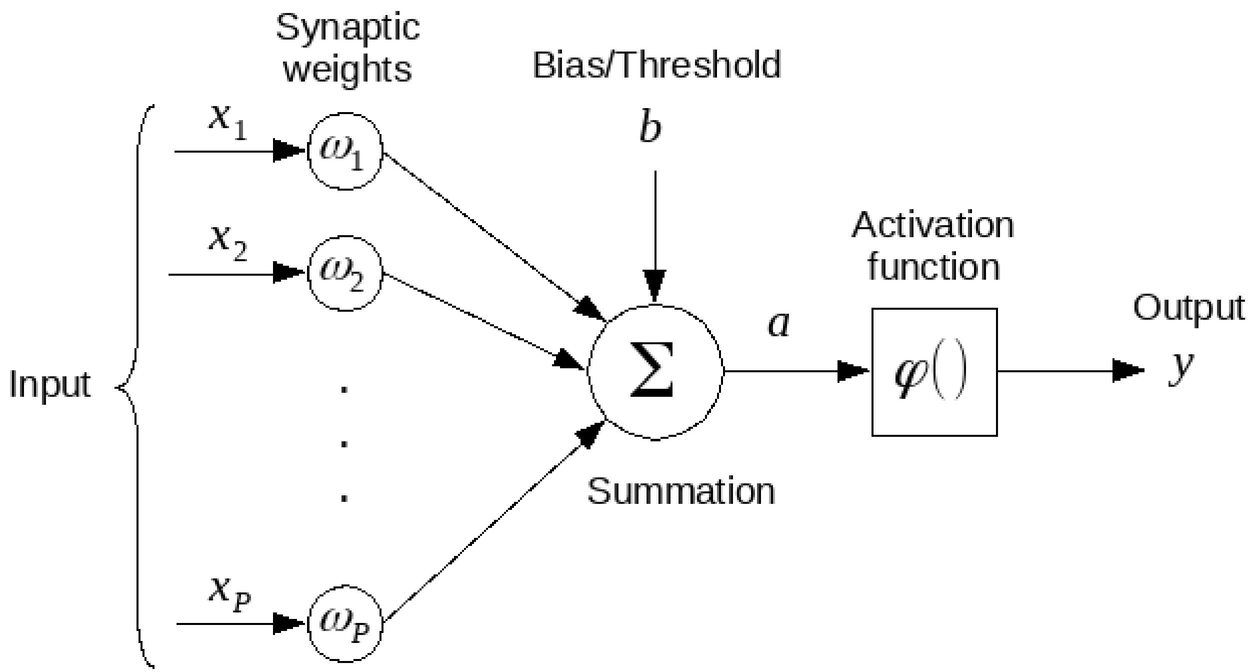
The MLP Architecture

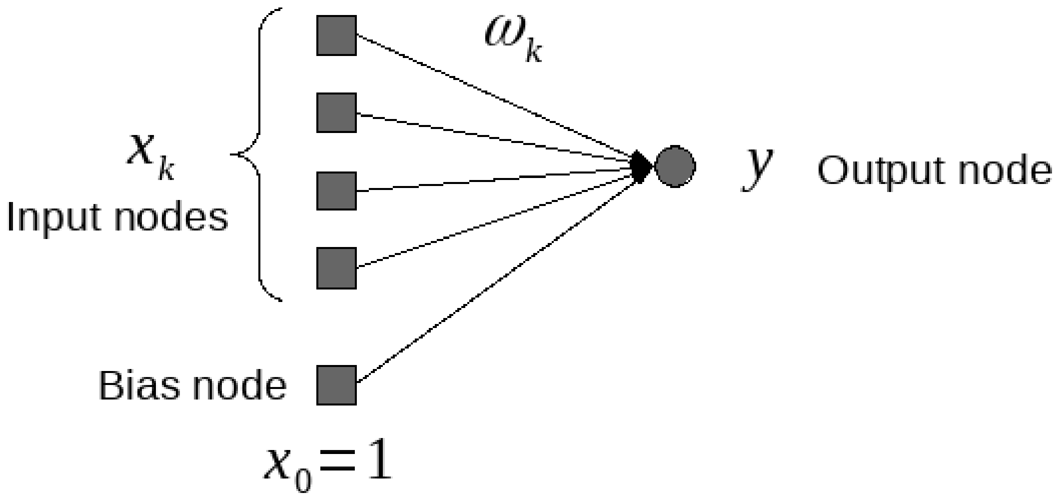
The simple perceptron

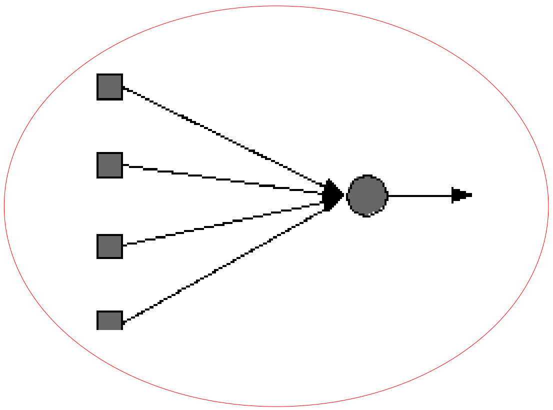


$$a = \sum_{k=1}^{P} \omega_k x_k + b$$

$$y = \varphi(a)$$

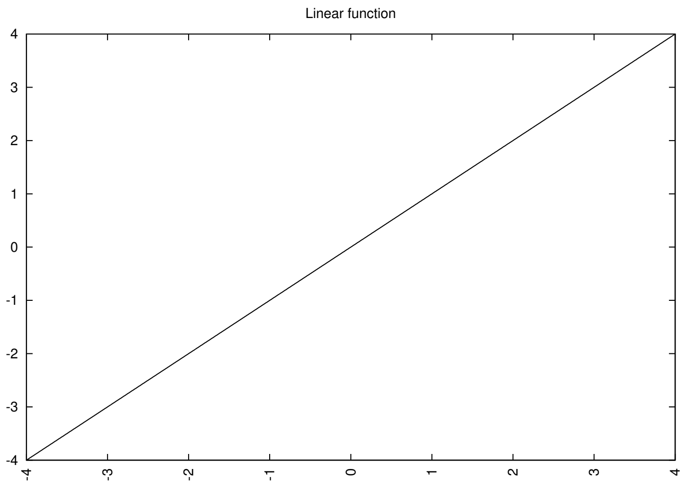
The perceptron, simpler notation

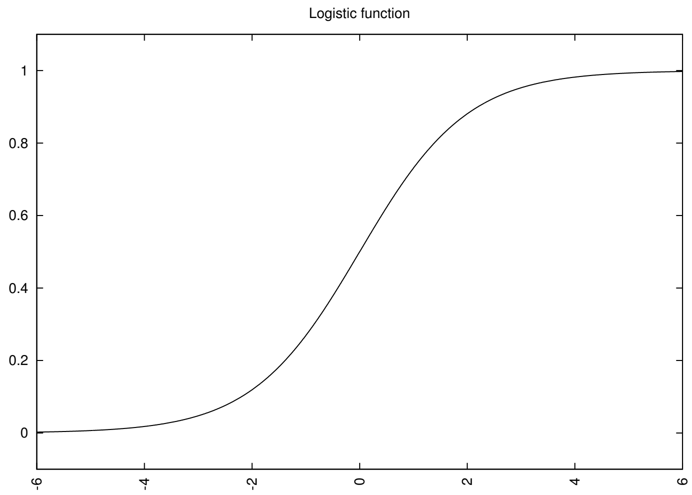




$$y(\mathbf{x}, \boldsymbol{\omega}) = \varphi_o\left(\sum_{k=1}^{P} \omega_k x_k + \omega_0\right) = \varphi_o\left(\sum_{k=0}^{P} \omega_k x_k\right) = \varphi_o(\boldsymbol{\omega}^T \mathbf{x})$$

For the perceptron we only have ”two” activation functions





Linear: $\varphi_o(x) = x$

Logistic: $\varphi_o(x) = \dfrac{1}{1 + e^{-x}}$

Training a perceptron

We need a training dataset: $\{\mathbf{x}_n, \mathbf{d}_n\}_{n=1,\ldots,N}$

"pattern" or
"data point"

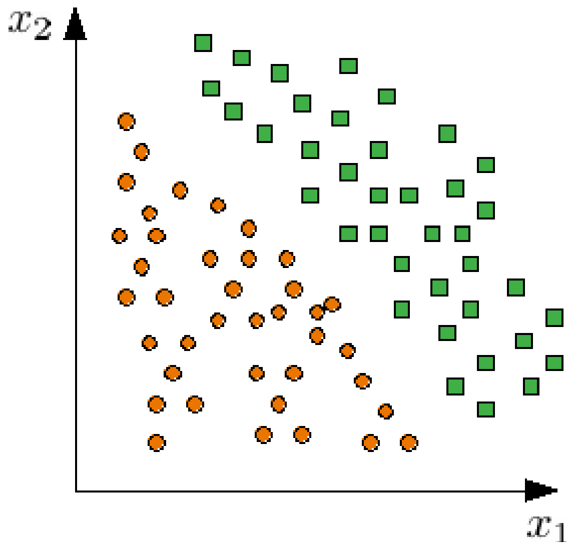
"target" or
"label" or
"gold standard"

Each data point typically consists of many values

$$\mathbf{x}_n = (x_{n1}, x_{n2}, \ldots, x_{nP})$$

Targets can be single- or multiple valued depending on the application.

Example of simple data sets

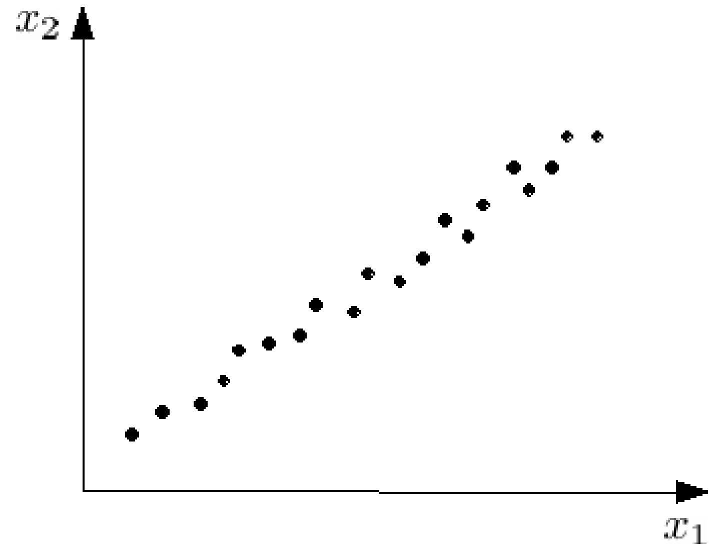


$$\mathbf{x}_n = (x_{n1}, x_{n2}) \text{ (inputs)}$$
$$d_n = \{0, 1\} \text{ (targets)}$$

Class ■

Class ●

Classification problem

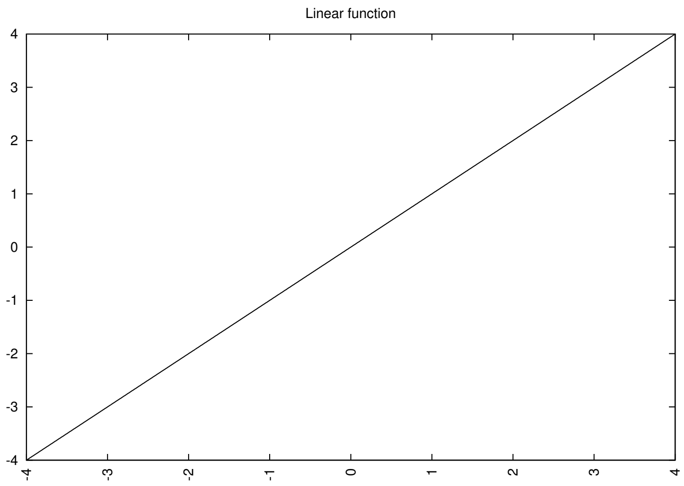


$$x_n = x_{n1} \text{ (inputs)}$$
$$d_n = x_{n2} \text{ (targets)}$$

Regression problem

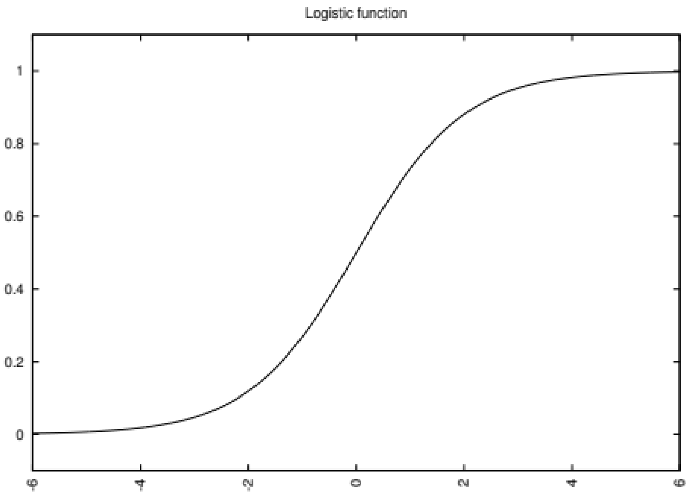
Choice of activation function

Regression problem



Linear: $\varphi_o(x) = x$

Classification problem



Logistic: $\varphi_o(x) = \dfrac{1}{1 + e^{-x}}$

Why these choices?

We have

$$y(\mathbf{x}, \boldsymbol{\omega}) = \varphi_o(\mathbf{x}, \boldsymbol{\omega}) \qquad \left\{ \mathbf{x}_n, d_n \right\}$$

Perceptron

Training data

Denote: $y_n = y(\mathbf{x}_n, \boldsymbol{\omega})$

Task!

$$y_n = d_n \quad , \forall n$$

How?

A very common approach is to
construct an error (loss) function

$$E = \frac{1}{N}\sum_{n=1}^{N}(y_n - d_n)^2$$

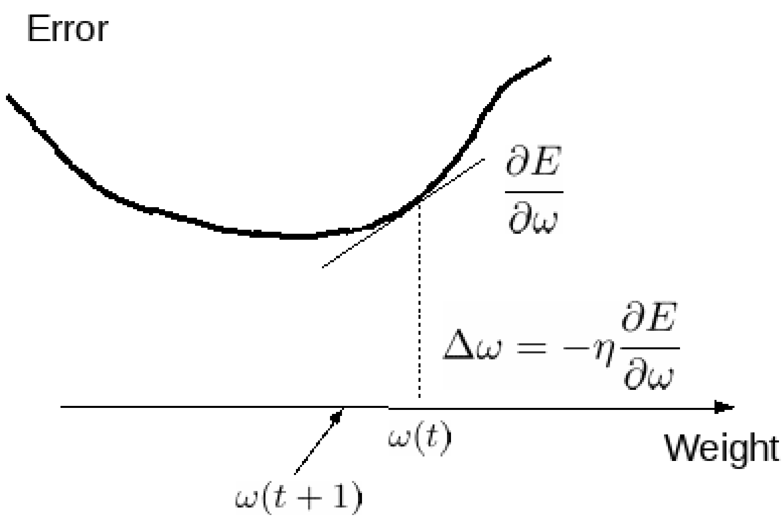
Consider as a function
of the weights!

An example of an
error/loss function!

Minimizing *E* means "solving" the task
(or at least an attempt to solve it)

How?

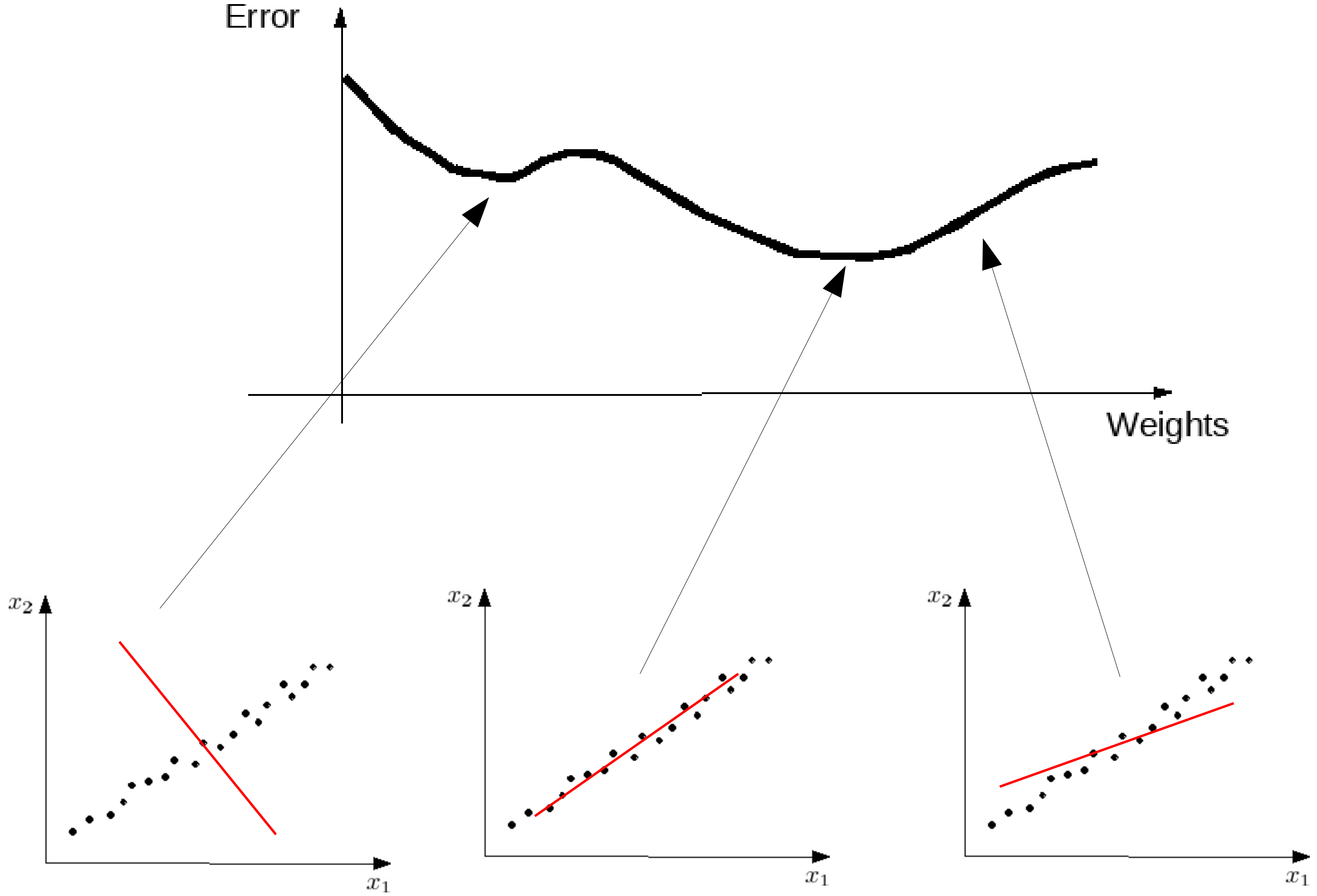
Most common approaches are "**gradient descent**" methods



$$\Delta\omega_i = -\eta \frac{\partial E(\boldsymbol{\omega})}{\partial \omega_i}$$

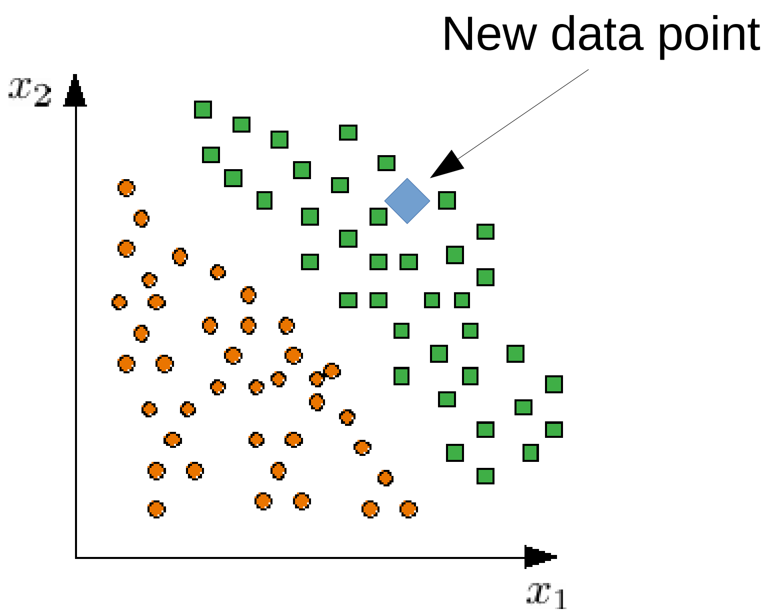
$$\Delta\omega_i = -\eta \frac{1}{N} \sum_{n=1}^{N} \frac{\partial E_n(\boldsymbol{\omega})}{\partial \omega_i}$$

Illustration!



