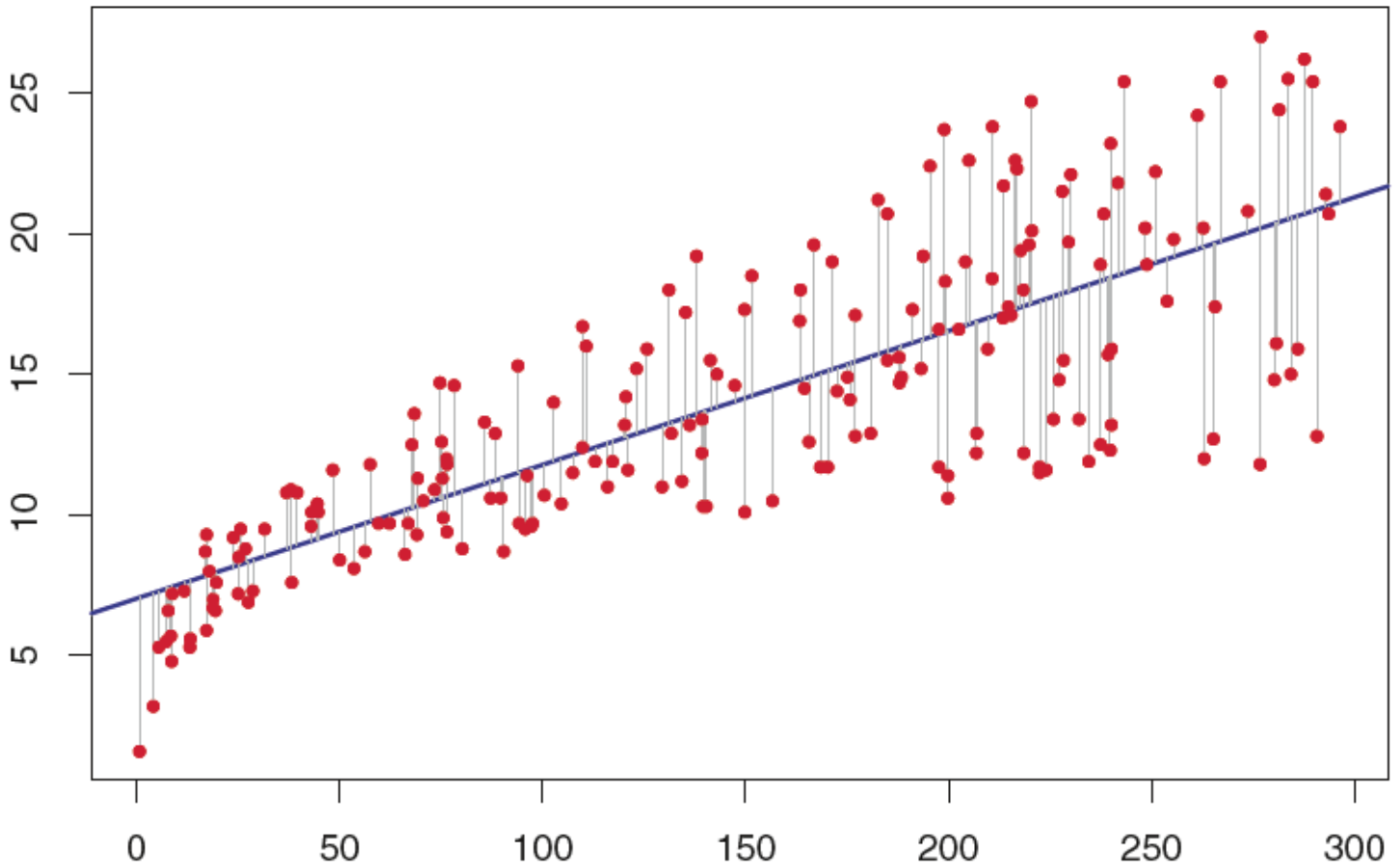
NLP with Neural Networks

Part 1

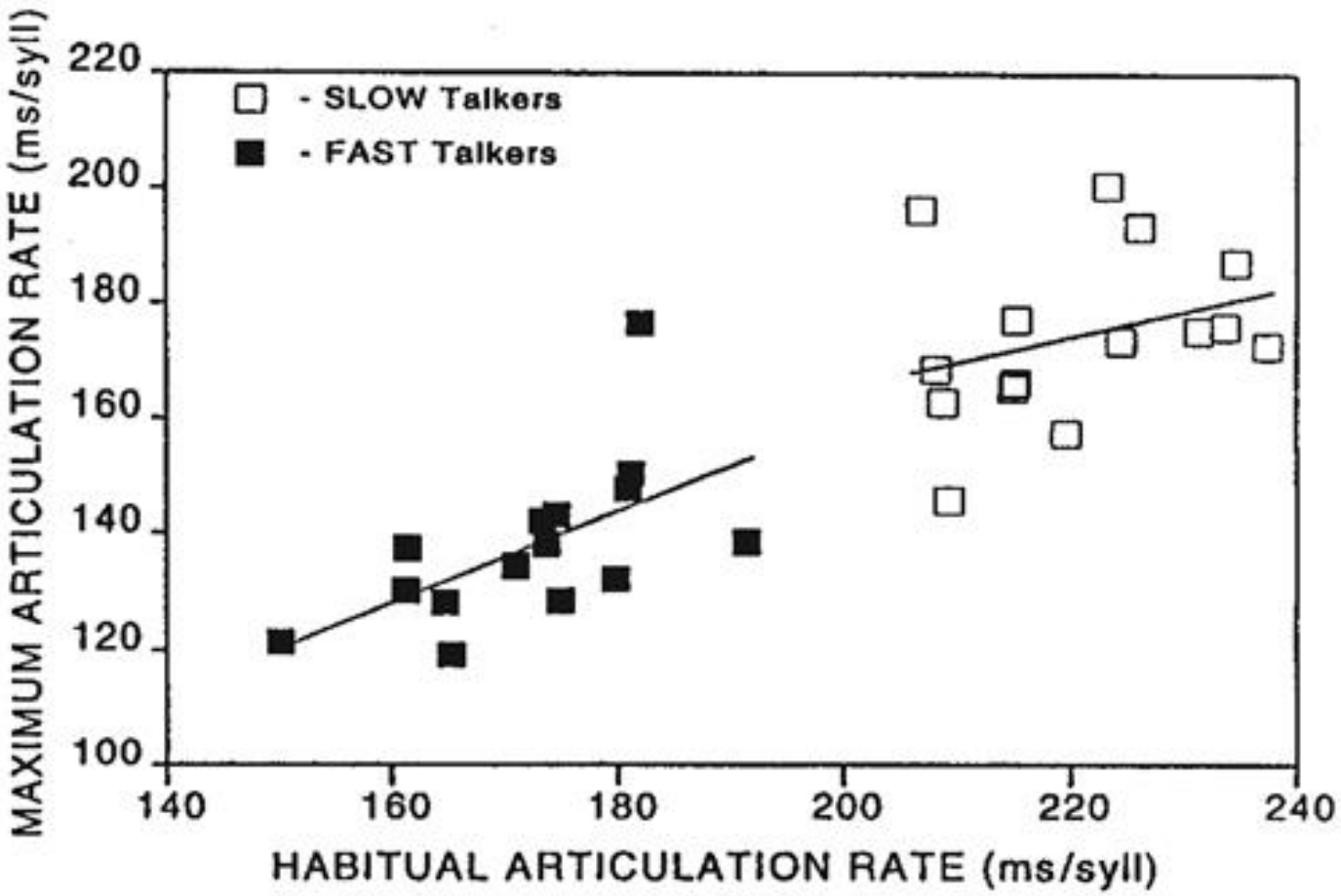
[J&M'17 Ch. 9 (3rd edition), available here]

Simple linear regression (without error)

$$\hat{y} = (x \times \beta_1) + \beta_0$$



NN


- SLOW Talkers
- FAST Talkers
MAXIMUM ARTICULATION RATE (ms/syll)
220
200
180
160
140
120
100
140
160
180
200
220
240
HABITUAL ARTICULATION RATE (ms/syll)

One way to estimate β_1 and β_0, the Residual Sum of Squares (RSS):

$$\beta_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

$$\beta_0 = \bar{y} - \beta_1 \bar{x}$$

where

$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ (= sample mean of x)

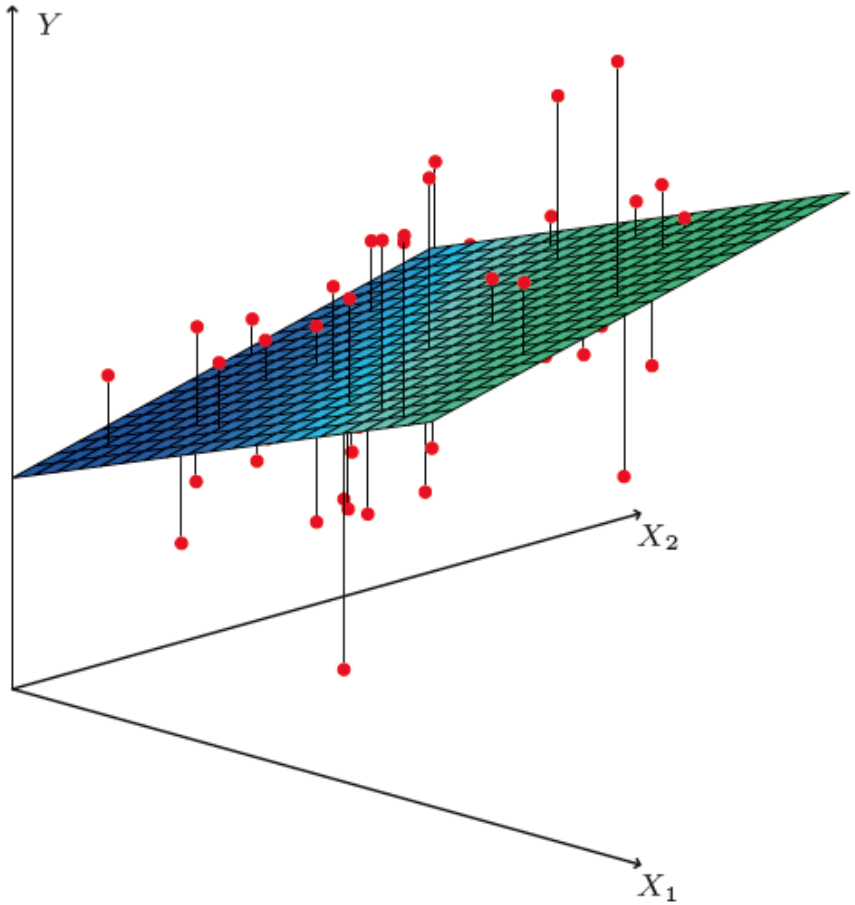
$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ (= sample mean of y)

To evaluate quality of β's, split data in two:

- Use 80% of data to train the model (training set)
- Use 20% of data to test accuracy (held-out set)
- Optionally, split held-out set into two:
  - Validation/Development set
    (to tweak/improve model after training, or to compare sub-models)
  - Test set
    (one final run to check quality)

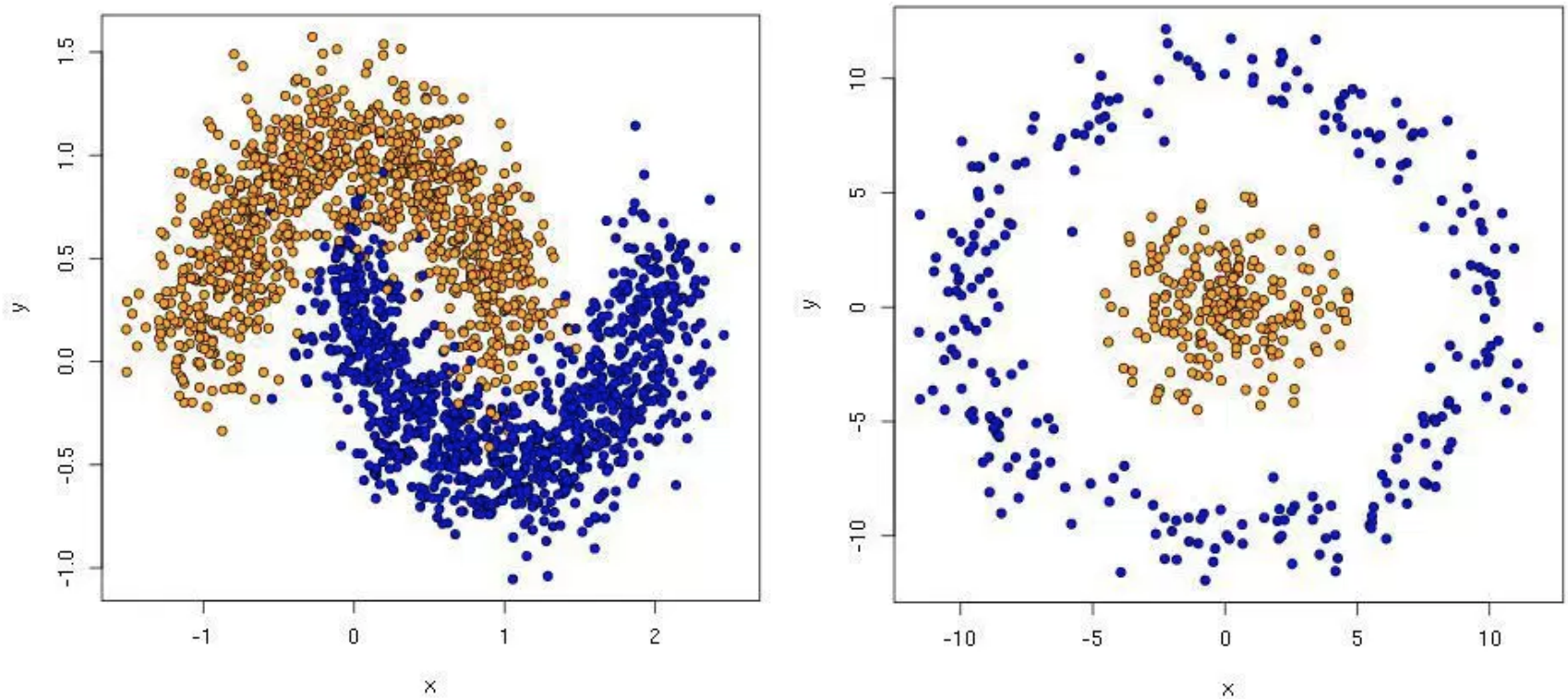
Multilinear regression (without error)

$\hat{y} = (x_1 \times \beta_1) + (x_2 \times \beta_n) + \beta_0$



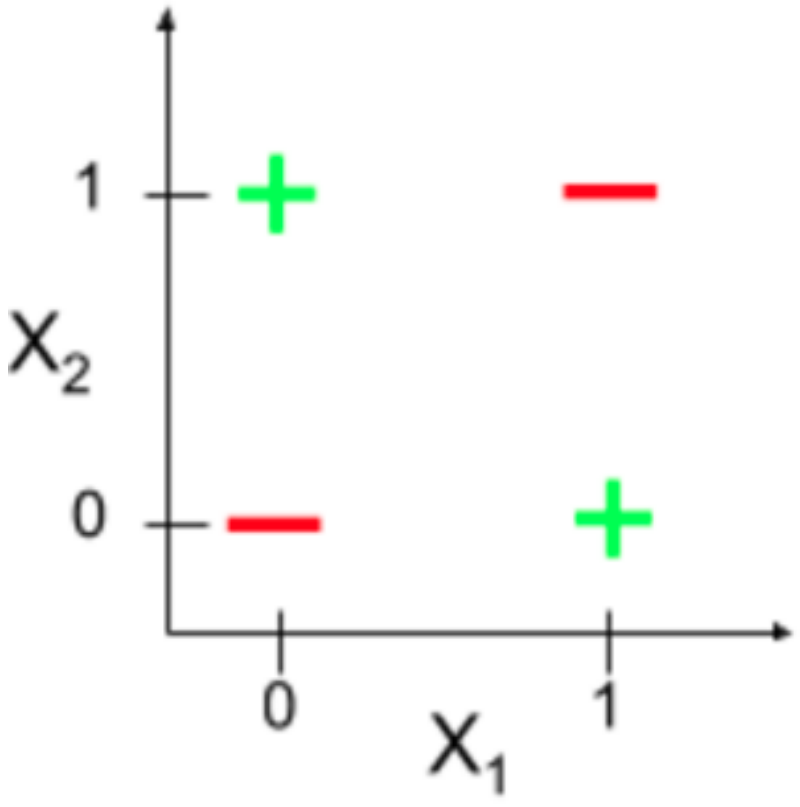
(For more on linear regression in CL see Ch6 of J&M'08)

