Lecture 5

Neural Networks

G. Englebienne
M. Poel

University of Twente

Linear and Non-linear models

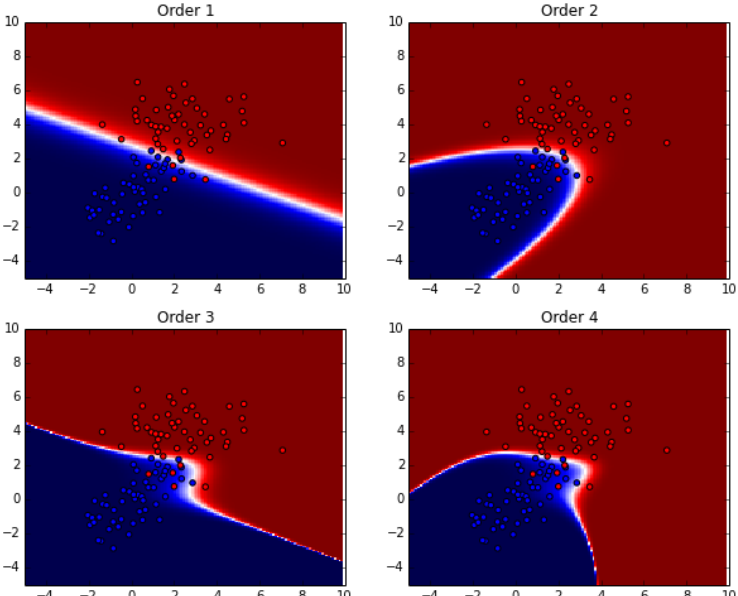
Linear models:

- Easy to train Θ
- Simple, tend to generalise well
- Too simple, may not fit the problem

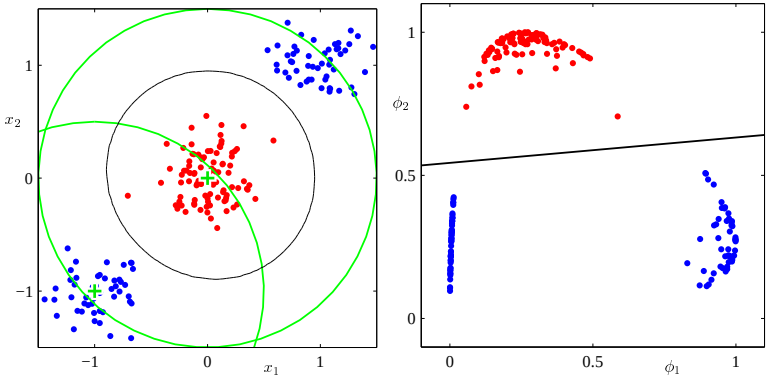
Extending models to non-linear

- Non-linear in function of observations
- Linear in function of parameters

$$y(\mathbf{x}, \mathbf{w}) \rightarrow y(\phi(\mathbf{x}), \mathbf{w})$$


Order 1
Order 2
Order 3
Order 4

Basis Functions



Adaptive Basis Functions

Today we look at a technique to find the basis functions automatically

- (Artificial) Neural Networks
- Inspired from biology (neurons)
  - Their biological plausibility has often been exaggerated
  - Nevertheless some of the problems they have are also shown by biological systems (*e.g.* Moiré effect)
  - Being biologically implausible does not affect the usefulness as artificial learning systems
- Based on the perceptron (cf. lecture 2)

Perceptrons

Perceptrons:

- Output: step function of linear combination of inputs

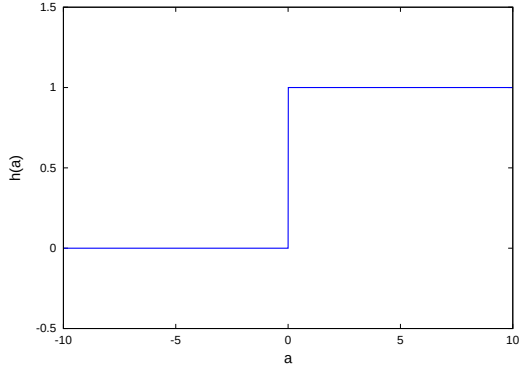
$$y(x) = h(\mathbf{w}^\top \mathbf{x})$$