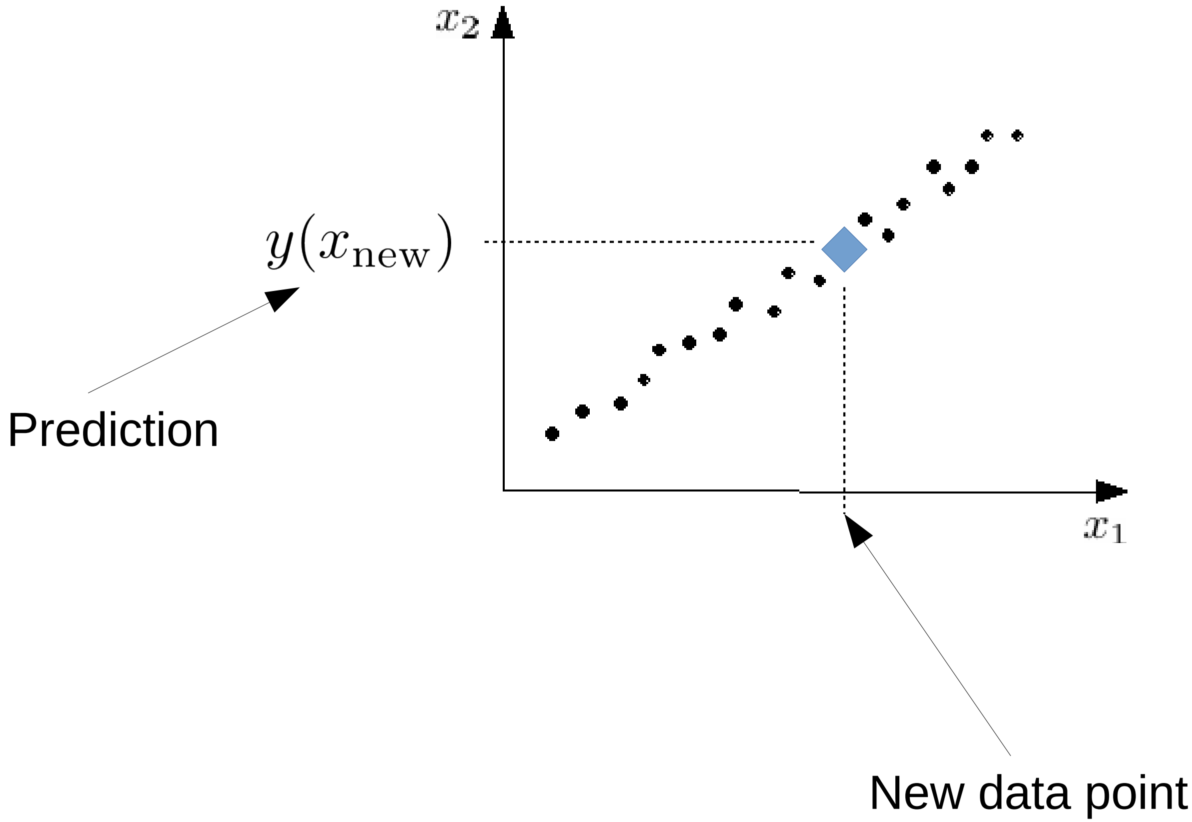
See https://playground.tensorflow.org/ for more illustrations!

How do we use the perceptron?



$$\text{prediction of } \mathbf{x}_{\text{new}} \begin{Bmatrix} \text{class 1} & \text{if} & y(\mathbf{x}_{\text{new}}) > \text{cut value} \\ \text{class 0} & \text{if} & y(\mathbf{x}_{\text{new}}) \leq \text{cut value} \end{Bmatrix}$$

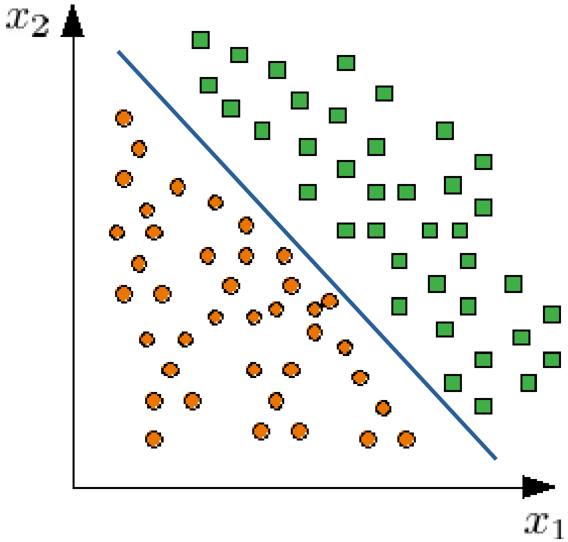
How do we use the perceptron?

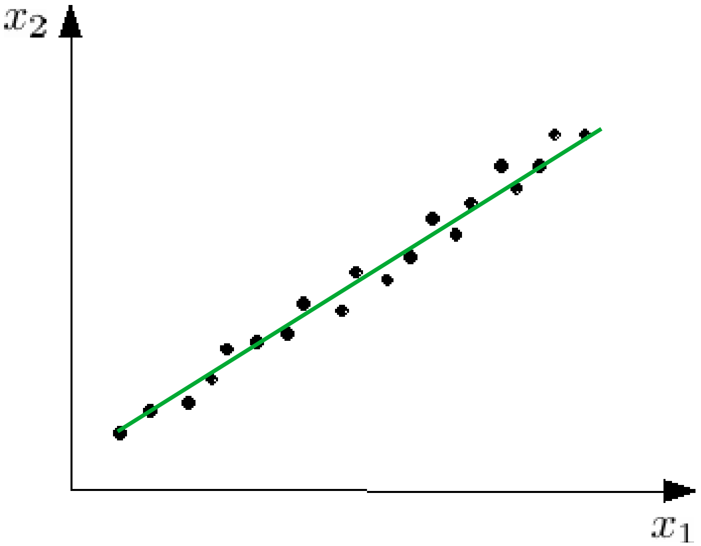


Why do we not always use the perceptron?

Fundamental limitation!

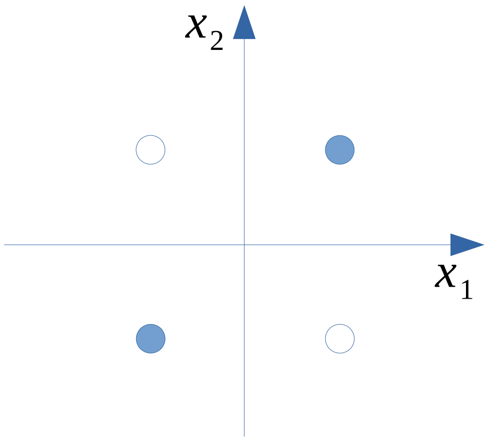
Linear boundary and linear regression!



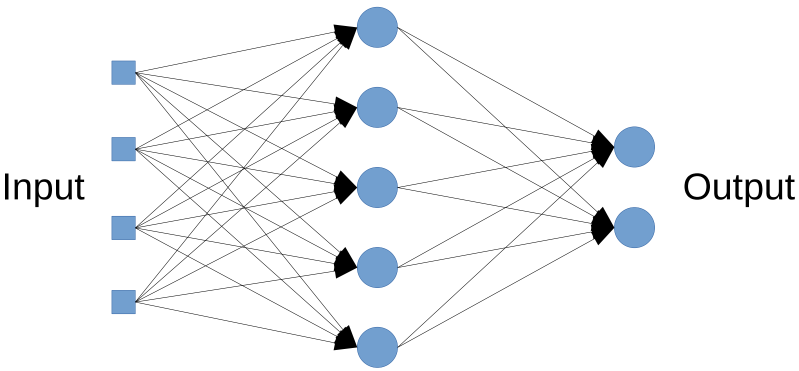


Can we understand why we
have a linear boundary?

The XOR-problem cannot be solved by the perceptron!

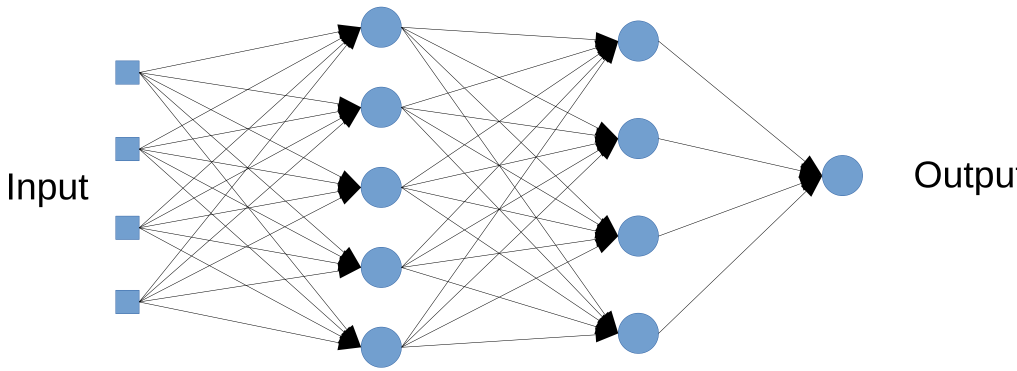


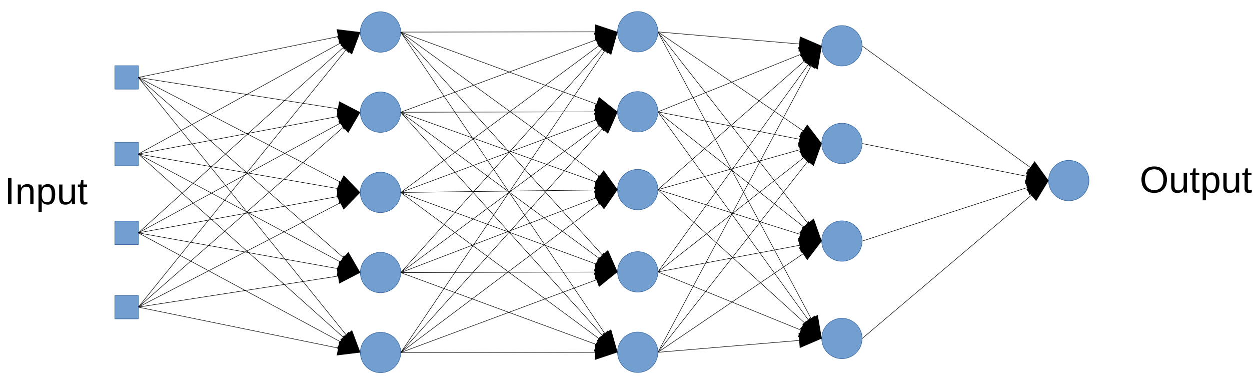
We need the Multi-layer perceptron (MLP)



One hidden layer

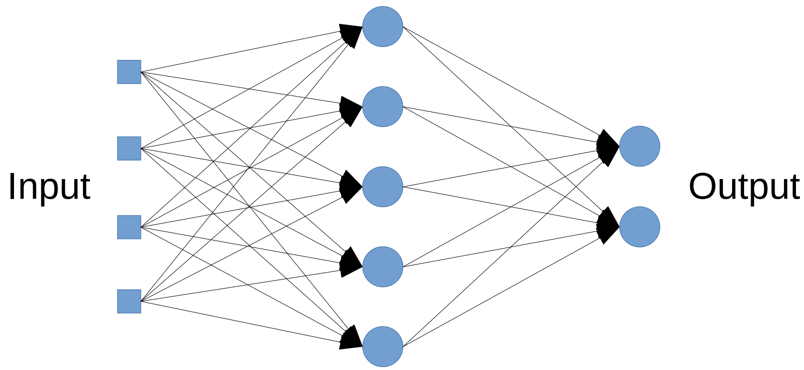
Two hidden layers



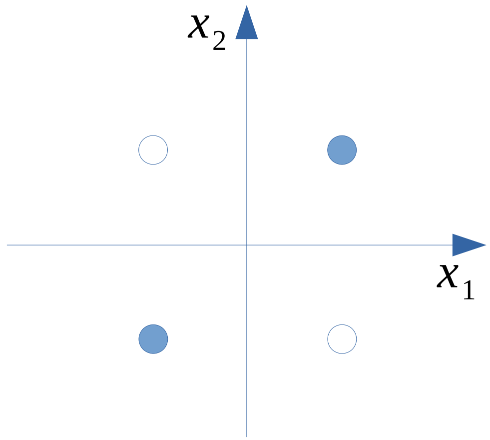


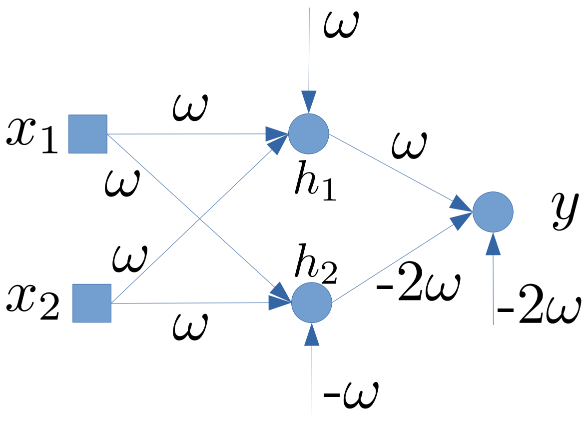
Three hidden layers (towards deep MLP)

It is also important that we have **non-linear** activation functions in the hidden layer

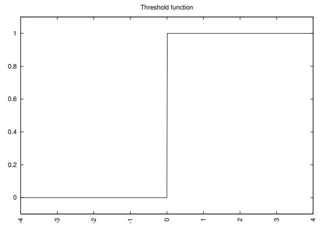


The XOR-problem can be solved by this MLP!

