Adaptive basis function models

Agenda

- Introduction
- Classification and regression trees (CART)
- Feedforward neural networks

Introduction

- Try to learn useful features $\phi(x)$ directly from the input data
- We will create a model called **adaptive basis-function model** (ABM), which has the form:

$$f(\mathbf{x}) = w_0 + \sum_{m=1}^{M} w_m \phi_m(\mathbf{x})$$

where $\phi_m(\mathbf{x})$ is the m’th basis function, which is learned from the data

- Typically the basis functions are parametric – $\phi_m(\mathbf{x}) = \phi(\mathbf{x}; \mathbf{v}_m)$ where $\mathbf{v}_m$ are the parameters of the basis function itself
- The entire parameter set is $\boldsymbol{\theta} = (w_0, \mathbf{w}_{1:M}, \{\mathbf{v}_m\}_{m=1}^{M})$

Introduction

- The resulting model is not linear-in-the-parameters anymore
- We will only be able to compute a locally optimal MLE or MAP estimate of θ
- Such models often significantly outperform linear models

Classification and regression trees (CART)

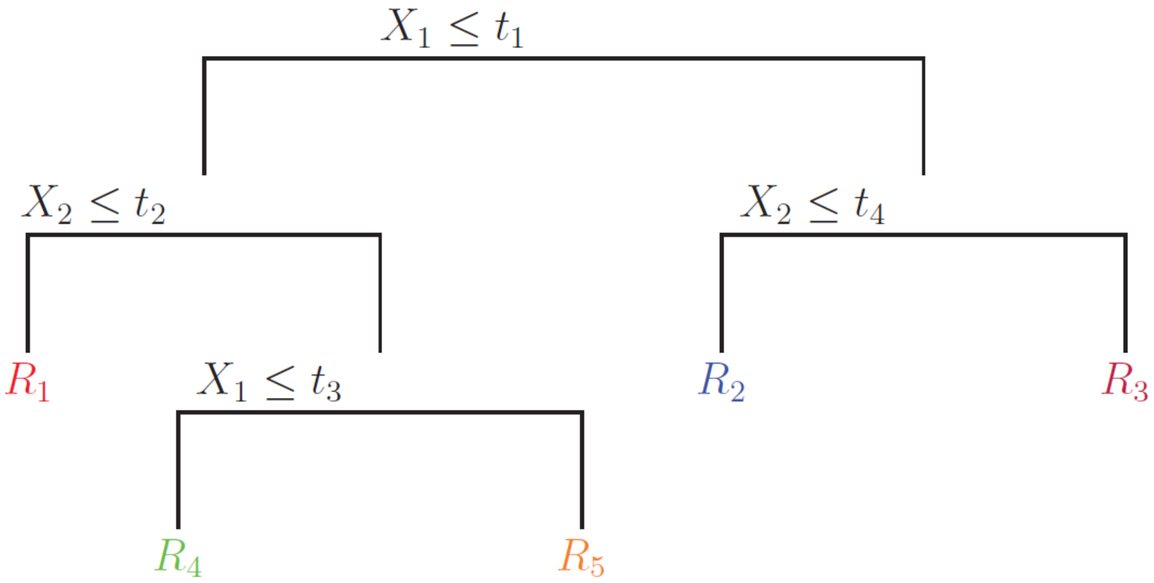
- CART od decision trees are defined by **recursively partitioning the input space**, and defining **local model in each resulting region** of input space
- Represented as a tree, with one leaf per region

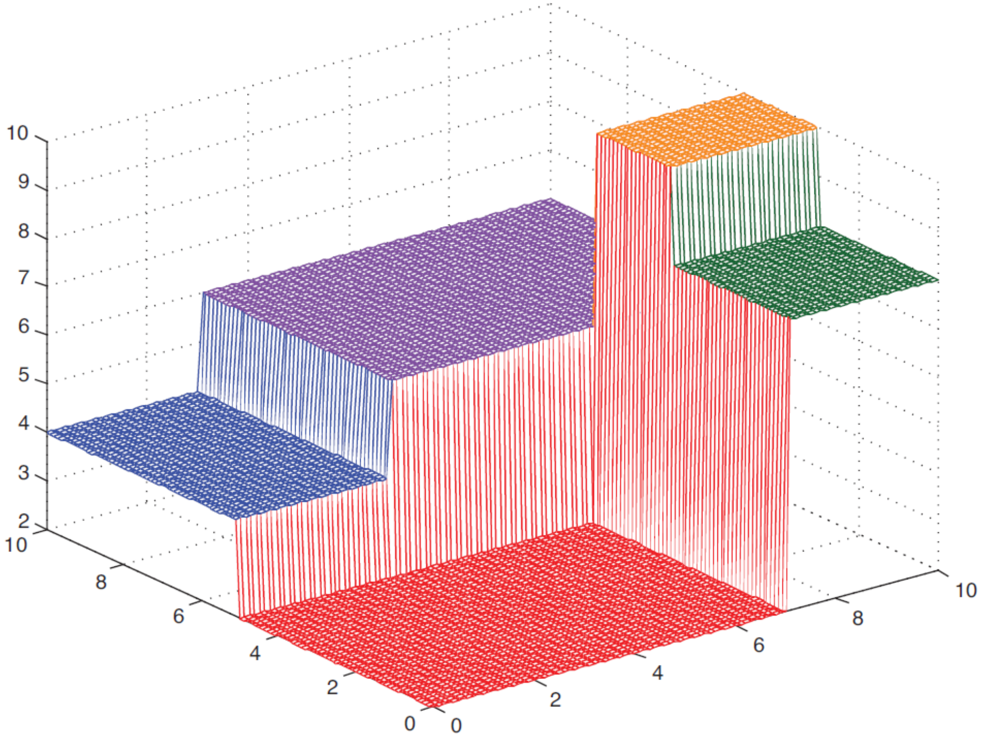
Basics

- Axis parallel split – partition 2d space into 5 regions
- Associate mean response with each of these regions

$$f(\mathbf{x}) = \mathbb{E}\left[y|\mathbf{x}\right] = \sum_{m=1}^{M} w_m \phi(\mathbf{x}; \mathbf{v}_m)$$

where w_m is the **mean response in region *m*** and $\mathbf{v}_m$ encodes the **choice variable** to split on, and the **threshold value** on the path





Basics

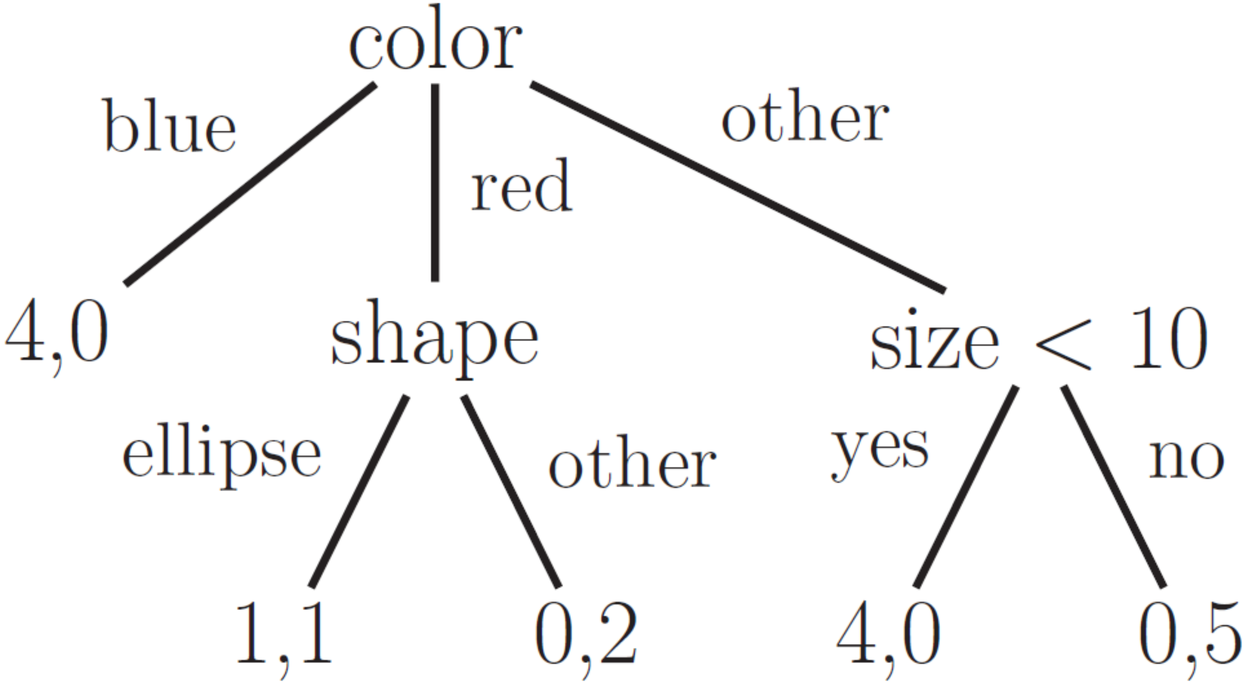
- Generalize to classification – instead of mean response, store the **distribution over class labels in each leaf**
- Example, color->blue, predict

$$p(y = 1|\mathbf{x}) = 4/4$$

- Color-blue, shape-other

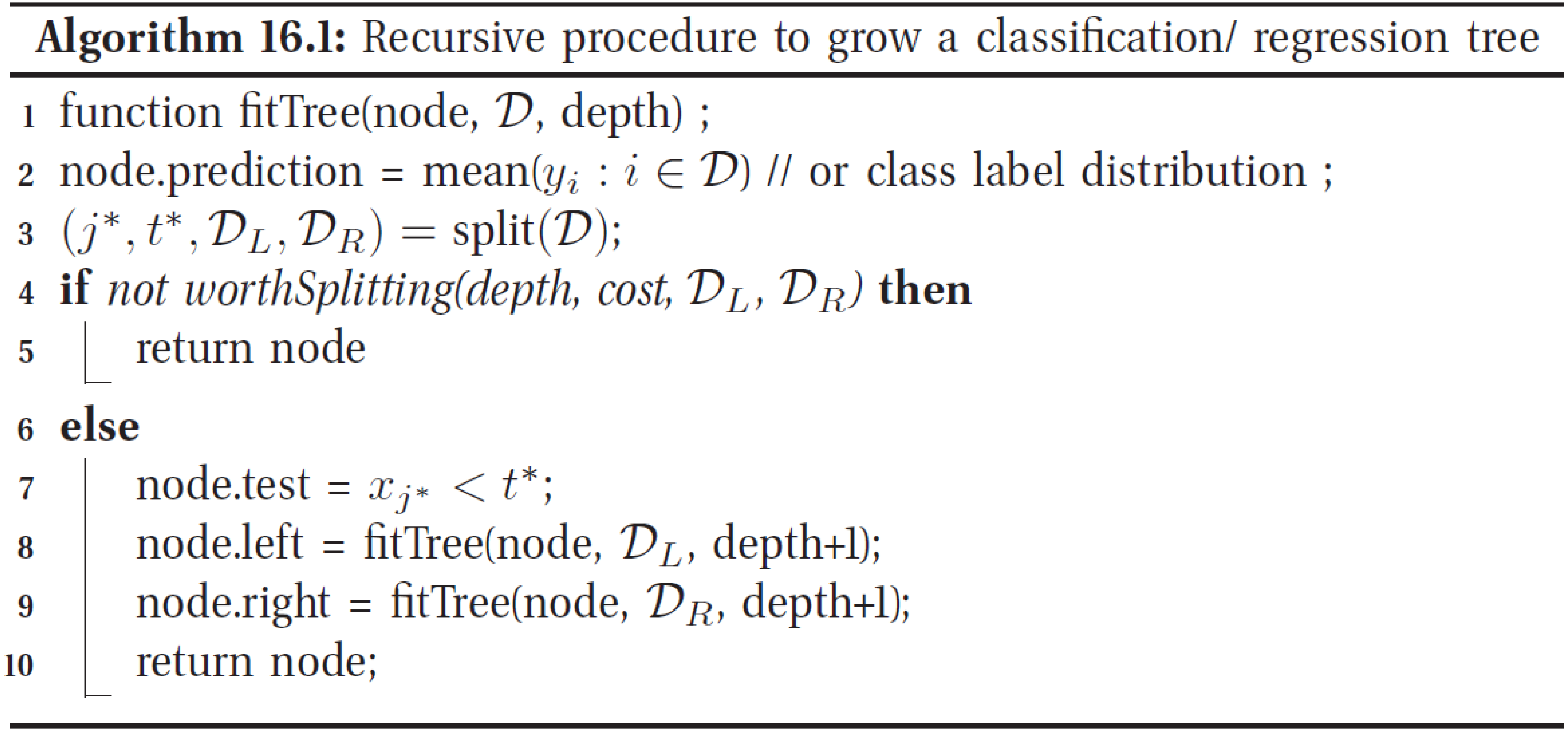
$$p(y = 1|\mathbf{x}) = 0/2$$

- These probabilities – empirical fraction of positive examples that satisfy each conjunction of feature values