- Step function $y(\cdot) \Rightarrow$ non-linear
- Multiple layers would make complex functions possible
  - non-linear functions of non-linear functions
- Training of single layer is problematic
  - Convergence
  - non-separable training data
  - Solution depends on initialisation
- Training of multiple layers would be next to impossible

By using a differentiable activation function, we can make training much easier

- For example: logistic activation function:

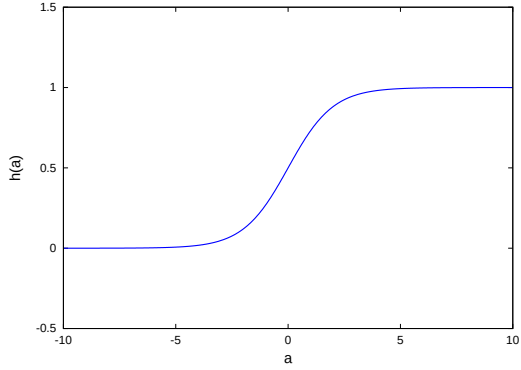
$$\sigma(a) = \frac{1}{1 + \exp(-a)}$$



By using a differentiable activation function, we can make training much easier

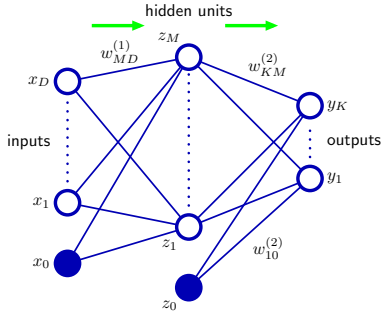
- For example: logistic activation function:

$$\sigma(a) = \frac{1}{1 + \exp(-a)}$$



Multi-layer perceptrons

With a clever application of the chain rule of derivations we can combine multiple layers and still train the network.



- Multi-layer perceptrons (MLP) — not really perceptrons at all

Architecture

The architecture is constrained

- In order to be trainable, a *feed-forward* architecture is required
- Can be sparse
- Can have skip-layer connections

This is clearly much more constrained than biological neural networks

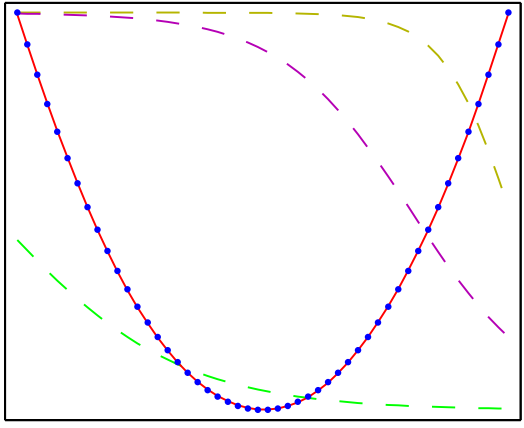
Combining two layers results in function of the form

$$y_k(\mathbf{x}, \mathbf{w}) = h_2\left(\sum_{j=0}^{M} w_{kj}^{(2)} h_1\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)\right)$$

- The combined, weighted non-linearities make very complex functions possible
- A two-layer network with "linear" output activation function can approximate any continuous function within a compact domain with arbitrary precision
  - If the hidden layer has sufficient units
  - Holds for many activation functions of the hidden units (but not polynomials)