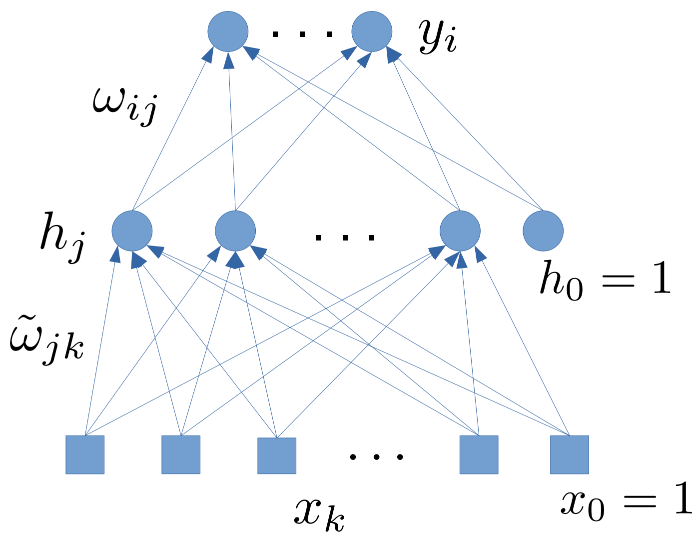




(Activation functions = threshold function)



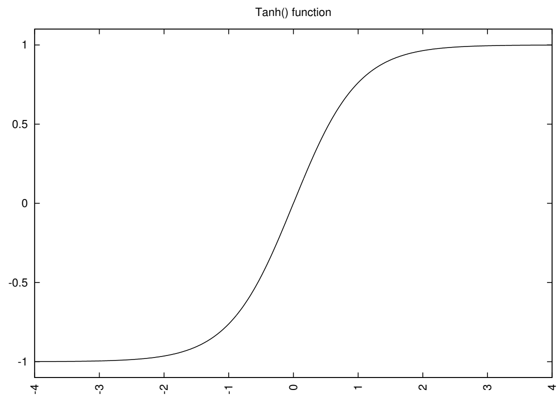
The details of a one hidden layer MLP



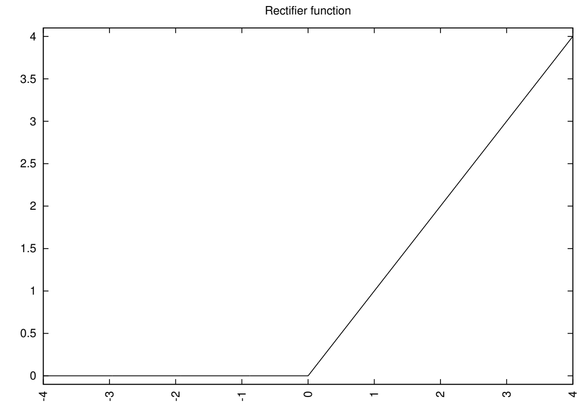
$$y_i(\mathbf{x}_n) = \varphi_o\left(\sum_j \omega_{ij}\varphi_h\left(\sum_k \tilde{\omega}_{jk}x_{nk}\right)\right)$$

Common activation functions for the hidden layers

“Old and shallow”



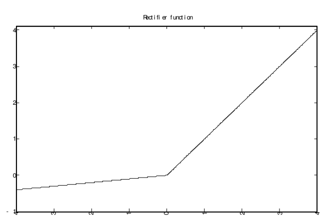
Tanh: $\dfrac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$



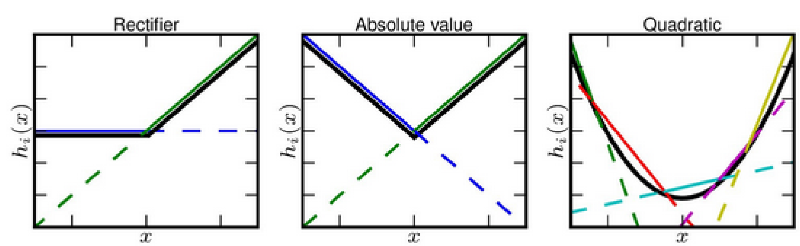
“New and deep”

ReLU: $\max(0, x)$

Also possible



Leaky ReLU



Maxout units

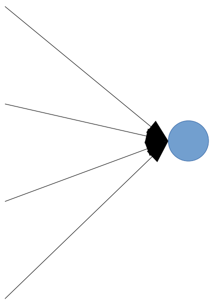
The MLP is mostly used for two kinds of tasks:

Classification: The algorithm is asked to predict one of k classes for which the input belongs to

Regression: Predict numerical outputs given an input

For classification problems we typically
make a distinction between binary
and multiple classification problems

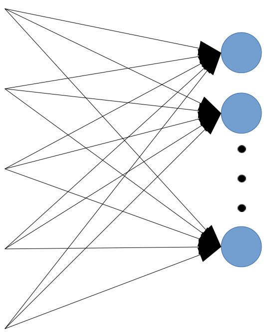
Binary



A single output node:

Class 0: target value = 0
Class 1: target value = 1

M classes

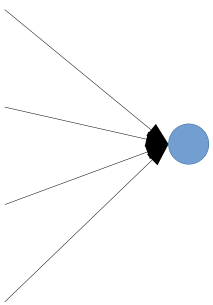


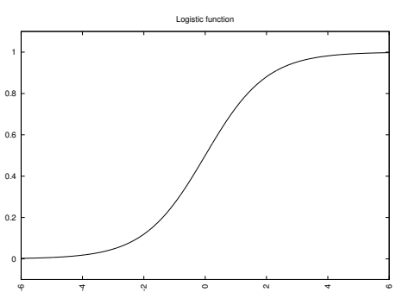
M output nodes:
One-hot-encoding

Class 1: [1 0 0 … 0]
Class 2: [0 1 0 … 0]
…
Class M: [0 0 0 … 1]

Output activation functions for the MLP

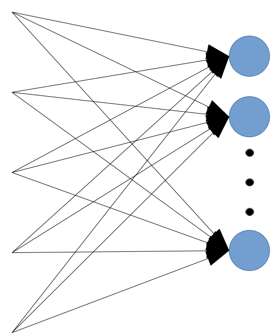
Binary classification

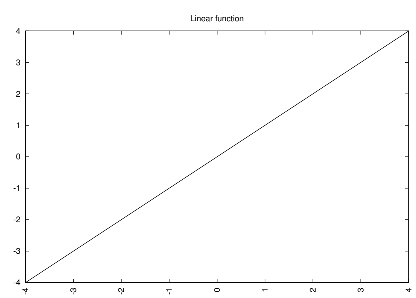




Logistic

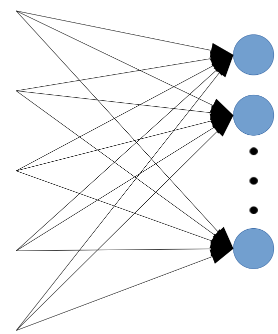
Regression





Linear

M classes



Softmax

$$y_{ni} = \frac{e^{a_{ni}}}{\sum_{i'} e^{a_{ni'}}}$$

a_{ni} = net input to output node i
for pattern n

What about error/loss functions?

For binary classification we commonly use *binary* cross-entropy error/loss:

$$E(\boldsymbol{\omega}) = -\frac{1}{N}\sum_{n=1}^{N}\Big(d_n \log y_n + \big(1 - d_n\big)\log\big(1 - y_n\big)\Big)$$

For M-class classification we commonly use *categorical* cross-entropy error/loss:

$$E(\boldsymbol{\omega}) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{M} d_{ni}\log y_{ni}$$

And for regression we have mean squared error/loss:

$$E(\boldsymbol{\omega}) = \frac{1}{2N}\sum_{n}\sum_{i}\left(d_{ni} - y_{ni}\right)^2$$

How do we decide the error function?

A very common approach is to use the ***Maximum Likelihood principle***

$X = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$ Training data drawn from $p_{\text{data}}(\mathbf{x})$

$p_{\text{model}}(\mathbf{x}; \theta)$ Family of distributions modeled by θ

Maximum Likelihood

$$\theta = \underset{\theta}{\operatorname{argmax}} \; p_{\text{model}}(X; \theta)$$

$$= \underset{\theta}{\operatorname{argmax}} \; \prod_{n}^{N} p_{\text{model}}(\mathbf{x}_n; \theta)$$

Products are numerically tricky, better to take the log ...

$$\theta = \arg\max_{\theta} \sum_{n}^{N} \log p_{\mathrm{model}}(\mathbf{x}_n; \theta)$$

We can define the loss function to be

$$E(\theta) = -\sum_{n}^{N} \log p_{\mathrm{model}}(\mathbf{x}_n; \theta)$$

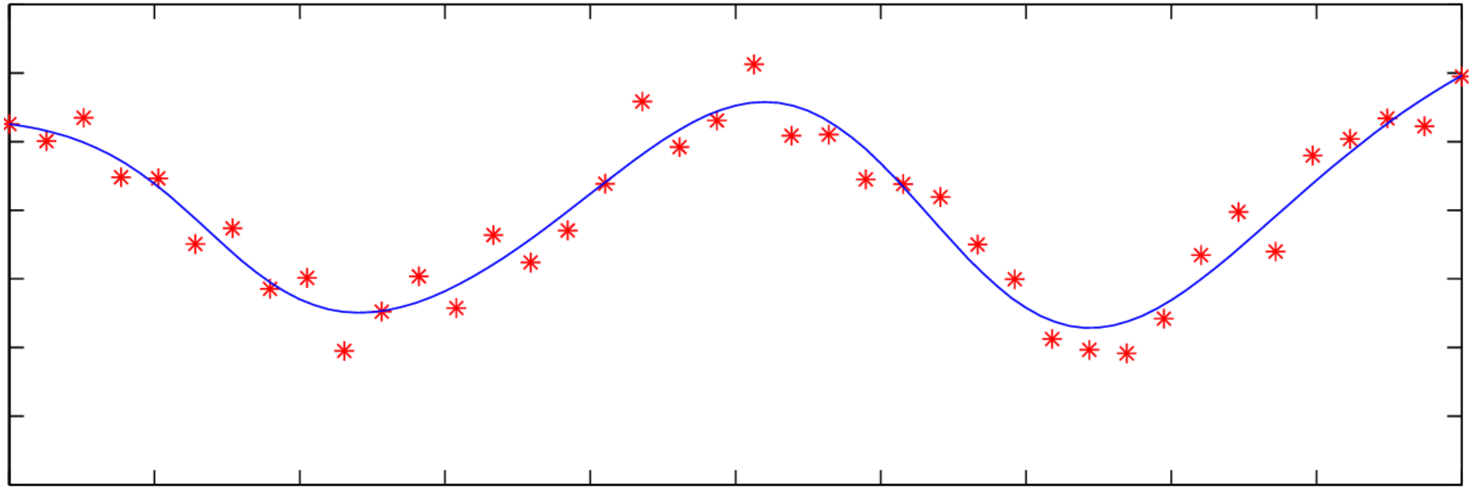
Conditional log likelihood

$$E(\theta) = -\sum_{n}^{N} \log P_{\mathrm{model}}(\mathbf{d}_n | \mathbf{x}_n; \theta)$$

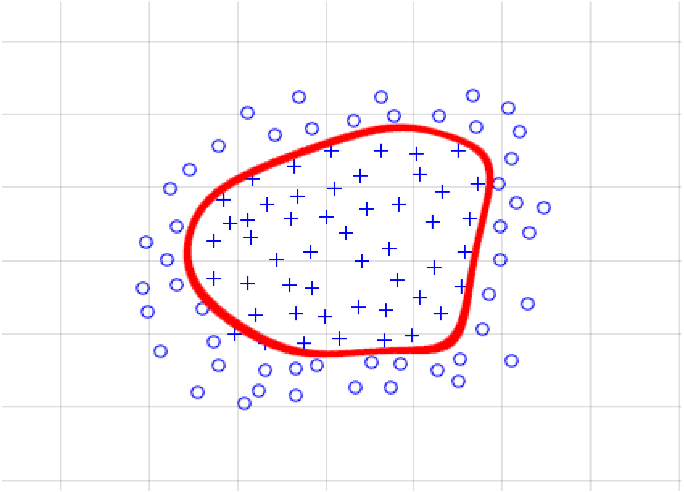
Some simple examples

MLP:
1 – 4 – 1
(tanh - linear)

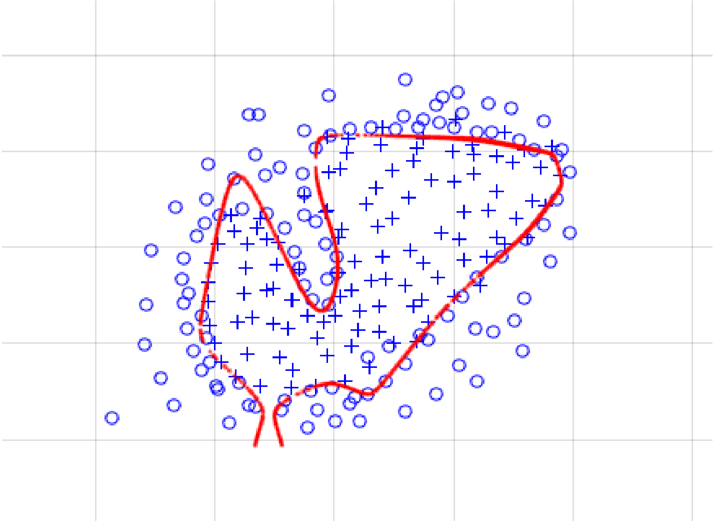
1D regression



MLP:
2 – 4 – 1
(tanh - logistic)

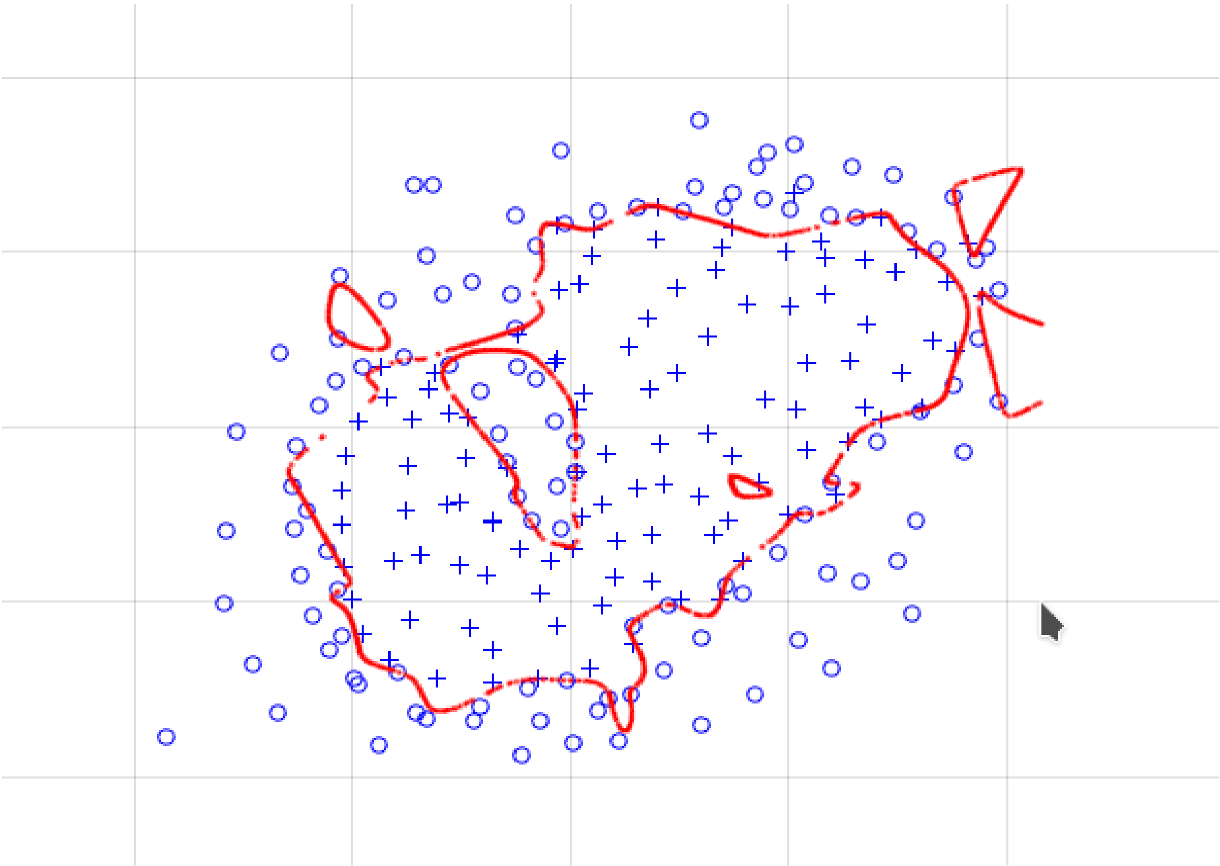


MLP:
2 – 7 – 1
(tanh - logistic)



How can I draw the red line that is
the boundary between the classes?

We can also do this!!



What do we call this situation?

The approximation theorem

Let $\varphi(\cdot)$ be a sigmoidal function and let $f(x) \in \mathcal{C}(I_m)$ where $\mathcal{C}(I_m)$ is the set of all continuous functions defined over the m-dimensional hypercube $I_m = [0,1]^m$. For any $\varepsilon > 0$ there exists an integer N_h and a set of real constants: $\omega_j, \tilde{\omega}_{jk}, b_j$ $(j = 1, \cdots, N_h,\ k = 1, \cdots, m)$ such that

$$|F(x_1, x_2, \cdots, x_m) - f(x_1, x_2, \cdots, x_m)| < \varepsilon \quad \forall x \in I_m$$

where

$$F(x_1, x_2, \cdots, x_m) = \sum_{j}^{N_h} \omega_j \varphi \left(\sum_{k=1}^{m} \tilde{\omega}_{jk} x_k + b_j \right).$$

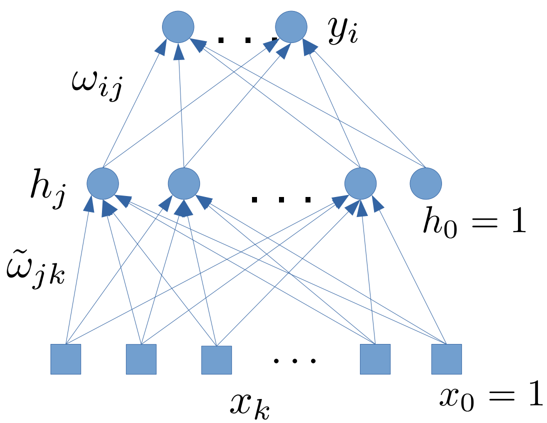
Note 1: No information about the number of nodes needed
Note 2: It also applies for classification problems
Note 3: An updated version exists for ReLU activation functions

More on training the MLP. Some details of minimizing error/loss functions.