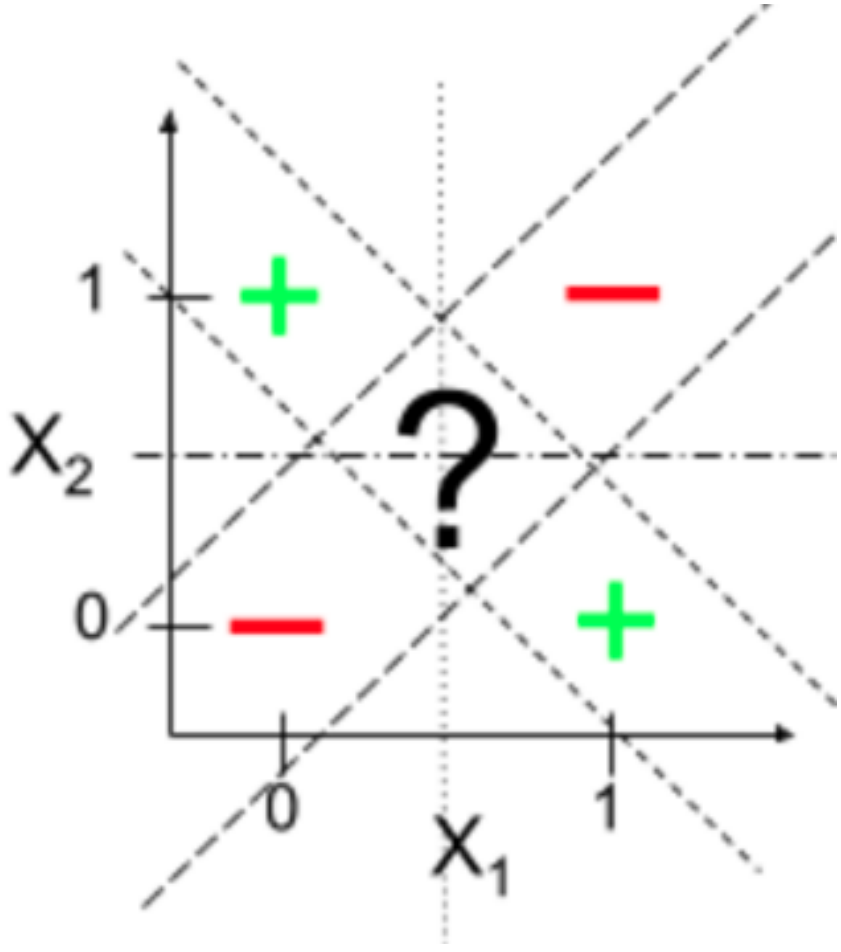
But what if y is far from linear?



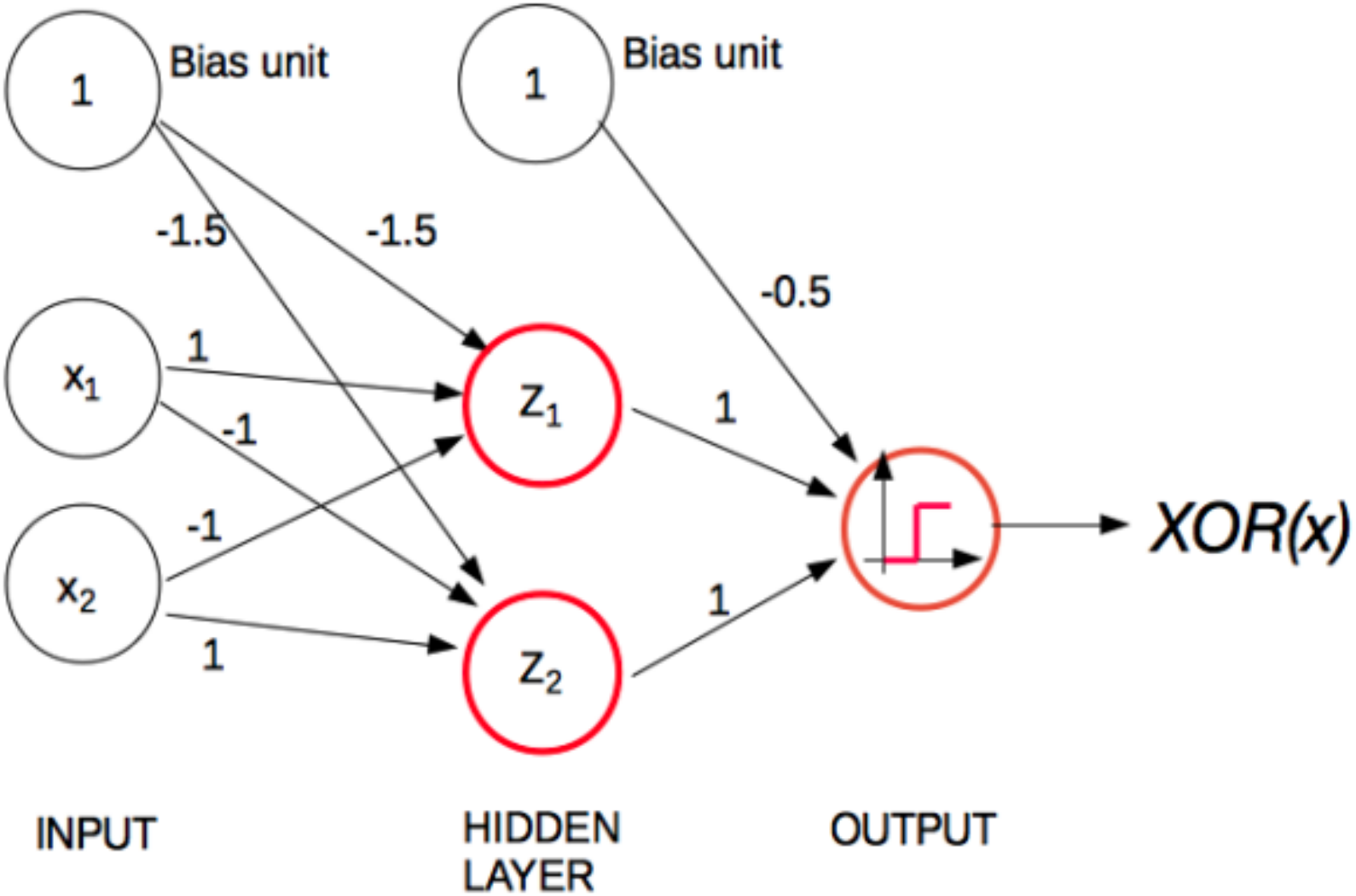
Solution: transformed hierarchical linear models.

Classic example: XOR (exclusive or)