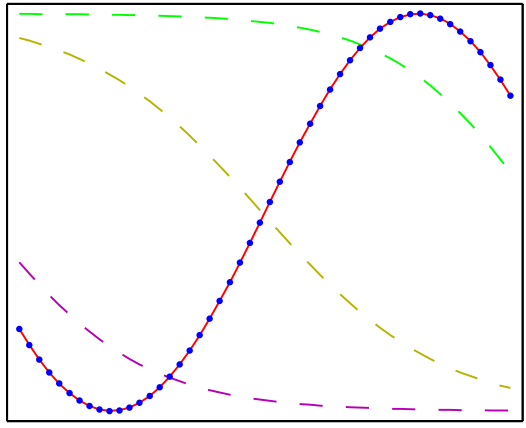
Universal Function Approximators

Example: 3 hidden units and tanh activation



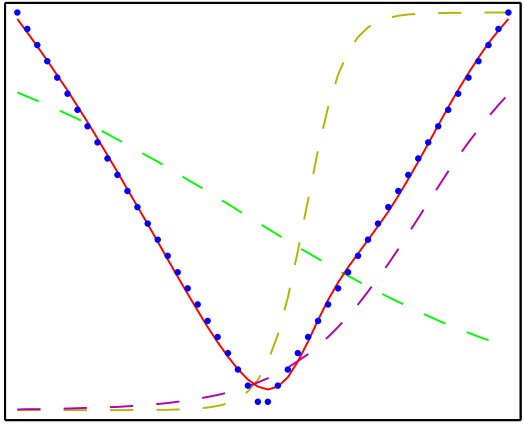
Universal Function Approximators

Example: 3 hidden units and tanh activation



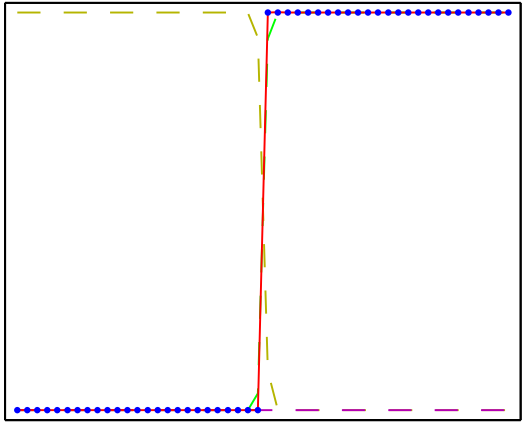
Universal Function Approximators

Example: 3 hidden units and tanh activation



Universal Function Approximators

Example: 3 hidden units and tanh activation



Training methods

Choose an error function E and adapt the parameters in order to minimise it.

- Strongly non-linear, with many optima
  - No closed-form solution for the parameters
  - Numerical, iterative procedure
- Efficient gradient-based methods (Gradient Descent, Quasi-Newton, Adam, ...)
- Stochastic gradient descent has advantages over batch methods:
  - More efficient at handling redundancy
  - Escapes local minima more easily
- So how do we compute the gradient?

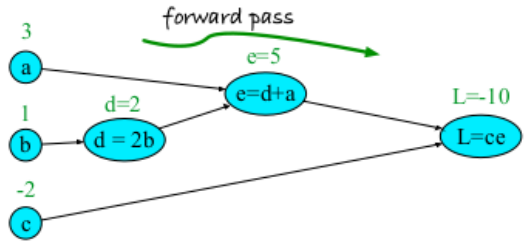
Computational graphs

Consider computing the function:

$$L(a, b, c) = c(a + 2b)$$

We can decompose this into temporary variables and compute:

$$d = 2\,b \qquad e = a + d \qquad L = c\,e$$



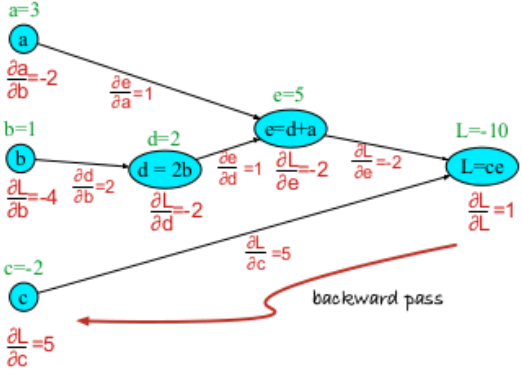
Chain rule of derivatives

Consider the function

$$f(x) = u(v(w(x)))$$

the derivative of f with respect to x can then be decomposed into:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx}$$



Backpropagation

Backpropagation works in two passes:

Forward pass : computing the activations of the hidden and output units.