Growing a tree

- Greedy procedure to find the optimum partitioning of the data



Algorithm 16.1: Recursive procedure to grow a classification/ regression tree

```
function fitTree(node, $\mathcal{D}$, depth) ;
node.prediction = mean($y_i$ : $i \in \mathcal{D}$) // or class label distribution ;
$(j^*, t^*, \mathcal{D}_L, \mathcal{D}_R) = \text{split}(\mathcal{D})$;
**if** *not worthSplitting(depth, cost, $\mathcal{D}_L$, $\mathcal{D}_R$)* **then**
    return node
**else**
    node.test = $x_{j^*} < t^*$;
    node.left = fitTree(node, $\mathcal{D}_L$, depth+1);
    node.right = fitTree(node, $\mathcal{D}_R$, depth+1);
    return node;
```

Growing a tree

- The split function – chooses best feature and best value for that feature

$$(j^*, t^*) = \arg \min_{j \in \{1,\ldots,D\}} \min_{t \in \mathcal{T}_j} \text{cost}(\{\mathbf{x}_i, y_i : x_{ij} \leq t\}) + \text{cost}(\{\mathbf{x}_i, y_i : x_{ij} > t\})$$

- The set of possible thresholds $\mathcal{T}_j$ for feature *j* is obtained by sorting the unique values of x_{ij} (e.g. feature 1 has values {4.5, -12, 72, -12}, then $\mathcal{T}_j$= {-12,4.5.72})

- If inputs are categorical – the splits are of the form $x_{ij} = c_k$ and $x_{ij} \neq c_k$ for each possible class label c_k

Growing a tree

- Stopping heuristics
  - Is the reduction cost too small (normalized measure of the reduction cost):

$$\Delta \triangleq \text{cost}(\mathcal{D}) - \left(\frac{|\mathcal{D}_L|}{|\mathcal{D}|} \text{cost}(\mathcal{D}_L) + \frac{|\mathcal{D}_R|}{|\mathcal{D}|} \text{cost}(\mathcal{D}_R) \right)$$

  - Has the tree exceeded the maximum desired depth
  - Is the distribution of the response in either $\mathcal{D}_L$ or $\mathcal{D}_R$ sufficiently homogeneous (e.g. all labels are the same, so the distribution is pure)
  - Is the number of examples in either $\mathcal{D}_L$ or $\mathcal{D}_R$ too small

Growing a tree - costs

- Regression cost - $\mathrm{cost}(\mathcal{D}) = \sum_{i \in \mathcal{D}} (y_i - \overline{y})^2$

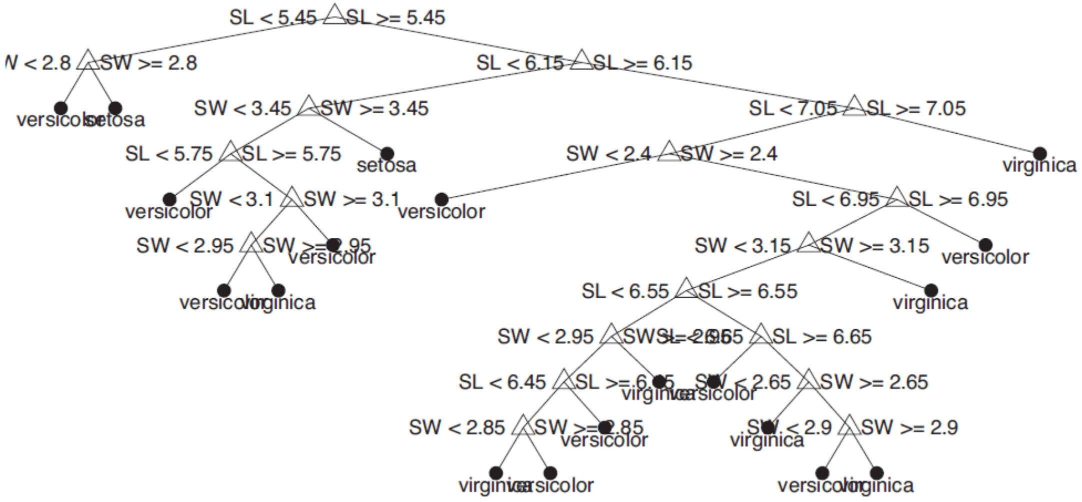
  where $\overline{y} = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} y_i$ is the mean of the response variable in the specified set of data

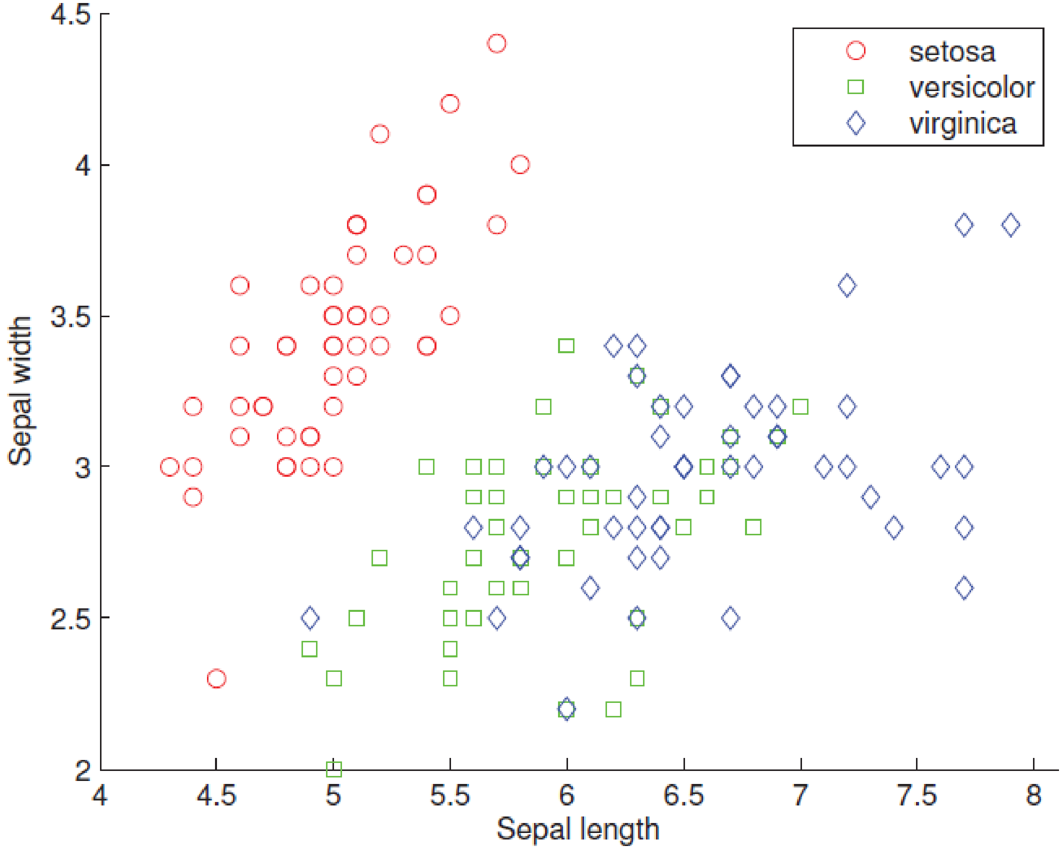
Growing a tree - costs

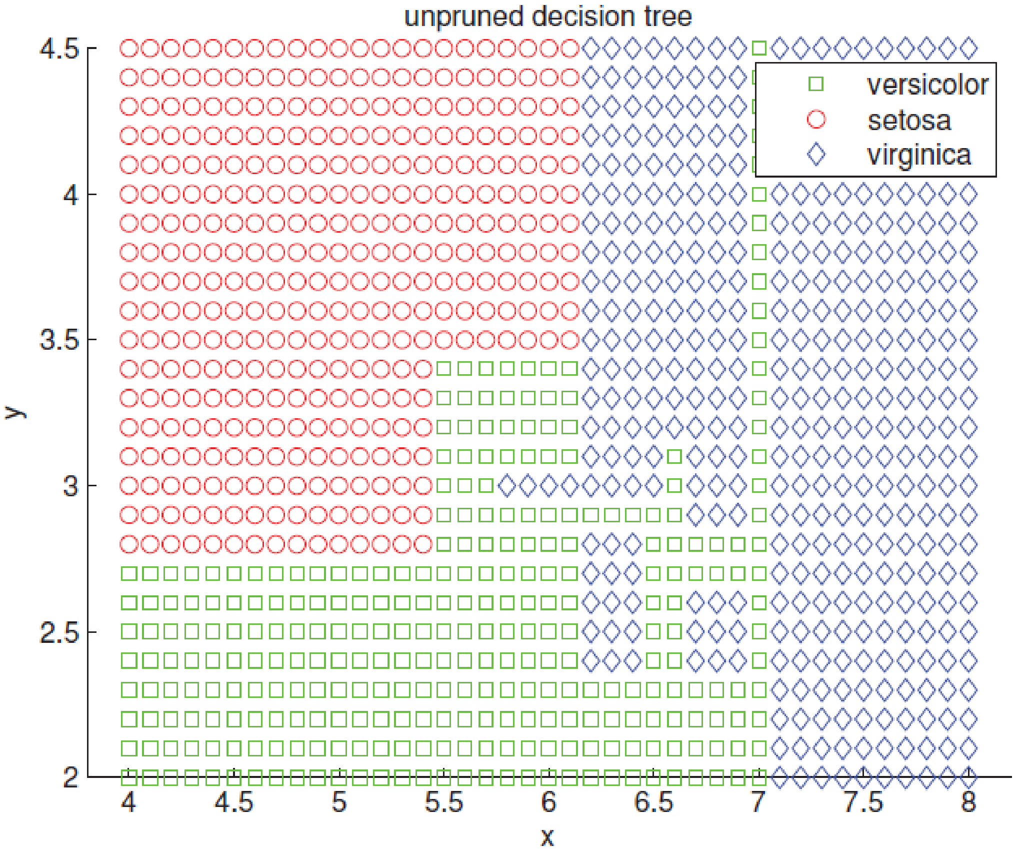
- Classification costs
  - Probability a random entry in the leaf belongs to class c $$\hat{\pi}_c = \frac{1}{|\mathcal{D}|}\sum_{i\in\mathcal{D}} \mathbb{I}(y_i = c)$$
  - **Misclassification rate** $$\frac{1}{|\mathcal{D}|}\sum_{i\in\mathcal{D}} \mathbb{I}(y_i \neq \hat{y}) = 1 - \hat{\pi}_{\hat{y}}$$
    - Most probable class label $\hat{y}_c = \operatorname{argmax}_c \hat{\pi}_c$
  - **Entropy –** amount of information $$\mathbb{H}\left(\hat{\boldsymbol{\pi}}\right) = -\sum_{c=1}^{C} \hat{\pi}_c \log \hat{\pi}_c$$
  - **Gini index –** expected error rate $$\sum_{c=1}^{C} \hat{\pi}_c(1-\hat{\pi}_c) = \sum_c \hat{\pi}_c - \sum_c \hat{\pi}_c^2 = 1 - \sum_c \hat{\pi}_c^2$$

Example

- Iris dataset (using 2 features)