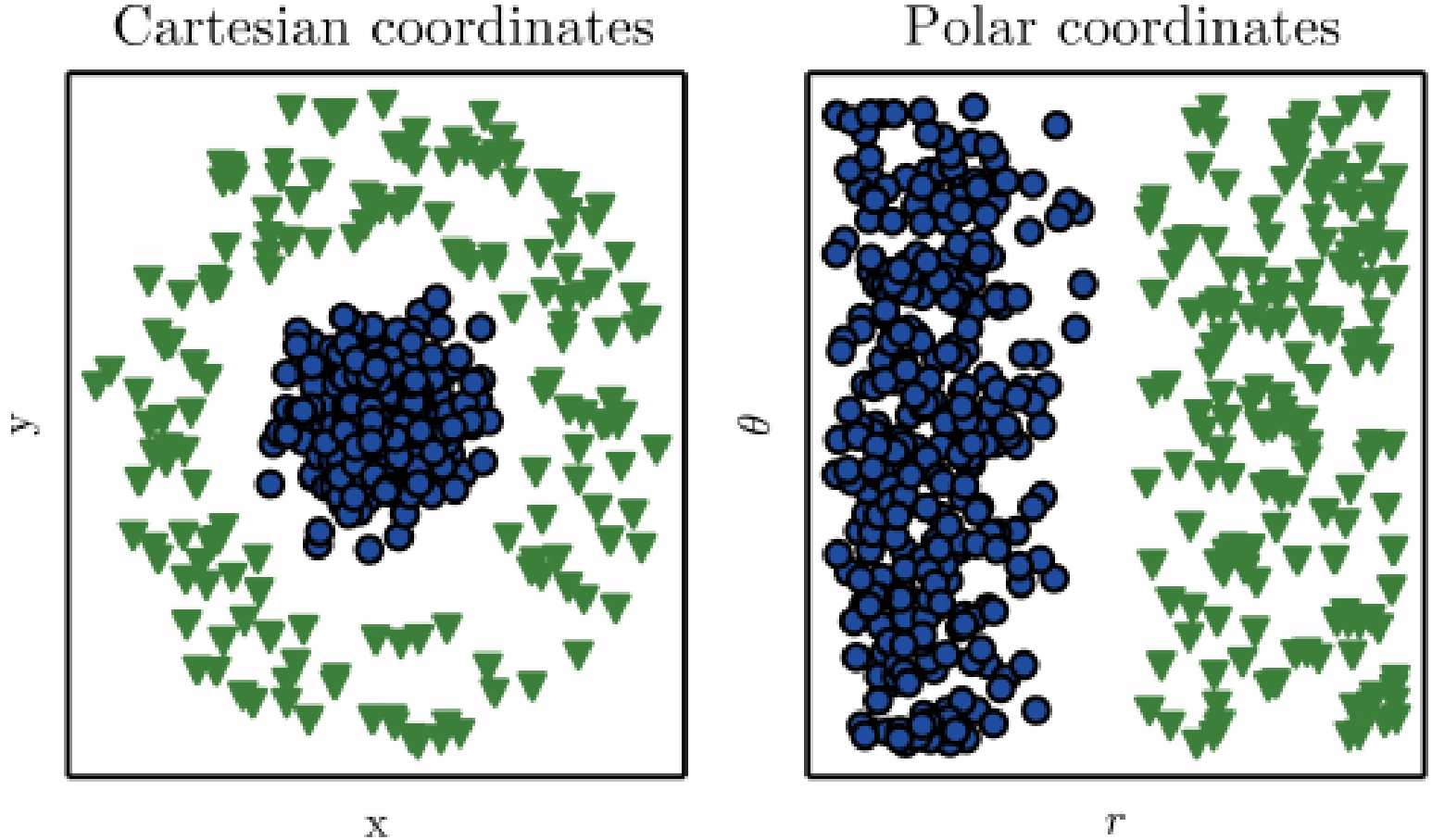


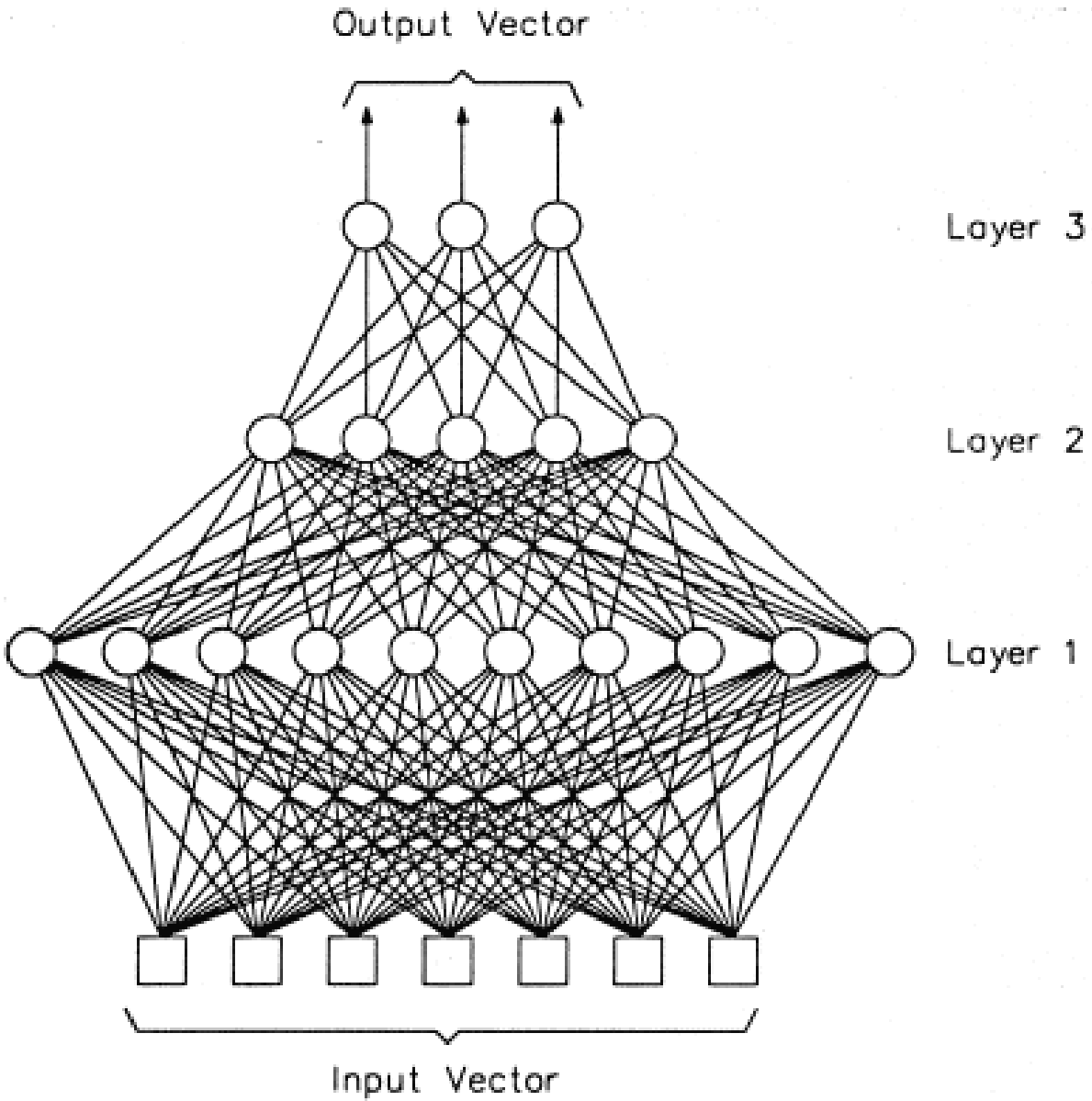


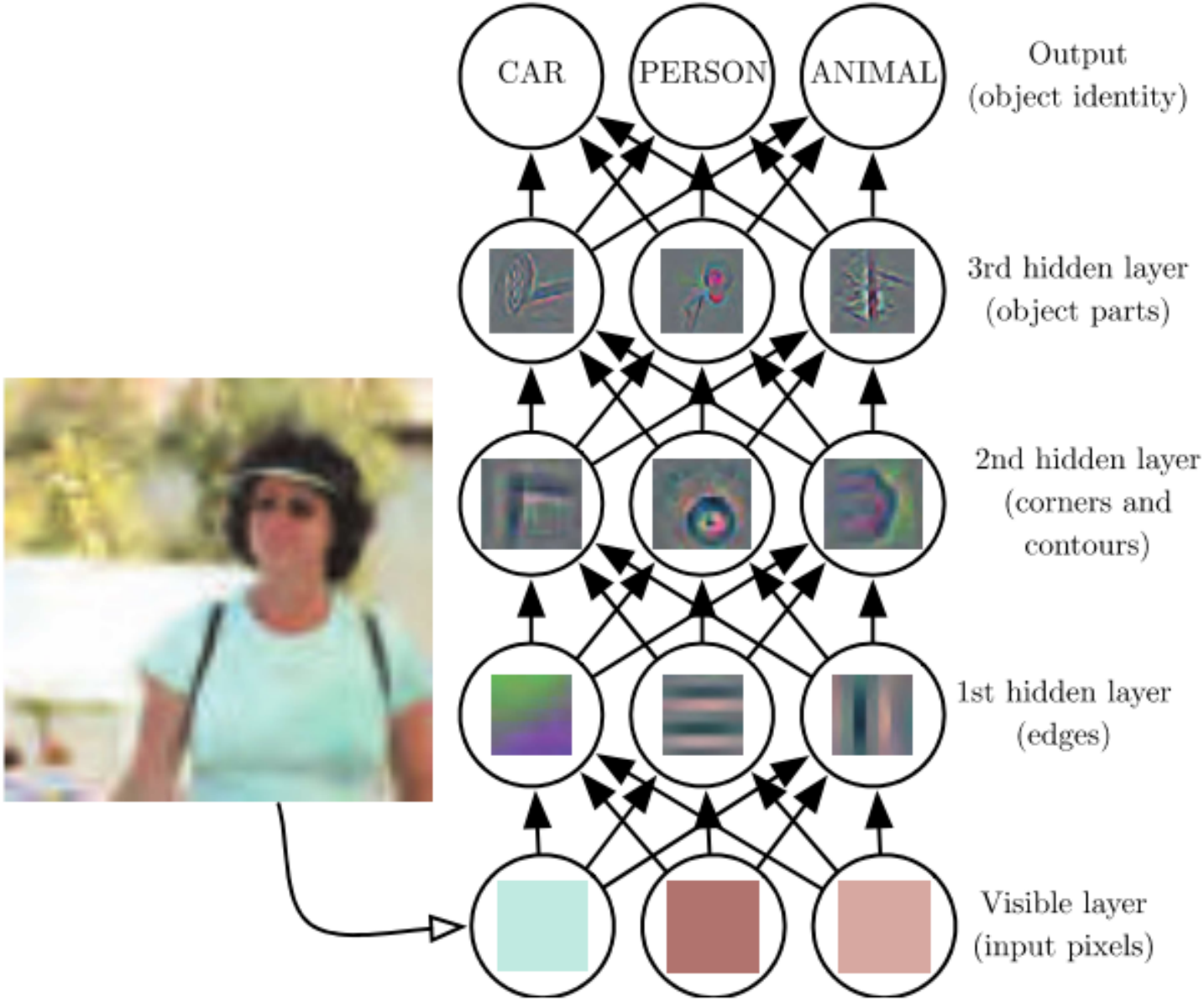
The multi-layer XOR **perceptron**:



The goal is to learn the right transformation (coordinate change).




Output Vector
Layer 3
Layer 2
Layer 1
Input Vector


CAR
PERSON
ANIMAL
Output
(object identity)
3rd hidden layer
(object parts)
2nd hidden layer
(corners and
contours)
1st hidden layer
(edges)
Visible layer
(input pixels)

Some running examples:

- Vision example
- Speech synthesis example
- Lip reading example

(Interview with Geoffrey Hinton, Toronto University)

First, some notation:

- Bold uppercase letters represent matrices

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix}$$

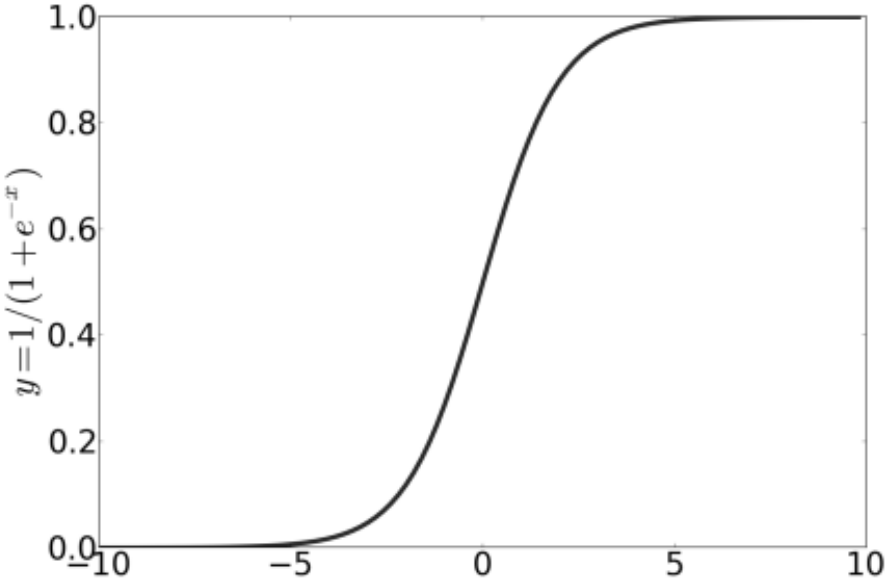
- Bold lowercase letters represent (row) vectors

$$\mathbf{x} = [x_1\, x_2\, \dots\, x_n]$$

- An index corresponds to the value of the vector/matrix at that position, e.g. $\mathbf{X}_{mn} = x_{mn}$ and $\mathbf{x}_2 = x_2$
- A linear model $\hat{y} = \sum_{i=1}^{n}(x_i \times w_i) + b$ can succinctly be written in vector notation as $\hat{y} = \mathbf{x} \cdot \mathbf{W} + \mathbf{b}$
- '$\cdot$' is the dot product (distributed over the columns of $\mathbf{W}$).

The output of $\hat{y} = \mathbf{x} \cdot \mathbf{W} + \mathbf{b}$ is in the range $[-\infty, +\infty]$.
To convert this into a probability, apply the sigmoid function (σ):

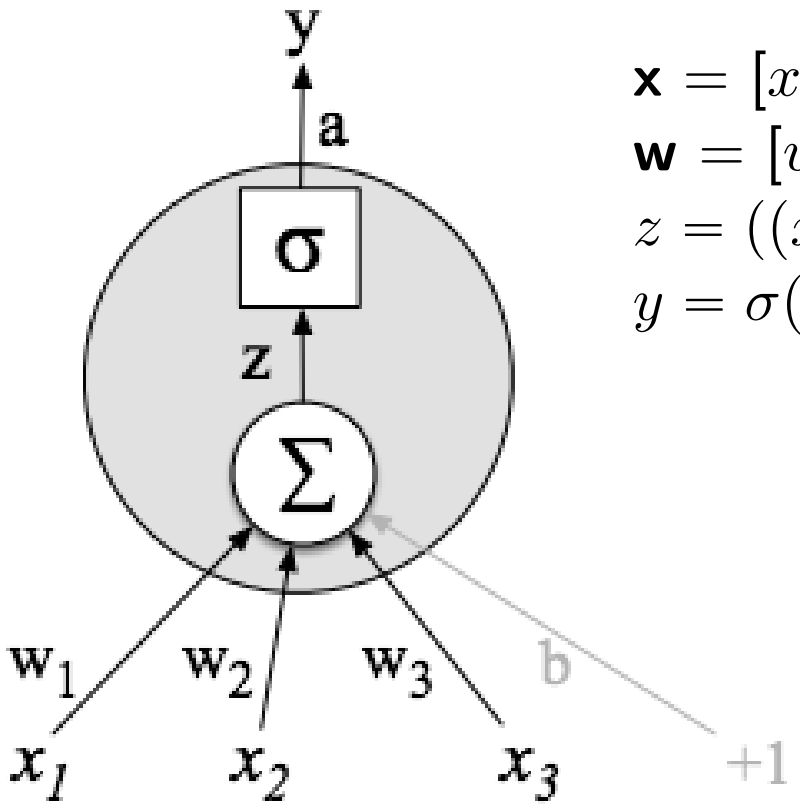
$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad \text{(aka. the logistic)}$$



$$\hat{y} = \sigma(\tilde{y}) = \sigma(\mathbf{x} \cdot \mathbf{W} + \mathbf{b}) = \frac{1}{1 + e^{-(\mathbf{x} \cdot \mathbf{W} + \mathbf{b})}}$$

($\tilde{y}$ is $\hat{y}$ before applying a transformation like σ)

A (simple) sigmoidal perceptron:



$$\mathbf{x} = [x_1, x_2, x_3]$$
$$\mathbf{w} = [w_1, w_2, w_3]$$
$$z = ((x_1 \times w_1) + (x_2 \times w_2) + (x_3 \times w_4)) + 1$$
$$y = \sigma(z)$$

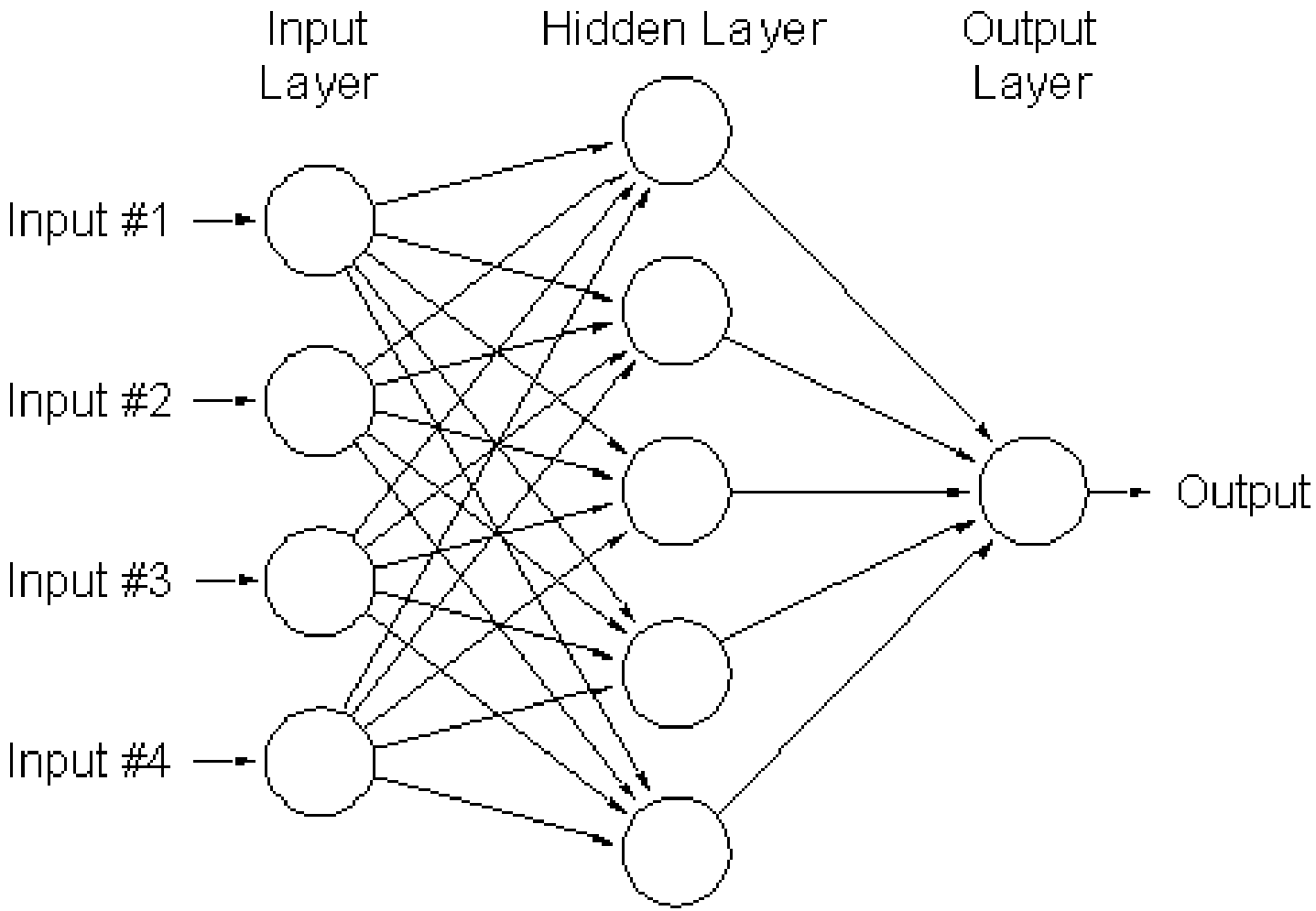
Suppose we have the following inputs and weights:

- $\mathbf{w} = [0.2, 0.3, 0.9]$
- $b = 1$

If we feed it $\mathbf{x} = [0.1, 0.6, 0.1]$ we get:

$$\hat{y} = \sigma(\mathbf{x} \cdot \mathbf{w} + b) = \frac{1}{1 + e^{-0.1 \times 0.2 + 0.6 \times 0.3 + 0.1 \times 0.9 + 1}} = 0.22$$

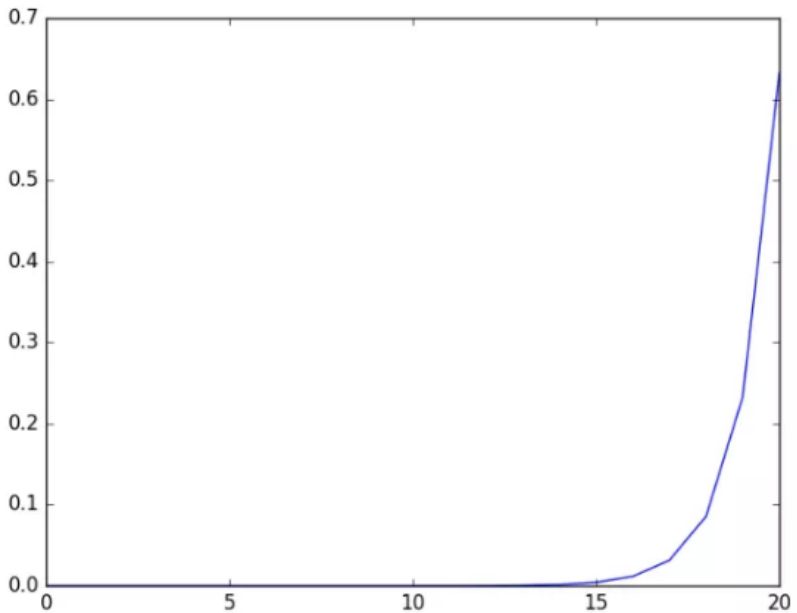
Bigger example: suppose we want to learn how to detect the language in which a text was written, using only the characters.



Softmax (aka. Multinomial Logistic, MaxEnt)

$$\text{softmax}(\mathbf{x}_i) = \frac{e^{\mathbf{x}_i}}{\sum_{j=1}^{n} e^{\mathbf{x}_j}}$$

This forces values to be positive and add up to 1, making them interpretable as a probability distribution.



Training set consists of several documents per language, from which we extract character bigram models (underscore = space):

Assuming we have an alphabet of 28 characters, each document is represented as a vector **x** with 784 dimensions ($= 28 \times 28$) where each entry $\mathbf{x}_i$ represents the normalized bigram count of a particular character combination in a given document:

$$E_1 = [x_{\_a} x_{\_d} x_{\_s} x_{\_t} x_{d\_} x_{de} x_{en} \ldots x_{\_z}]$$
$$E_2 = [x_{\_a} x_{\_d} x_{\_s} x_{\_t} x_{d\_} x_{de} x_{en} \ldots x_{\_z}]$$
$$\vdots$$
$$E_n = [x_{\_a} x_{\_d} x_{\_s} x_{\_t} x_{d\_} x_{de} x_{en} \ldots x_{\_z}]$$
$$\vdots$$
$$G_1 = [x_{\_a} x_{\_d} x_{\_s} x_{\_t} x_{d\_} x_{de} x_{en} \ldots x_{\_z}]$$
$$G_2 = [x_{\_a} x_{\_d} x_{\_s} x_{\_t} x_{d\_} x_{de} x_{en} \ldots x_{\_z}]$$
$$\vdots$$
$$G_n = [x_{\_a} x_{\_d} x_{\_s} x_{\_t} x_{d\_} x_{de} x_{en} \ldots x_{\_z}]$$

$x_{AB} = \dfrac{\text{Count}(AB)}{|D|}$... where $|D|$ is the document's char length.