Backward pass : computing the gradients of the error function

In a feed-forward network, each node computes

$$a_j = \sum_i w_{ji} z_i, \tag{1}$$

which is transformed by an activation function, so that

$$z_j = h(a_j)$$

For each input $\mathbf{x}_n$ in the training set, we have an associated target t_n and corresponding error E_n. The partial derivative of the error with respect to a weight w_{ji} can be decomposed using the chain rule:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$

From (1) we have $\frac{\partial a_j}{\partial w_{ji}} = z_i$ and we introduce $\delta_j \triangleq \frac{\partial E_n}{\partial a_j}$ so that:

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i \tag{2}$$

If we choose the sum-of-squared error function (summing over k outputs)

$$E_n = \frac{1}{2}\sum_k (y_{nk} - t_{nk})^2$$

with $y_{nk} = \mathbf{w}^\top \mathbf{z}_n = \sum_j w_{kj} z_{nj}$, the gradient $\frac{\partial E_n}{\partial w_{kj}} = (y_{nk} - t_{nk}) z_{nj}$, so that

$$\delta_k \triangleq \frac{\partial E_n}{\partial a_k} = y_k - t_k$$

We can then compute the derivative with respect to the previous layer as:

$$\delta_j \triangleq \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

where $a_k = \sum_j w_{jk} h(a_j)$, so that for a single node j

$$\delta_j = h'(a_j) \sum_k w_{jk} \delta_k \tag{3}$$

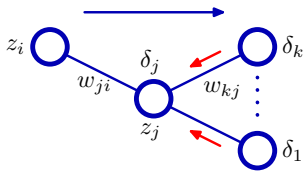
We can then compute the derivative with respect to the previous layer as:

$$\delta_j \triangleq \frac{\partial E_n}{\partial a_j} = \sum_k \delta_k \frac{\partial a_k}{\partial a_j}$$

where $a_k = \sum_j w_{jk} h(a_j)$, so that for a single node j

$$\delta_j = h'(a_j) \sum_k w_{jk} \delta_k \tag{3}$$

Summary



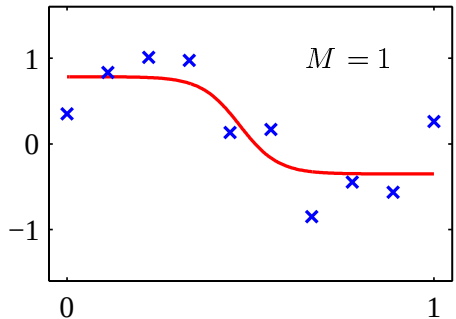
Error Backpropagation

1. Forward propagate an input vector $\mathbf{x}_n$ to find the activations for the hidden units
2. Evaluate δ_k for all output units
3. Backpropagate the δ_k using (3) to obtain δ_j for all hidden units
4. Use (2) to find the derivatives with respect to the weights

- Backpropagation can also be used to compute other derivatives of the error function, second derivatives, ...
- In practice, it is easy and useful to check the validity of an implementation using the method of finite differences.

The number of input and output units is generally imposed by the problem, but the number of hidden units may vary

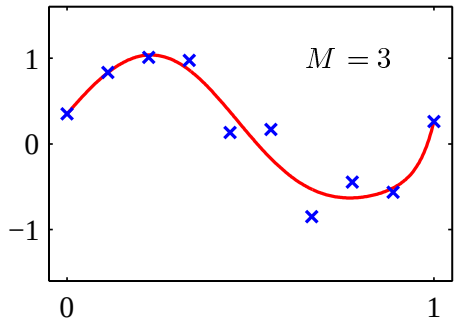
Example



Regularisation

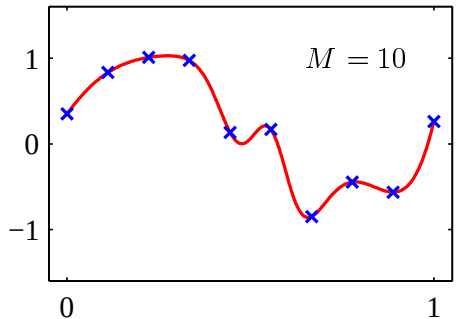
The number of input and output units is generally imposed by the problem, but the number of hidden units may vary