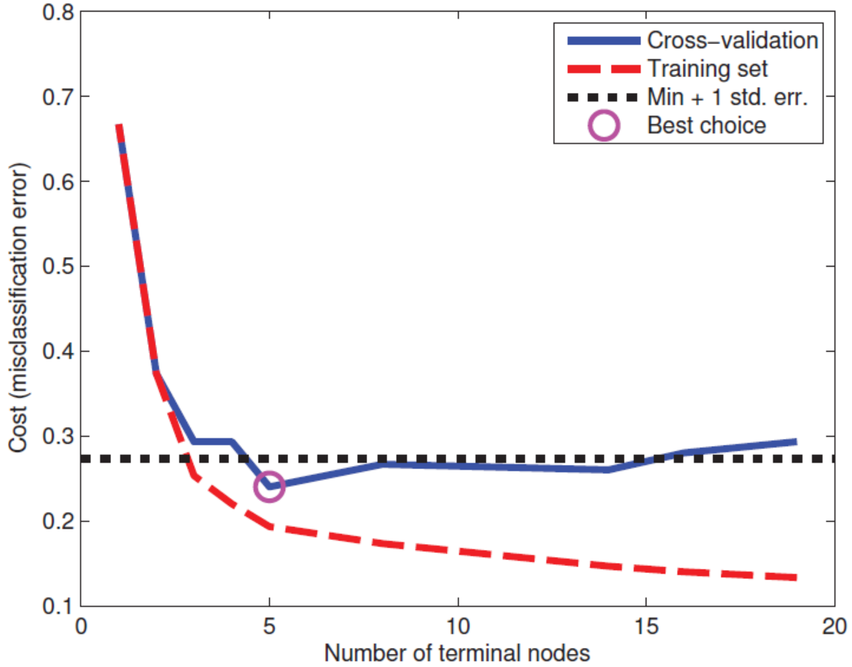


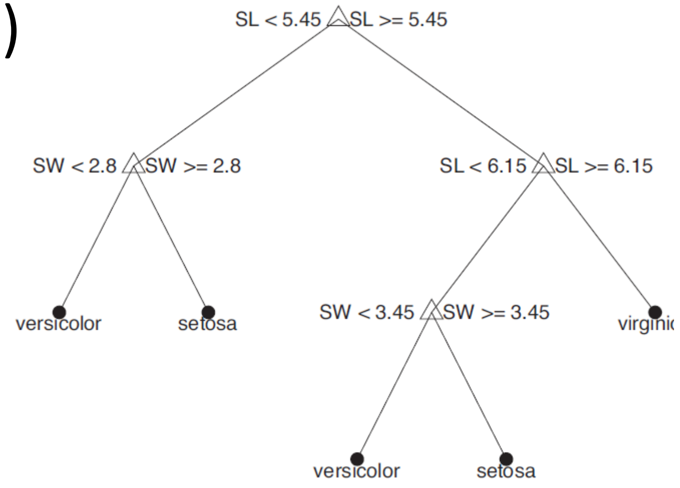
Pruning a tree

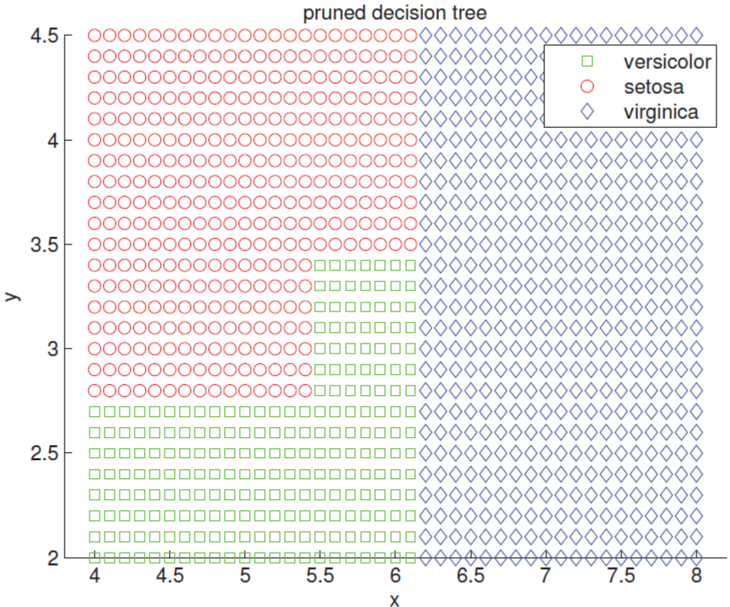
- Standard approach – grow a “full” tree and then perform **pruning** to avoid overfitting
  - Prune the branches giving the least increase in the error
  - How far to prune back? - evaluate the cross-validated error on each such subtree, and then pick the tree whose CV error is within one standard error of the minimum (heuristic based on sence)
  - Choose the simplest model

    whose accuracy is comparable

    with the best model







Pros and cons of trees

- Pros
  - Easy to interpret
  - Easily handle mixed discrete and continuous inputs
  - Perform variable selection
  - Scale well to large data sets
- Cons
  - Do not predict very accurately compared to other models
  - The trees are **unstable –** small changes in the input data can have large effect on the structure of the tree, (errors at the top affect the rest of the tree)
  - The trees are high variance estimators

Bootstrap Aggregation / Bagging

- Individual models (e.g. decision trees) may have high variance along with low bias

- Construct M different subset of data, chosen randomly with displacement
- Train separate copy of predictive model on each
- Average prediction over copies

$$f(x) = \frac{1}{M}\sum_{i} f_m(x)$$

- This technique is called **bagging** or **bootstrap aggregating**
- If the errors are uncorrelated, then bagged error reduces linearly with M

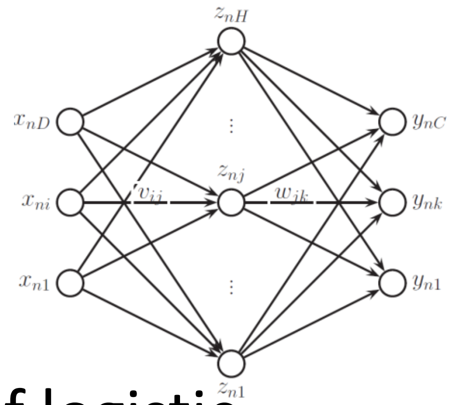
Random Forests

- Training same algorithm on bootstraps creates correlated errors

- Randomly choose (a) subset of variables and (b) subset of training data

- Good predictive accuracy
- Loss in interpretability
- Widely used in many applications

Feedforward neural networks



- Feedforward neural network or multi-layer perceptron is a series of logistic regression models stacked on top of each other
- The **final layer** is either another logistic regression or a linear regression models, depending on whether we are solving a classification or regression problem
- For example, if we have two layers, and have a regression problem, the models has the form

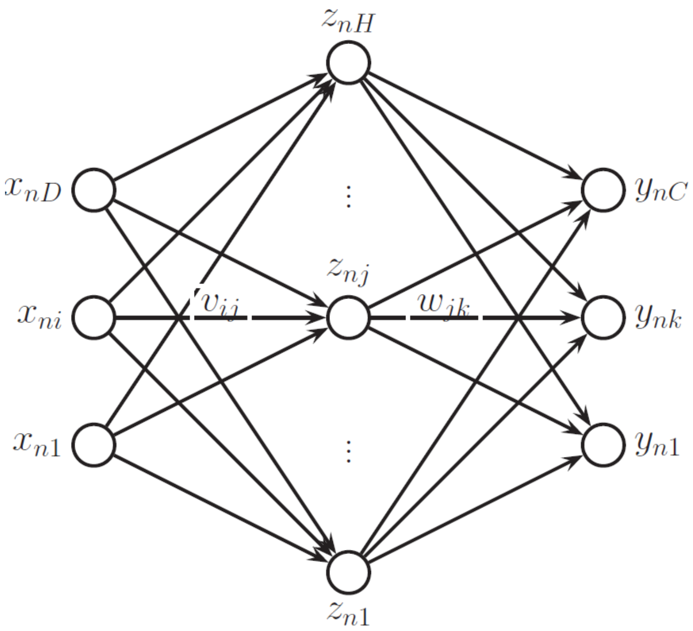
$$
\begin{aligned}
p(y|\mathbf{x}, \boldsymbol{\theta}) &= \mathcal{N}(y|\mathbf{w}^T \mathbf{z}(\mathbf{x}), \sigma^2) \\
\mathbf{z}(\mathbf{x}) &= g(\mathbf{V}\mathbf{x}) = [g(\mathbf{v}_1^T \mathbf{x}), \ldots, g(\mathbf{v}_H^T \mathbf{x})]
\end{aligned}
$$

where *g* is a non-linear **activation** or **transfer** function (usually logistic), $z(x)$ is hidden layer (deterministic f-on on the inputs), *H* is number of hidden units, **V** – weight matrix from the inputs to the hidden nodes, and **w** is weight vector from the hidden nodes to the output

Feedforward neural networks

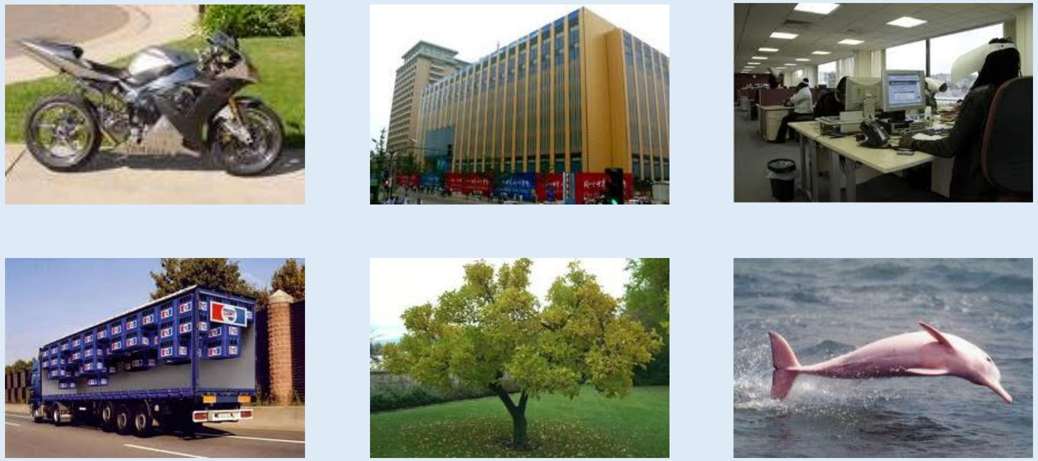
- It is important that *g* be non-linear, otherwise the whole model collapses into a large linear regression model of the form $y = \mathbf{w}^T(\mathbf{V}\mathbf{x})$
- It can be shown that MLP is **universal approximator** meaning it can model any suitably smooth function, given enough hidden units, to any desired accuracy
- To handle binary classification, we pass the output through a sigmoid:

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathrm{Ber}(y|\mathrm{sigm}(\mathbf{w}^T\mathbf{z}(\mathbf{x})))$$



Компјутерска визија: Детекција на автомобил

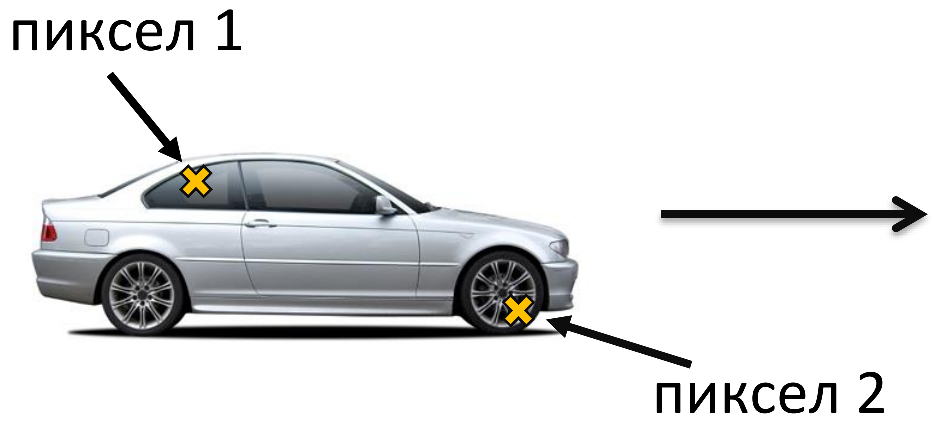
Автомобил



Не е автомобил

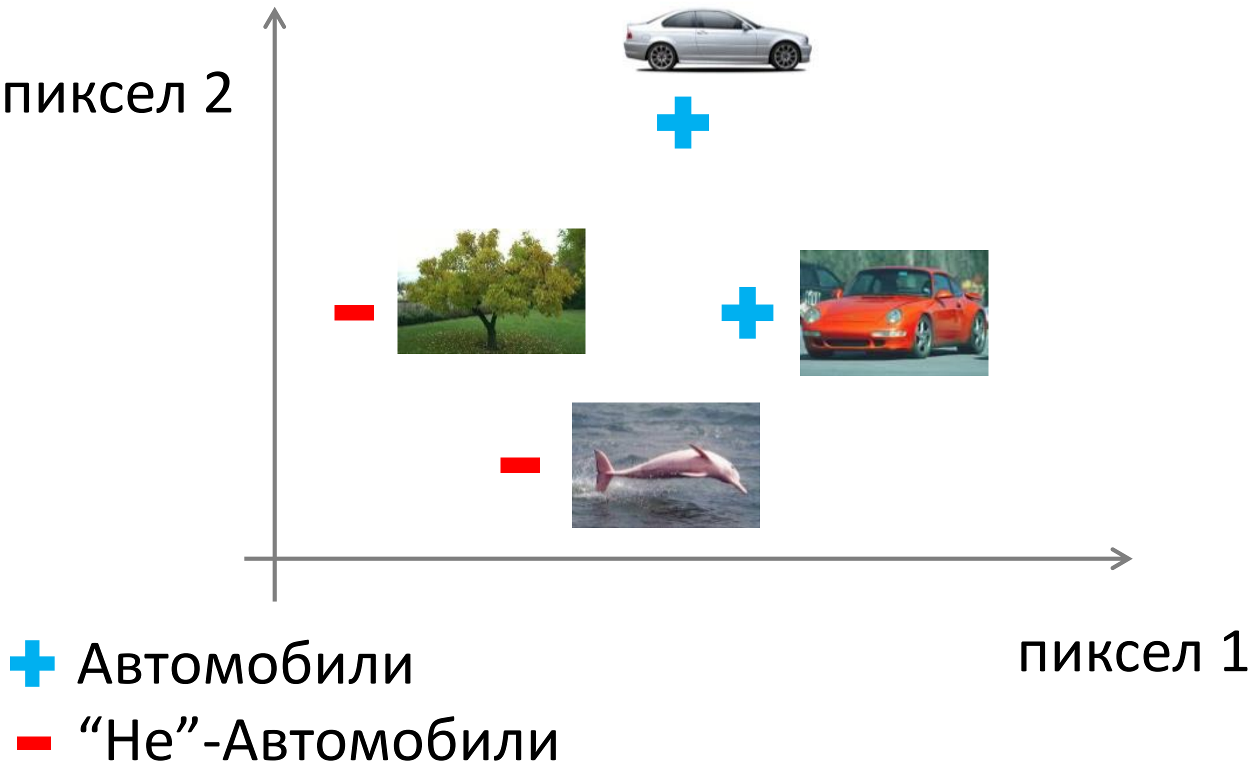
Тестирање:

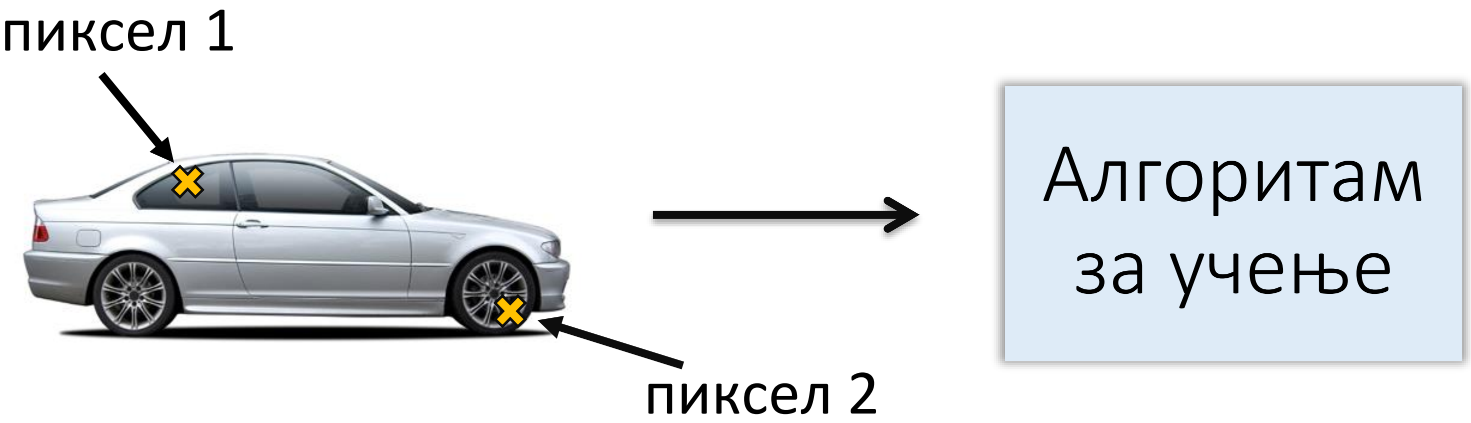
Што е ова?



→

Алгоритам за учење

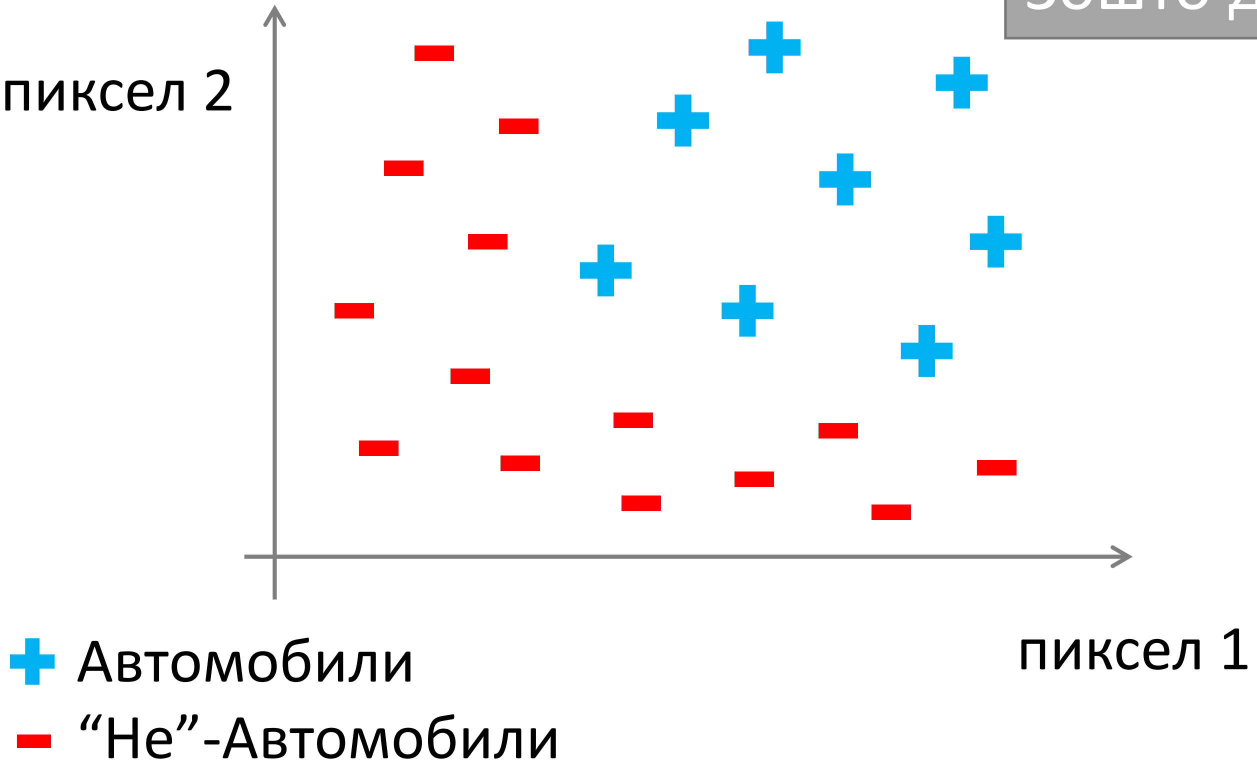




Алгоритам за учење

Зошто да не примениме логистичка регресија?

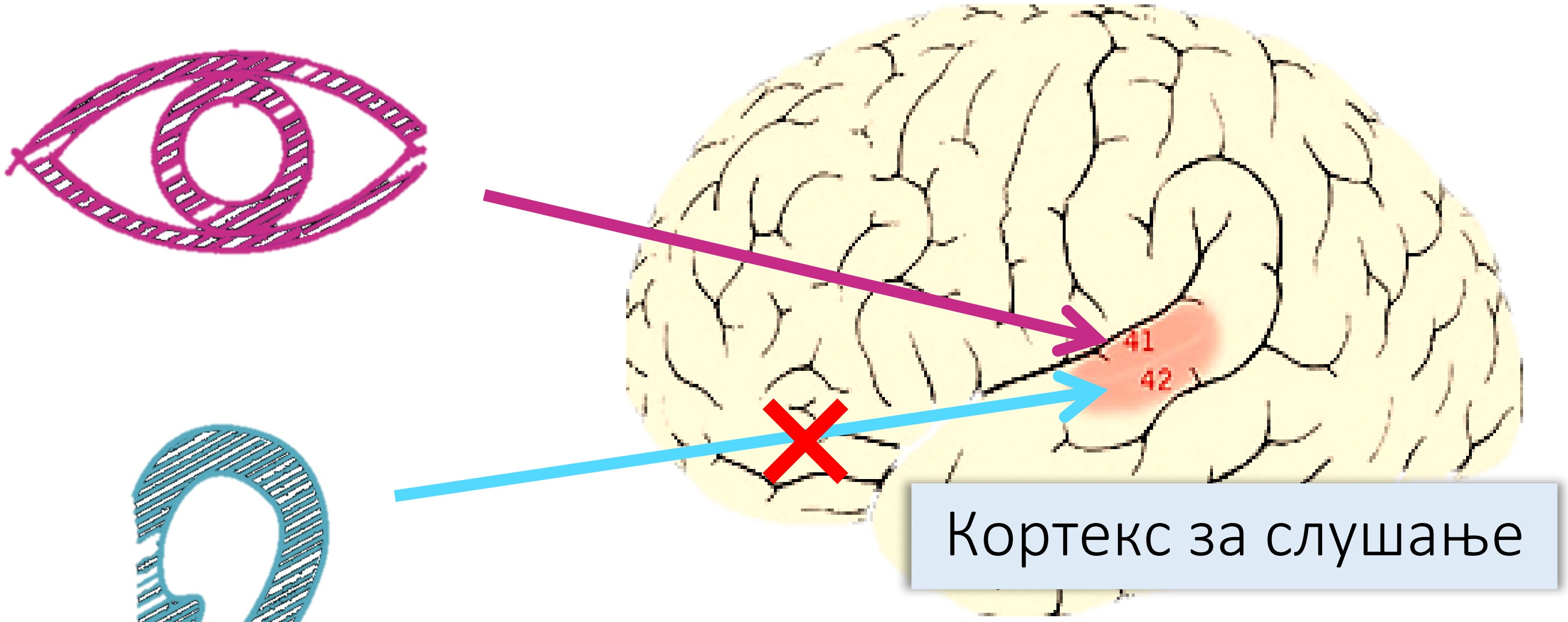
слика со 50 x 50 пиксели→ 2500 пиксели
(7500 ако е RGB)



$$x = \begin{bmatrix} \text{пиксел 1 интензитет} \\ \text{пиксел 2 интензитет} \\ \vdots \\ \text{пиксел 2500 интензитет} \end{bmatrix}$$

Квадратни карактеристики($x_i \times x_j$): ≈
3 милиони карактеристики

Хипотезата на “единствен алгоритам за учење”

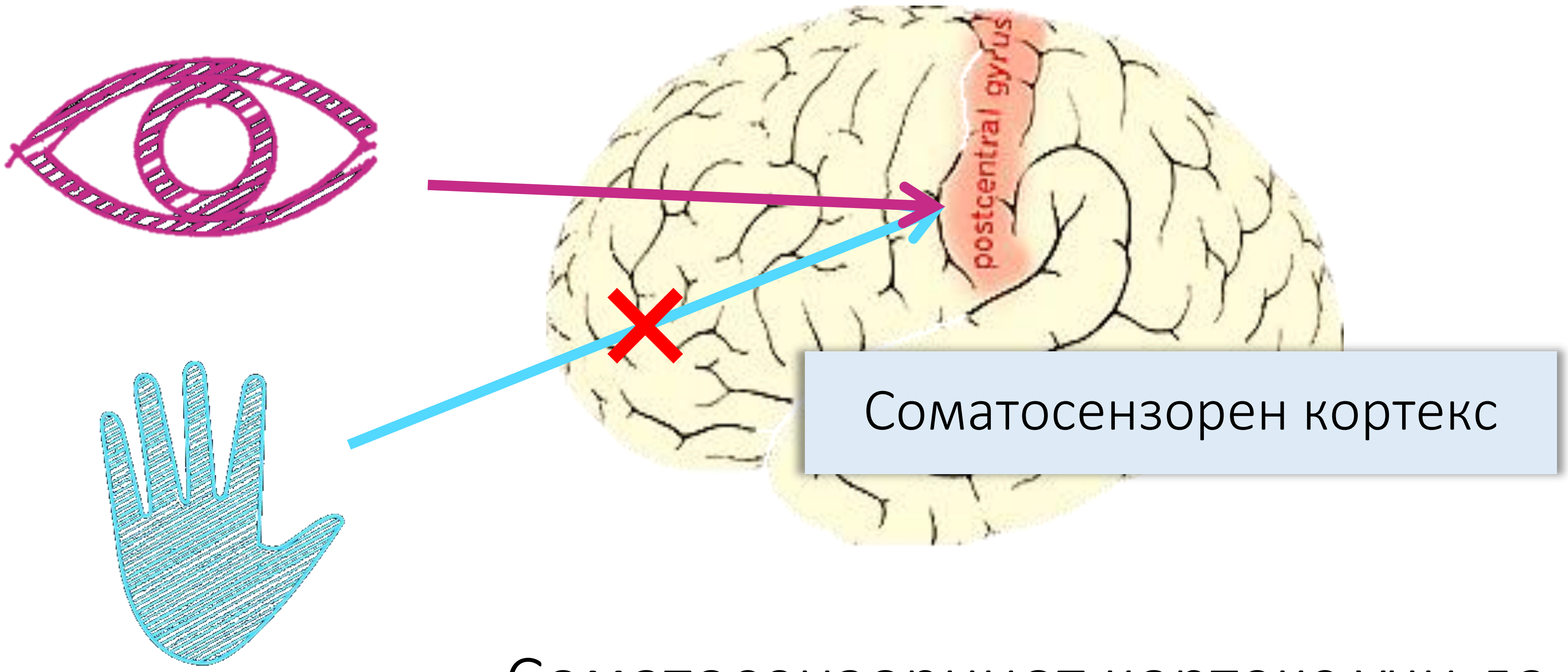


Кортекс за слушање

Кортексот за слушање учи да гледа

[Roe et al., 1992]

