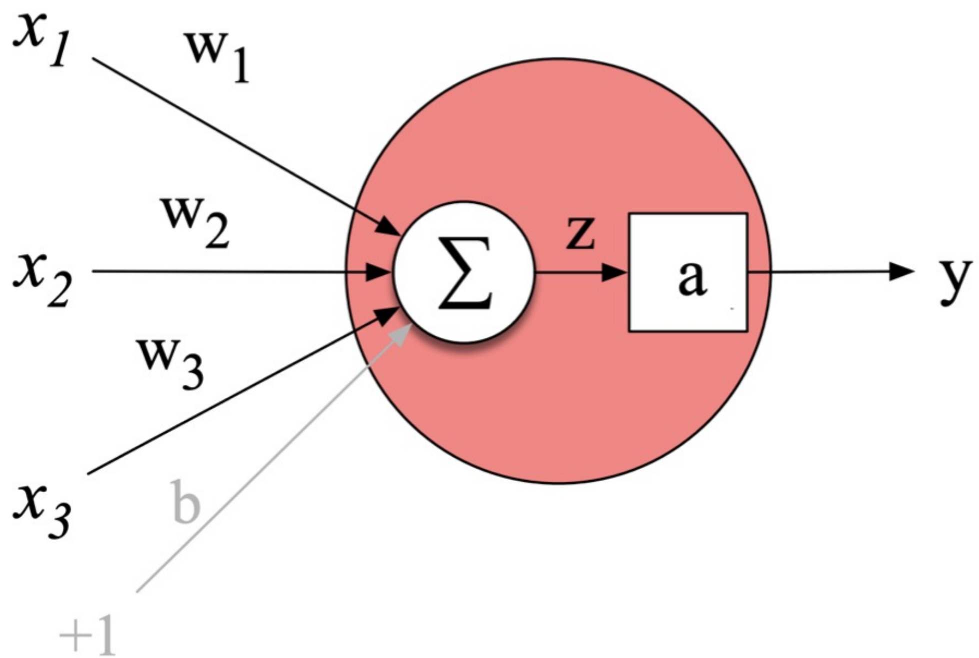


$y \in [0, 1]$. To see why:

$\lim_{z\to\infty} \sigma(z) = 1$

$\lim_{z\to-\infty} \sigma(z) = 0$

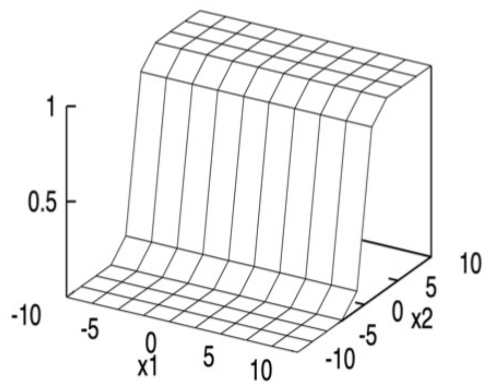
Visualization of a neuron



From J&M3, §7.1

Example: 2-dimensional input

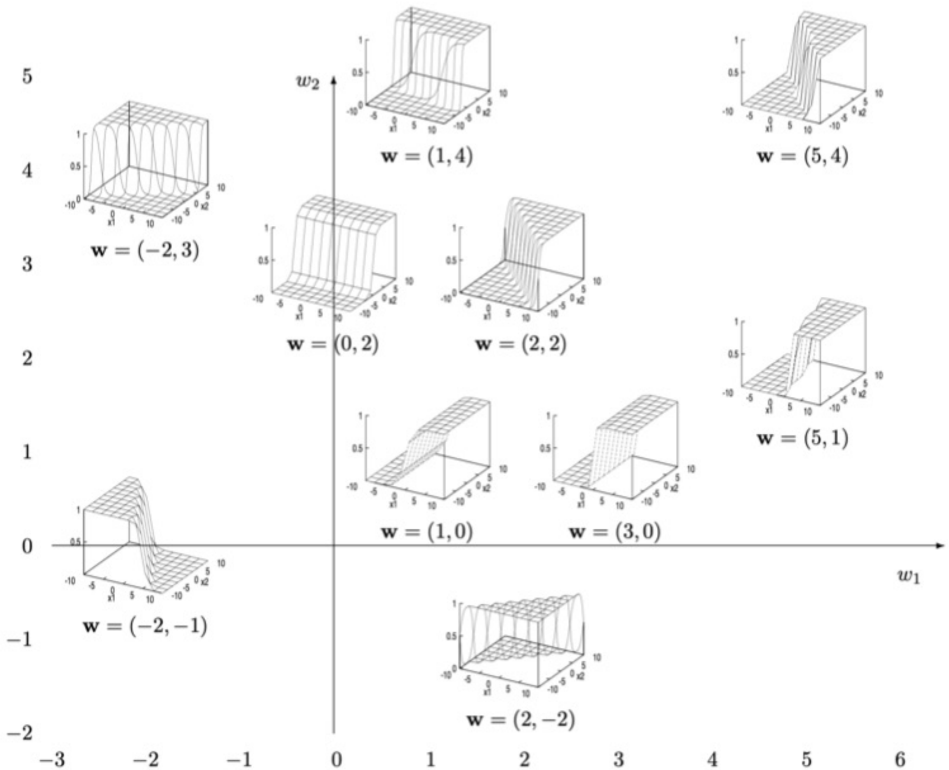
The full neural network is $y = \frac{1}{1+e^{-(w_1 x_1 + w_2 x_2 + b)}}$ If we set $\mathbf{w} = (0, 2)$:



From MacKay, §39.2

Each choice of $\mathbf{w}$ (a point in $\mathbf{w} \in \Theta$) indexes a function from the space $f(\cdot) : \mathcal{X} \to \mathcal{Y}$.

Parameter space



From MacKay, §39.2

Input: word embeddings

Word embeddings are a parameter matrix $E \in \mathbb{R}^{d \times |\mathcal{V}|}$ (with as many columns as words in the vocabulary)

For any word, we can fetch the corresponding column in E to obtain its representation.

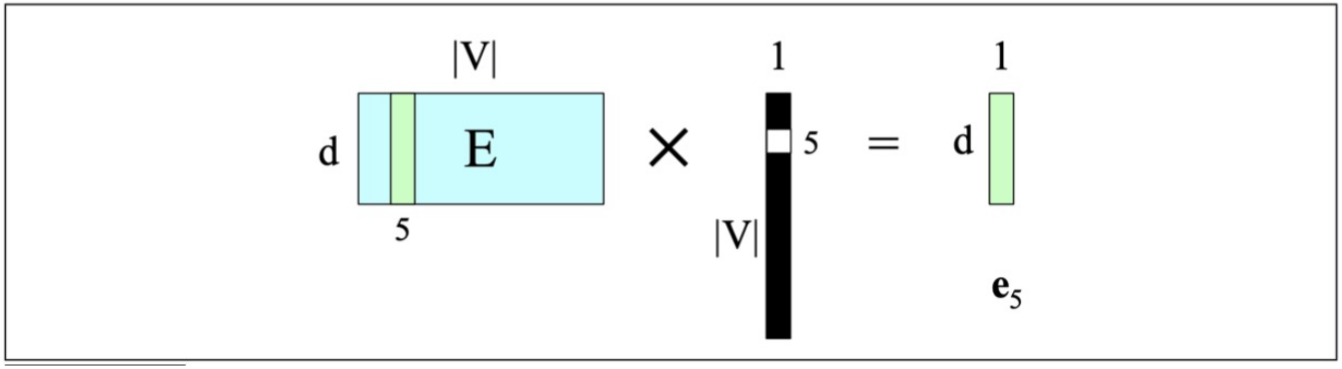
As a pre-processing step, we encode each $x \in \mathcal{V}$ as a distinct **one-hot vector** $\text{one-hot}(x)$. E.g., for $\mathcal{V} = \{all, happy, families\}$:

- $\text{one-hot}(all) = [1, 0, 0]$
- $\text{one-hot}(happy) = [0, 1, 0]$
- $\text{one-hot}(families) = [0, 0, 1]$

Input: word embeddings

During training and inference, any word in a context can be embedded via matrix-vector multiplication.