The gradient descent has the following basic update formula:

$$\Delta\omega_i = -\eta \frac{\partial E}{\partial \omega_i}$$

As an example for a one hidden layer MLP



input-to-hidden weights $\quad \Delta\tilde{\omega}_{jk} = -\eta \frac{\partial E}{\partial \tilde{\omega}_{jk}}$

hidden-to-output weights $\quad \Delta\omega_{ij} = -\eta \frac{\partial E}{\partial \omega_{ij}}$

As an example for MSE error/loss function

$$E(\boldsymbol{\omega}) = \frac{1}{2N}\sum_{n=1}^{N}\sum_{i}\left(d_{ni} - y_i(\mathbf{x}_n)\right)^2,$$

and with output from the MLP given as

$$y_i(\mathbf{x}_n) = \varphi_o\left(\sum_j \omega_{ij}\varphi_h\left(\sum_k \tilde{\omega}_{jk}x_{nk}\right)\right) = \varphi_o\left(\sum_j \omega_{ij}h_{nj}\right)$$

we can easily compute the weight updates

$$\begin{aligned}
\Delta\omega_{ij} &= -\eta\frac{\partial E}{\partial \omega_{ij}} = ? \\
\Delta\tilde{\omega}_{jk} &= -\eta\frac{\partial E}{\partial \tilde{\omega}_{jk}} = ?
\end{aligned}$$

As an example for MSE error/loss function

$$E(\boldsymbol{\omega}) = \frac{1}{2N} \sum_{n=1}^{N} \sum_{i} \left(d_{ni} - y_i(\mathbf{x}_n)\right)^2 ,$$

and with output from the MLP given as

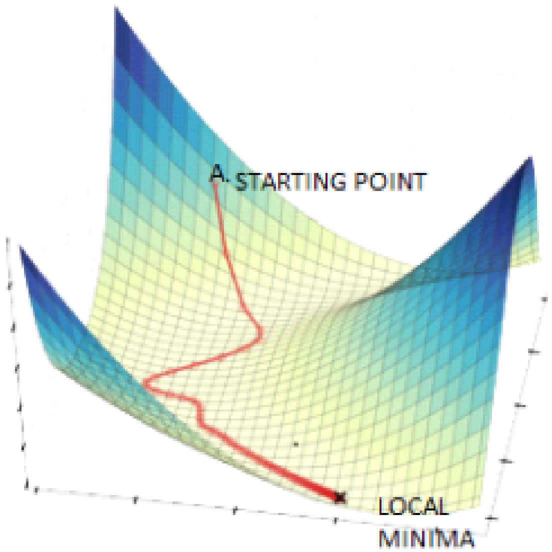
$$y_i(\mathbf{x}_n) = \varphi_o\left(\sum_j \omega_{ij} \varphi_h\left(\sum_k \tilde{\omega}_{jk} x_{nk}\right)\right) = \varphi_o\left(\sum_j \omega_{ij} h_{nj}\right)$$

we can easily compute the weight updates

$$\begin{aligned}
\Delta\omega_{ij} &= -\eta \frac{\partial E}{\partial \omega_{ij}} = \eta \frac{1}{N} \sum_n \underbrace{(d_{ni} - y_i(\mathbf{x}_n))\varphi_o'\left(\sum_{j'} \omega_{ij'} h_{nj'}\right)}_{\equiv \delta_{ni}} h_{nj} \\
\Delta\tilde{\omega}_{jk} &= -\eta \frac{\partial E}{\partial \tilde{\omega}_{jk}} = \eta \frac{1}{N} \sum_n \sum_i \delta_{ni} \; \omega_{ij} \; \varphi_h'\left(\sum_{k'} \tilde{\omega}_{jk'} x_{nk'}\right) x_{nk}
\end{aligned}$$

How do we compute gradients for a general feed-forward architecture?

Training an ML model, especially neural network models, involves minimizing the loss function with respect to model parameters



Very often one have to rely on numerical minimization procedures. The most common approach is ***gradient descent*** based methods

$$\Delta\omega_i = -\eta \frac{\partial E(\boldsymbol{\omega})}{\partial \omega_i}$$

Now very often

$$E(\boldsymbol{\omega}) = \frac{1}{N}\sum_{n}^{N} E_n(\boldsymbol{\omega})$$

We can write

$$\Delta\omega_i = \frac{1}{N}\sum_{n} \Delta\omega_{ni}$$

per pattern update

Gradient descent improvements 1:
Stochastic gradient descent

Gradient descent (GD)

$$\Delta\omega_i = -\eta \frac{1}{N} \sum_{n=1}^{N} \frac{\partial E_n(\boldsymbol{\omega})}{\partial \omega_i}$$

Stochastic gradient descent (SGD)

$$\Delta\omega_i = -\eta \frac{1}{P} \sum_{p=1}^{P} \frac{\partial E_p(\boldsymbol{\omega})}{\partial \omega_i}$$

$$P << N$$

The collection P samples to use
if called a **mini-batch**

Gradient descent improvements 2: **Momentum**

The momentum term adds a part of the previous update to the current

$$\Delta\omega_i(t+1) = -\eta\frac{\partial E}{\partial \omega_i} + \alpha\Delta\omega_i(t)$$

(Why is it called momentum?)

Gradient descent improvements 3:
Individual learning rates - RPROP

Individual learning rates can handle the problem of different gradient sizes in different directions

$$\Delta\omega_{ij} = -\eta_{ij}\frac{\partial E}{\partial \omega_{ij}}$$

RPROP = Resilient PROPagation

$$\eta_{ij}(t) = \begin{cases} \gamma^{+}\eta_{ij}(t-1) & \text{if} \quad \frac{\partial E(t)}{\partial \omega_{ij}} \cdot \frac{\partial E(t-1)}{\partial \omega_{ij}} > 0 \\ \gamma^{-}\eta_{ij}(t-1) & \text{if} \quad \frac{\partial E(t)}{\partial \omega_{ij}} \cdot \frac{\partial E(t-1)}{\partial \omega_{ij}} < 0 \end{cases}$$

with $0 < \gamma^{-} < 1 < \gamma^{+}$

But not used so much….

Gradient descent improvements 4:
RMSPROP

RMSPROP (Root Mean Square Propagation) only uses one common learning rate, but it keeps a running average of the squared gradient for each weight that is used to normalize the magnitude of the gradient, thereby effectively introducing individual weights.

Running average

$$v_i(t) = \gamma v_i(t-1) + (1-\gamma)\left(\frac{\partial E(t)}{\partial \omega_i}\right)^2$$

Update the weight

$$\omega_i(t+1) = \omega_i(t) - \frac{\eta}{\sqrt{v_i(t)}}\frac{\partial E(t)}{\partial \omega_i}$$

To be computed using SGD

Gradient descent improvements 5:
ADAM (ADAptive Moment estimation)

In addition to keeping a running average of the square of the past gradients, as RMSPROP does, Adam also keeps a running average of the past gradients. We define,

$$m_i(t+1) = \beta_1 m_i(t) + (1-\beta_1)\frac{\partial E(t)}{\partial \omega_i}$$

$$v_i(t+1) = \beta_2 v_i(t) + (1-\beta_2)\left(\frac{\partial E(t)}{\partial \omega_i}\right)^2$$

and

$$\hat{m}_i = \frac{m_i(t+1)}{1-\beta_1^t}$$

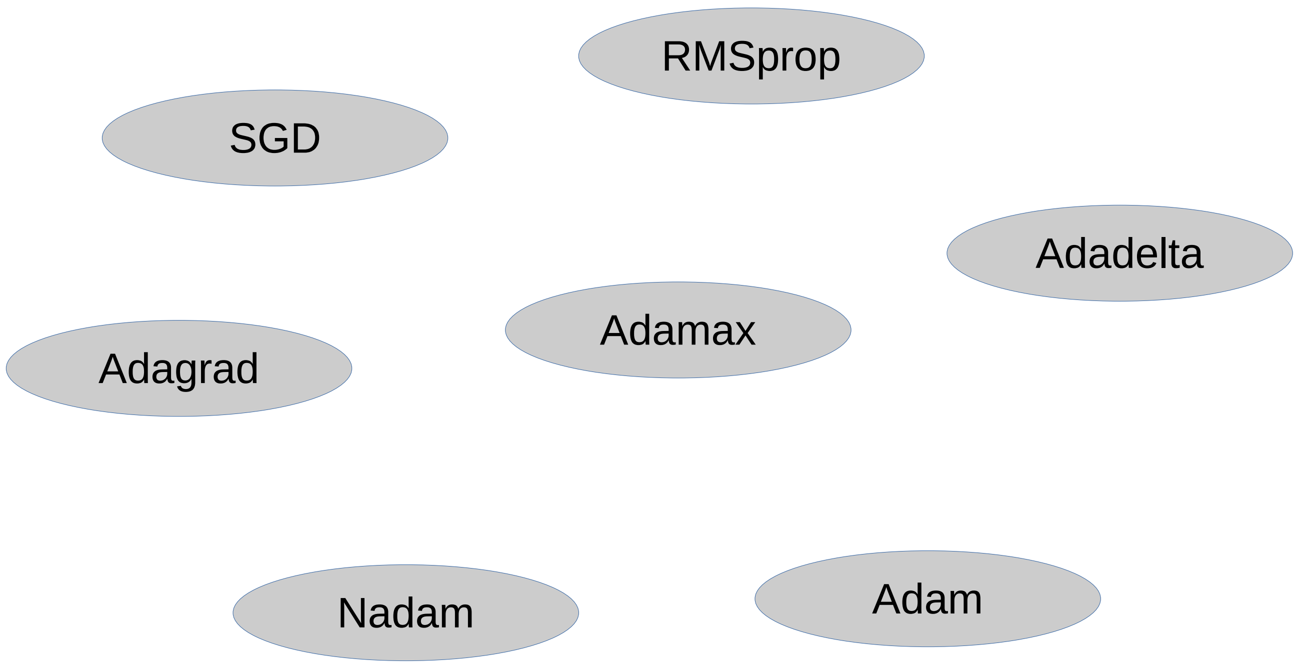
$$\hat{v}_i = \frac{v_i(t+1)}{1-\beta_2^t}$$

$$\omega_i(t+1) = \omega_i(t) - \eta\frac{\hat{m}_i}{\sqrt{\hat{v}_i}+\epsilon}$$

$$(\beta_1 = 0.9, \beta_2 = 0.999 \text{ and } \epsilon = 10^{-8})$$

Adam is very popular!!

Many different methods



Are we now ready to start training?

OK!

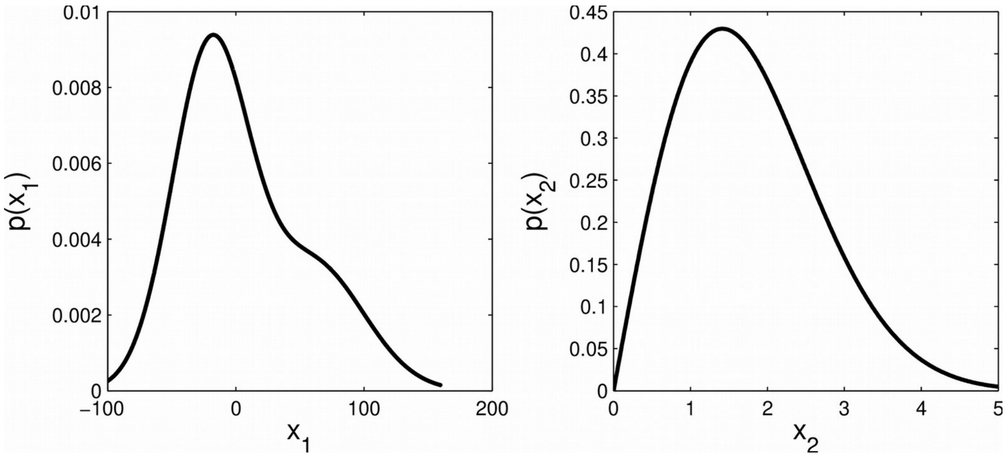
- Dataset
- Choice of architecture
- Choice of activation functions
- Choice of error/loss function
- How to minimize

What about?

- Pre-processing of input data
- Measuring the performance

Pre-processing of input data

We need to compensate for different input sizes



Compute

Mean $$\mu_k = \frac{1}{N}\sum_{n=1}^{N} x_{nk}$$

Std $$\sigma_k = \sqrt{\frac{1}{N}\sum_{n=1}^{N}\left(x_{nk} - \mu_k\right)^2}$$

Transform $$x_{nk} \to \frac{x_{nk} - \mu_k}{\sigma_k} \quad \forall\, n, k$$

More pre-processing of input data

- Missing data imputation
- Encoding
- Dimensionality reduction
- Feature selection
- More pre-processing

Encoding

Feature:	**Input**:
Numerical value	Numerical value
Binary category	Often 0/1 encoding
Many categories	Often one-hot-encoding
< 5 years	[1 0 0]
[5, 10] years	[0 1 0]
> 10 years	[0 0 1]
Text	“Many possibilities”, e.g. word2vec
….	….

How to measure performance – regression problems

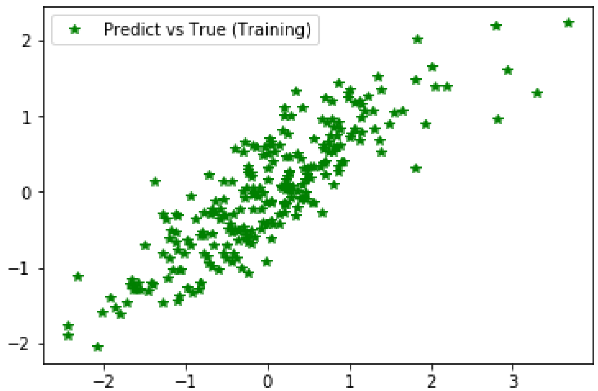
RMSE

$$E = \sqrt{\frac{1}{N}\sum_{n}^{N}\left\|\mathbf{d}_n - \mathbf{y}(\mathbf{x}_n, \boldsymbol{\omega}^*)\right\|^2}$$

Normalized MSE

$$E = \frac{\sum_n \left\|\mathbf{d}_n - \mathbf{y}(\mathbf{x}_n, \boldsymbol{\omega}^*)\right\|^2}{\sum_n \left\|\mathbf{d}_n - < \mathbf{d} >\right\|^2}$$

Scatterplot
True vs. predicted



Compute correlation!

How to measure performance – binary classification problems

Confusion matrix

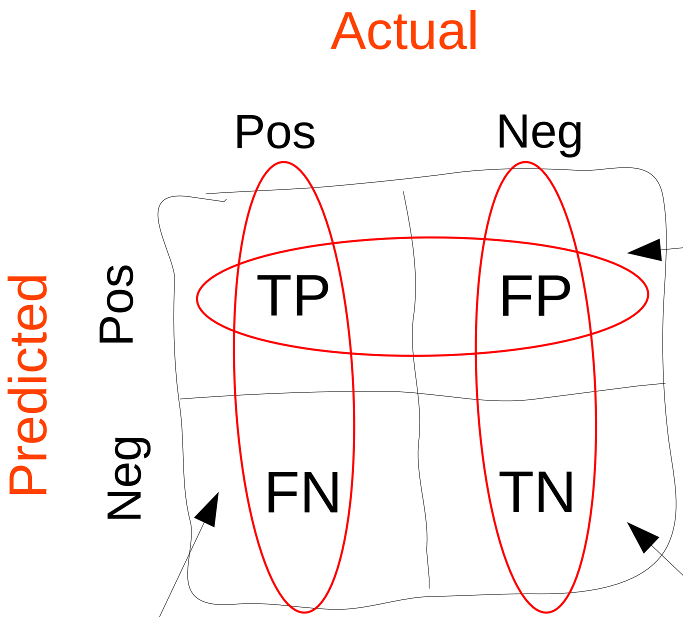
		Actual	
		Pos	Neg
Predicted	Pos	TP	FP
	Neg	FN	TN

TP = True Positives
TN = True Negatives
FP = False Positives
FN = False Negatives

		Actual	
		Pos	Neg
Predicted	Pos	TP	FP
	Neg	FN	TN

$$\text{Accuracy} = \frac{TP + TN}{TP+FP+FN+TN}$$

Why can accuracy be misleading?

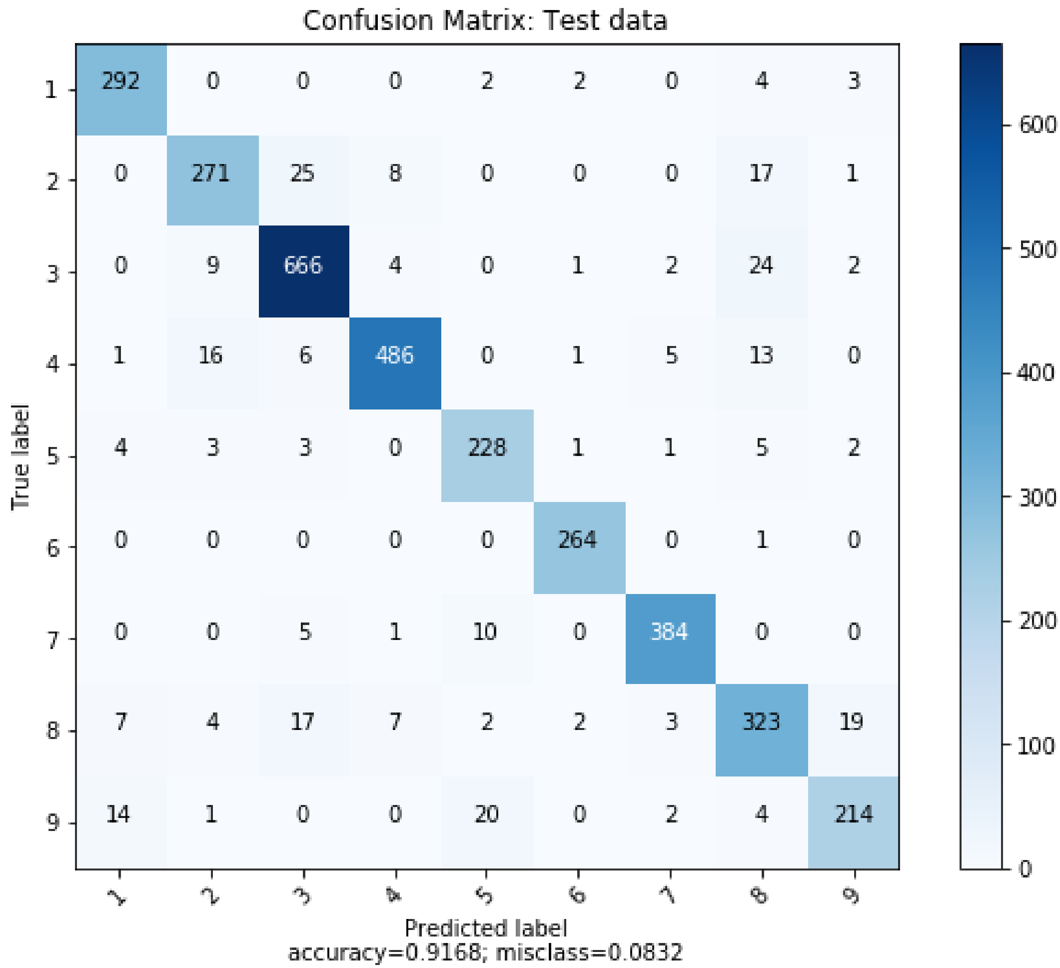


$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Sensitivity (Recall)} = \frac{TP}{TP + FN}$$

$$\text{Specificity} = \frac{TN}{FP + TN}$$

Confusion matrix, also for many classes



True label \ Predicted label	1	2	3	4	5	6	7	8	9
1	292	0	0	0	2	2	0	4	3
2	0	271	25	8	0	0	0	17	1
3	0	9	666	4	0	1	2	24	2
4	1	16	6	486	0	1	5	13	0
5	4	3	3	0	228	1	1	5	2
6	0	0	0	0	0	264	0	1	0
7	0	0	5	1	10	0	384	0	0
8	7	4	17	7	2	2	3	323	19
9	14	1	0	0	20	0	2	4	214