Хипотезата на “единствен алгоритам за учење”

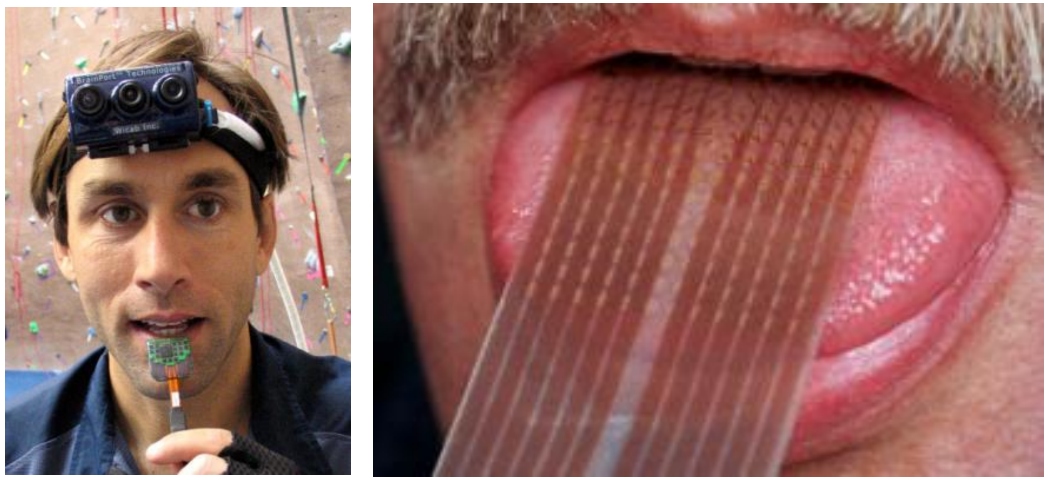


Соматосензорен кортекс

Соматосензорниот кортекс учи да гледа

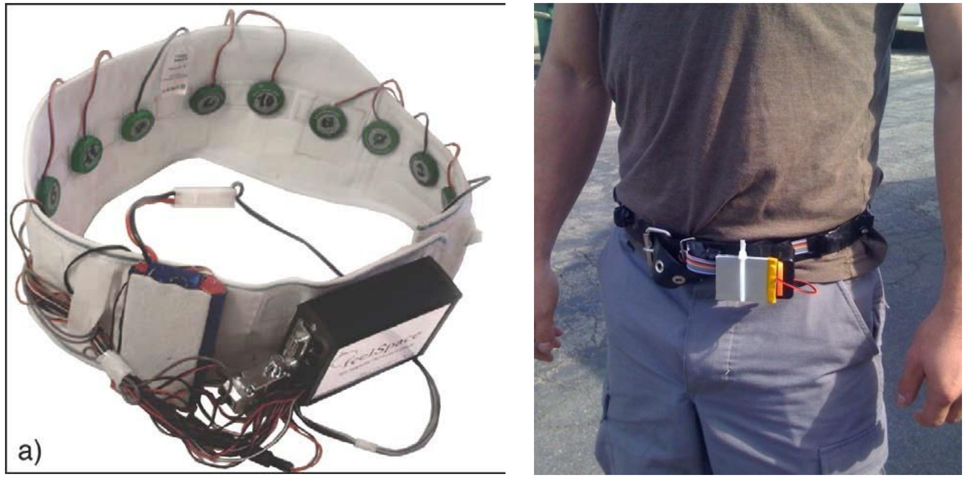
[Metin & Frost, 1989]

Сензорски репрезентации во мозокот

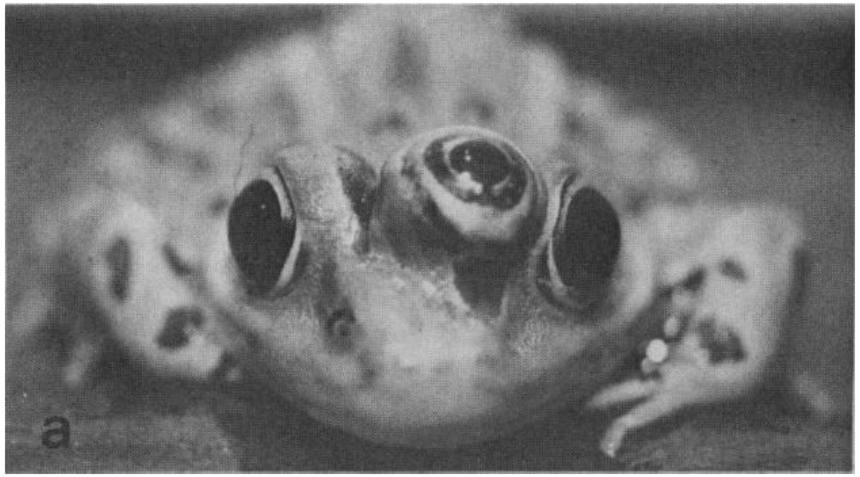


Гледање со јазикот

Човечка ехолокација (сонар)



Тактилен појас: осет за насока

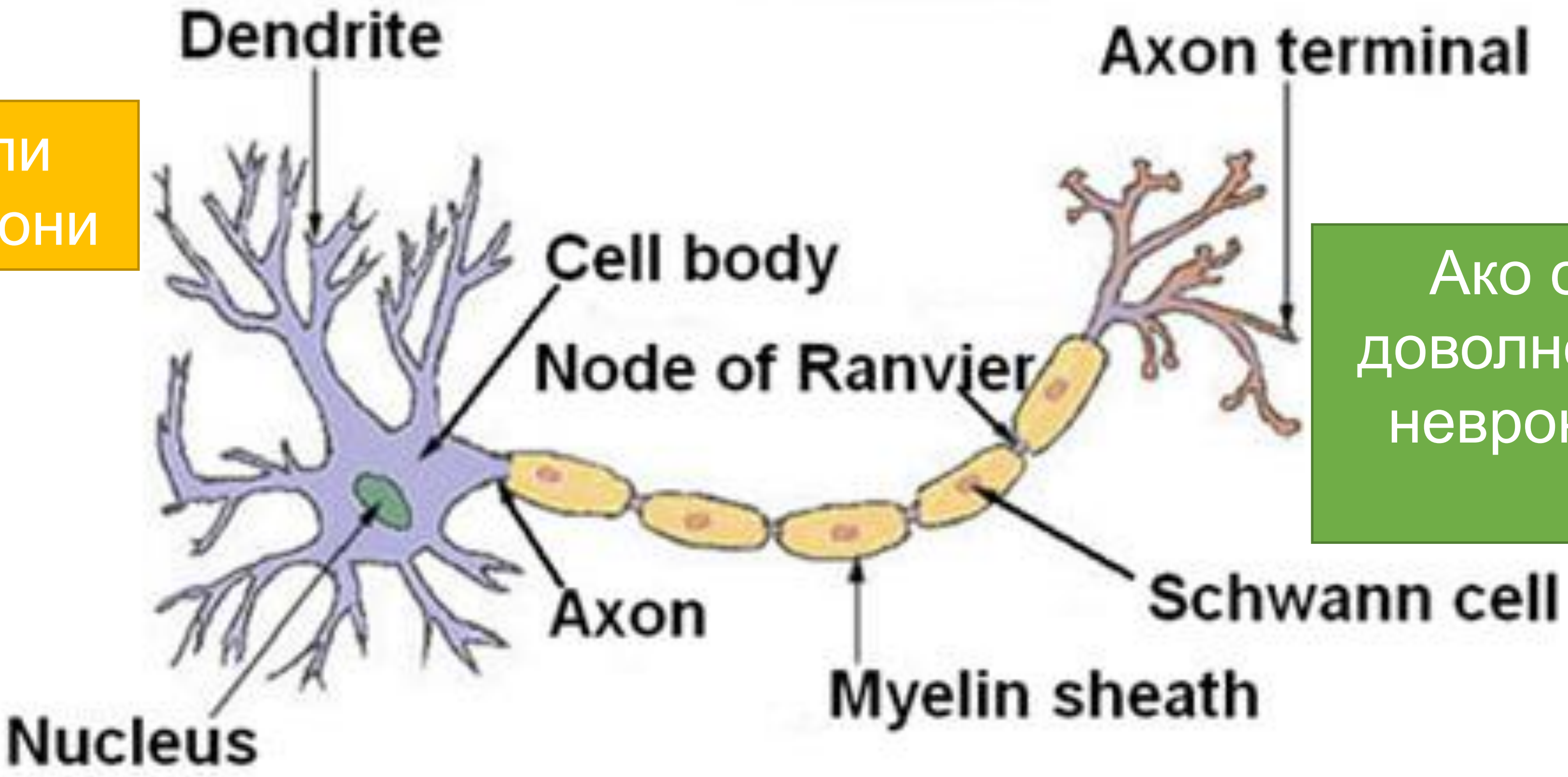


Имплантирање на трето око

[BrainPort; Welsh & Blasch, 1997; Nagel et al., 2005; Constantine-Paton & Law, 2009]

Невроните во мозокот

Влезни сигнали од други неврони

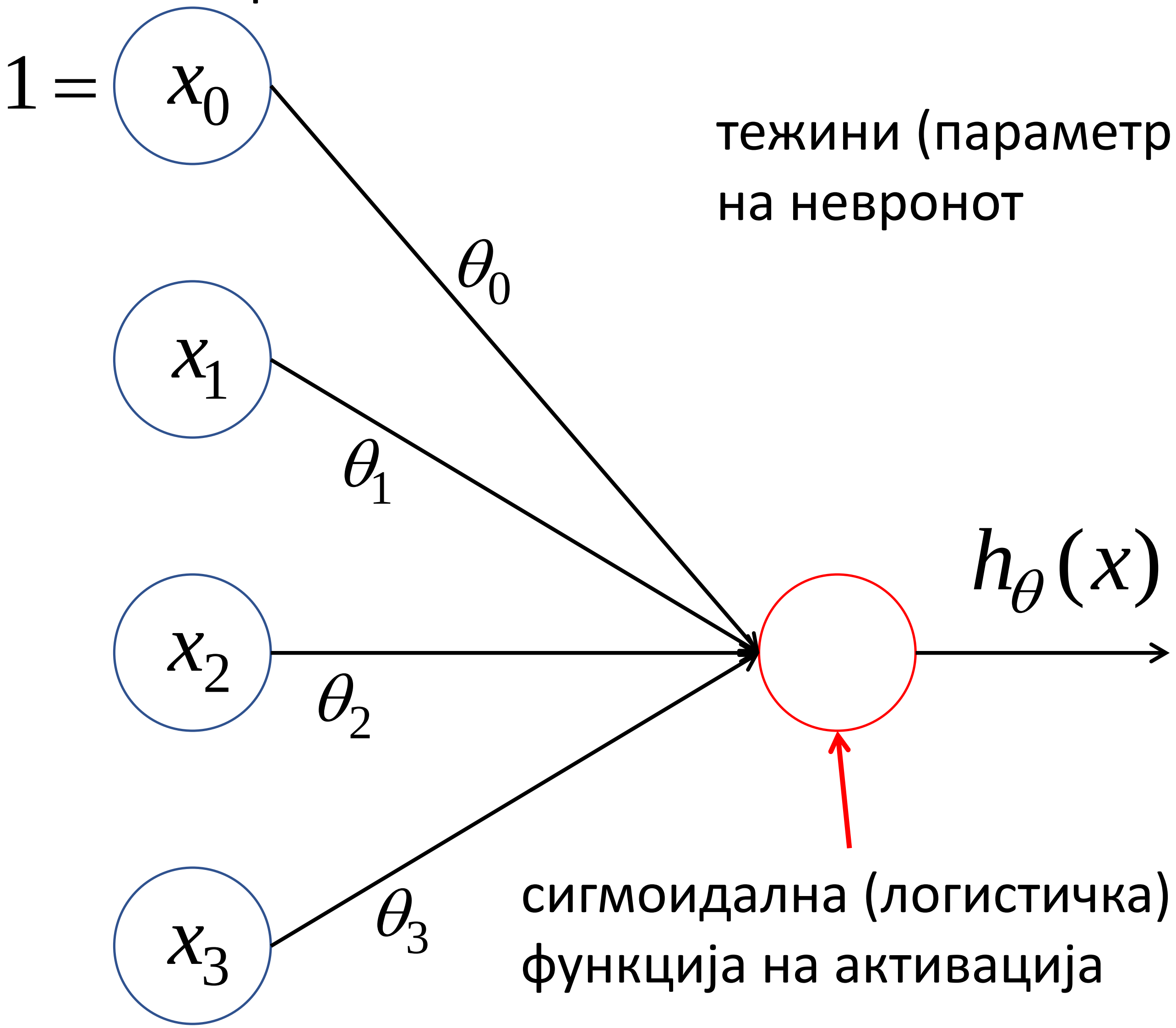


Ако се акумулира доволно голем сигнал, невронот „испалува“ сигнал

Јачината на врските определува како се акумулираат сигналите

Модел на неврон (логистичка единица)