Example



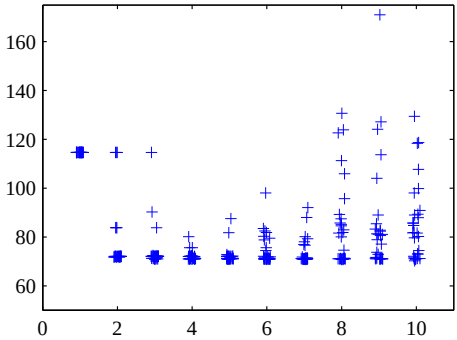
The number of input and output units is generally imposed by the problem, but the number of hidden units may vary

Example



Yet the generalisation performance is not a simple function of M

Example: error on left-out data

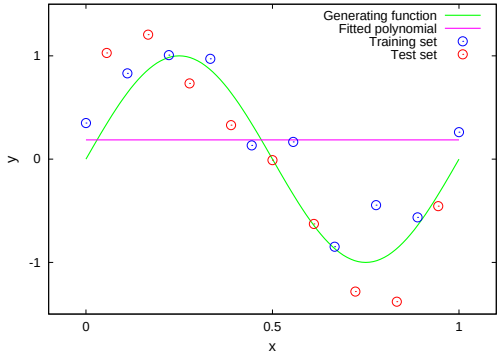


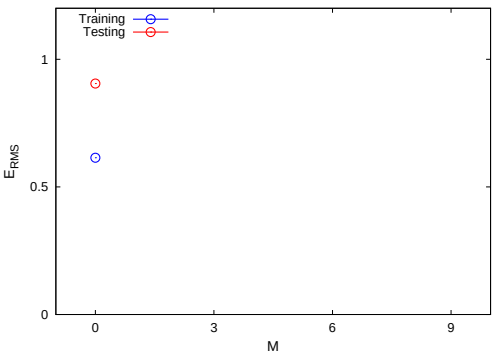
- 30 random starts per size
- Initial weights sampled from a Gaussian distribution