For binary classification problems it is common to use the Receiver Operating Characteristics (ROC) curve and the area under it (AUC)

To make a decision for a class, we need a cut value (C)

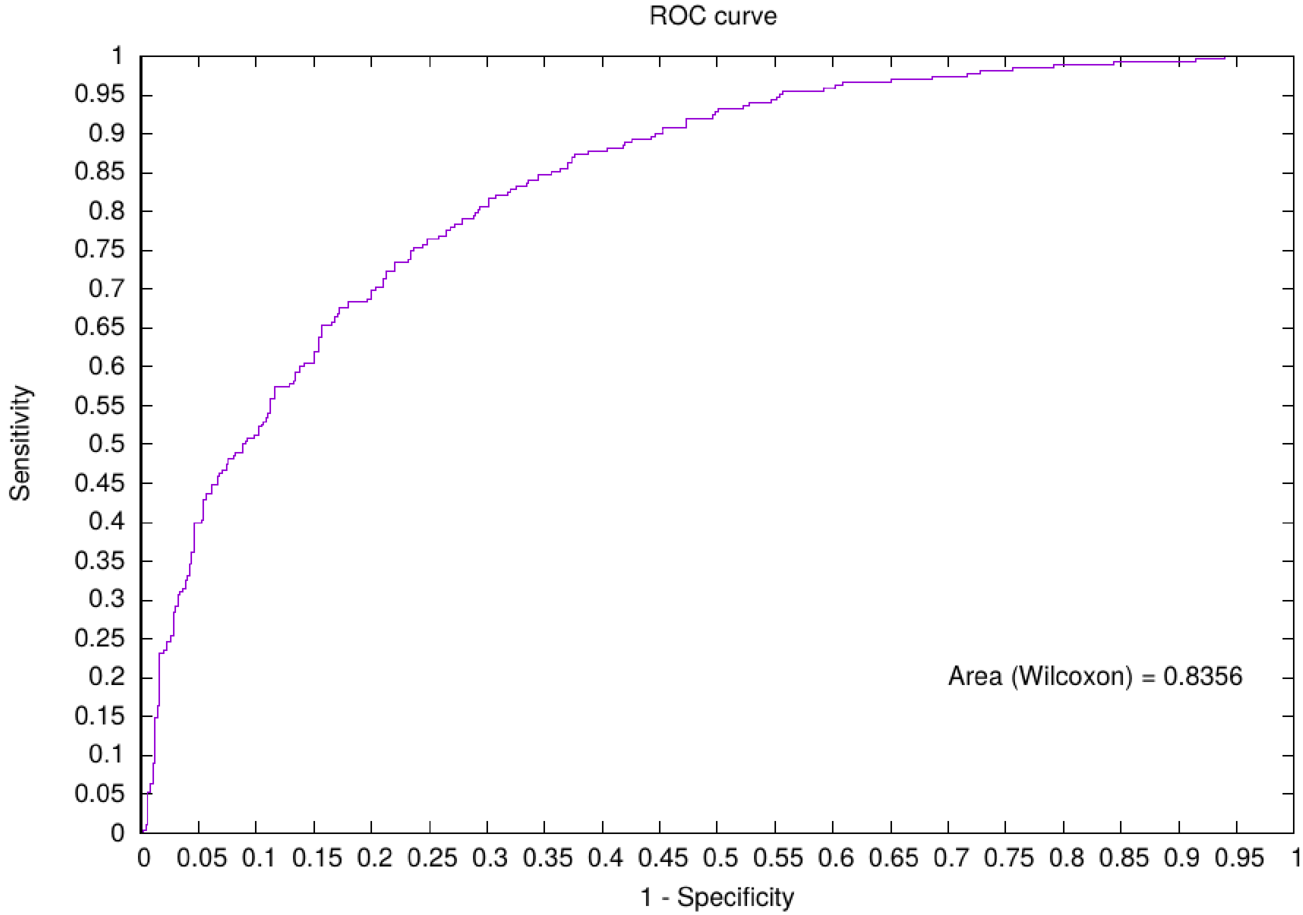
$$\left\{ \begin{array}{lll} \text{class 1 (pos)} & \text{if} & y(\mathbf{x}) > \mathrm{C} \\ \text{class 0 (neg)} & \text{if} & y(\mathbf{x}) \leq \mathrm{C} \end{array} \right\}$$

For each C we get a Sensitivity / Specificity pair

Vary C between [0,1] and plot all Sens vs (1-Spec)

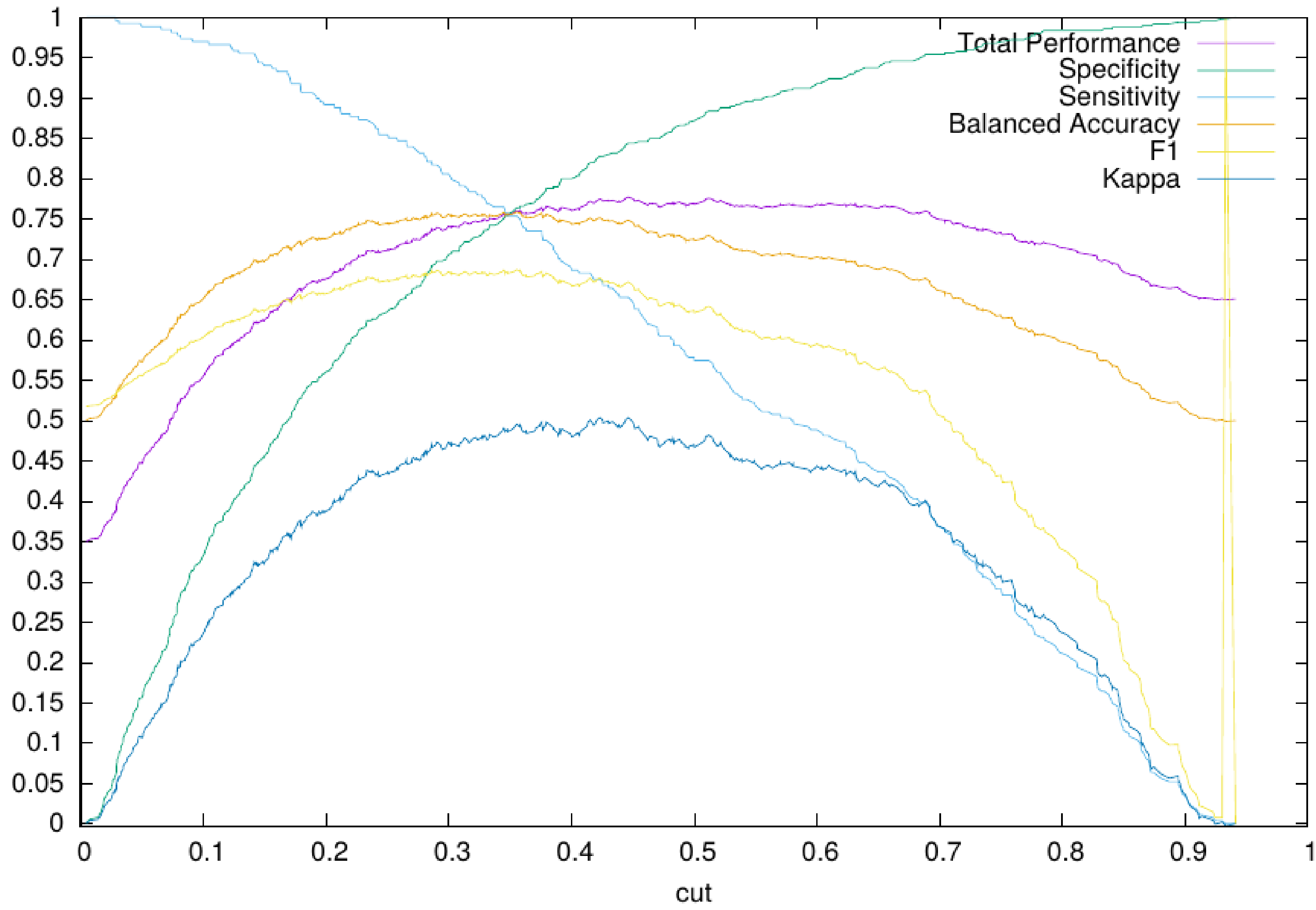
This is the ROC curve!

An example



What does the area mean?

We can plot more things!



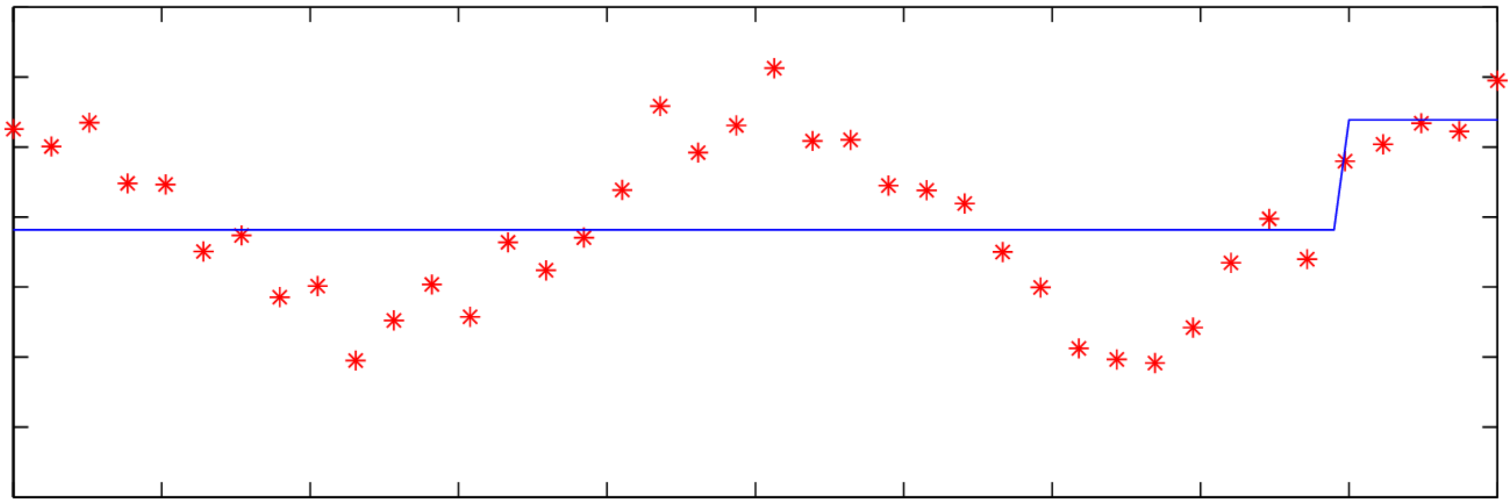
We now ready to start training!

OK!

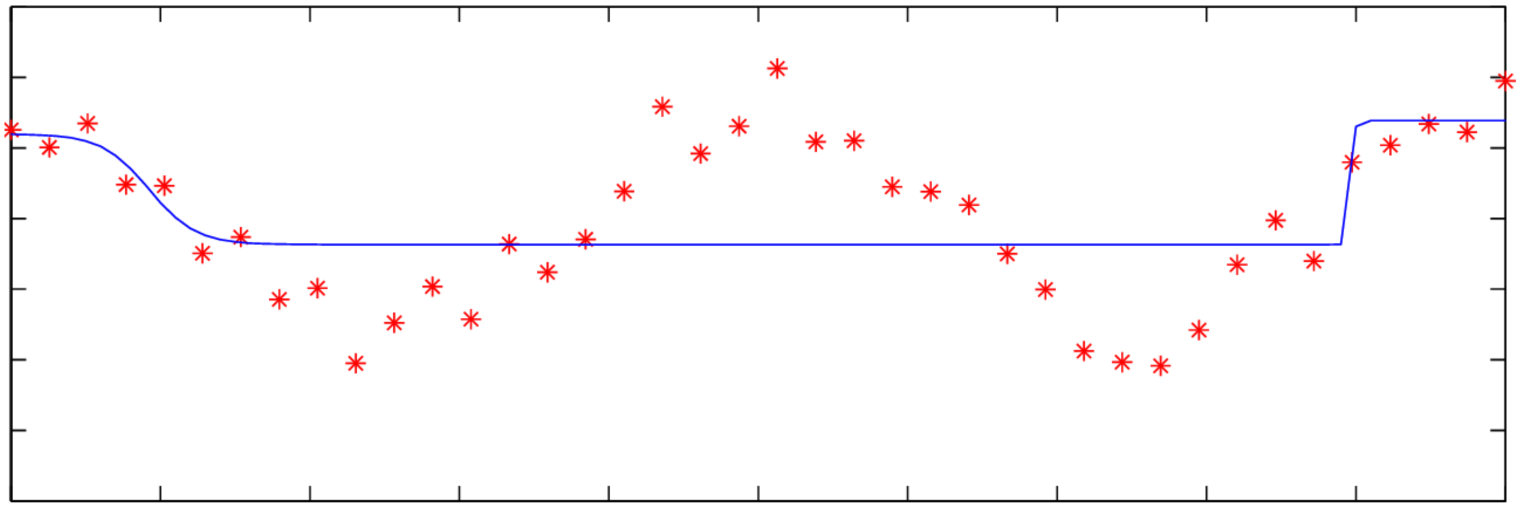
- Dataset
- Choice of architecture
- Choice of activation functions
- Choice of error/loss function
- How to minimize
- Pre-processing of input data
- Measuring the performance

Some examples!