E.g., for the word with 1 in its 5th dimension of the one-hot vector:



So $\text{enc}(x) = E\,\text{one-hot}(x)$

Composing word embedding in a doc representation

To construct the representation of a document length $n-1$, we could just concatenate the encodings of the corresponding words:

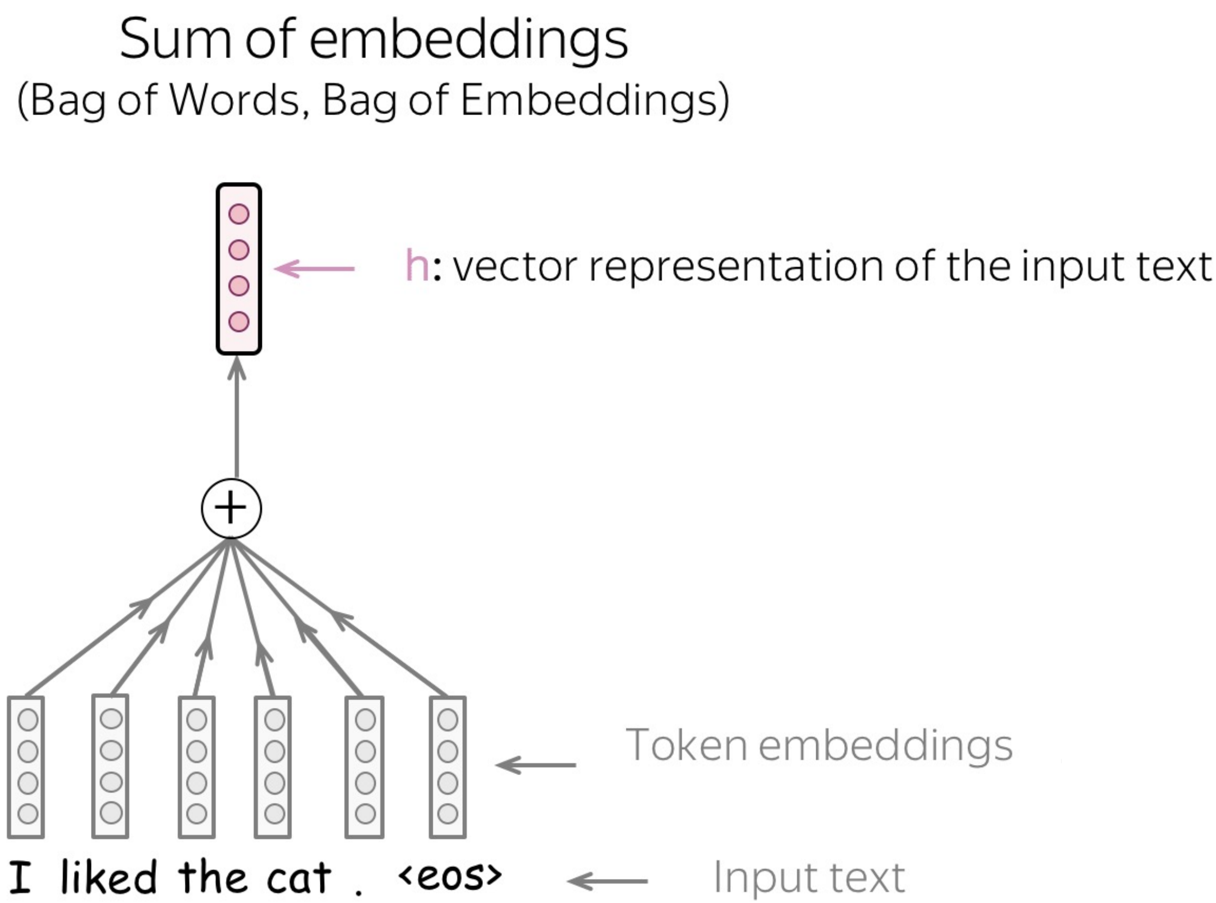
$$\text{enc}(x_1, \dots, x_n) = \text{enc}(x_1) \circ \cdots \circ \text{enc}(x_n).$$

So $\text{enc}(x_1, \dots, x_n) \in \mathbb{R}^{d()}$

This may not be a great idea as the document length can be very long and the number of words varies across documents

So not really what we do for classification (but this architecture will make much more sense what we will get to 'language modeling', i.e. predicting next word).