One hidden neuron per language, and output neuron chooses the one that outputs the biggest y:

$$\hat{y} = \text{arg max}_{L \in \{En,Fr,Gr,It,Sp\}} \mathbf{x} \cdot \mathbf{w}^L + b^L$$

By arranging all the $\mathbf{w}^L$ into a matrix (one column per language), we can view this as a parallel network:

$$\hat{\mathbf{y}} = \mathbf{x} \cdot \mathbf{W} + \mathbf{b}$$

i.e.

$$[x_1 x_2 ... x_{784}] \begin{bmatrix} w_1^{Eng} & w_1^{Fr} & w_1^{Gr} & w_1^{Sp} & \\ w_2^{Eng} & w_2^{Fr} & w_2^{Gr} & w_2^{Sp} & \\ ... & ... & ... & ... & ... \\ w_{784}^{Eng} & w_{784}^{Fr} & w_{784}^{Gr} & w_{784}^{Sp} & \end{bmatrix} = [y^{Eng} y^{Fr} y^{Gr} y^{Sp}] = \hat{\mathbf{y}}$$

$$\hat{y} = \text{arg max}_i \ \hat{\mathbf{y}}_i$$

Applying the *softmax* function, the values in $\hat{\mathbf{y}}$ will be positive and sum to 1, making them interpretable as a probability distribution.

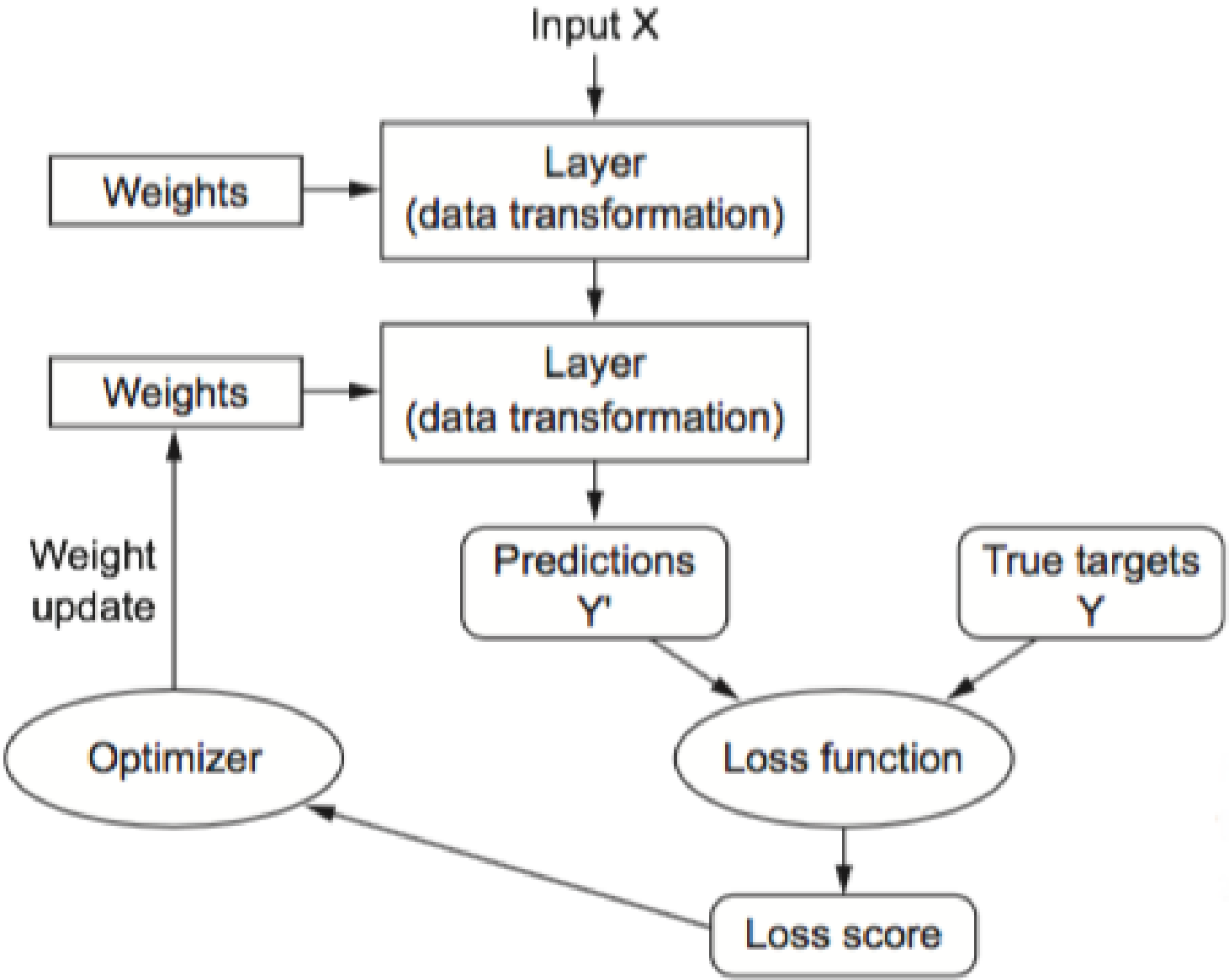
$$\hat{\mathbf{y}} = \mathit{softmax}(\mathbf{x} \cdot \mathbf{W} + \mathbf{b})$$

$$\hat{y}_i = \frac{e^{(\mathbf{x}\cdot\mathbf{W}+\mathbf{b})_i}}{\sum_j e^{(\mathbf{x}\cdot\mathbf{W}+\mathbf{b})_j}}$$

How to set **W** and **b** so that the model correctly classifies the input?

First, the basic intuition:

1. Pick small random values for the parameters **W** and **b**;
2. Compute difference between output and intended output;
3. Adjust parameters **W** and **b**;
4. Repeat until no improvement is possible.


Input X
Weights
Layer
(data transformation)
Weights
Layer
(data transformation)
Weight
update
Predictions
Y'
True targets
Y
Optimizer
Loss function
Loss score

The details are complex, but crucial. So let's dive in.

First, some more notation:

- Let's call refer to the model's parameters **W** and **b** as Θ.
- $f(\mathbf{x}; \Theta)$ is the function that yields $\hat{y}$ given **x** and Θ.
- The loss function $L(\hat{y}, y)$ measures how much $\hat{y}$ differs from correct (i.e. training) y.

Given a labeled training set of **x** and **y**, the overall loss $\mathcal{L}$ with respect to Θ is the average loss over all training examples:

$$\mathcal{L}(\Theta) = \frac{1}{n} \sum_{i=1}^{n} L(f(\mathbf{x}_i; \Theta), \mathbf{y}_i)$$

Our goal is to find the $\hat{\Theta} = \arg\min_{\Theta} \mathcal{L}(\Theta)$

Some standard loss functions:

1. Hinge (binary classification; where $\hat{y} \in \{-1, 1\}$)

$$L_{\text{hinge}}(\tilde{y}, y) = \max(0, 1 - y \times \tilde{y})$$

$$\hat{y} = \text{sign}(\tilde{y}) = \begin{cases} +1 : \text{if } \tilde{y} > 0 \\ 0 \quad : \text{if } \tilde{y} = 0 \\ -1 : \text{if } \tilde{y} < 0 \end{cases}$$

2. Hinge (multi-class; output is a one-hot vector $\hat{\mathbf{y}} = \hat{\mathbf{y}}_1 ... \hat{\mathbf{y}}_n$)

$$L_{\text{hinge(multi-class)}}(\hat{\mathbf{y}}, \mathbf{y}) = \max(0, 1 - (\hat{\mathbf{y}}_t - \hat{\mathbf{y}}_k))$$

$\mathbf{y}$ is a one-hot vector for the correct class, $\hat{\mathbf{y}}_t$ is the correct class, $\hat{\mathbf{y}}_k$ is the highest scoring class such that $\hat{\mathbf{y}}_k \neq \hat{\mathbf{y}}_t$

Loss functions (continued):

1. Binary cross entropy (binary classification with conditional probability outputs; $y \in \{0, 1\}$)

$$L_{\text{logistic}}(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

σ is applied to the classifier output $\tilde{y}$, and the result is interpreted as $\hat{y} = \sigma(\tilde{y}) = P(y = 1|\mathbf{x})$

2. Categorical cross-entropy (multi-class generalization)
Where $\mathbf{y} = \mathbf{y}_1 ... \mathbf{y}_n$ is a vector representing true multinomial distribution over labels $1...n$, and $\hat{\mathbf{y}} = \hat{\mathbf{y}}_1 ... \hat{\mathbf{y}}_n$ is the (softmax transformed) classifier output.

$$L_{\text{cross-entropy}}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_i \mathbf{y}_i \log(\hat{\mathbf{y}}_i)$$

So, $\hat{\mathbf{y}} = P(y = i|\mathbf{x})$

$$\hat{\Theta} = \text{arg min}_{\Theta}\mathcal{L}(\Theta) = \text{arg min}_{\Theta} = \frac{1}{n}\sum_{i=1}^{n} L(f(\mathbf{x}_i;\Theta), \mathbf{y}_i)$$

But the idea that loss must be minimized at all costs may lead to *overfitting*, i.e. learning quirks of the training set very well, to the point of failing to correctly classify new examples.

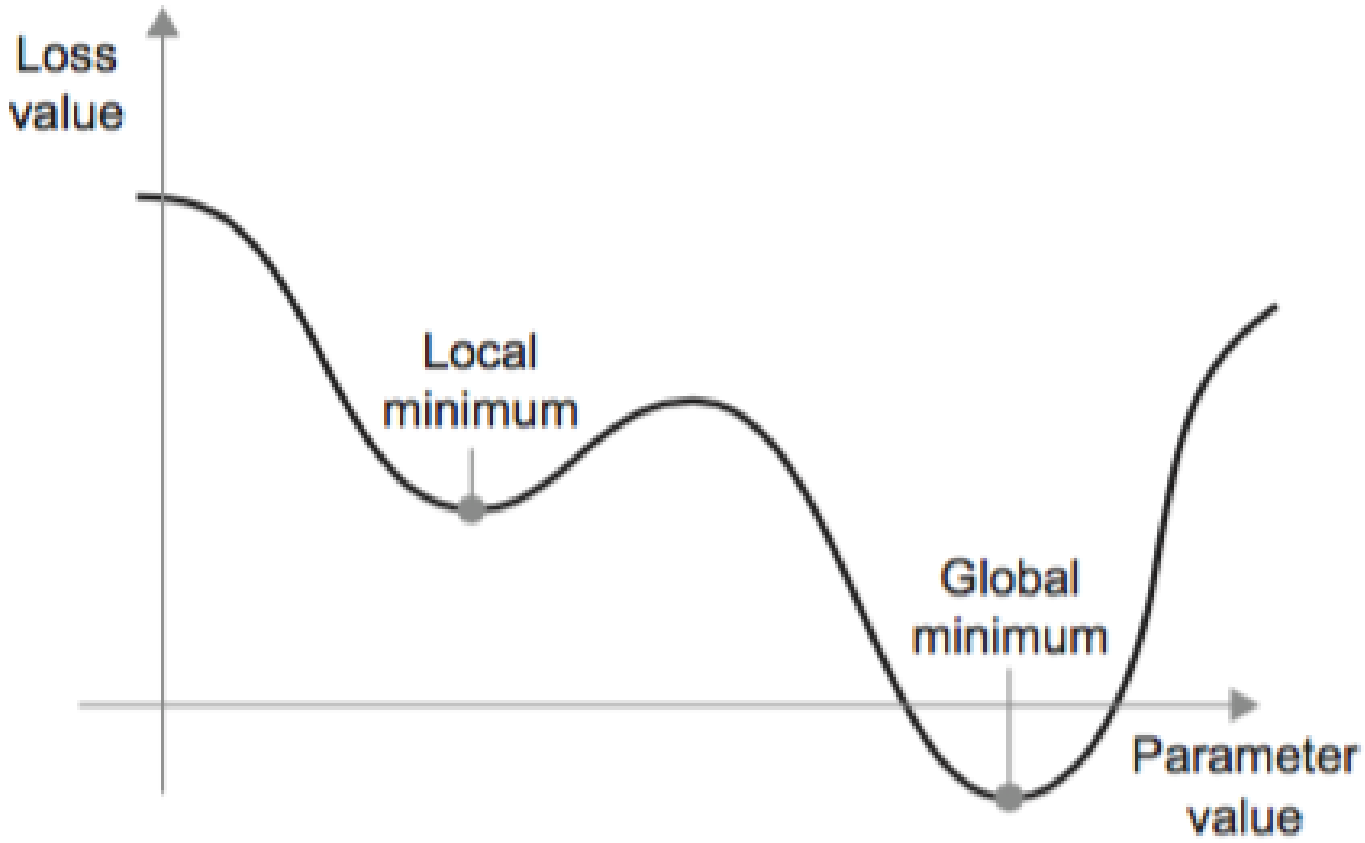
To counteract this problem, a regularization function is added, which returns the complexity of Θ. Ideally, the latter is low.

$$\hat{\Theta} = \text{arg min}_{\Theta}\mathcal{L}(\Theta) = \text{arg min}_{\Theta} = \frac{1}{n}\sum_{i=1}^{n} L(f(\mathbf{x}_i;\Theta), \mathbf{y}_i) + \lambda R(\Theta)$$

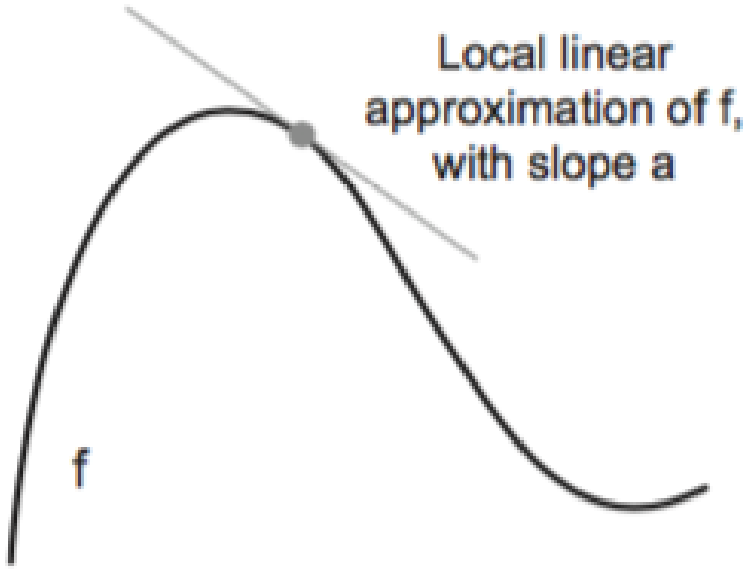
- The regularizer R equates complexity with large weights (the norms of the weight matrix), e.g.:

$$R(\mathbf{W}) = \sum_{i,j} \mathbf{W}_{i,j}^2 \text{ (L2 norm; gaussian prior)}$$