bias неврон



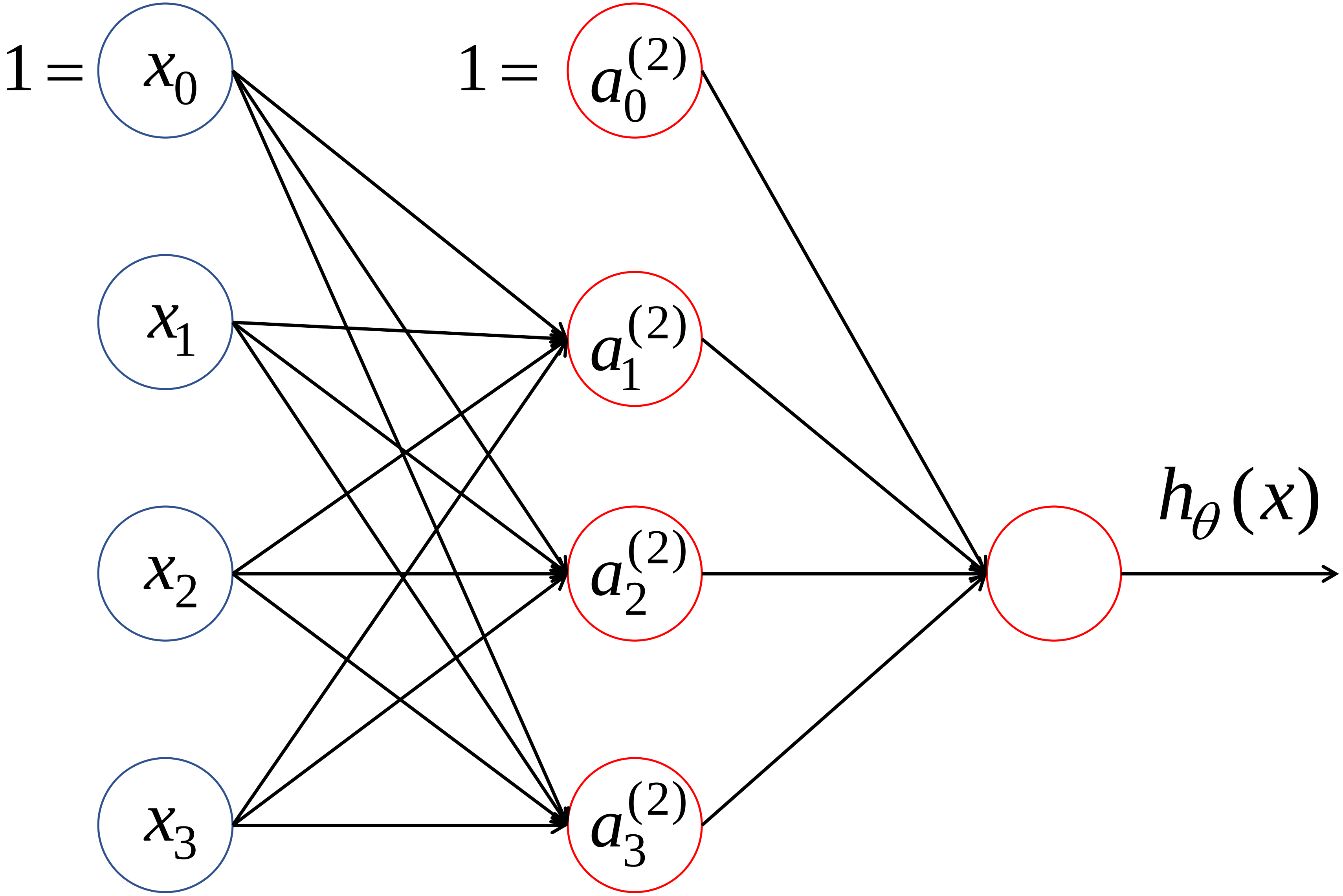
сигмоидална (логистичка) функција на активација

тежини (параметри) на невронот

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix} \qquad x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$$

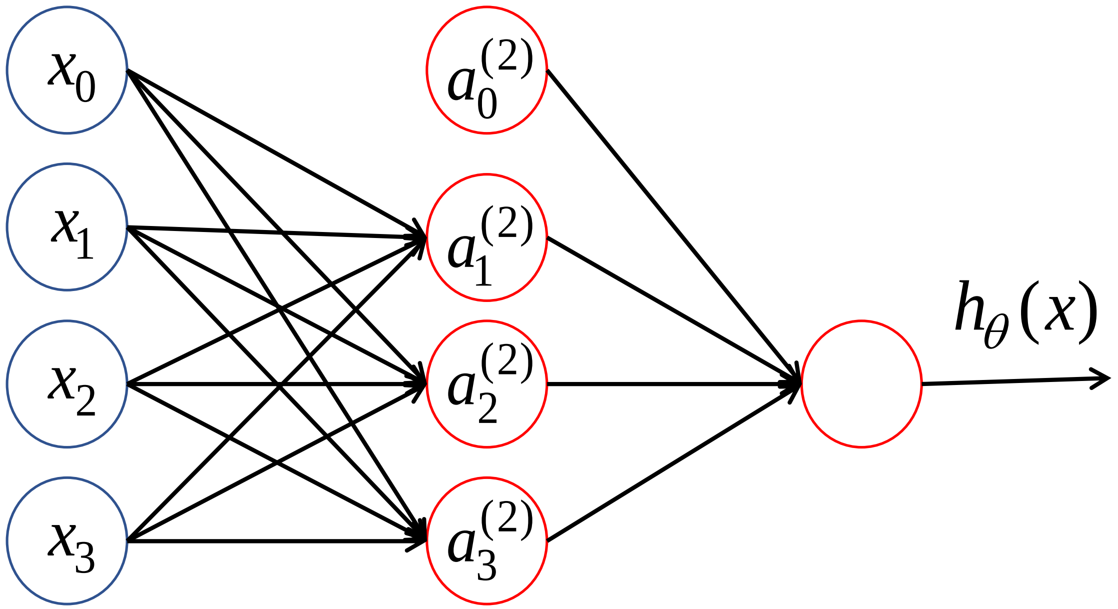
Модел на невронска мрежа



слој 1 – влезен слој

слој 2 – скриен слој

слој 3 – излезен слој



$a_i^{(j)}$ - активација на невронот i во слојот j

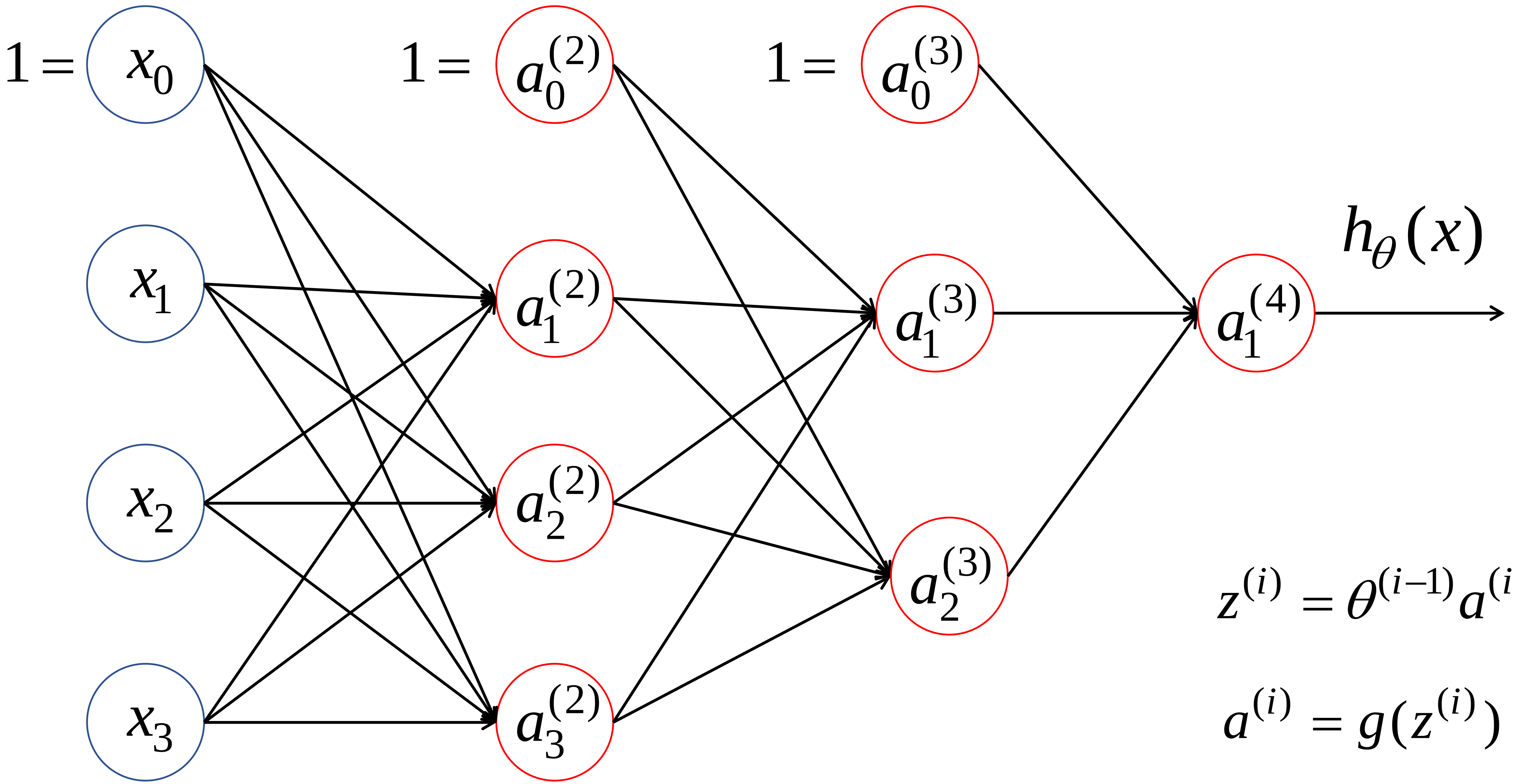
$\theta^{(k)}$ - матрица на тежини за врските помеѓу слојот k и слојот $(k+1)$

$$a_1^{(2)} = g(\theta_{10}^{(1)} x_0 + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2 + \theta_{13}^{(1)} x_3)$$

$$a_2^{(2)} = g(\theta_{20}^{(1)} x_0 + \theta_{21}^{(1)} x_1 + \theta_{22}^{(1)} x_2 + \theta_{23}^{(1)} x_3)$$

$$a_3^{(2)} = g(\theta_{30}^{(1)} x_0 + \theta_{31}^{(1)} x_1 + \theta_{32}^{(1)} x_2 + \theta_{33}^{(1)} x_3)$$

$$h_\theta(x) = a_1^{(3)} = g(\theta_{10}^{(2)} a_0^{(2)} + \theta_{11}^{(2)} a_1^{(2)} + \theta_{12}^{(2)} a_2^{(2)} + \theta_{13}^{(2)} a_3^{(2)})$$