In this particular case, the lowest validation error was for $M = 8$

Weight decay and Overfitting

Example

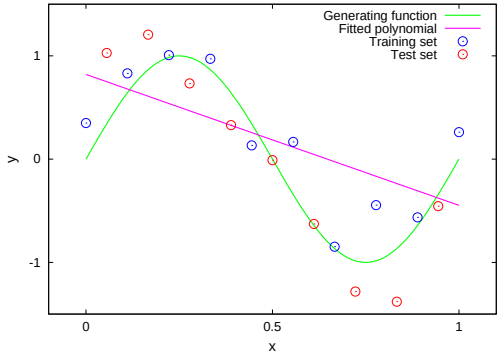


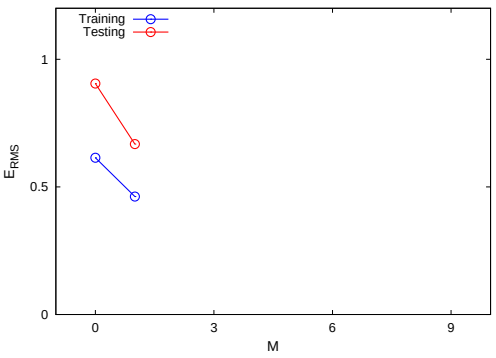


$M = 0$

Weight decay and Overfitting

Example

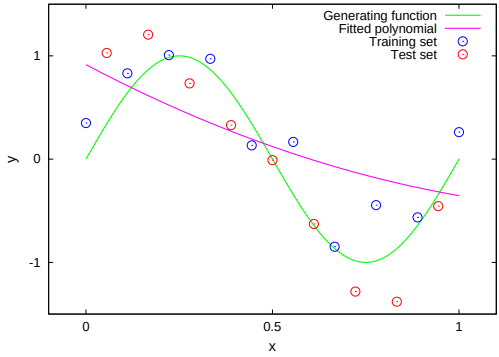


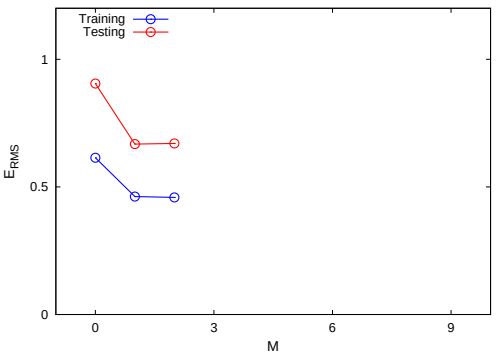


$M = 1$

Weight decay and Overfitting

Example

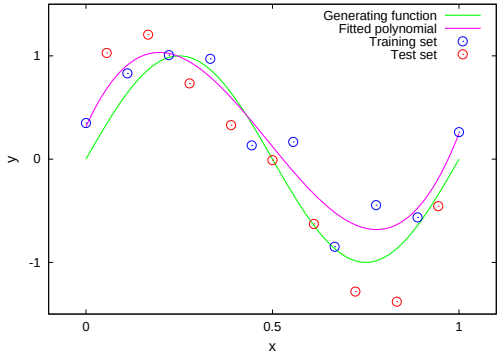


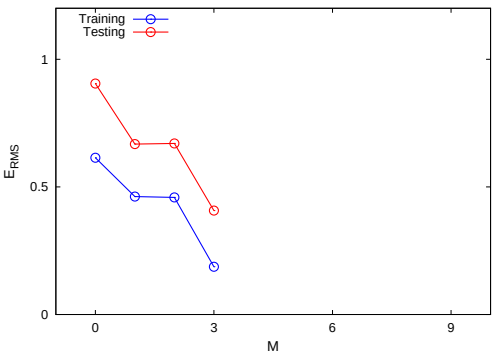


$M = 2$

Weight decay and Overfitting

Example

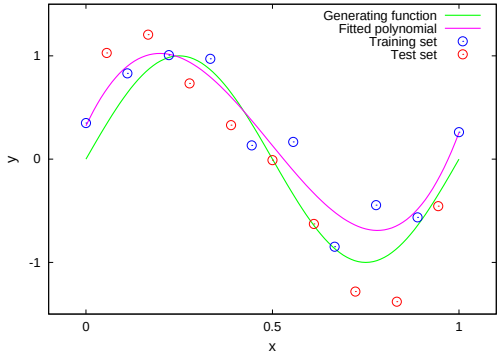


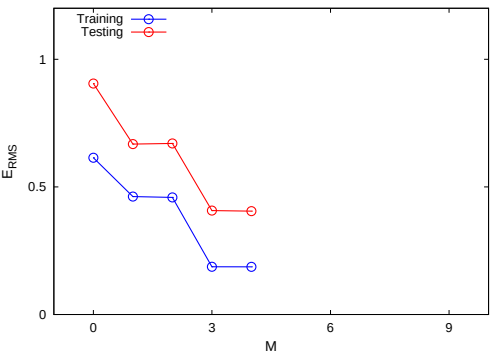


$M = 3$