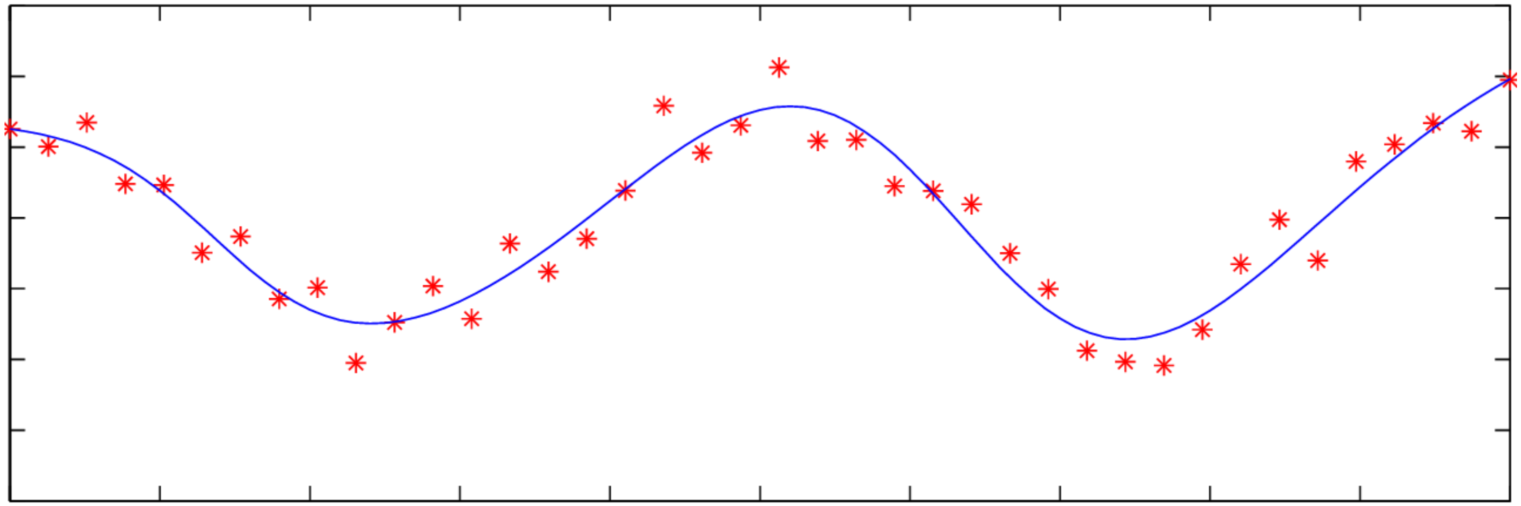
MLP: 1 hidden node



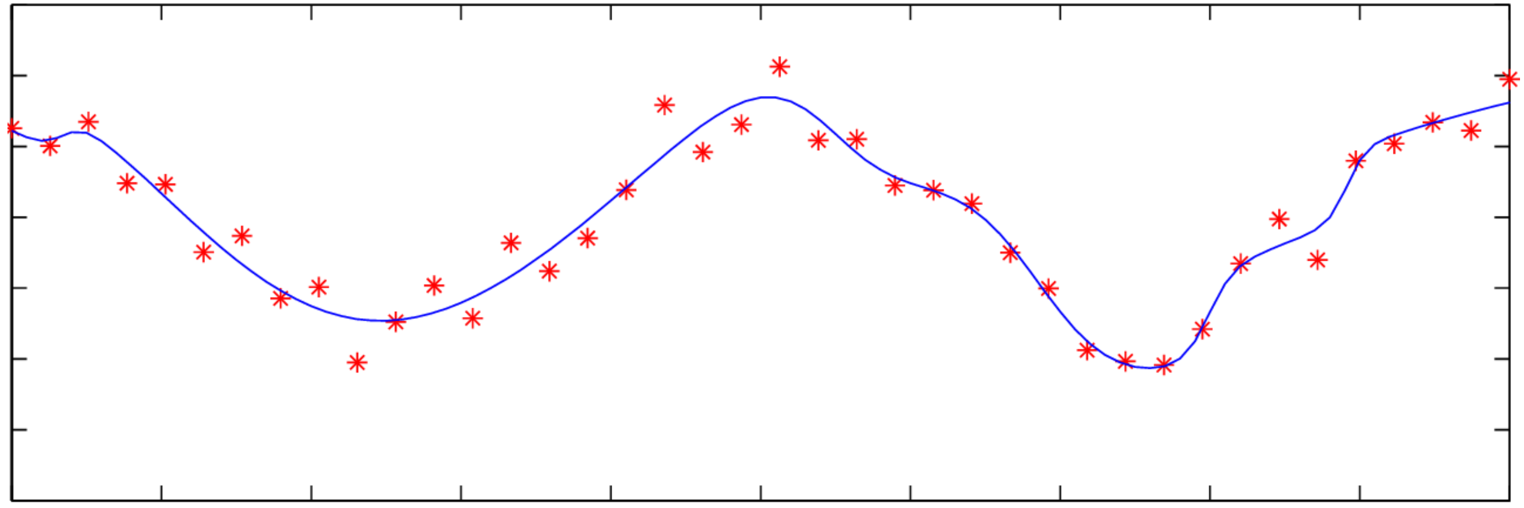
MLP: 2 hidden nodes



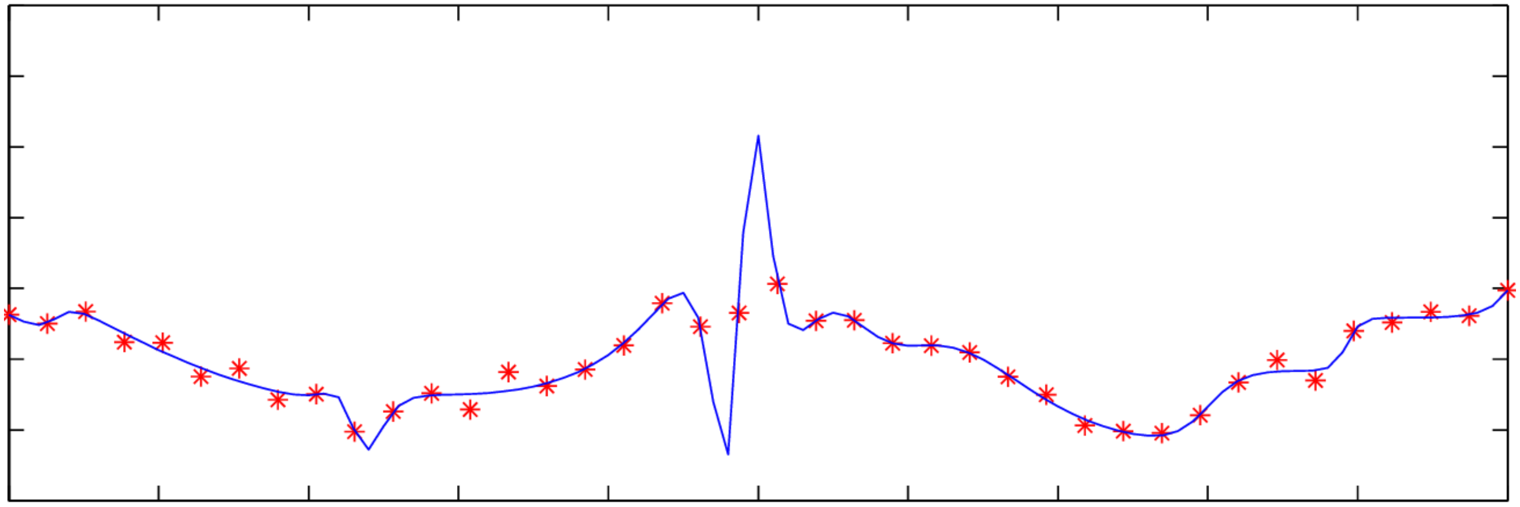
MLP: 4 hidden nodes



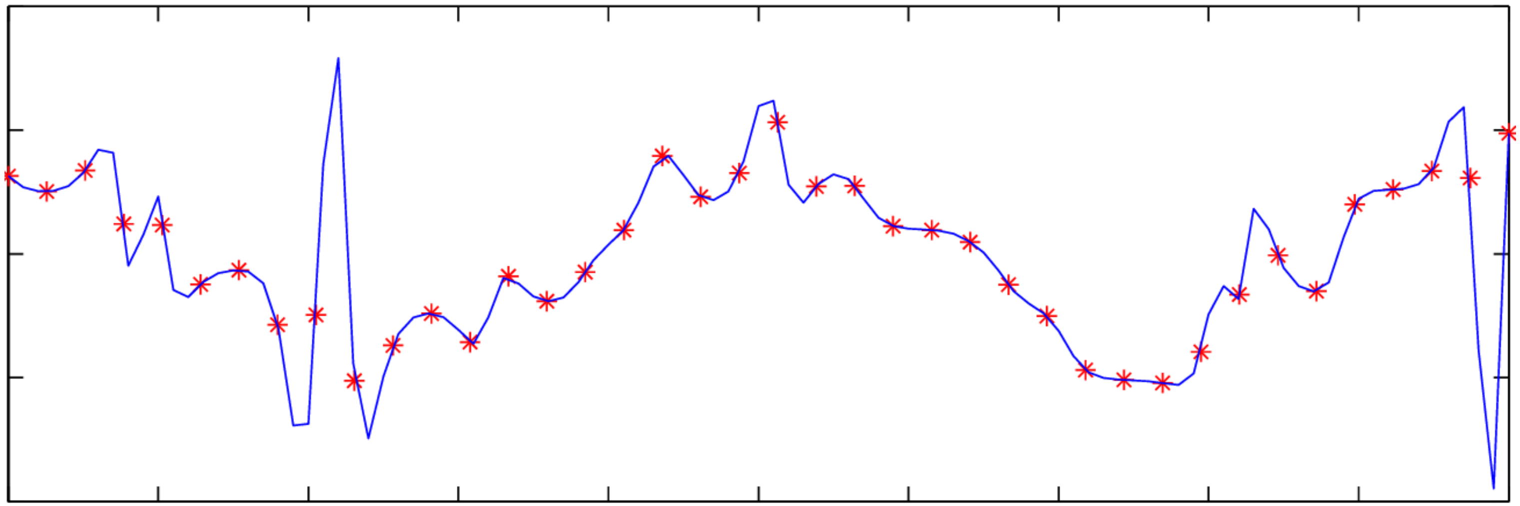
MLP: 7 hidden nodes



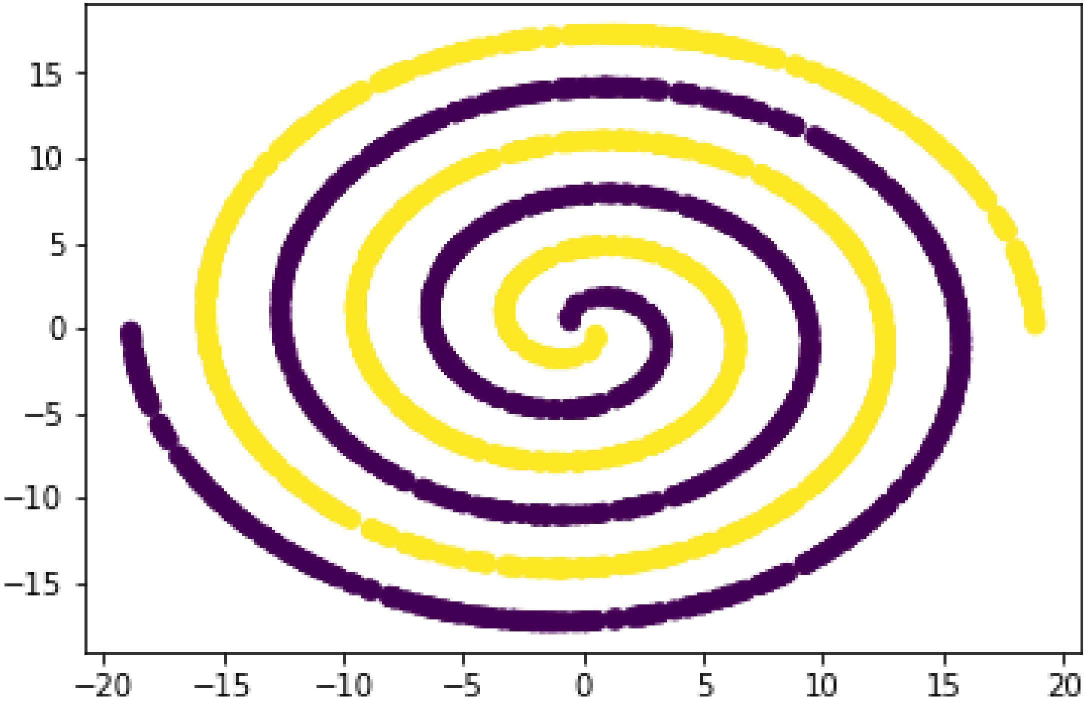
MLP: 12 hidden nodes



MLP: 25 hidden nodes

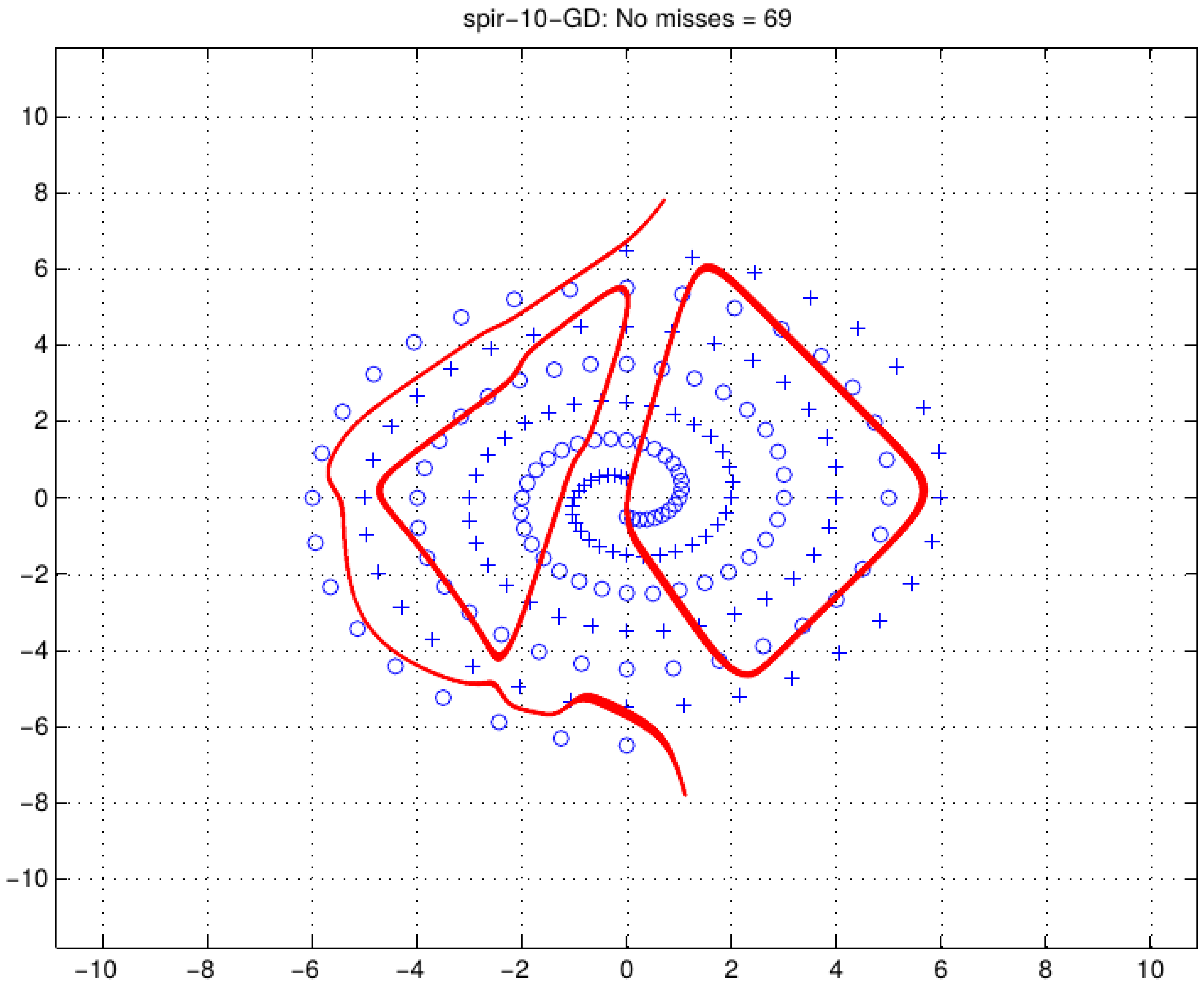


The 2D spiral classification problem



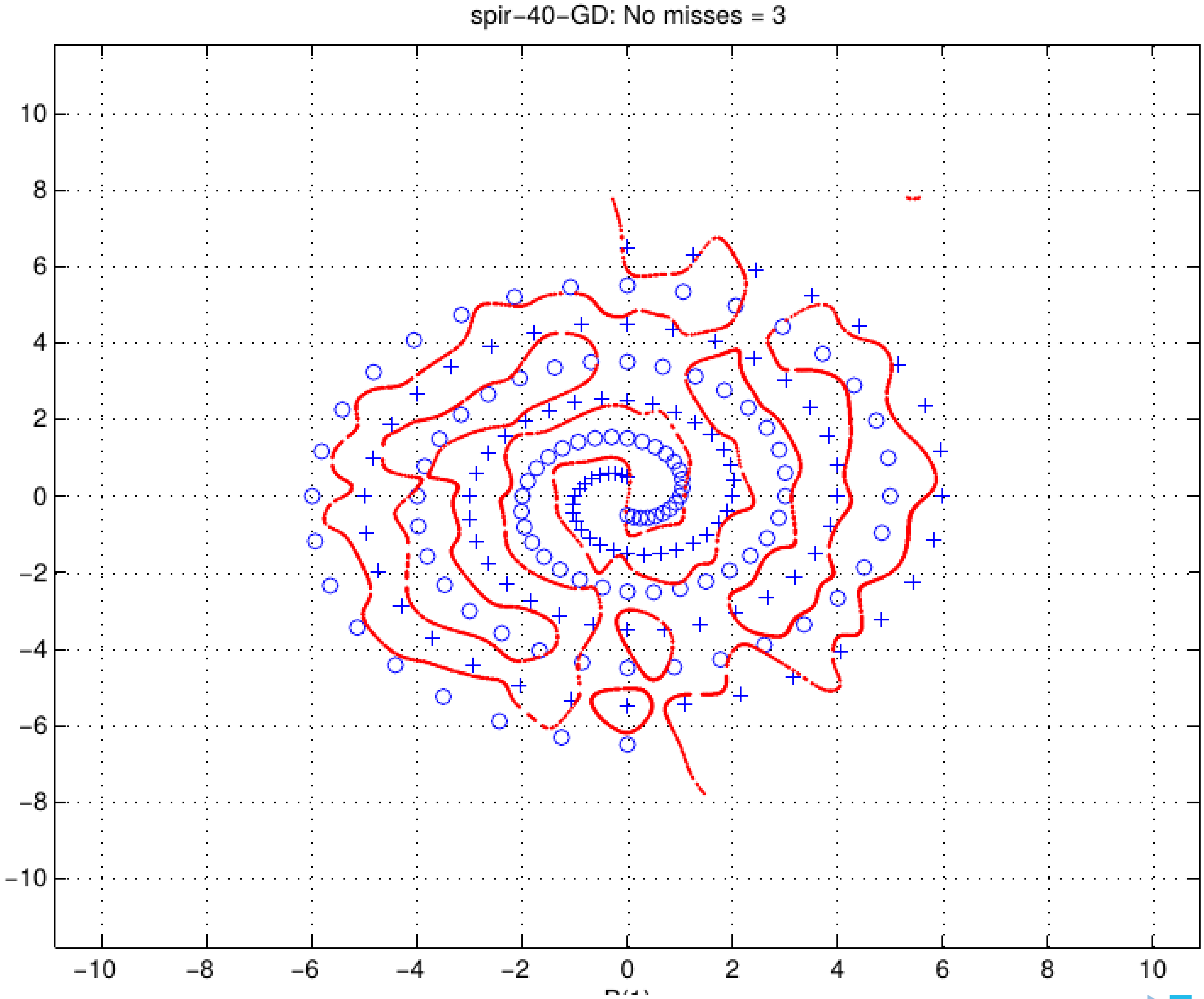
HALMSTAD UNIVERSITY

MLP: 2 – 10 – 1

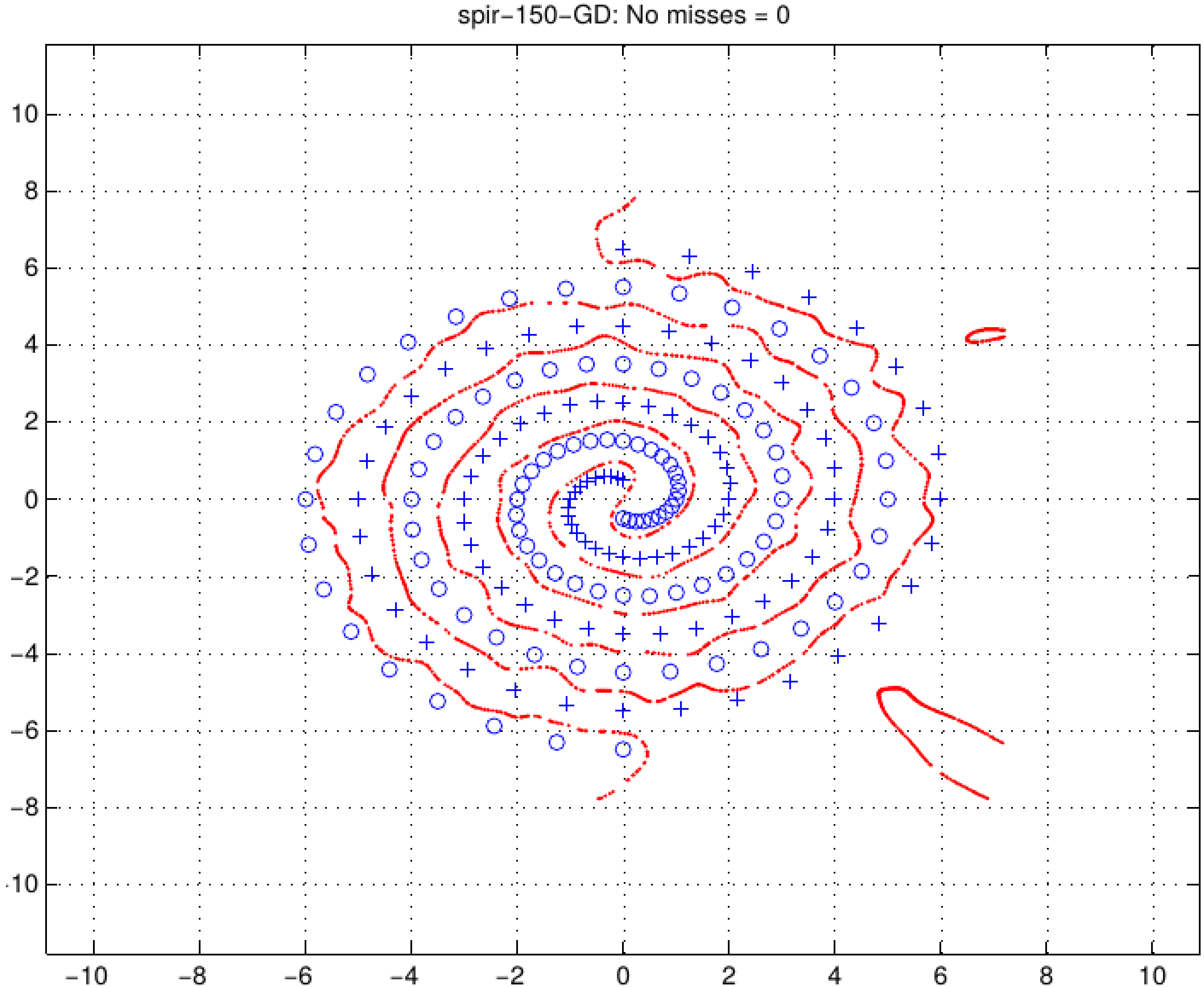


HALMSTAD UNIVERSITY

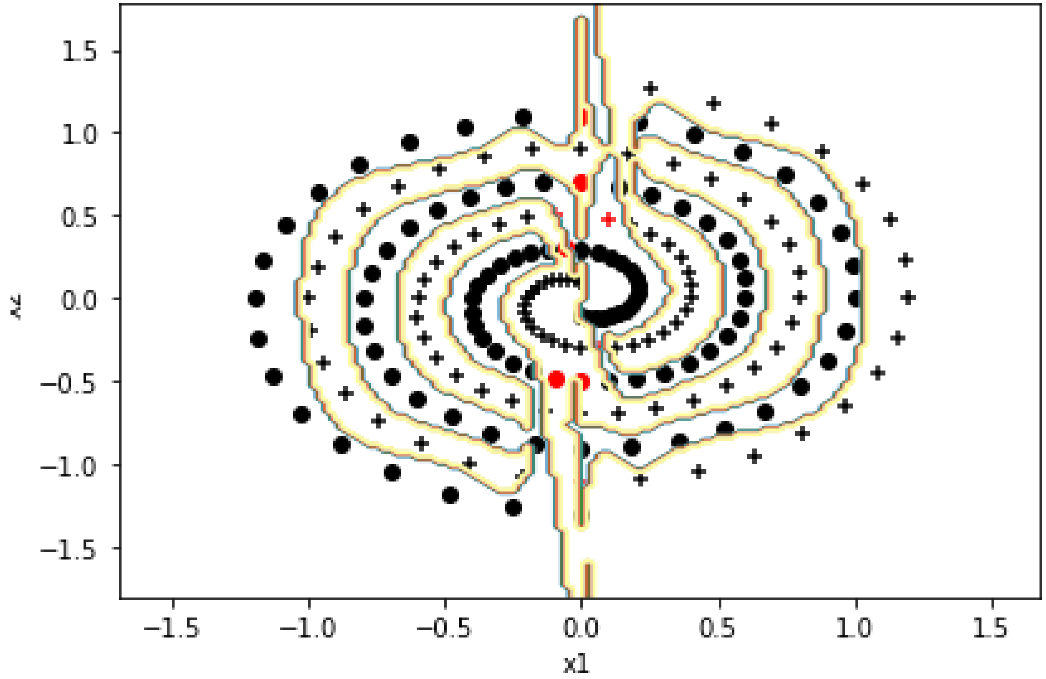
MLP: 2 – 40 – 1



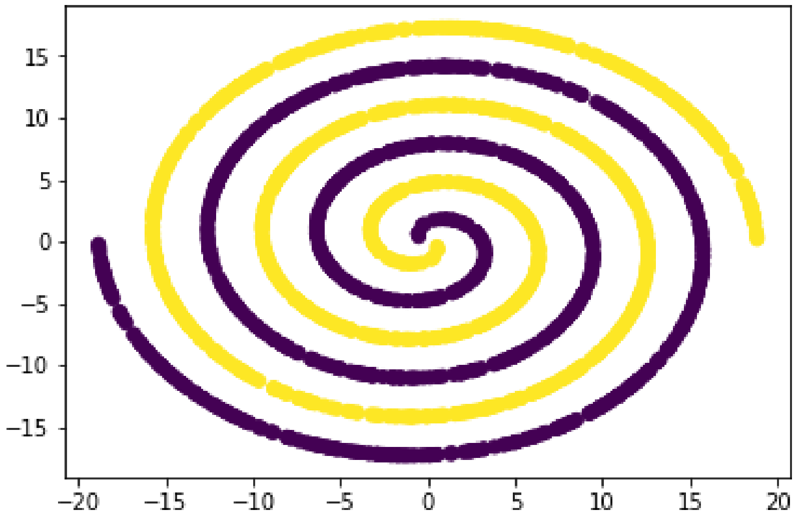
MLP: 2 – 150 – 1