Training the model = finding Θ with the global minimum loss

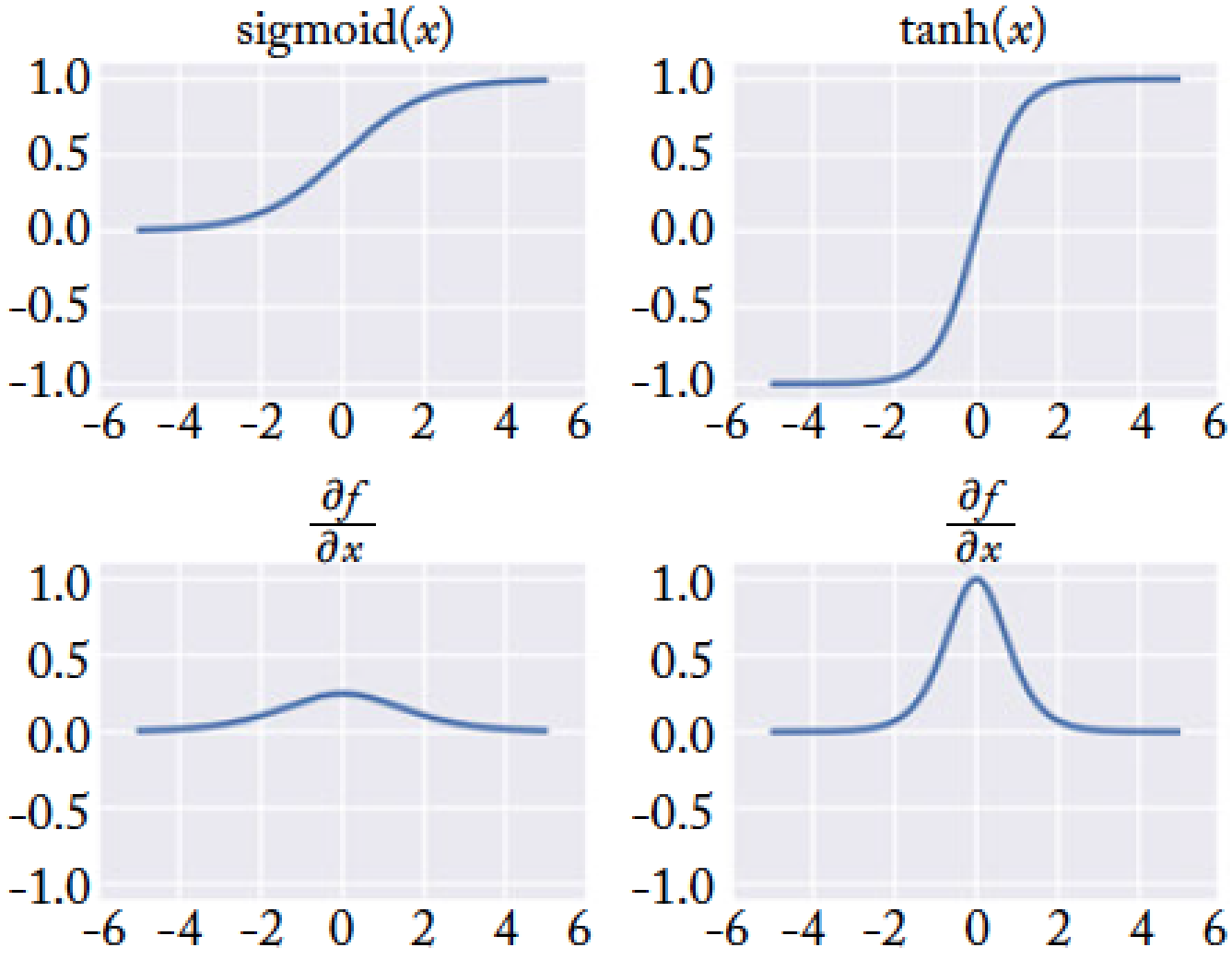


Finding global minimum: partial derivative of $\mathcal{L}(\Theta)$ wrt **W**.
The sign and magnitude of the slope (or gradient) tells us how y changes as x increases.

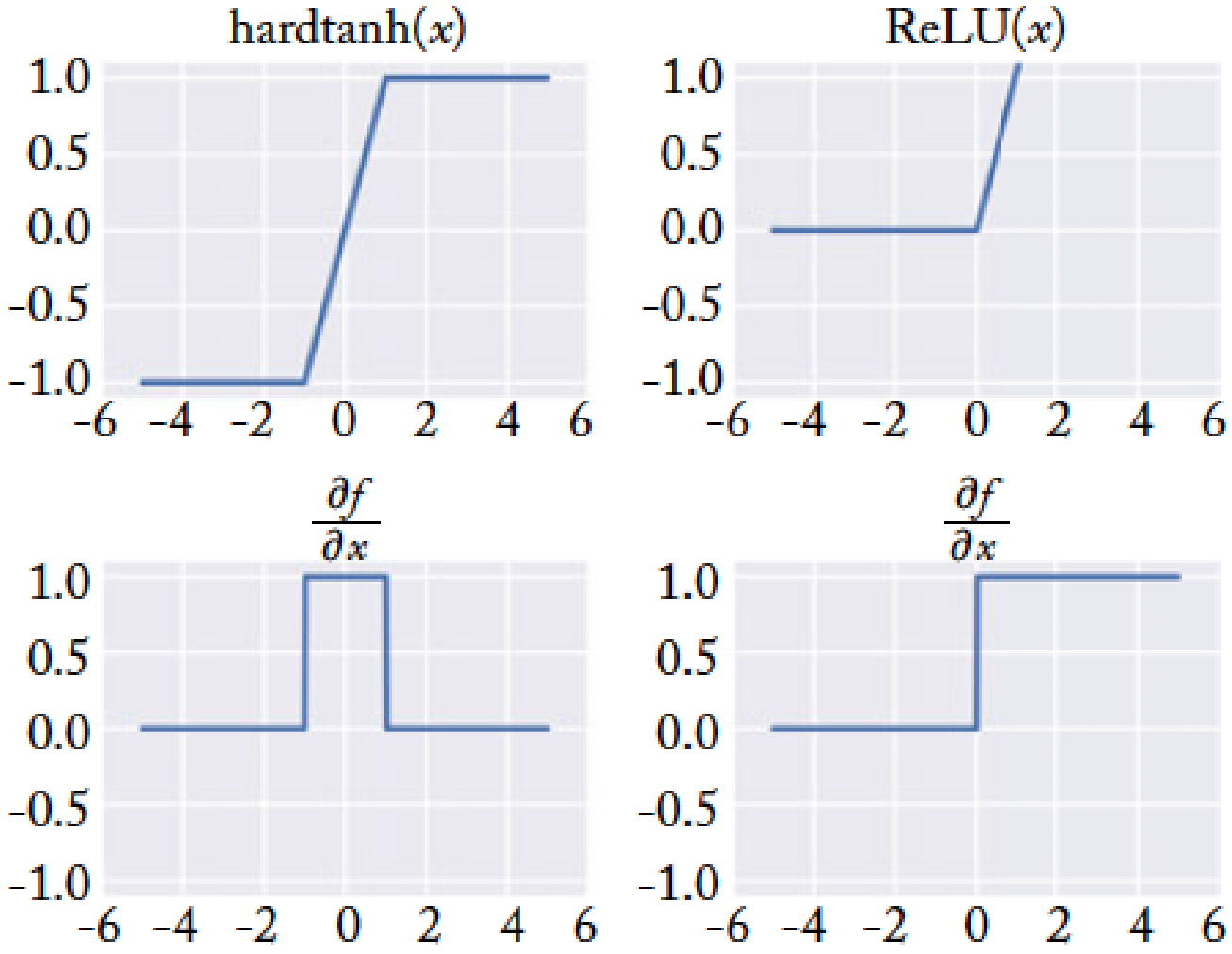


Solution may not be found if the function is not convex, but in practice this approach works well.

In order for this to work, the activation functions used by the network need to be differentiable, eg.:



ReLU turns out to be quite efficient to compute, and works well.

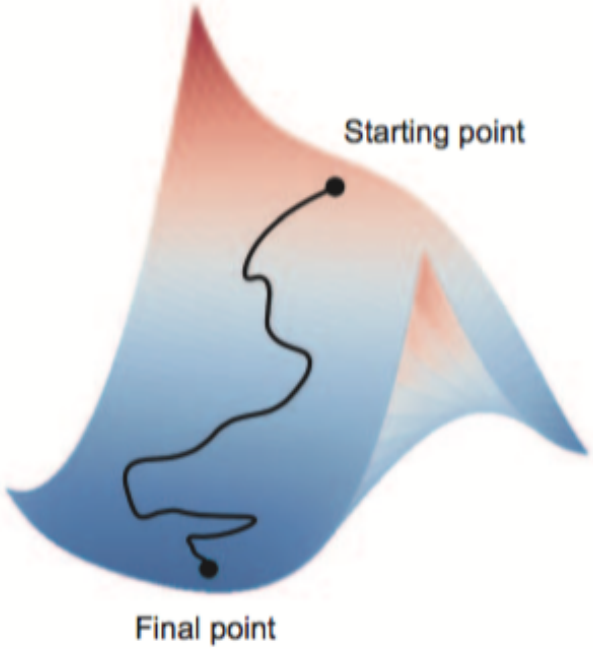


Adjusting the gradients:

- For simple networks: *stochastic gradient descent*
  ... Iteratively update the weight parameters in the direction of the gradient of the loss function until a minimum is reached.

$$w = w - \mu u$$

Where μ is a learning rate parameter (small number between 0 and 1), and u is the derivative. Example.



Adjusting the gradients:

- For networks with hidden layers: *backpropagation* (based on the chain rule of differentiation; example)

  ... start with the final loss and work backwards from the top layer to the bottom layer, applying the chain rule to compute the contribution of each parameter in the loss value.

  Minibatches: updates can occur with every training example, or with every k training examples (e.g. $k = 30$).

Our character bigram multi-class linear classifier for languages:

$$\hat{y} = \text{arg max}_i \hat{\mathbf{y}}_i$$

$$\hat{\mathbf{y}} = \mathbf{x} \cdot \mathbf{W} + \mathbf{b}$$

$$L(\hat{\mathbf{y}}, \mathbf{y}) = \text{max}(0, 1 - (\hat{\mathbf{y}}_i - \hat{\mathbf{y}}_k))$$

Note that if $1 - (\hat{\mathbf{y}}_i - \hat{\mathbf{y}}_k) \leqslant 0$ then the loss is 0, and so is the gradient.

Otherwise, we take the derivatives of $\frac{\partial L}{\partial \mathbf{b}_i}$ and $\frac{\partial L}{\partial \mathbf{W}_{i,j}}$.

Unpacking:

$$L(\hat{\mathbf{y}}, \mathbf{y}) = \max(0, 1-(\hat{\mathbf{y}}_i-\hat{\mathbf{y}}_k)) = \max(0, 1-((\mathbf{x}\cdot\mathbf{W}+\mathbf{b})_t-(\mathbf{x}\cdot\mathbf{W}+\mathbf{b})_k)) =$$

$$\max\Big(0, 1 - \Big(\Big(\sum_i \mathbf{x}_i \cdot \mathbf{W}_{i,t} + \mathbf{b}_t\Big) - \Big(\sum_i \mathbf{x}_i \cdot \mathbf{W}_{i,k} + \mathbf{b}_k\Big)\Big)\Big) =$$

$$\max\Big(0, 1 - \sum_i \mathbf{x}_i \cdot \mathbf{W}_{i,t} - \mathbf{b}_t + \sum_i \mathbf{x}_i \cdot \mathbf{W}_{i,k} + \mathbf{b}_k\Big) =$$

1. $\frac{\partial L}{\partial \mathbf{b}_n}$

   If $n \neq k, t$ the term $\mathbf{b}_n$ does not contribute to the loss, and so its derivative is 0. If $n = k$ the derivative is 1, and for $n = t$, we get -1.

2. $\frac{\partial L}{\partial \mathbf{W}_{i,m}}$

   For $\mathbf{W}_{i,m}$, only $m = k$ and $m = t$ contribute to the loss.

   If $m = t$ then $\frac{\partial(-\mathbf{x}_i \cdot \mathbf{W}_{i,t})}{\partial \mathbf{W}_{i,t}} = -\mathbf{x}_i$

   If $m = k$ then $\frac{\partial(\mathbf{x}_i \cdot \mathbf{W}_{i,k})}{\partial \mathbf{W}_{i,k}} = \mathbf{x}_i$

   Otherwise, 0

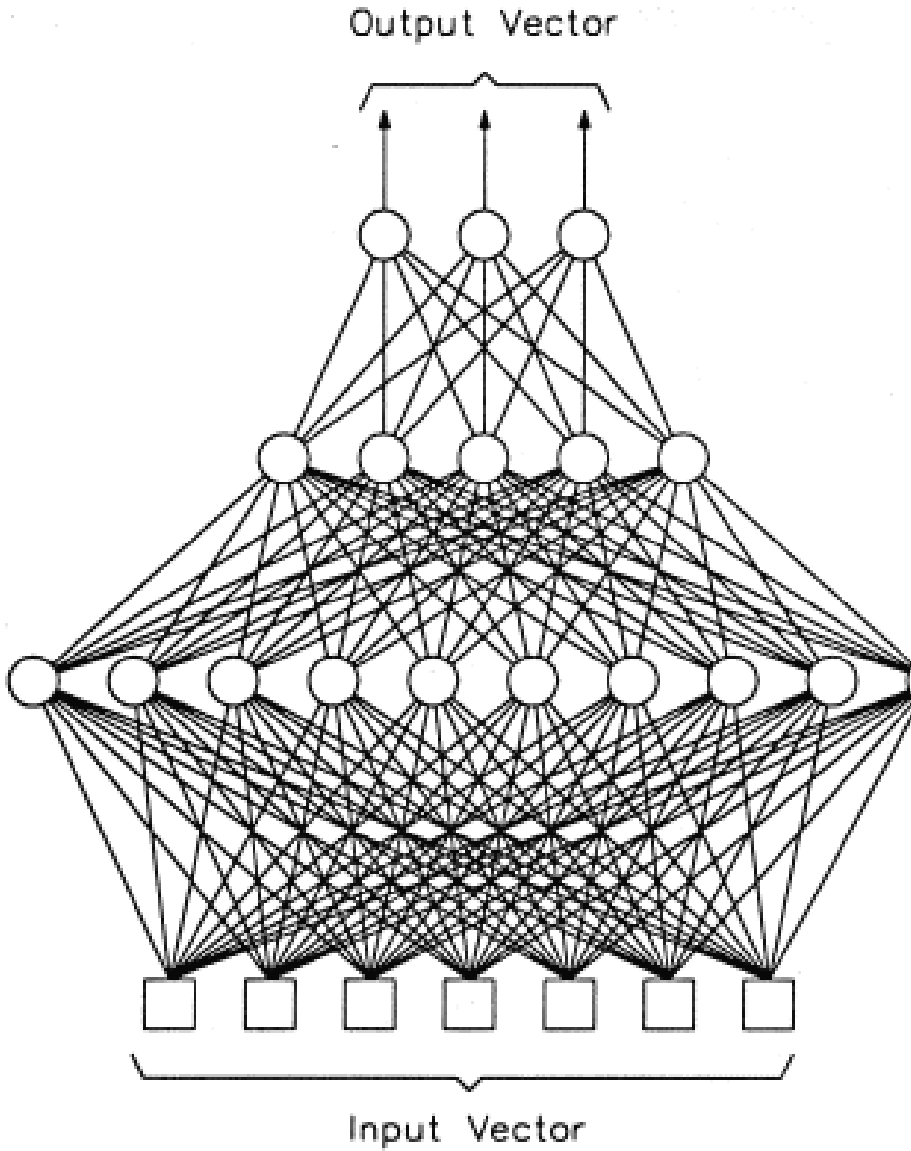
$$\text{MLP2}(\mathbf{x}) = \mathbf{y}$$

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

$$\mathbf{h}^2 = g^2(\mathbf{h}^1 \cdot \mathbf{W}^2 + \mathbf{b}^2)$$

$$\mathbf{y} = \mathbf{h}^2 \cdot \mathbf{W}^3$$

Where the gs are non-linear transformations (sigmoid, tanh, RectifierLU, etc.)



This MLP2 is a fully-connected (affine) feed-forward multi-layer perceptron.

Suppose you wanted to use MLP2 to create a document classification system. The output is one of the following classes: Economy, Politics, Sports, Leisure, Gossip, Lifestyle, Other.

1. What is your input, and how do you encode it?
2. What is your training set?
3. How many output neurons?
4. Which activation function do you use for the last layer?
5. Which loss function do you use for the last hidden layer?