Basic models: bags of words (= Embeddings)



We will see much more power methods soon

Output: categorical distribution

The neuron returns a scalar output,

- real (if linear);
- or in $[0, 1]$ (if log-linear).

Instead, to yield a **Categorical** distribution over classes, we will need:

- to output multiple units (the number of classes K)
- to choose an activation which ensures that the output is a valid probability (sums to 1).

The perceptron

To create multivariate outputs, we can join multiple neurons, by concatenating their weight vector and bias scalar row-wise. Hence, $W = [\mathbf{w}_1, \ldots, \mathbf{w}_{|\mathcal{K}|}]$.

Each entry W_{ij} (i-th row and j-th column) is the importance of the connection between the i-th output unit to the j-th input unit.

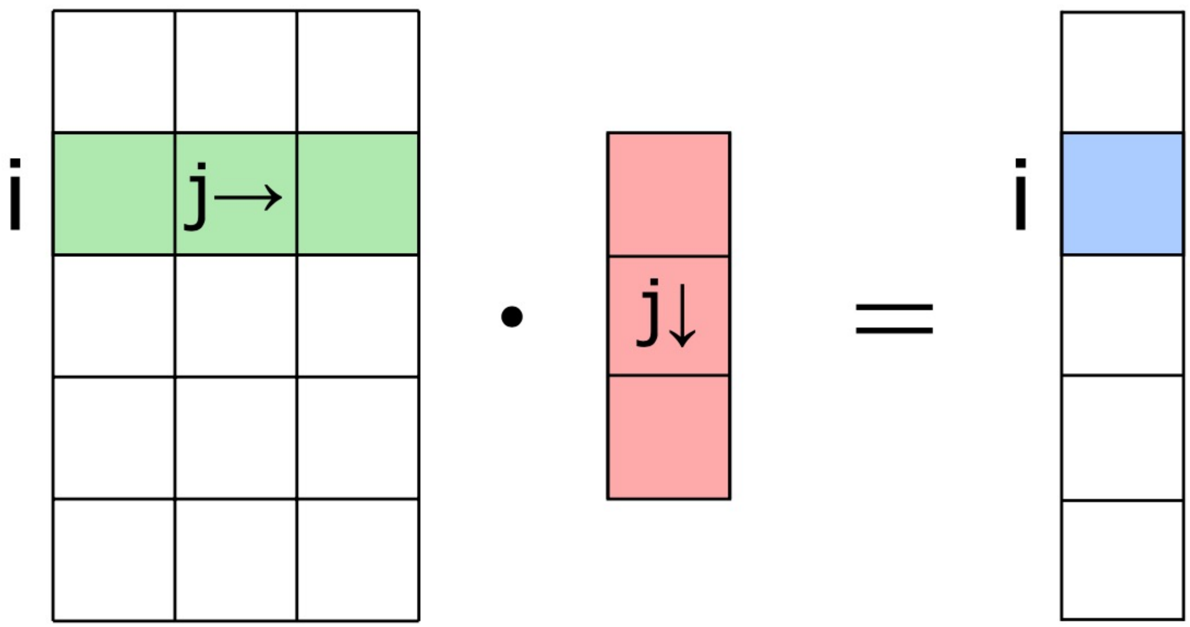
The number of output units is the **width** of the perceptron.

$$f(\mathbf{x}) = a(\mathbf{z}) = a(W\mathbf{x} + \mathbf{b}) = a(\sum_{j=1}^{d} W_{ij}\mathbf{x}_j + \mathbf{b}_i)$$

Thus, $\mathbf{z} \in \mathbb{R}^K$, $W \in \mathbb{R}^{K d(n-1)}$, and $\mathbf{b} \in \mathbb{R}^K$

This architecture is known as the **perceptron**.