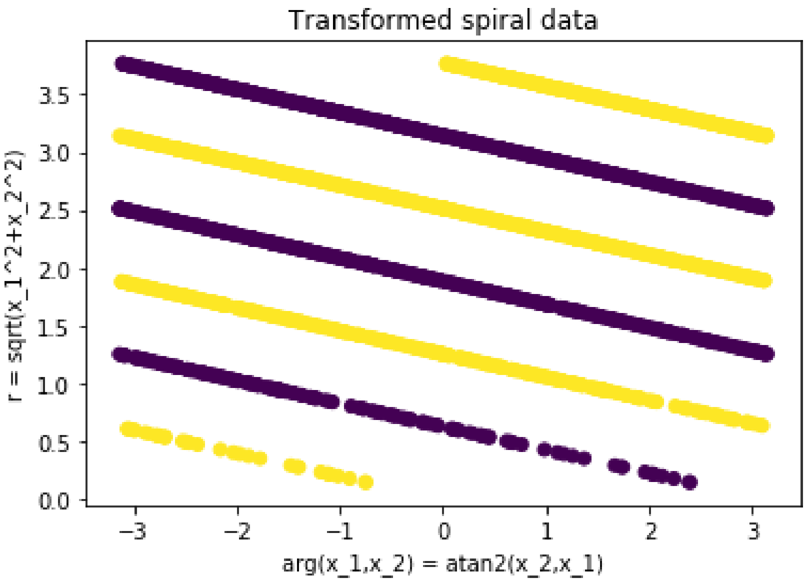


MLP: 2 – 5 – 5 – 5 – 1



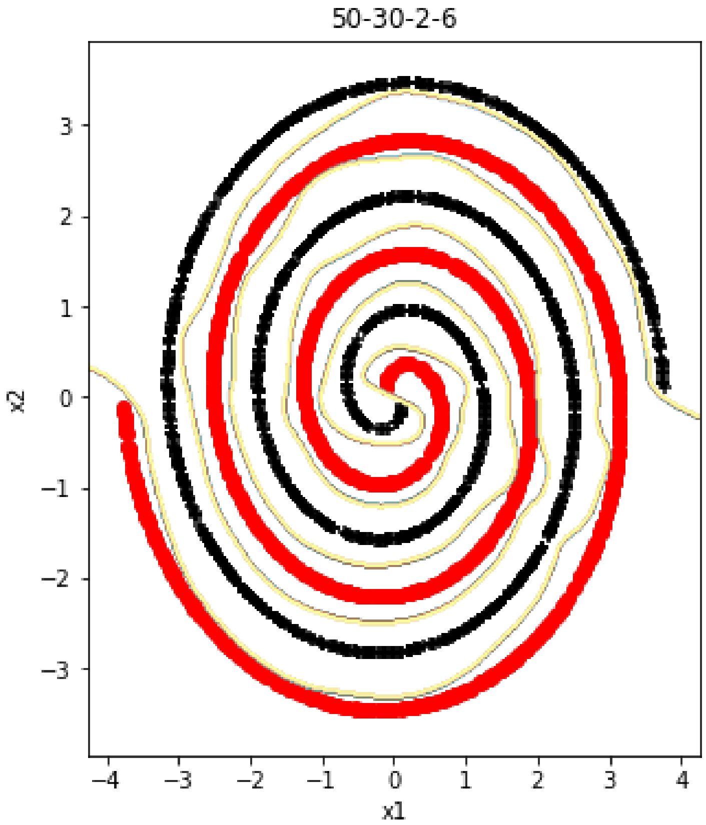
Pre-processing helps



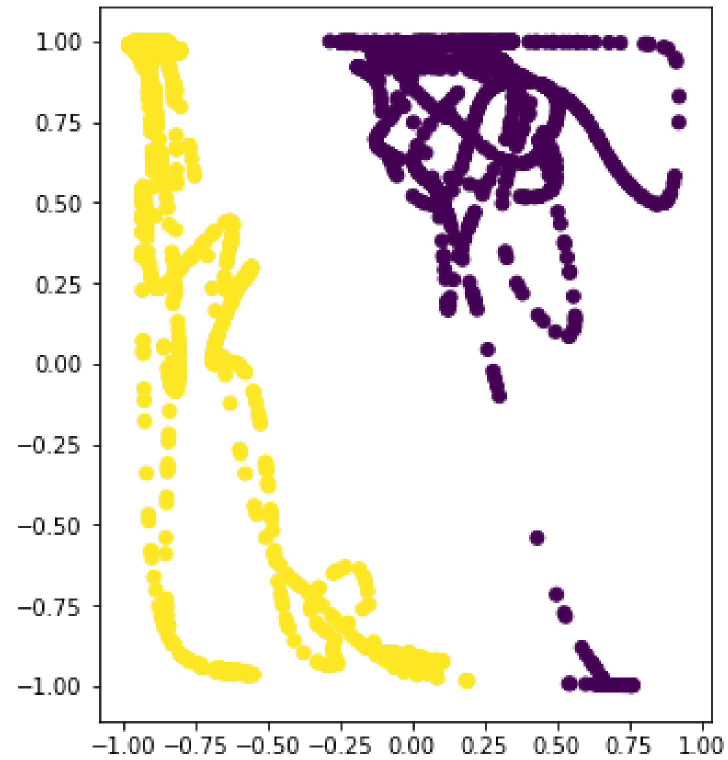


This can be solved by an MLP: 2 – 6 – 1

MLP: 2 – 50 – 30 – 2 – 6 – 1

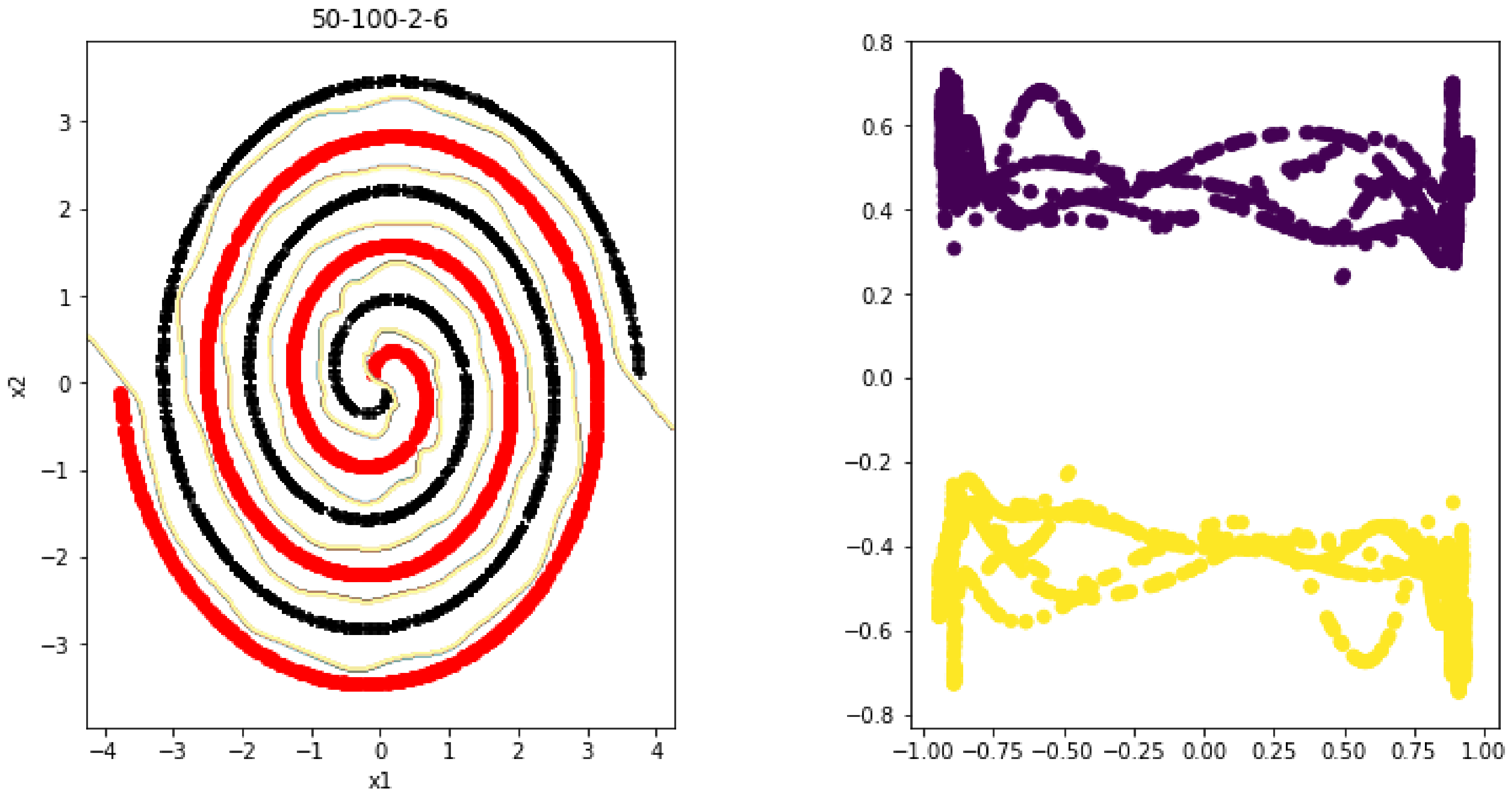


Third hidden layer values

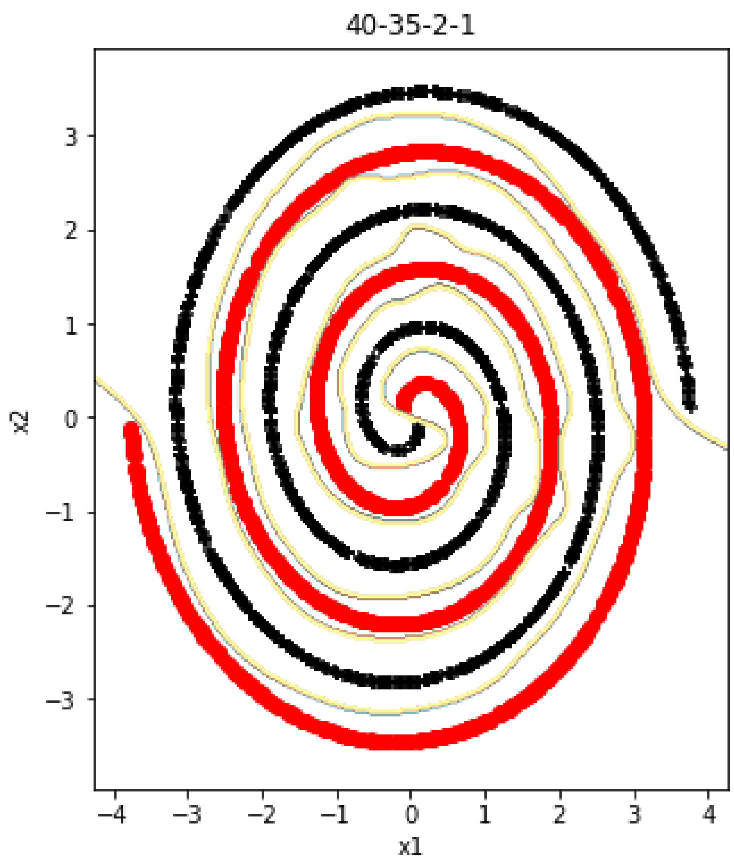


MLP: 2 – 50 – 100 – 2 – 6 – 1

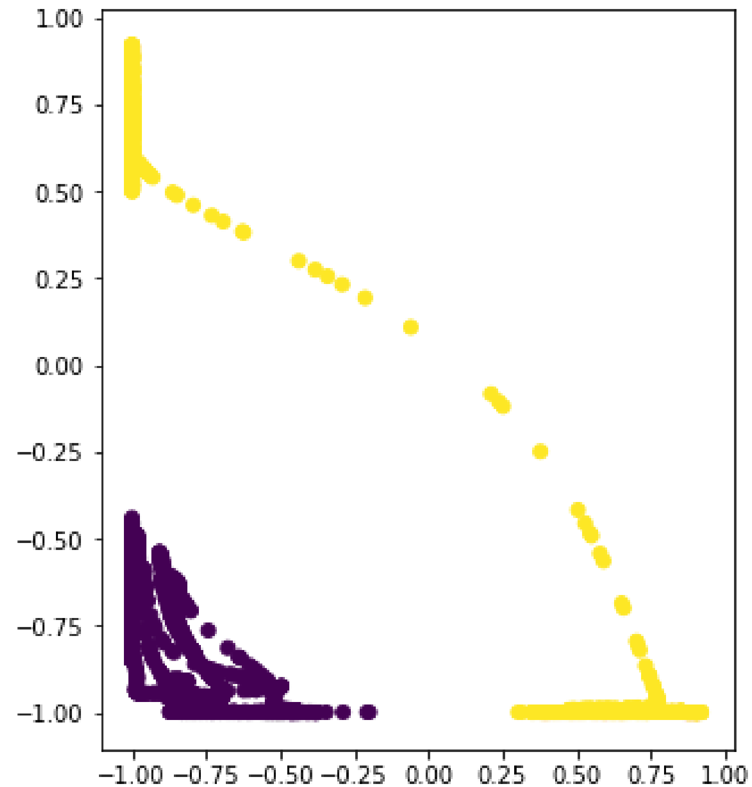
Third hidden layer values



MLP: 2 – 40 – 35 – 2 – 1 – 1



Third hidden layer values



More on regularization

L2 and L1 regularization

$$\tilde{E}(\mathbf{w}, \alpha) = E(\mathbf{w}) + \alpha\Omega$$

L2 norm (weight decay)

$$\Omega = \frac{1}{2}\sum_i \omega_i^2$$

L1 norm (lasso)