NN

Suppose you wanted to use MLP2 to create a document classification system. The output is one of the following classes: Economy, Politics, Sports, Leisure, Gossip, Lifestyle, Other.

1. What is your input, and how do you encode it?
   Input is a document encoded as a vector; each position in the vector corresponds to a word (or lemma). Additional positions may be used to represent normalized bigram counts.
2. What is your training set?
3. How many output neurons?
4. Which activation function do you use for the last layer?
5. Which loss function do you use for the last hidden layer?

NN

Suppose you wanted to use MLP2 to create a document classification system. The output is one of the following classes: Economy, Politics, Sports, Leisure, Gossip, Lifestyle, Other.

1. What is your input, and how do you encode it?
   Input is a document encoded as a vector; each position in the vector corresponds to a word (or lemma). Additional positions may be used to represent normalized bigram counts.
2. What is your training set?
   Class-labeled documents encoded as a vector (like above).
3. How many output neurons?
4. Which activation function do you use for the last layer?
5. Which loss function do you use for the last hidden layer?

NN

Suppose you wanted to use MLP2 to create a document classification system. The output is one of the following classes: Economy, Politics, Sports, Leisure, Gossip, Lifestyle, Other.

1. What is your input, and how do you encode it?
   Input is a document encoded as a vector; each position in the vector corresponds to a word (or lemma). Additional positions may be used to represent normalized bigram counts.
2. What is your training set?
   Class-labeled documents encoded as a vector (like above).
3. How many output neurons?
   7
4. Which activation function do you use for the last layer?
5. Which loss function do you use for the last hidden layer?

Suppose you wanted to use MLP2 to create a document classification system. The output is one of the following classes: Economy, Politics, Sports, Leisure, Gossip, Lifestyle, Other.

1. What is your input, and how do you encode it?
   Input is a document encoded as a vector; each position in the vector corresponds to a word (or lemma). Additional positions may be used to represent normalized bigram counts.
2. What is your training set?
   Class-labeled documents encoded as a vector (like above).
3. How many output neurons?
   7
4. Which activation function do you use for the last layer?
   Softmax, so that we have multinomial probabilities
5. Which loss function do you use for the last hidden layer?

NN

Suppose you wanted to use MLP2 to create a document classification system. The output is one of the following classes: Economy, Politics, Sports, Leisure, Gossip, Lifestyle, Other.

1. What is your input, and how do you encode it?
   Input is a document encoded as a vector; each position in the vector corresponds to a word (or lemma). Additional positions may be used to represent normalized bigram counts.
2. What is your training set?
   Class-labeled documents encoded as a vector (like above).
3. How many output neurons?
   7
4. Which activation function do you use for the last layer?
   Softmax, so that we have multinomial probabilities
5. Which loss function do you use for the last hidden layer?
   Categorial cross-entropy

Suppose you wanted to use MLP2 to create an authorship classification system. Given a text, the model should classify it as 'Native' or 'Non-native'.

Typical input features:

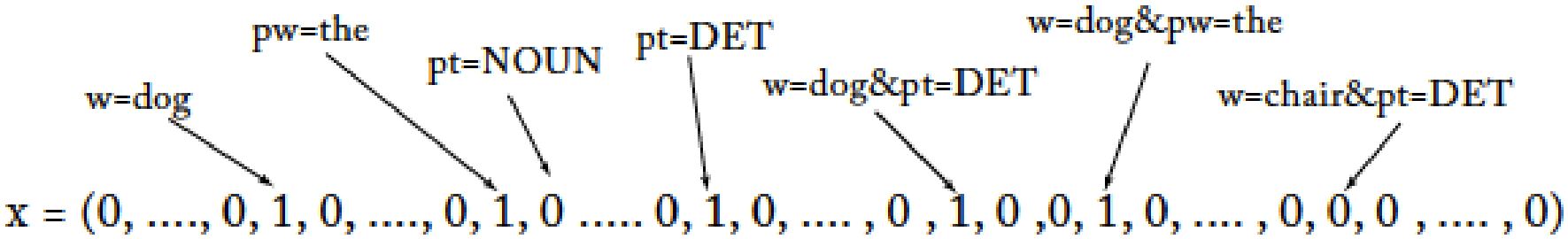
- (Bag of) POS tags
  ... ie. counts for each tag.
- (Bag of) Function words (stop words) & pronouns
- POS bigrams and trigrams
- Ratio between function words and content words in a window of text
- ..

(Bag of words : concatenation of word vectors, disregarding word order in text)

Part of speech tagging, MLP features:

- word=X
- 2-letter-suffix=X
- 3-letter-suffix=X
- 2-letter-prefix=X
- 3-letter-prefix=X
- word-is-capitalized
- word-contains-hyphen
- word-contains-digit
- For P in [-2,-1,+1,+2]:
  - Word at position P=X
  - 2/3-letter-prefix/suffix of word at position P=X
  - word at position P=X is capitalized, contains hyphen, contains digit
- Predicted POS of word at position -1 = X
- Predicted POS of word at position -2 = X

In such tasks, the input vector is a gigantic concatenation of large (and extremely sparse one-hot) vectors:

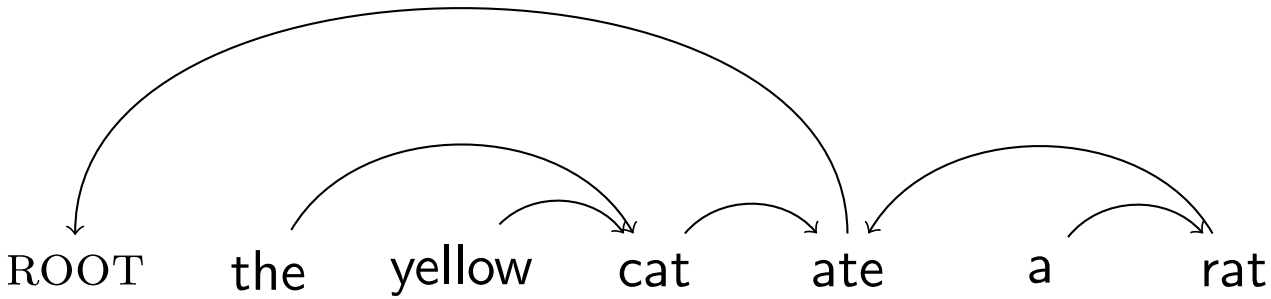


... with a moving window of 4 words (2 to the left of word of interest, and 2 to the right).

Dependency parsing task

Input: sentence

Output: dependency parse



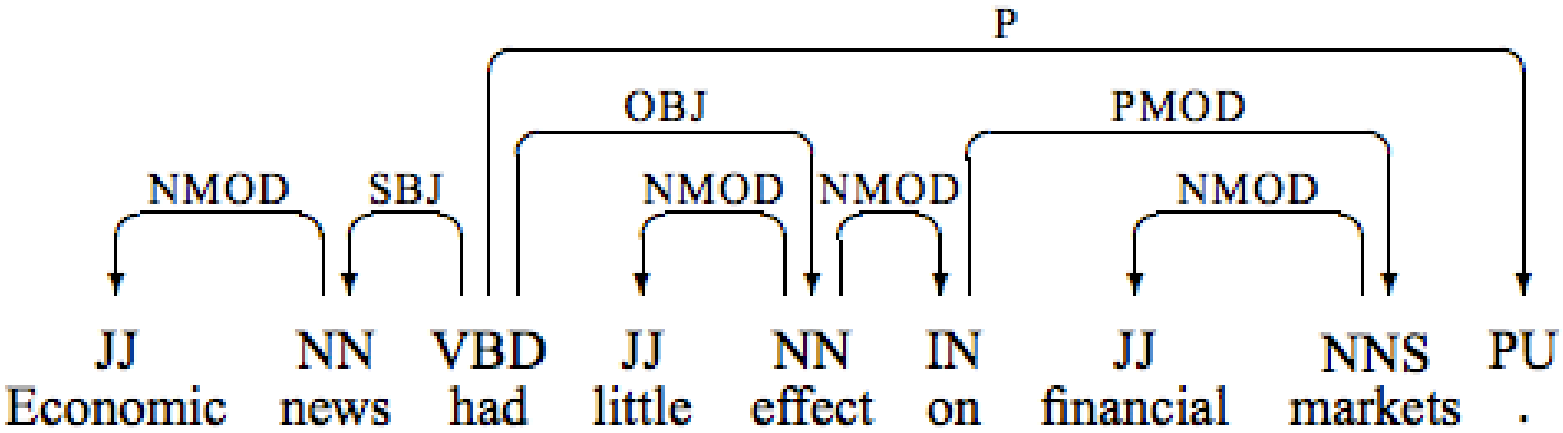
In Dependency Grammar there are no trees, only directed binary relations, where 'X → Y' means 'X depends on Y':

arc(cat-3,the-1)	arc(cat-3,yellow-2)
arc(ROOT-0, ate-4)	arc(ate-4,cat-3)
arc(ate-4,rat-6)	arc(rat-6,a-5)

See a demo here

Dependency Grammar

There are many variations wrt direction, typing and rooting:



Dependency Grammar

A dependency structure is a graph $G = \langle V, E, O \rangle$ where

- V is a set of nodes, e.g.
  $\{n(the, n_1), n(cat, n_2), n(snores, n_3), n(root, n_4)\}$
- E is a set of arcs (edges), e.g.
  $\{arc(n_2, n_1), arc(n_3, n_2), arc(n_4, n_3)\}$
  Alternatively, labeled in some way, eg.
  $\{arc_{det}(n_2, n_1), arc_{subj}(n_3, n_2), arc_{root}(n_4, n_3)\}$
- O is a linear precedence order on V, e.g.
  $\{n_1 < n_2, n_2 < n_3, n_3 < n_4\}$

Lexical entries:

- $x(w_1, ..., *, ..., w_k)$ means $w_1 ... w_k$ are dependent on x
  V(N, $*$, N) : *ate*
  N(DT, $*$) : *cat*
- x(*) means: x is a leaf node
  N($*$) : *Robin*
  Det($*$): *the*

Greedy Discriminative Transition-Based Dependency Parsing

- Actions: Shift, Reduce
- Storage:
  - A stack σ, written with top to the right, e.g. $\sigma|w$
    It is initialized with the ROOT symbol.
  - A buffer β, written with top to the left, e.g. $w|\beta$
    It is initialized with the input sentence $w_1...w_n$.
  - A set of dependency arcs D, initialized as empty: $\varnothing$.
- A discriminative classifier (MaxEnt, NN) identifies the most likely upcoming dependency given the input and grammar rules.

Greedy Discriminative Transition-Based Dependency Parsing

arc-eager: heads can take a right dependent, before its dependents are found.

Start: $\sigma = [\text{ROOT}]$, $\beta = w_1, ..., w_n$, $D = \varnothing$

While $\beta \neq \varnothing$:

- Left-Arc: $\sigma|w_i,\ w_j|\beta,\ D \rightarrow \sigma,\ w_j|\beta,\ D \cup \{arc(w_j, w_i)\}$
  Precondition: $arc(w_k, w_i) \notin D,\ w_i \neq \text{ROOT}$
- Right-Arc:
  $\sigma|w_i,\ w_j|\beta,\ D \rightarrow \sigma|w_i|w_j,\ \beta,\ D \cup \{arc(w_i, w_j)\}$
- Shift: $\sigma,\ \ w_i|\beta,\ D \rightarrow \ \sigma|w_i,\ \beta,\ D$
- Reduce: $\sigma|w_i,\ \beta,\ D \rightarrow \sigma,\ \beta,\ D$
  Precondition: $arc(w_k, w_i) \in D$

Dependency Parsing

Example: The cat can eat the rat.

σ	β	$D = \varnothing$	**Action**
[ROOT]	[the, cat, can eat, the, rat]	D	
[ROOT, the]	[cat, can, eat, the, rat]	D	Shift
[ROOT]	[cat, can, eat, the, rat]	$D_1 = D \cup \{arc(cat, the)\}$	Left-Arc
[ROOT,cat]	[can, eat, the, rat]	D_1	Shift
[ROOT]	[can, eat, the, rat]	$D_2 = D_1 \cup \{arc(can, cat)\}$	Left-Arc
[ROOT, can]	[eat, the, rat]	D_2	Shift
[ROOT, can]	[eat, the, rat]	$D_3 = D_2 \cup \{arc(\text{ROOT}, can)\}$	Right-Arc
[ROOT, can, eat]	[the, rat]	$D_4 = D_3 \cup \{arc(can, eat)\}$	Right-Arc
[ROOT, can, eat, the]	[rat]	D_4	Shift
[ROOT, can, eat]	[rat]	$D_5 = D_4 \cup \{arc(rat, the)\}$	Left-Arc
[ROOT, can, eat, rat]	[]	$D_6 = D_5 \cup \{arc(rat, the)\}$	Right-Arc
[ROOT, can, eat]	[]	D_6	Reduce
[ROOT, can]	[]	D_6	Reduce
[ROOT]	[]	D_6	Reduce

Arc-factored approach to dependency parsing

- Each of the n^2 word-word arcs is assigned a score
- Pick the parse with a valid tree and the highest score

Goal: train a scoring function $\textsc{ArcScore}(h, m, s)$, where h (head) and d (dependent) are (the indices of) two words in sentence s that are being considered as candidates for attachment

NN

The input sentence is the sequence of tokens and the respective POS tags: $(w_1 ... w_n, p_1, ..., p_n)$.

$\textsc{ArcScore}(h, m, s)$ is trained with the following features.

- Word form and POS of head word
- Word form and POS of the dependent word
- Word (and/or POS tags) in a window of -2 and $+2$ words relative to w_h, including their relative positions
- Word (and/or POS tags) in a window of -2 and $+2$ words relative to w_d, including their relative positions
- Distance between w_h and w_d (i.e. $|h - d|$)
- Direction of the dependency
- Word forms and their POSs appearing between w_h and w_d (using a bag-of-words approach)

Word prediction

Let w be a target word, and c_1^k be an ordered list of (context) words. Both w and c_1^k are encoded as one-hot vectors.