Reminder: Matrix-vector multiplication



Softmax activation

A choice of $a(\mathbf{z})$ that will 'squash' the scores of $\mathbf{z}$ into the range $[0, 1]$ and normalise them to sum to 1 (thus, yielding a Categorical distribution) is **softmax**.

$$f(\mathbf{x})_i = \text{softmax}(\mathbf{z}) = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)}$$

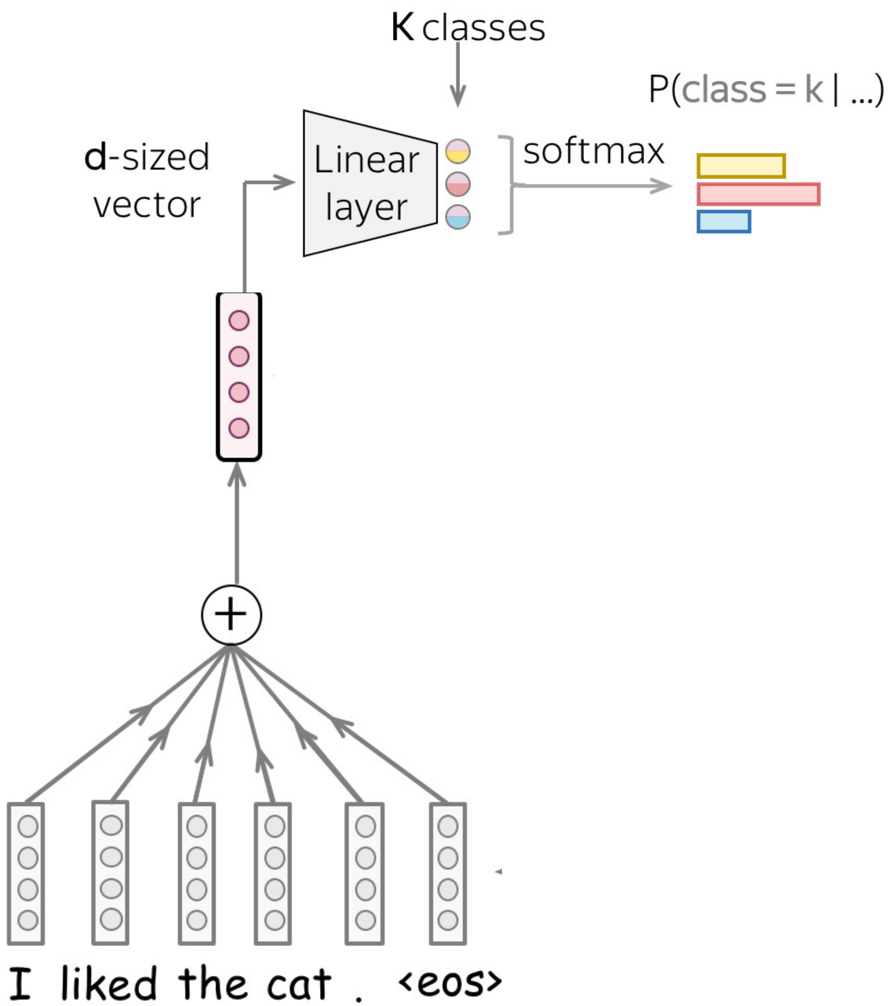
for the i-th output dimension.

(Can be thought of as a multivariate generalisation of sigmoid)

Putting everything together, we have constructed a basic bag-of-word classifier:

$p(class \mid x_1 \ldots x_n) = \text{softmax}\left(W \sum_{i=1}^{n} \text{enc}(x_i) + \mathbf{b}\right)$

Schematic representation



Non-linear problems

Yet, there exists a class of problems that perceptrons cannot solve.[1]