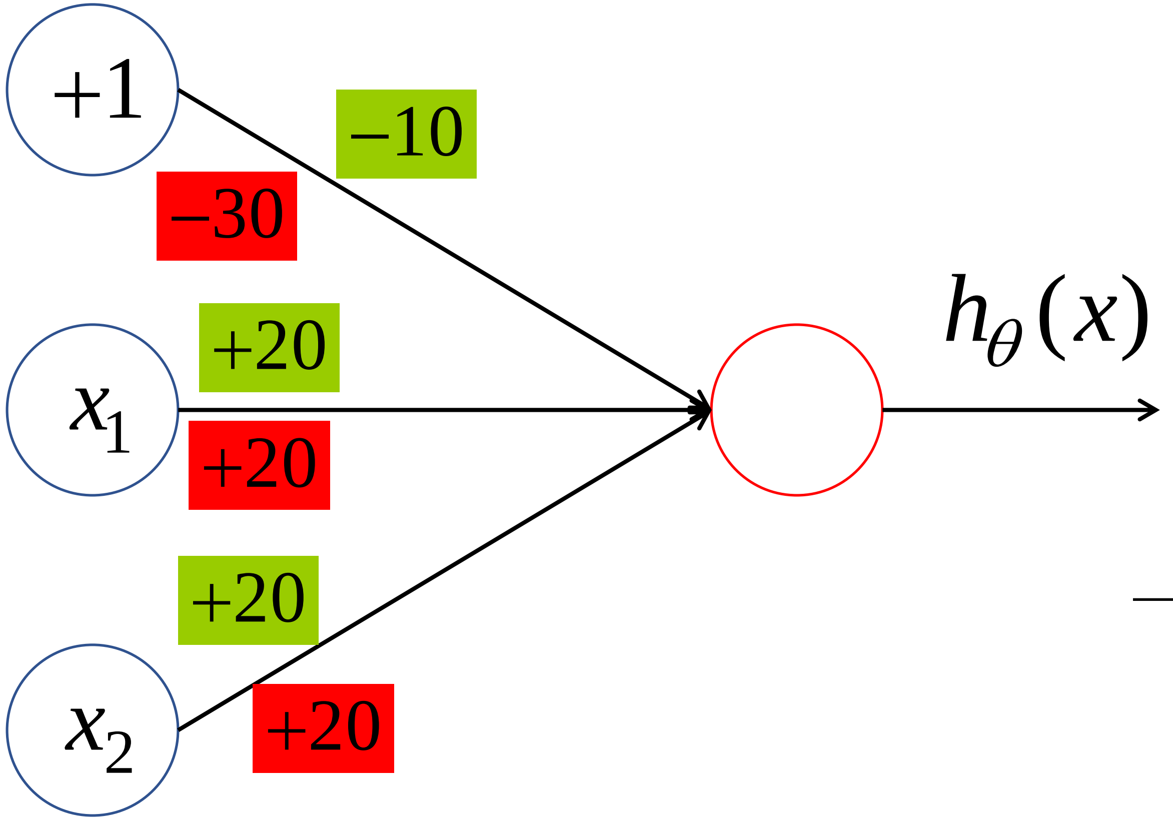
$$z^{(i)} = \theta^{(i-1)} a^{(i-1)}$$

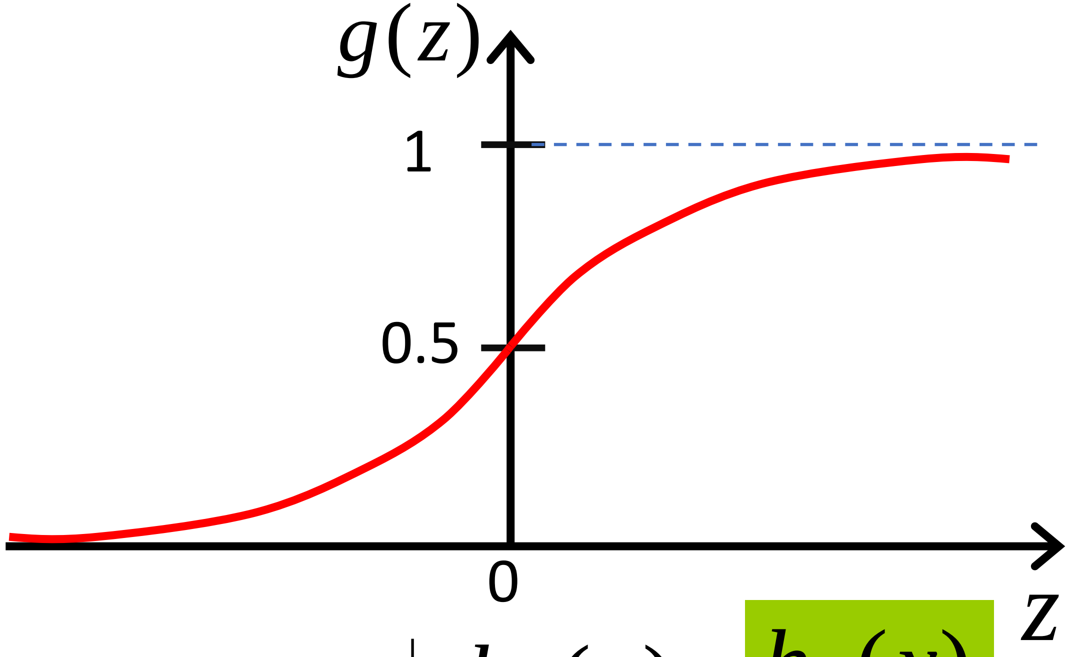
$$a^{(i)} = g(z^{(i)})$$

Едноставни примери на невронски мрежи: AND, OR

$y = x_1 \; AND \; x_2$

$y = x_1 \; OR \; x_2$

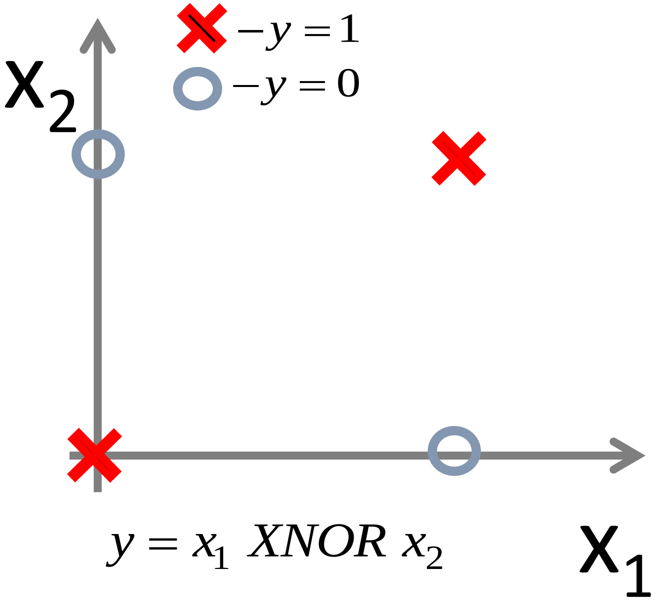




x_1	x_2	$h_\theta(x)$	$h_\theta(x)$
0	0	$g(-30)\approx 0$	$g(-10)\approx 0$
0	1	$g(-10)\approx 0$	$g(10)\approx 1$
1	0	$g(-10)\approx 0$	$g(10)\approx 1$
1	1	$g(10)\approx 1$	$g(30)\approx 1$

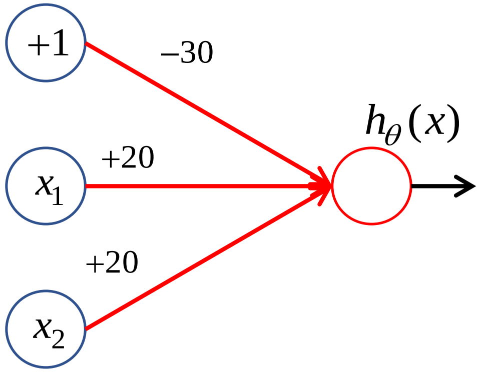
$$h_\theta(x) = g(-30 + 20x_1 + 20x_2)$$

Пример на нелинеарна класификација: XOR/XNOR

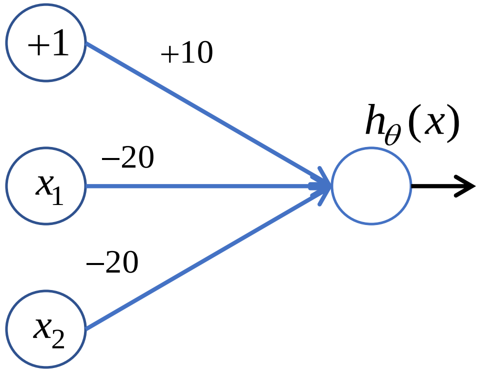


$y = x_1\ XNOR\ x_2$

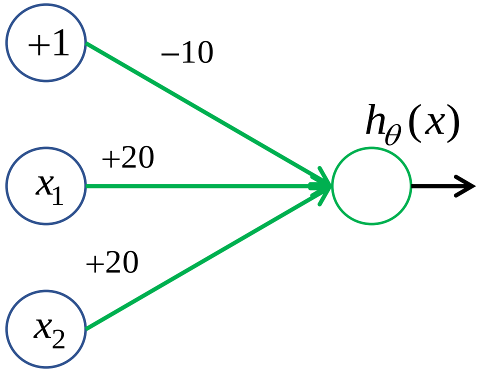
$$x_1\ XNOR\ x_2 = (x_1\ AND\ x_2)\ \ OR\ \ (NOT(x_1)\ AND\ NOT(x_2))$$