The goal is predict w from c_1^k. That is:

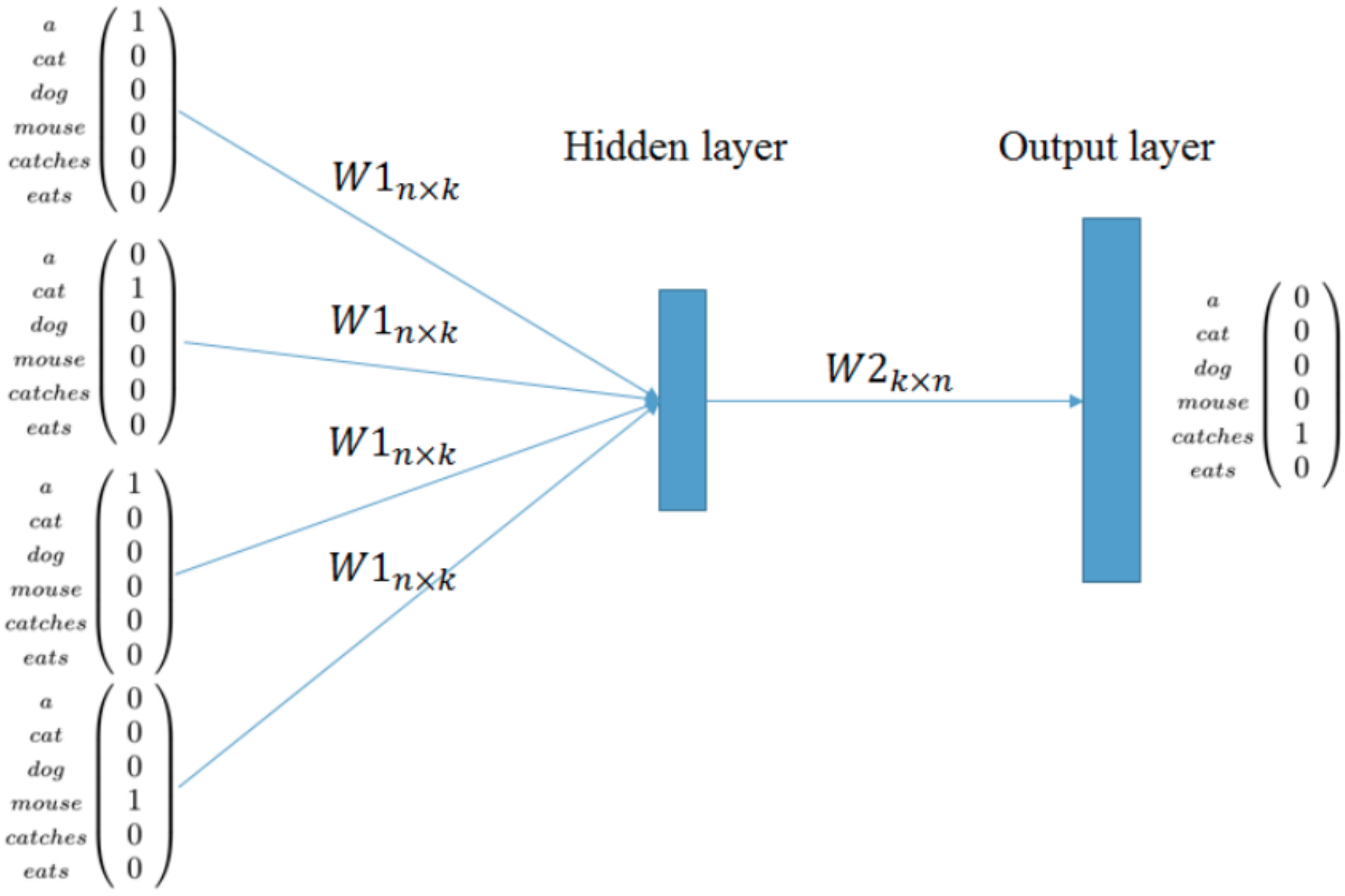
$$\hat{\mathbf{y}} = P(w|c_1 ... c_k)$$

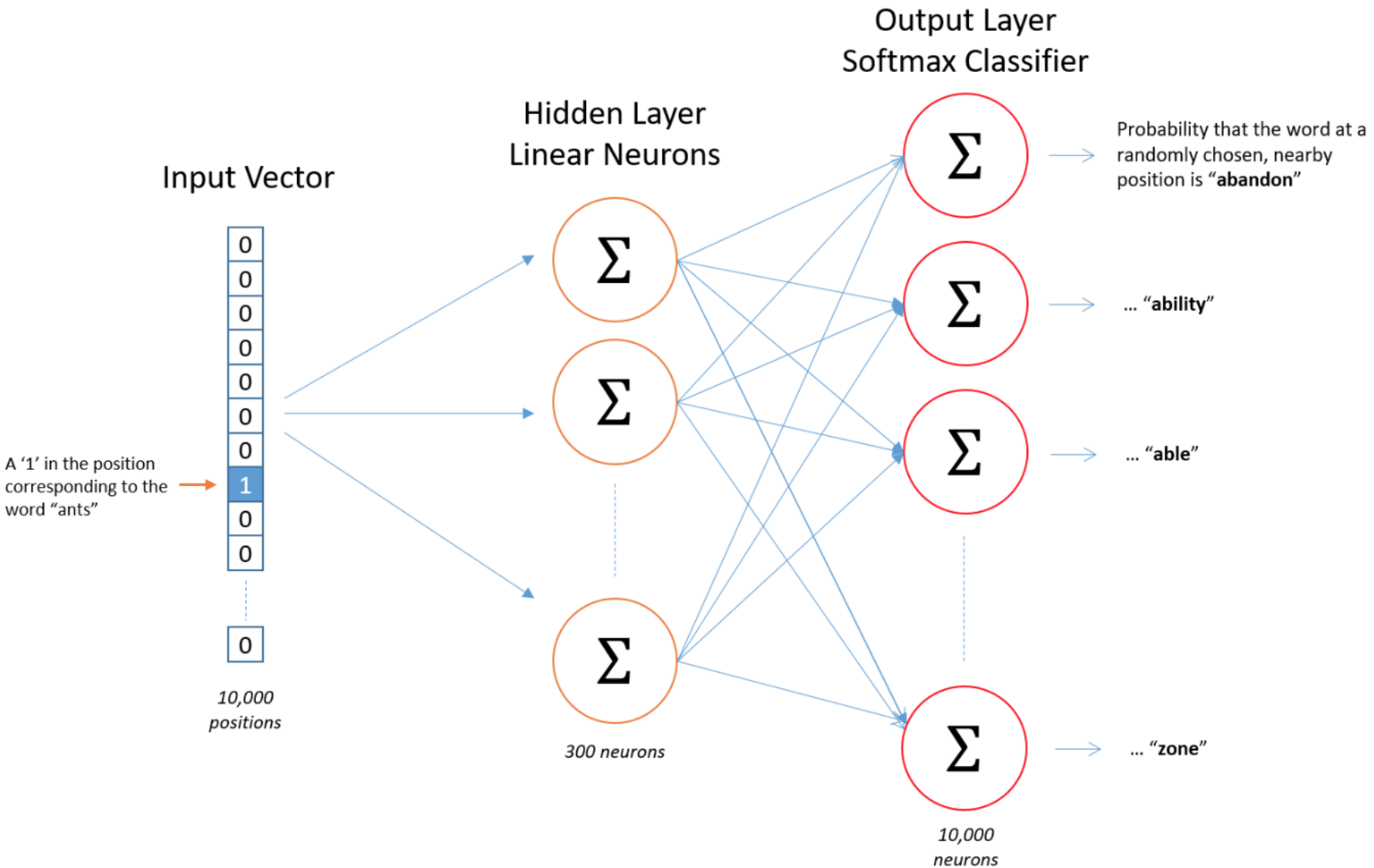
Generalizing to a vector $\mathbf{x}$ of all w's:

$$\text{MLP2}(\mathbf{x}) = \mathbf{y}$$
$$\mathbf{h}^1 = \sigma(\mathbf{x} \cdot \mathbf{W}^1 + \mathbf{b}^1)$$
$$\mathbf{h}^2 = \text{softmax}(\mathbf{h}^1 \cdot \mathbf{W}^2 + \mathbf{b}^2)$$
$$\mathbf{y} = \mathbf{h}^2 \cdot \mathbf{W}^3$$

Add special symbol 'Unk' for unknown words (randomly replace known words with 'Unk' for training), use sentence boundary symbols, and use cross-entropy loss.

NN


a cat dog mouse catches eats
1 0 0 0 0 0
a cat dog mouse catches eats
0 1 0 0 0 0
a cat dog mouse catches eats
1 0 0 0 0 0
a cat dog mouse catches eats
0 0 0 1 0 0
$W1_{n\times k}$
$W1_{n\times k}$
$W1_{n\times k}$
$W1_{n\times k}$
Hidden layer
$W2_{k\times n}$
Output layer
a cat dog mouse catches eats
0 0 0 0 1 0


Output Layer
Softmax Classifier
Hidden Layer
Linear Neurons
Input Vector
Probability that the word at a randomly chosen, nearby position is "abandon"
... "ability"
... "able"
... "zone"
A '1' in the position corresponding to the word "ants"
0
0
0
0
0
0
0
1
0
0
0
10,000 positions
300 neurons
10,000 neurons

Obtaining dense word vectors from $\mathbf{W}^2$:

- Remove the softmax (for efficiency purposes)
- Other layers use *ReLu* or *hardtanh*, for example.
- Use a word surrounding context:
  $P(w_3|w_1 w_2 * w_4 w_5)$ instead of $P(w_5|w_1 w_2 w_3 w_4 w_5 *)$

The resulting model yields a score $\text{MLP}(w, c_1^k)$

The Loss(w, c_1^k) is

$$L(w, c, x') = \max(0, 1 - (\text{MLP}(w, c_1^k) - \text{MLP}(w', c_1^k)))$$

where w' is a random word from the vocabulary. This randomly produces incorrect word-context pairs to minimize the loss.

W(“woman")–W(“man") ≈ W(“aunt")–W(“uncle")
W(“woman")–W(“man") ≈ W(“queen")–W(“king")

Figures from Mikolov, T., Yih, W., & Zweig, G. (2013). Linguistic Regularities in Continuous Space Word Representations

Some shortcomings of vector space semantics:

- What counts as similar depends on the goal.
  cat is similar to *dog* as both are pets. On the other hand, *cat* is similar to *tiger*, as both are felines.
- Antonyms
  good and *bad* tend to appear the the same contexts, but are not similar (in a sense).
- Biases
  Racial and gender stereotypes (i.e., European American names are closer to pleasant terms while African American names are closer to unpleasant terms; female names are more associated with family terms than with career terms; it is possible to predict the percentage of women in an occupation according to U.S. census based on the vector representation of the occupation name).
  (Caliskan et al. 2016)

Word embedding software:

- word2vec
- GenSim[1]
- GloVe

Pre-trained models:

- English, from Google
- English, from Stanford
- 137 languages, from Polyglot

[1] For whole documents use Doc2Vec.

Many freely available tools to implement NNs (Python, Java, Javascript, R, Lua, ...):

- Pytorch (Facebook)
- TensorFlow (Google) & TFJavascript
- CNTK (Microsoft)
- Keras (to run on top of TensorFlow and CNTK)
- Chainer
- DyNet
- Caffe2
- ...

And various repositories for datasets:

- Kaggle
- UCI Repository
- IMDB
- YELP
- DLJ4
- ...

Convolutional networks: capture the local aspects that are most informative for the task, and can deal with unbounded ngram vocabularies while keeping a bounded size embedding matrix.

Let $w_1 ... w_n$ be a sentence S, and let **E** be a matrix of the respective dense vectors (aka 'embeddings').

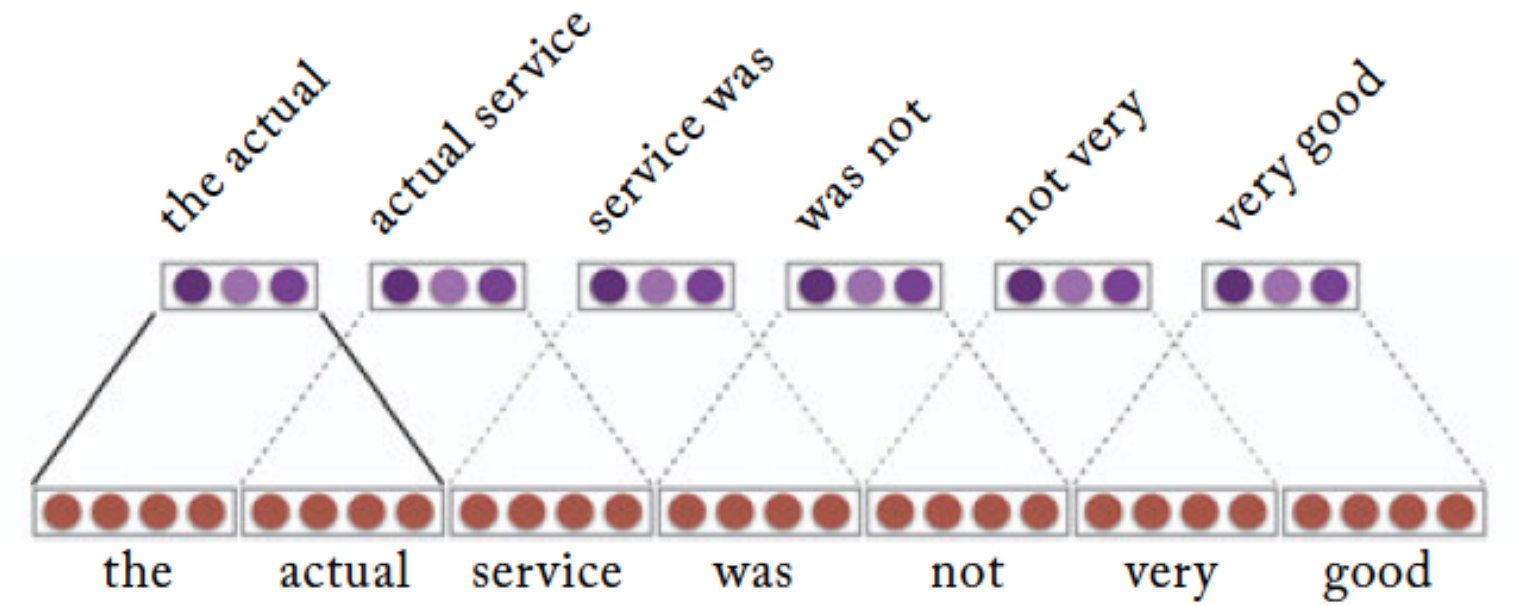
A (1D) convolution of size k is a sliding window of k words over S to which the following are applied:

1. vector concatenation ('$\oplus$')
2. dot-product with a weight vector
3. a non-linear function

$$\mathbf{x}_i = g(\oplus(w_i ... w_{i+k-1}) \cdot \mathbf{u})$$

This allows the identification of local predictors in a large structure, and combines them to produce a fixed size vector representation of the structure.

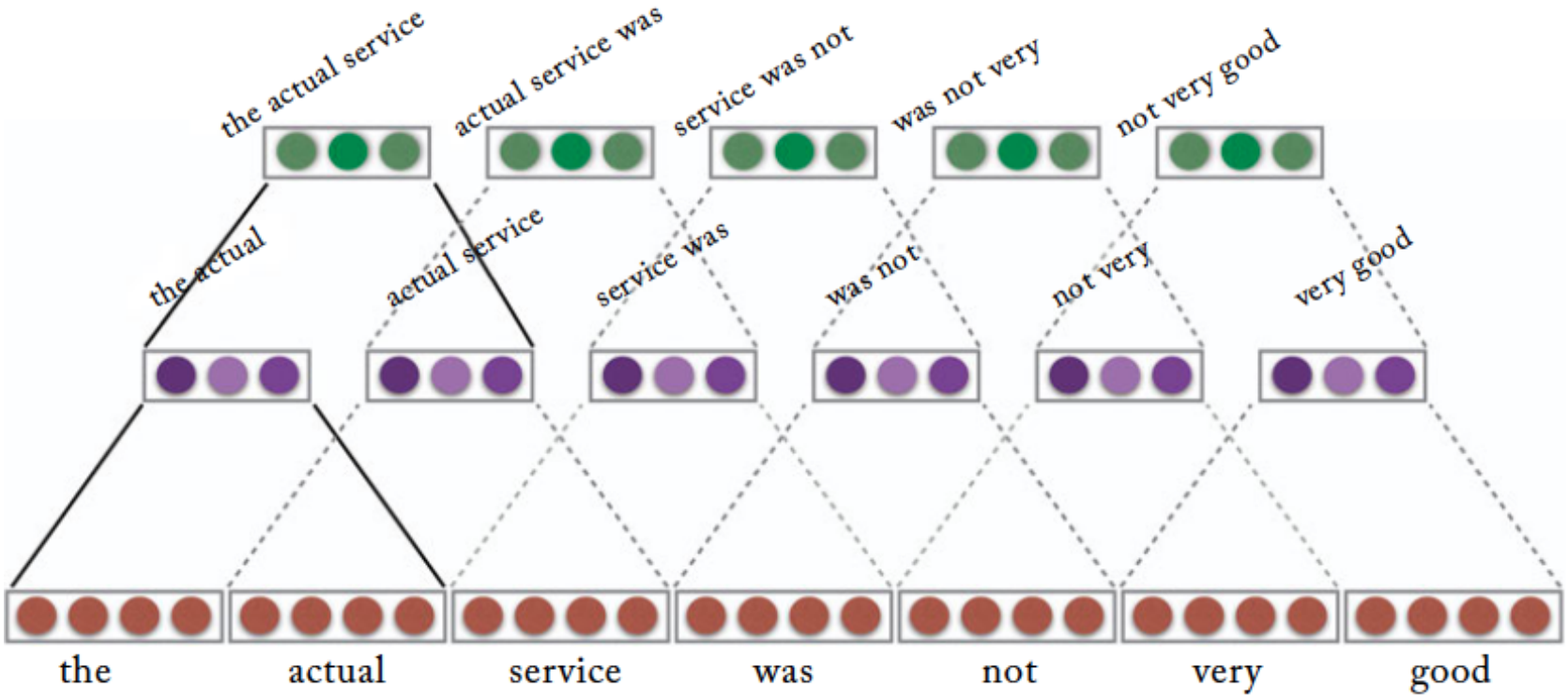
A narrow convolution of size $k = 2$



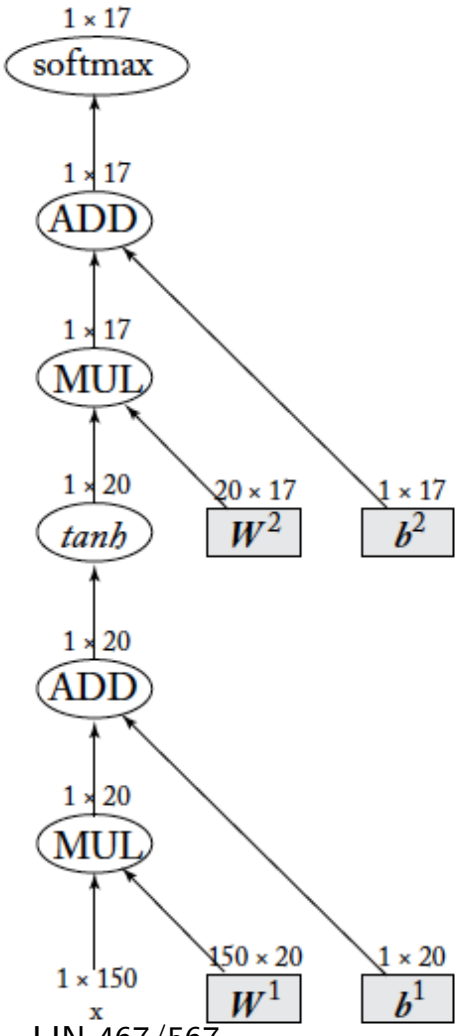
CNN's are used for:

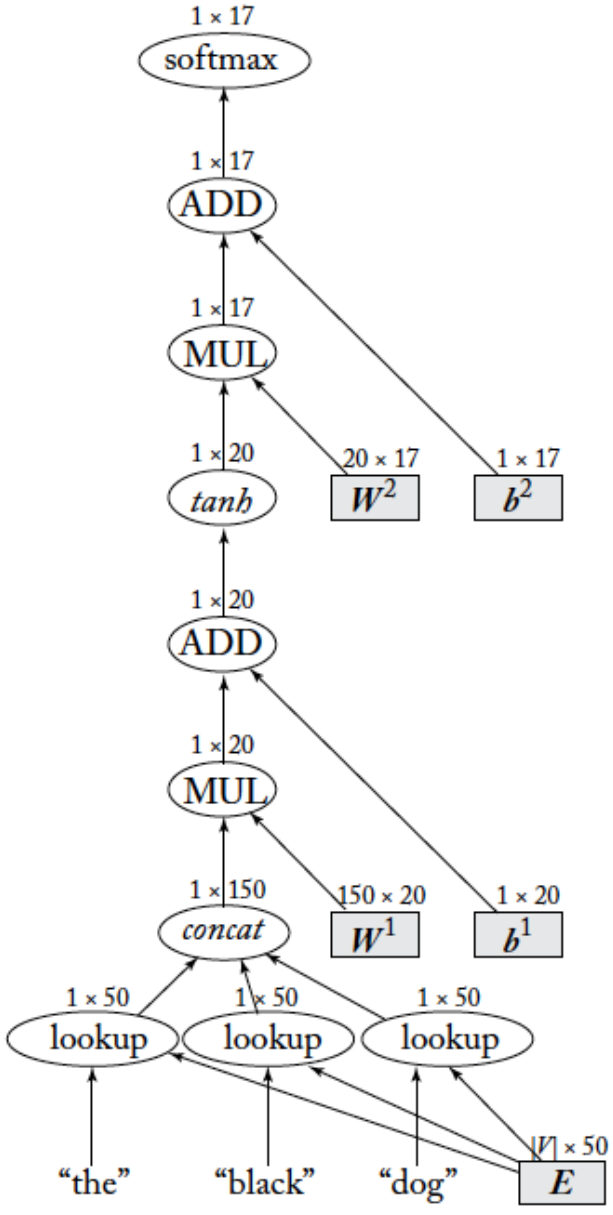
- Document classification (example)
- Sentiment classification (example)
- Question answering (example)

Convolutions are often hierarchical:

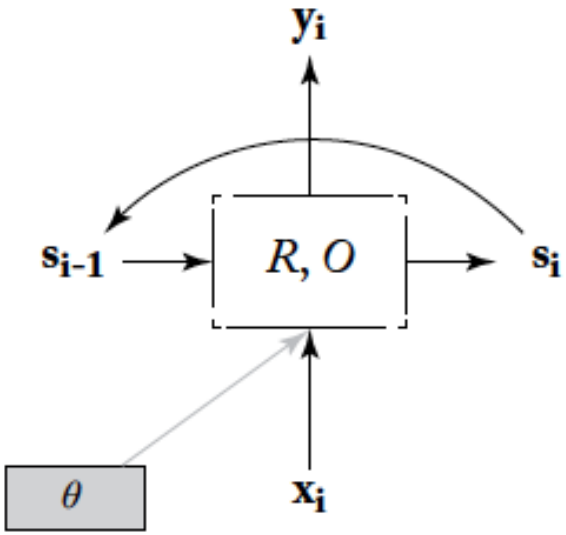


The computational graph abstraction:





Recurrent Neural Networks



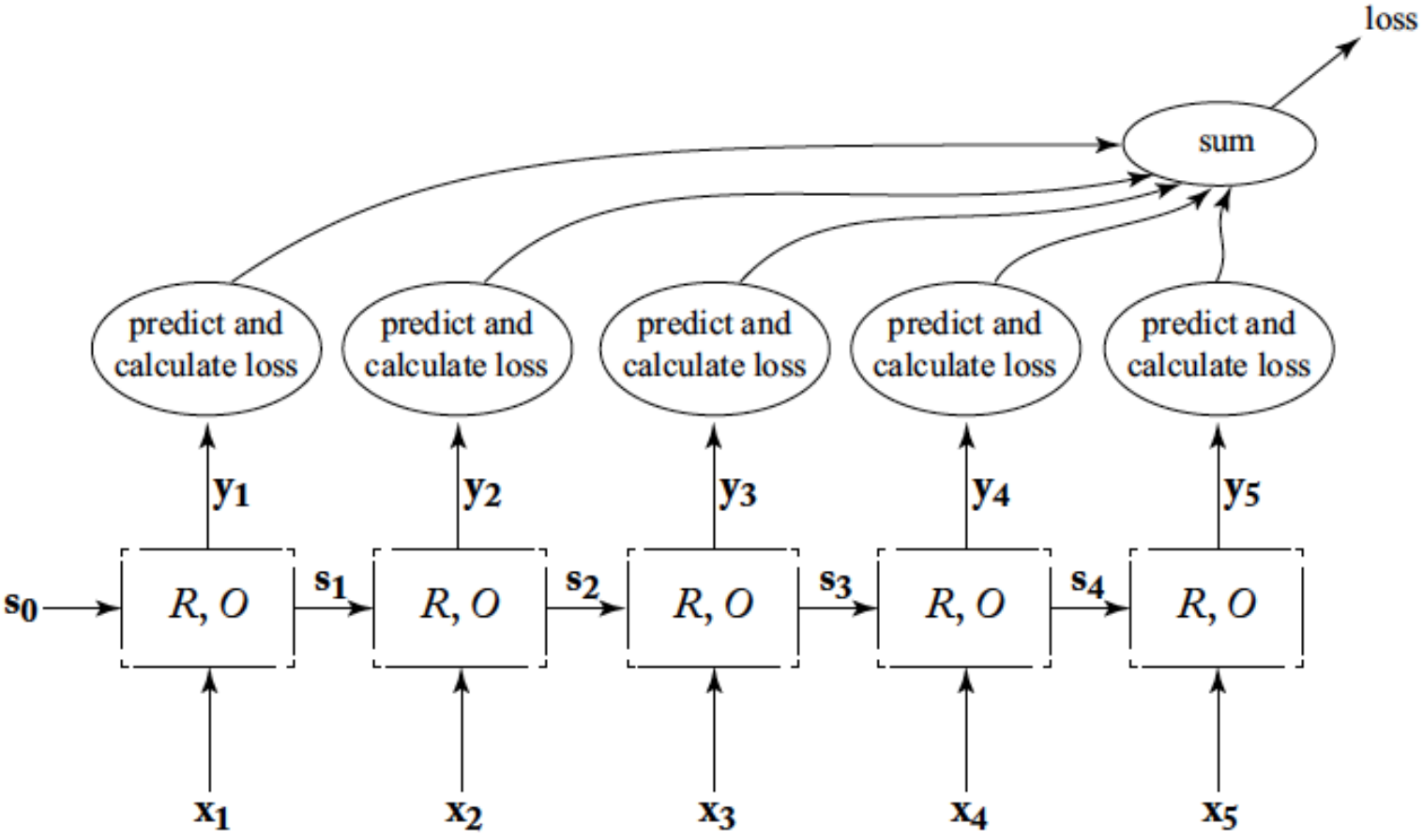
R is the network's (recursively defined) output
O is either identity or subset of the state

$$RNN(\mathbf{w}_{1:n}; \mathbf{s}) = \mathbf{y}_{1:n}$$

$$\mathbf{y}_i = O(\mathbf{s}_i)$$

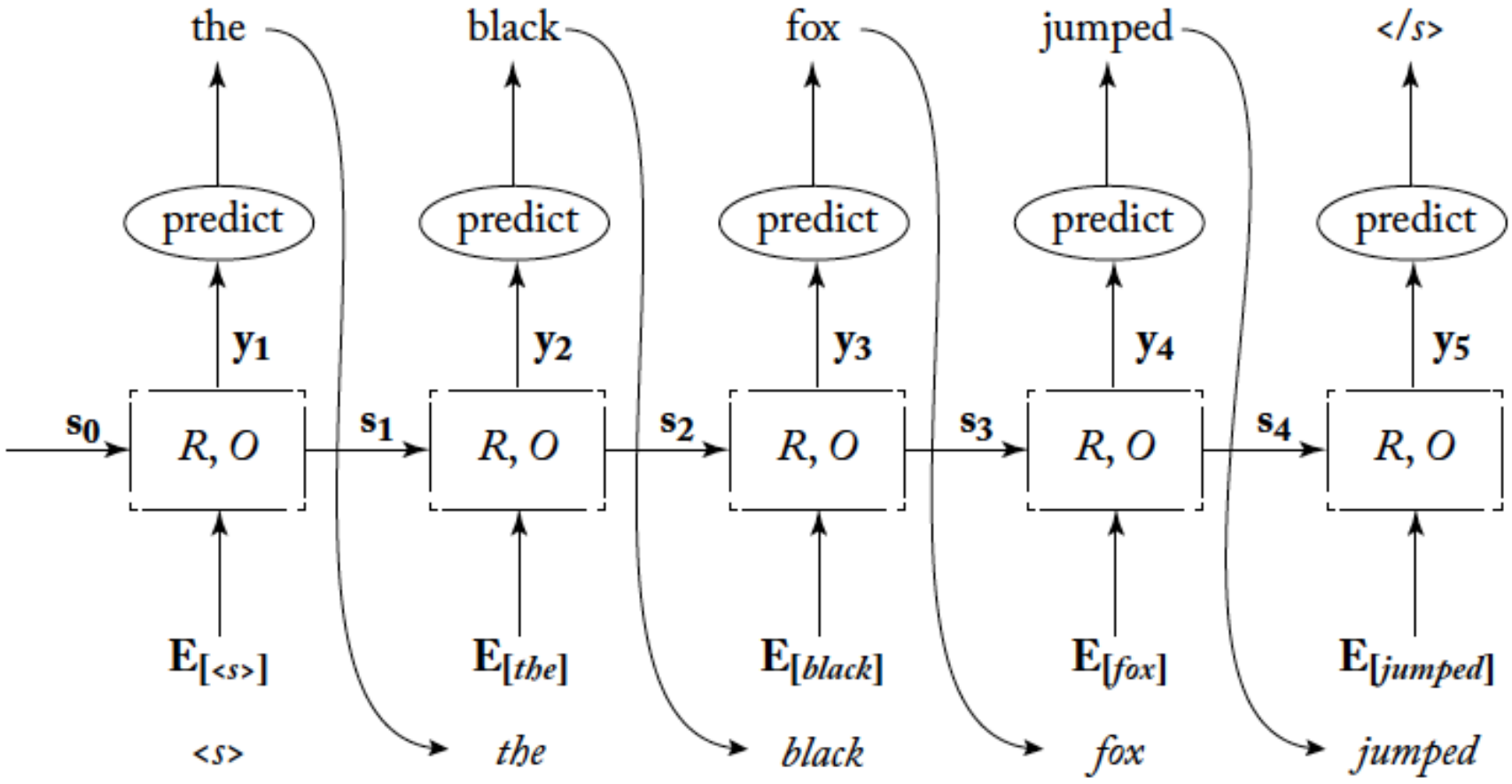
$$\mathbf{s}_i = R(\mathbf{s}_{i-1}, \mathbf{x}_i)$$

Recurrent Neural Network (unrolled)

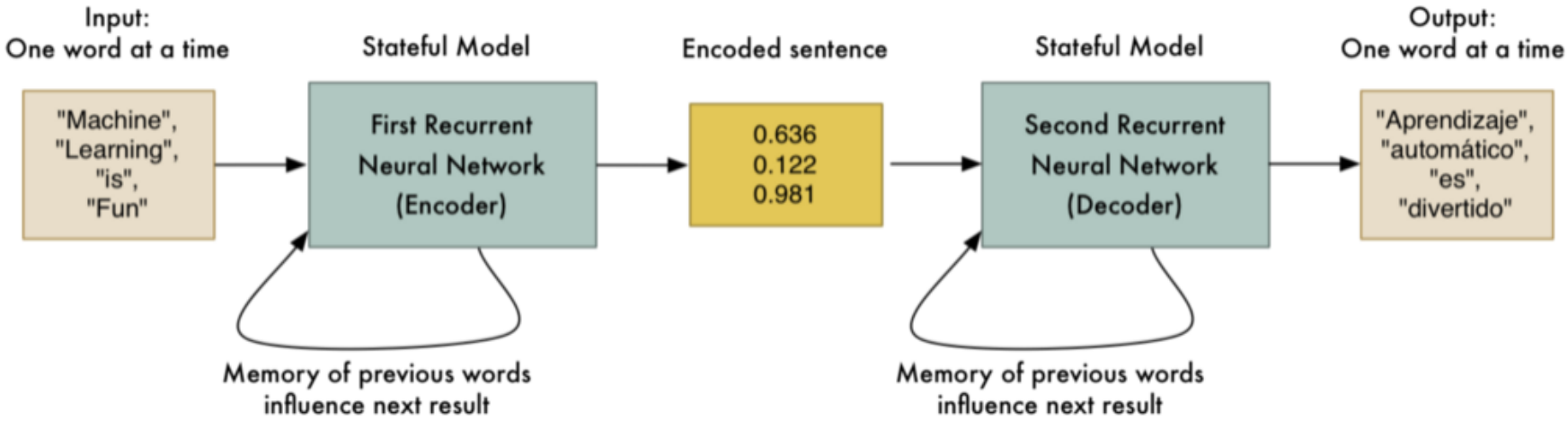


Such networks allow us to abandon the Markov assumption.

Recurrent Neural Network (predictor)



Recurrent Neural Network (translation)



Recurrent Neural Network (bidirectional)