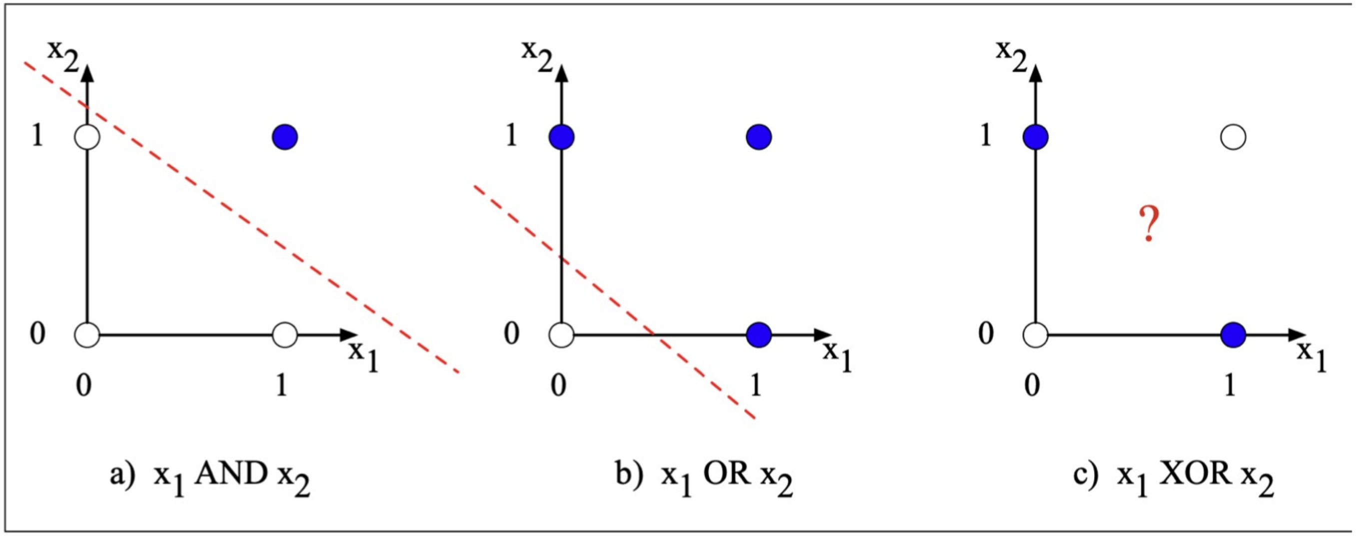
Historically, the first such problem connected to limitations of perceptrons is the XOR operator (Minsky and Papert 1969).

Consider the truth tables for various logical operators :

AND			OR			XOR		
x_1	x_2	z	x_1	x_2	z	x_1	x_2	z
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

[1] In the input space, but they can if extra features like $x_1 \cdot x_2$ are added.

Linear separability



For a threshold τ, we can draw a **decision boundary**, i.e., $y = 1$ if $w_1 x_1 + w_2 x_2 + b > \tau$, else $y = 0$.

This boundary is a line: $x_2 = (-w_1/w_2)x_1 + (-b/w_2) + \tau/w_2$

Multilayer perceptron

How to make neural networks more expressive, i.e., capable of indexing non-linear functions? Recipe:

1. stack perceptrons (layers)

2. use non-linear activations at the end of each layer

The resulting non-linear model is a **multi-layer perceptron** (MLP)!

Stacking layers

Perceptrons can be stacked. We refer to their ordered sequence as **layers**.

This creates **feedforward networks**, so that the output of layer l is passed as input to the next layer $l+1$,[2] but not to the previous ones $1, \ldots, l-1$.

In addition, this family of neural networks is **fully connected**, meaning that for each layer, each output unit is the weighted sum of **all** the input units.

The number of layers is the **depth** of the network (hence, the term deep learning!)

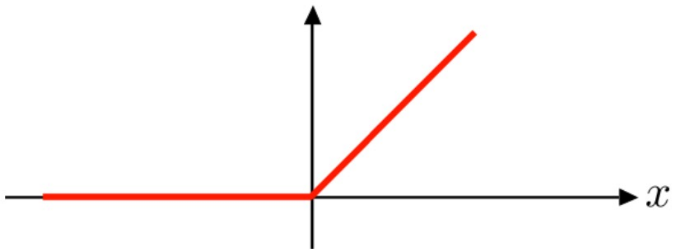
[2] Optionally, it can be passed also to any subsequent layer $l+1, \ldots, L$ (**skip connection**).

Non-linear activation functions

The output $\mathbf{z}$ is passed through a **non-linear function** $a(\cdot)$, which gives us the **hidden representation** $\mathbf{h} \in \mathbb{R}^h$ of intermediate layers.