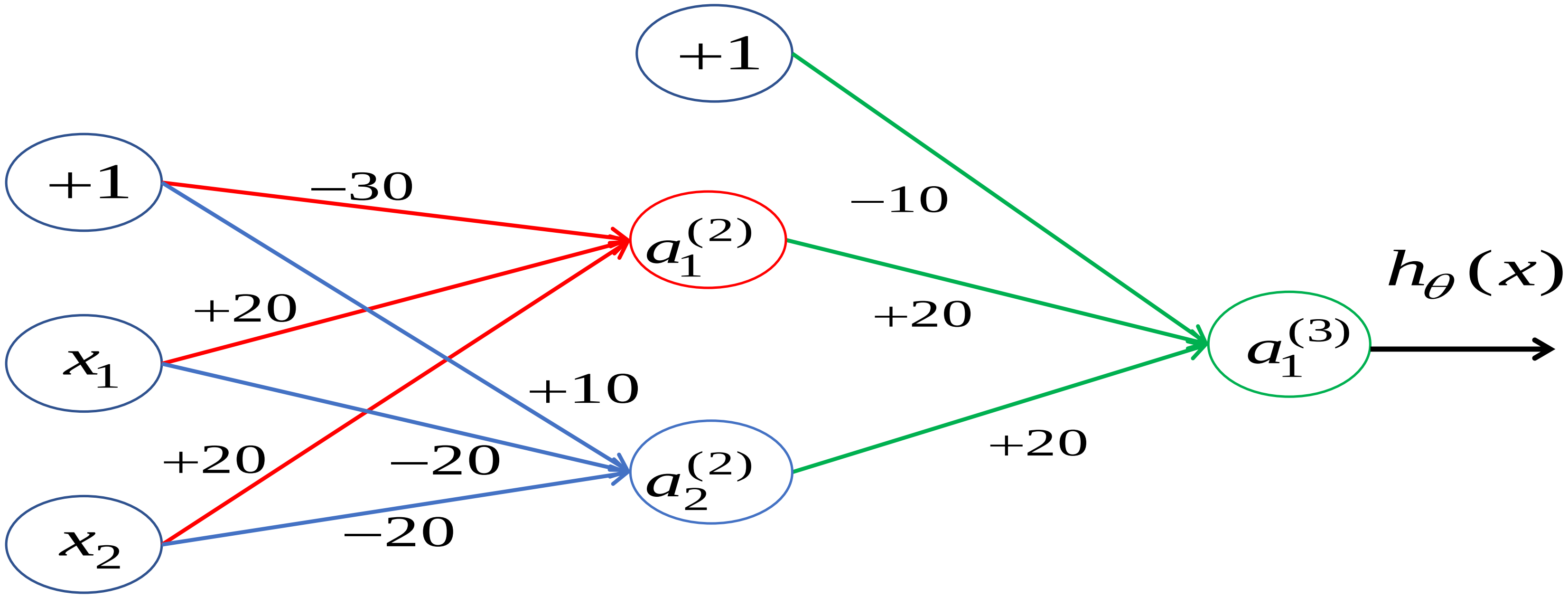
$x_1\ AND\ x_2$



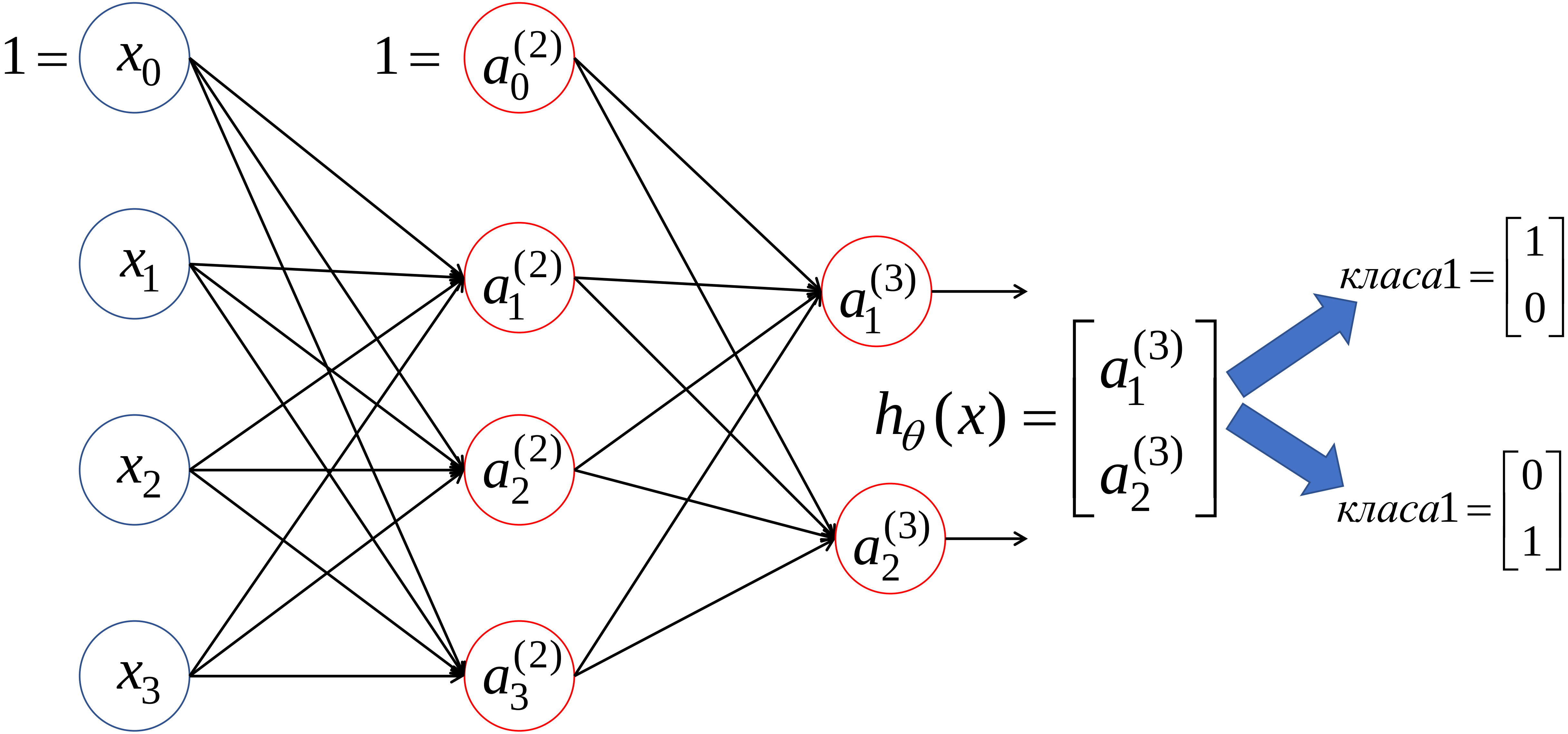
$NOT(x_1)\ AND\ NOT(x_2)$



$x_1\ OR\ x_2$



Повеќекласна класификација



$$h_\theta(x) = \begin{bmatrix} a_1^{(3)} \\ a_2^{(3)} \end{bmatrix}$$

$$\textit{класа}1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\textit{класа}1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Backpropagation algorithm

- The negative log likelihood of an MLP is a **non-convex function** of its parameters
- But we can find a **locally optimal** ML or MAP estimate using standard gradient-based optimization methods
- Since MLPs have lots of parameters, they are often **trained on very large data sets**
- It is common to use **first-order online methods**, such as stochastic gradient descent
- We will learn how to compute the gradient vector of the NLL by applying the chain rule – resulting algorithm is the backpropagation algorithm

Backpropagation algorithm

- For simplicity, we shell assume a model with just **one hidden layer**
- Distinguish the pre- and post-synaptic values of a neuron, that is before and after we apply the nonlinearity
- Input-to-hidden layer
  - $\mathbf{x}_n$ - *n*'th input data
  - $\mathbf{a}_n = \mathbf{V}\mathbf{x}_n$ - pre-synaptic hidden layer
  - $\mathbf{z}_n = g(\mathbf{a}_n)$ – post-synaptic hidden layer (*g*-transfer function)
- Hidden-to-output layer
  - $\mathbf{b}_n = \mathbf{W}\mathbf{z}_n$ - pre-synaptic output layer
  - $\hat{\mathbf{y}}_n = h(\mathbf{b}_n)$ – post-synaptic output layer

$$\mathbf{x}_n \overset{\mathbf{V}}{\to} \mathbf{a}_n \overset{g}{\to} \mathbf{z}_n \overset{\mathbf{W}}{\to} \mathbf{b}_n \overset{h}{\to} \hat{\mathbf{y}}_n$$

Backpropagation algorithm

- The parameters of the model are $\theta = (\mathbf{V}, \mathbf{W})$, first and second layer weight matrices
- The bias terms are accommodated by adding an element of $\mathbf{x}_n$ and $\mathbf{z}_n$ to 1
- In the regression case, with *K* outputs, the NLL is given by squared error:

$$J(\boldsymbol{\theta}) = -\sum_n \sum_k (\hat{y}_{nk}(\boldsymbol{\theta}) - y_{nk})^2$$

- In the classification case, with *K* classes, the NLL is given by the cross entropy:

$$J(\boldsymbol{\theta}) = -\sum_n \sum_k y_{nk} \log \hat{y}_{nk}(\boldsymbol{\theta})$$

$$\mathbf{x}_n \xrightarrow{\mathbf{V}} \mathbf{a}_n \xrightarrow{g} \mathbf{z}_n \xrightarrow{\mathbf{W}} \mathbf{b}_n \xrightarrow{h} \hat{\mathbf{y}}_n$$

Backpropagation algorithm

- Our task is to compute $\nabla_\theta J$.
- We will derive this for each *n* separately; the overall gradient is obtained by summing over *n*
- Start by considering the output layer weights:

$$\nabla_{\mathbf{w}_k} J_n = \frac{\partial J_n}{\partial b_{nk}} \nabla_{\mathbf{w}_k} b_{nk} = \frac{\partial J_n}{\partial b_{nk}} \mathbf{z}_n = \delta^w_{nk}\, \mathbf{z}_n$$

since $b_{nk} = \mathbf{w}_k^T \mathbf{z}_n$ and $\delta^w_{nk} = (\hat{y}_{nk} - y_{nk})$ is the error signal

- So the overall gradient is: $\nabla_{\mathbf{w}_k} J_n = \delta^w_{nk} \mathbf{z}_n$

$$\mathbf{x}_n \xrightarrow{\mathbf{V}} \mathbf{a}_n \xrightarrow{g} \mathbf{z}_n \xrightarrow{\mathbf{W}} \mathbf{b}_n \xrightarrow{h} \hat{\mathbf{y}}_n$$

Backpropagation algorithm

- For the input layer weights, we have $\nabla_{\mathbf{v}_j} J_n = \frac{\partial J_n}{\partial a_{nj}} \nabla_{\mathbf{v}_j} a_{nj} \triangleq \delta^v_{nj} \mathbf{x}_n$

  where $a_{nj} = \mathbf{v}_j^T \mathbf{x}_n$

- For the first level error signal we have:

$$\delta^v_{nj} = \frac{\partial J_n}{\partial a_{nj}} = \sum_{k=1}^{K} \frac{\partial J_n}{\partial b_{nk}} \frac{\partial b_{nk}}{\partial a_{nj}} = \sum_{k=1}^{K} \delta^w_{nk} \frac{\partial b_{nk}}{\partial a_{nj}}$$

- Now $b_{nk} = \sum_j w_{kj} g(a_{nj})$, so $\frac{\partial b_{nk}}{\partial a_{nj}} = w_{kj} g'(a_{nj})$

- Hence $\delta^v_{nj} = \sum_{k=1}^{K} \delta^w_{nk} w_{kj} g'(a_{nj})$ $\quad g'(a) = \frac{d}{da}\sigma(a) = \sigma(a)(1-\sigma(a))$

- The layer 1 error can be computed by passing the layer 2 error back

Putting it all together

- First perform a forward pass to compute $\mathbf{a}_n, \mathbf{z}_n, \mathbf{b}_n, \hat{\mathbf{y}}_n$
- Compute the error for the output layer $\boldsymbol{\delta}_n^{(2)} = \hat{\mathbf{y}}_n - \mathbf{y}_n$
- Compute the error for the hidden layer $\boldsymbol{\delta}_n^{(1)}$ by passing the error backwards through **W** and using

$$\delta_{nj}^{v} = \sum_{k=1}^{K} \delta_{nk}^{w} w_{kj} g'(a_{nj})$$

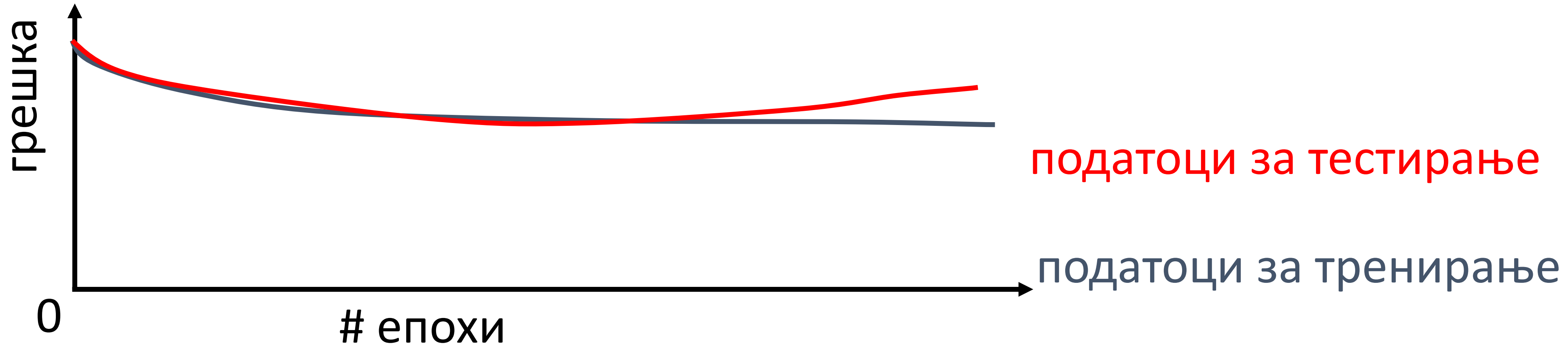
- Compute the overall gradient as follows $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_n [\boldsymbol{\delta}_n^v \mathbf{x}_n,\ \boldsymbol{\delta}_n^w \mathbf{z}_n]$

Коментари за алгоритамот за тренирање

- Нема гаранции дека ќе конвергира кон нулева грешка, може да конвергира во локален оптимум или да влезе во бесконечни осцилации.
- Во пракса, конвергира кон многу мали грешки за големи мрежи врз реални податоци.
- За тренирање на големи мрежи можат да бидат потребни многу епохи (илјадници) што може да трае дури со денови.
- За да избегнете проблеми со локални минимуми испробајте неколку варијанти со различни случајни тежини (*random restarts*)
    - вредностите да бидат блиски до нула [-ε,+ ε]

Спречување на преголемо тренирање

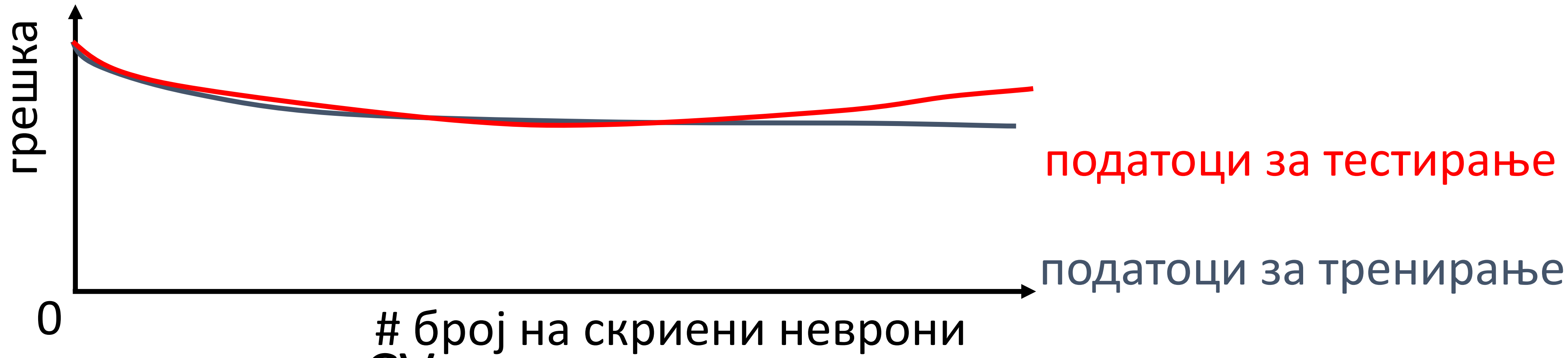
- Многу голем број на епохи може да резултира во over-fitting.



- Дел од податоците поставете го како множество за валидација и тестирајте ја прецизноста после секоја епоха. Запрете со тренирањето кога дополнителните епохи ќе почнат да ја зголемуваат грешката на валидација.
- За да избегнете губење на податоци за тренирање поради валидација:
  - Користете внатрешна за множеството за тренирање (валидација) за да го пресметате просечниот број на епохи што ја максимизира прецизноста на генерализацијата.
  - Крајното тренирање направете го со целото множество за тренирање користејќи го овој број на епохи.

Определување на најдобриот број на скриени неврони

- Вообичаено само еден скриен слој
  - ако има повеќе слоеви во секој слој ист број неврони
- Премалку скриени неврони можат да го оневозможат соодветното прилагодување кон податоците.
- Премногу скриени неврони можат да доведат до over-fitting.



- Користете внатрешна CV за емпириски да го определите оптималниот број на скриени неврони.