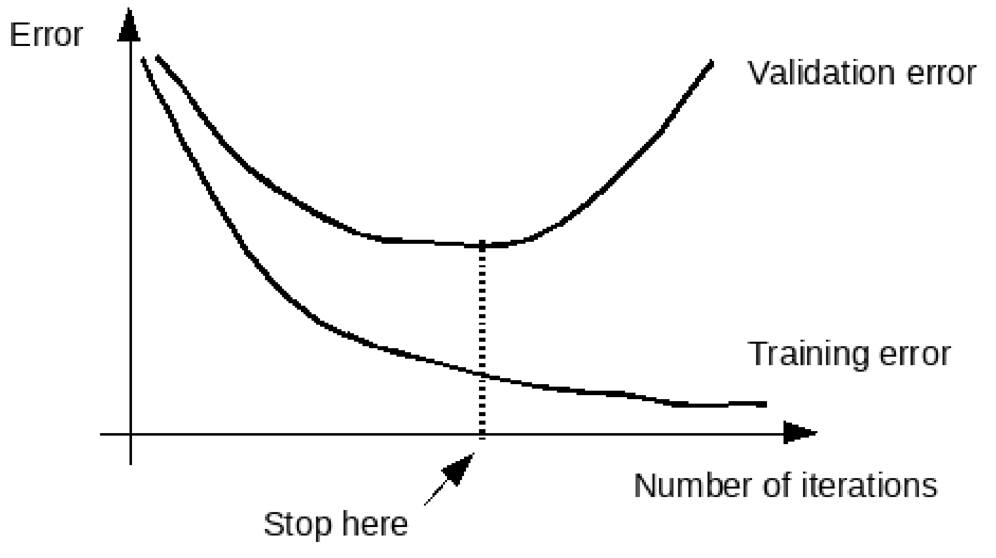
$$\Omega = \frac{1}{2}\sum_i |\omega_i|$$

Modified L2 norm

$$\Omega = \frac{1}{2}\sum_i \frac{(\omega_i/\omega_o)^2}{1 + (\omega_i/\omega_o)^2}$$

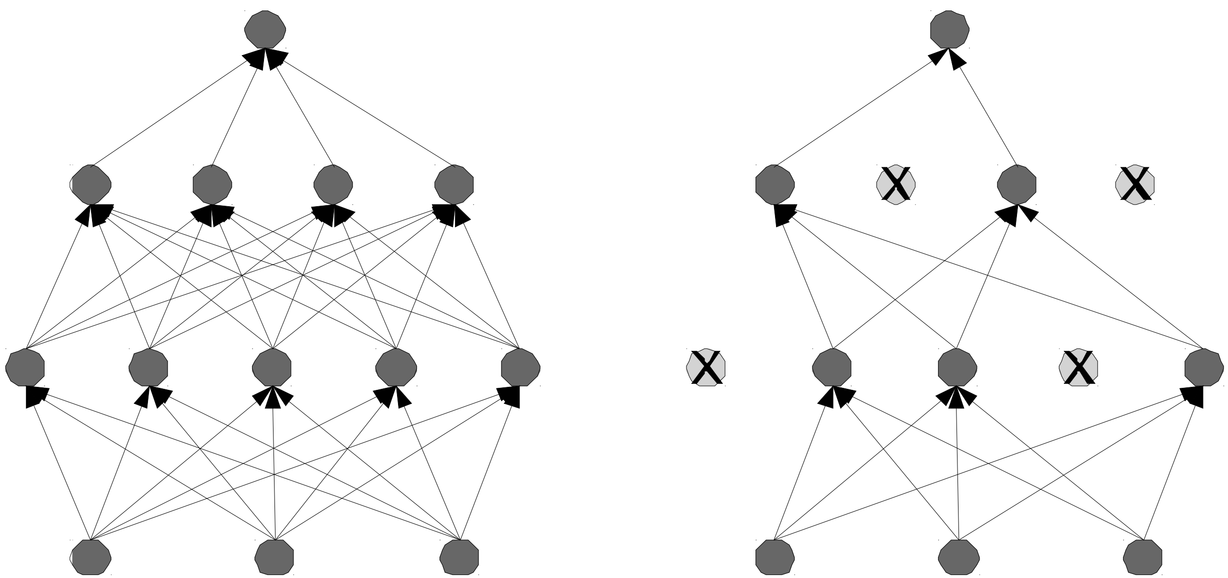
Early stop

Another alternative to regularization as a way of controlling the complexity of a network is the procedure of *early stopping*.



If we estimate the generalization error using a separate validation set we can stop the training when the validation error starts to increase.

Dropout regularization



Other regularization techniques
and ways of avoiding overfitting:

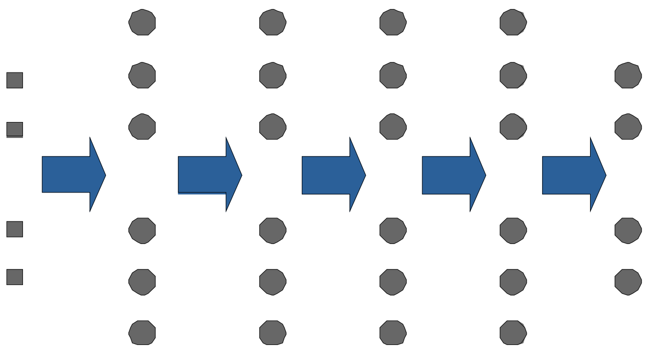
- Ensemble techniques
- Data augmentation
- ...

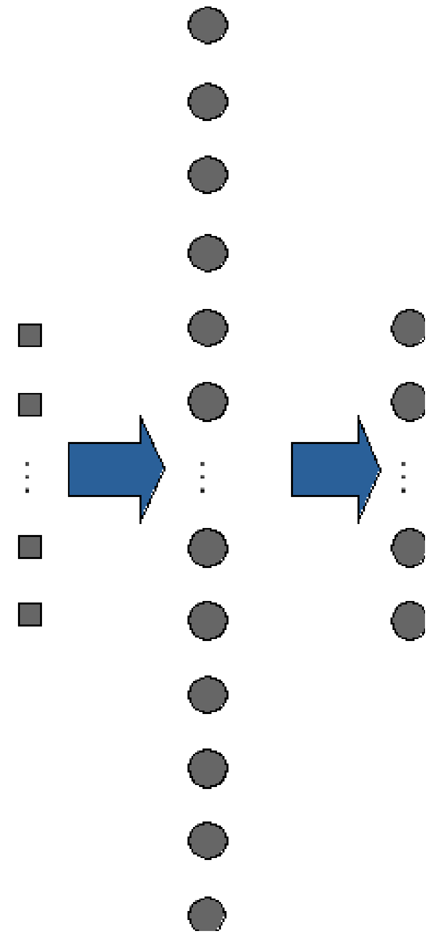
Some final words on deep MLPs

Deep MLPs

rather than this?

Why this

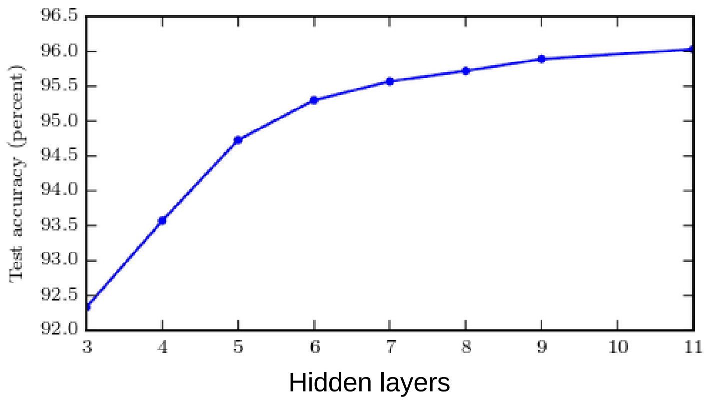


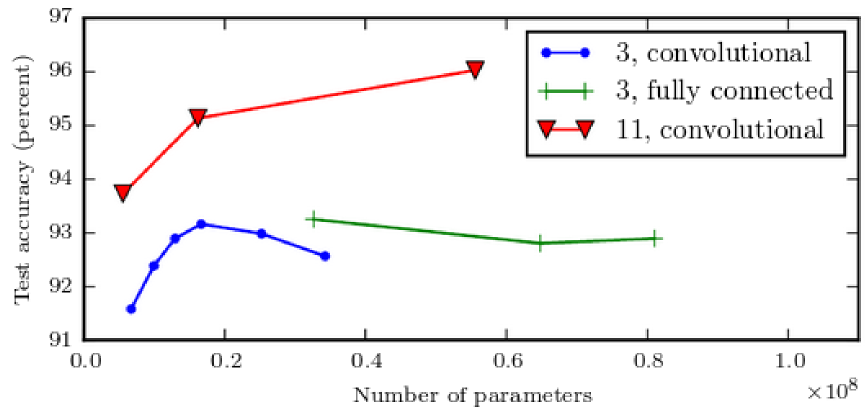


Deep MLPs

Empirically, deeper seems to give better generalization

(Image classification experiments)



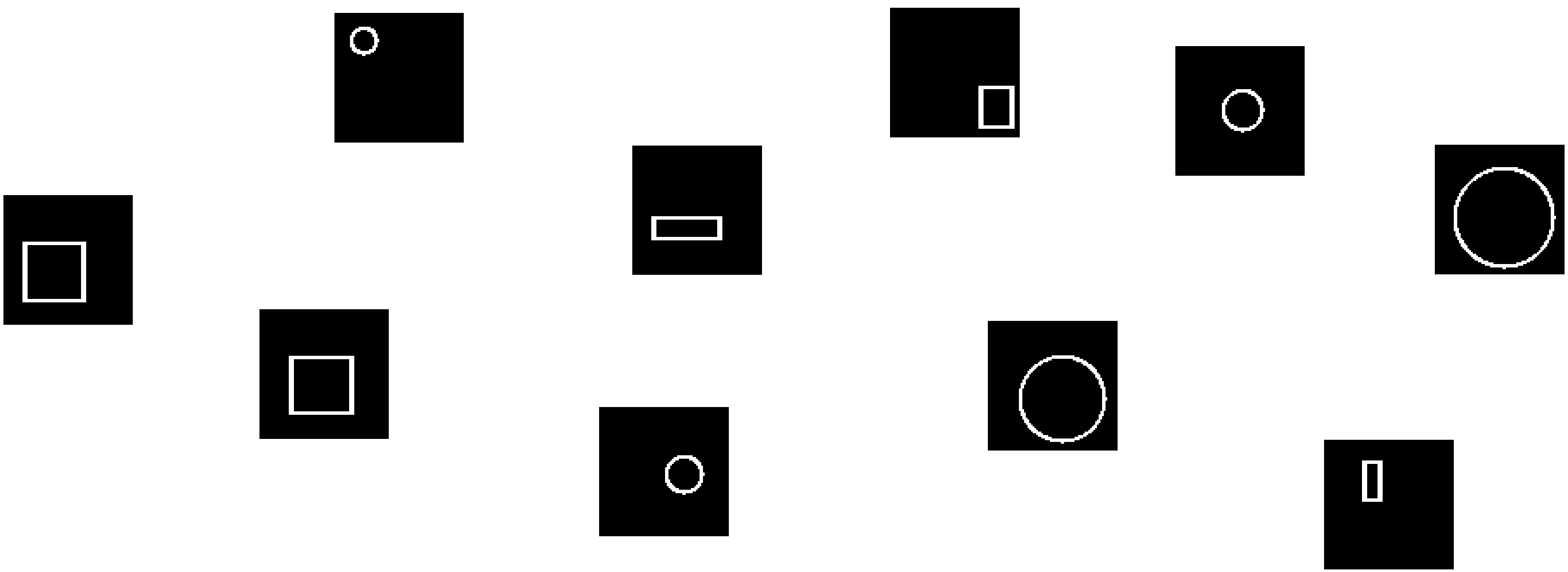


(From the deep learning book:
https://www.deeplearningbook.org/contents/mlp.html)

Deep learning = Representation learning

Rectangle or circle?

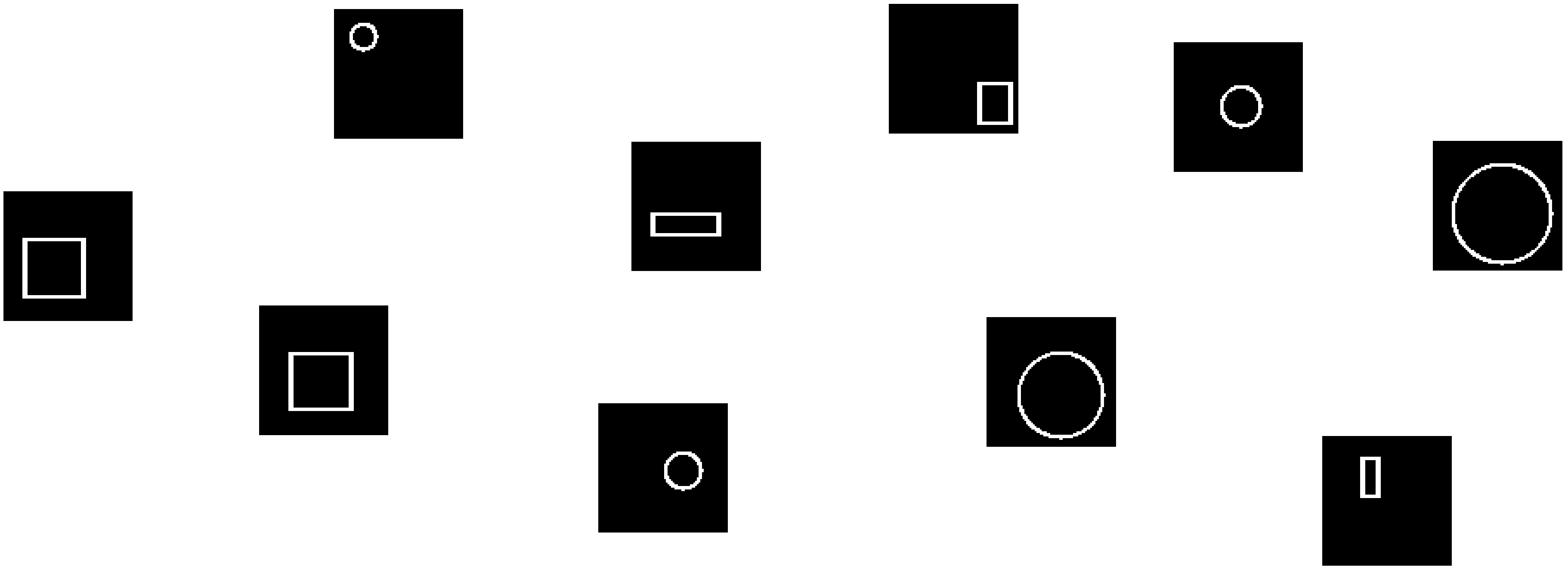
Which representation to use?

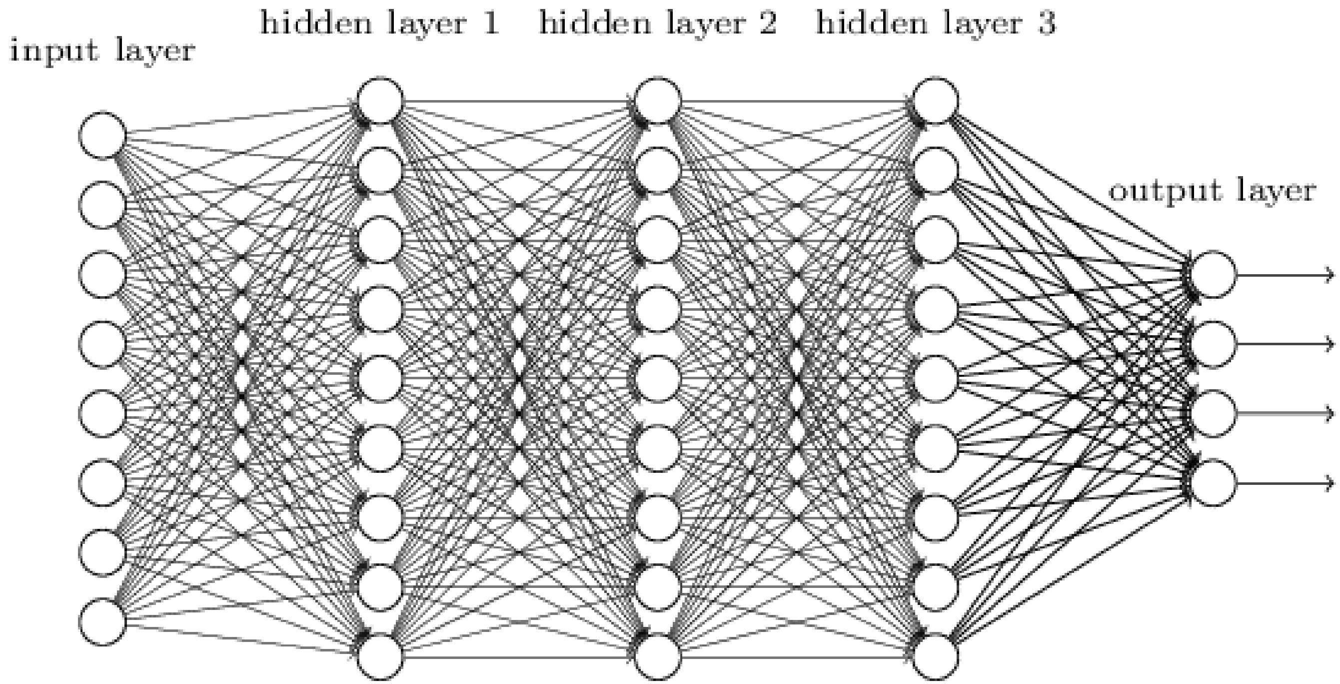


Deep learning = Representation learning

Small or large area?

Which representation to use?





Layer 1 | Layer2 | ... | Layer N

Simple features

.. building more complex features

.. building the “optimal” features for this specific task

Feature learning

Feature learning for “CNNs”