Any **differentiable**[3] non-linear function can be chosen. A common choice is ReLU (others are sigmoid, tanh, . . .):

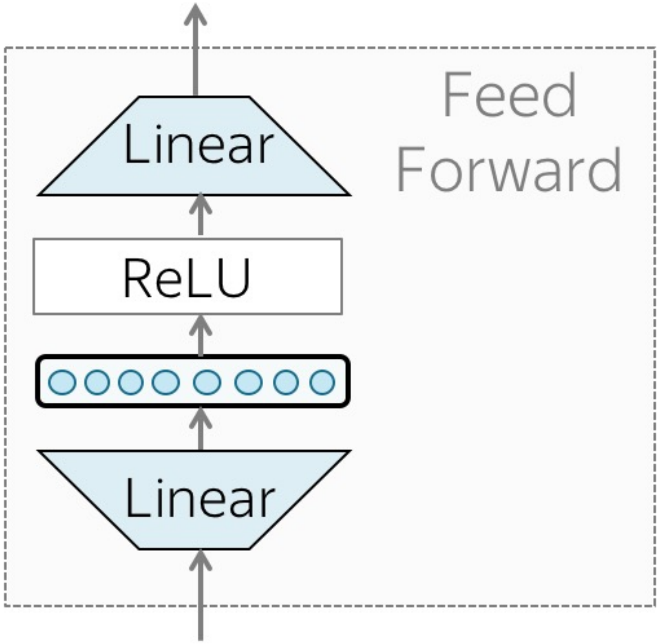
$$\mathrm{ReLU}(x) \triangleq \max(0, x)$$



Note: without non-linearities, a multi-layer network remains linear!

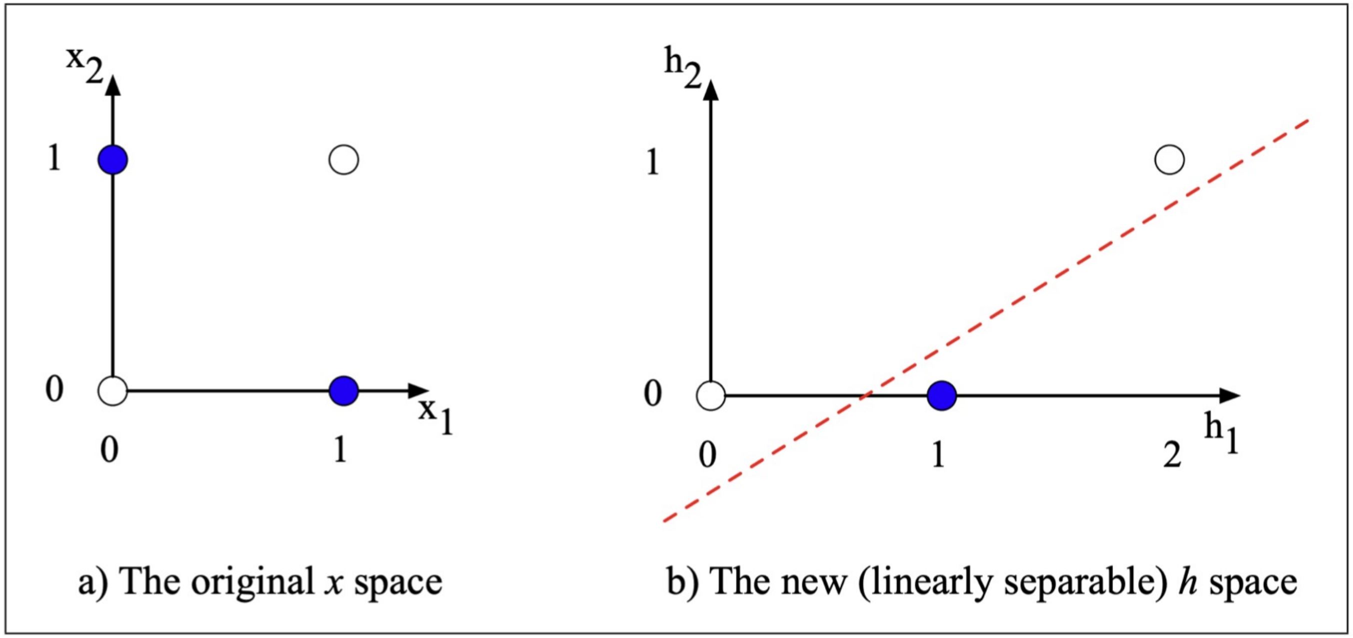
[3] We will see later why this is requirement for training the model

Example: 2-layer MLP (aka 2 layer Feed Forward)



$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

MLP solution to XOR

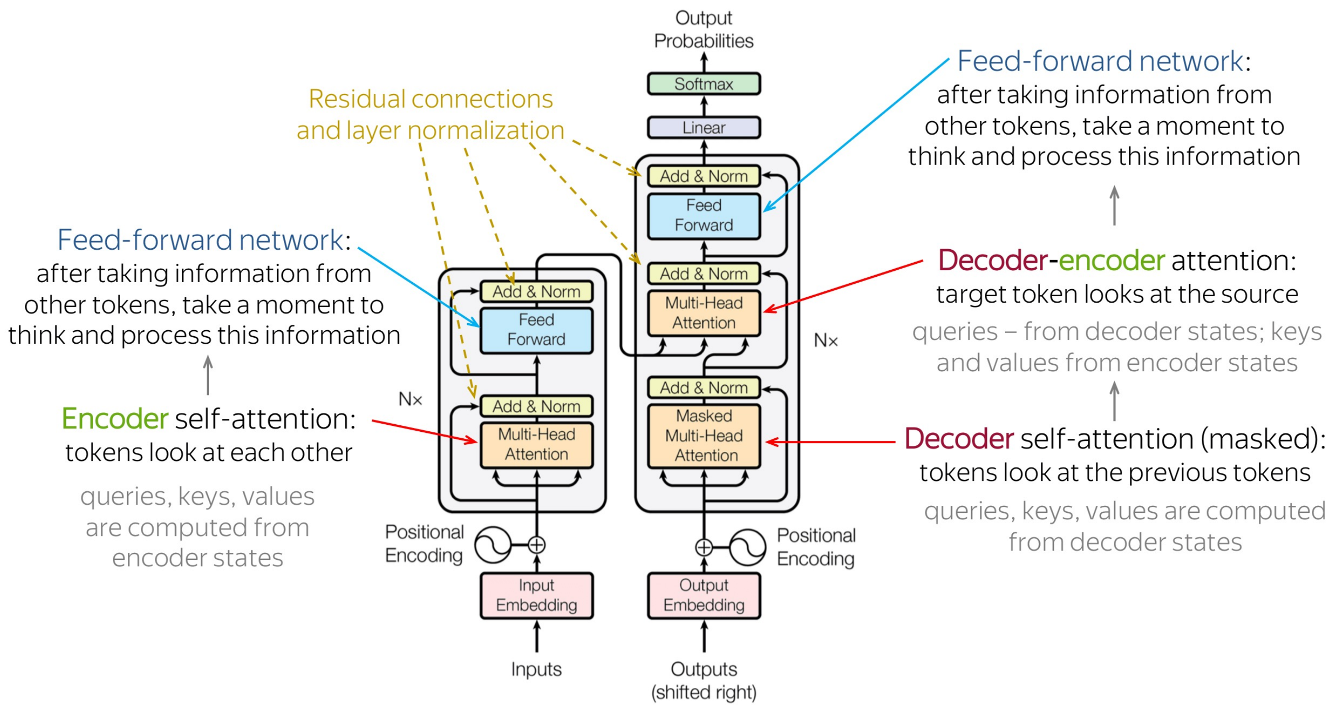


From J&M3, §7.3

Key idea: space folding! $\mathbf{h}$ is linearly separable in the next layer.

Soon more power NLP models!

But we already recognize perhaps half of their components



Conclusions

Text classification with neural networks

- Generalization of logistic regression
- Concept of word embeddings (will see and understand them much more later)
- Bag-of-word models for classification

Neural networks:

- NNs are built out of neurons, a function from many input dimensions to one output dimension.
- A multi-layer perceptron stacks multiple layers, each consisting of multiple units and with a non-linear activation.
- Each layer captures useful features for the next layers.