Weight decay and Overfitting

Example

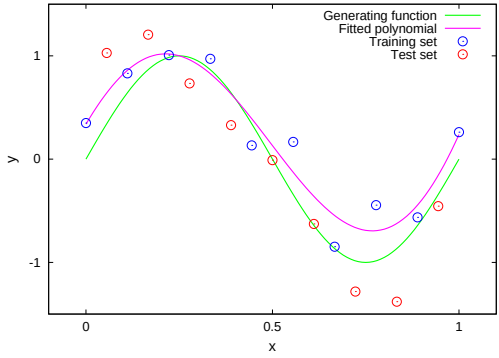


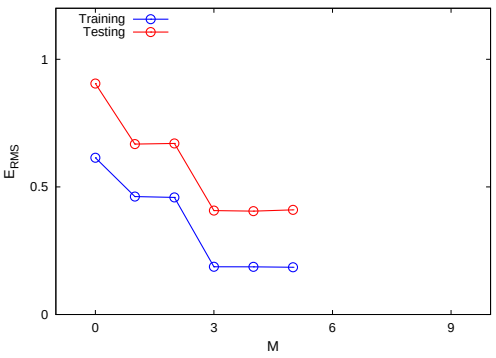


$M = 4$

Weight decay and Overfitting

Example

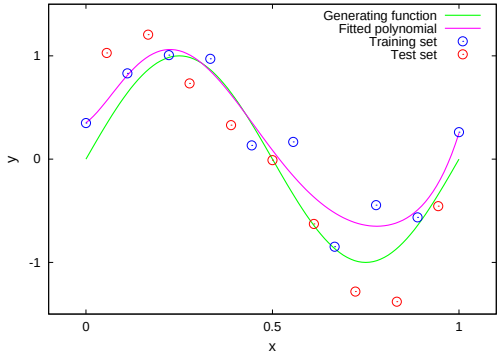


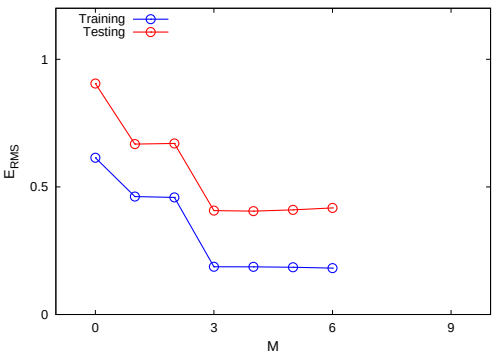


$M = 5$

Weight decay and Overfitting

Example

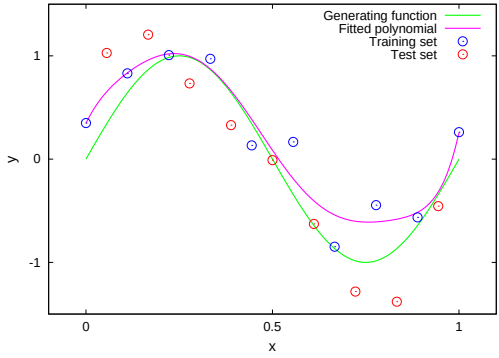


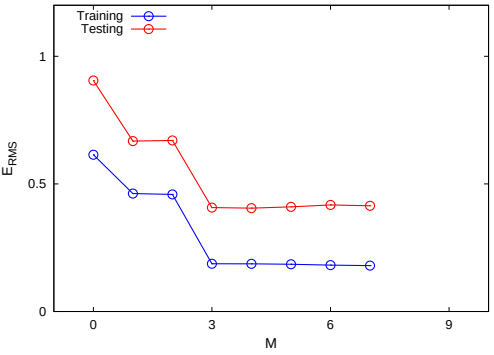


$M = 6$

Weight decay and Overfitting

Example

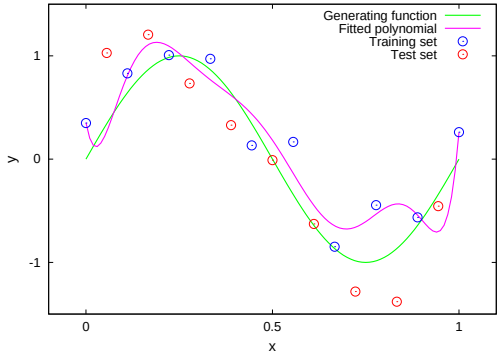


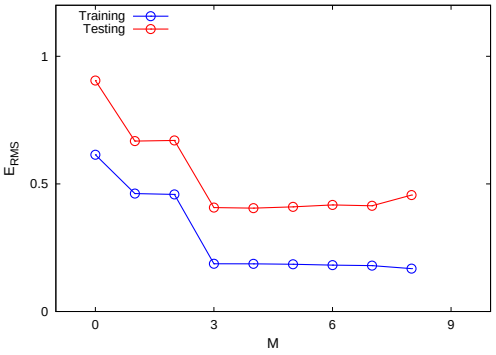


$M = 7$

Weight decay and Overfitting

Example

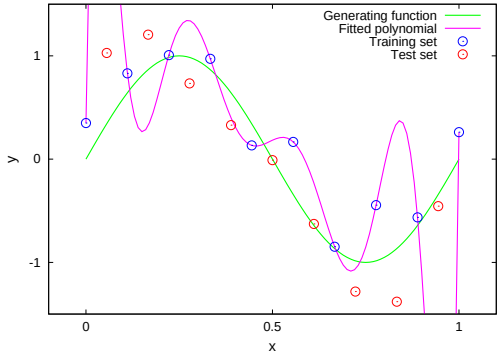


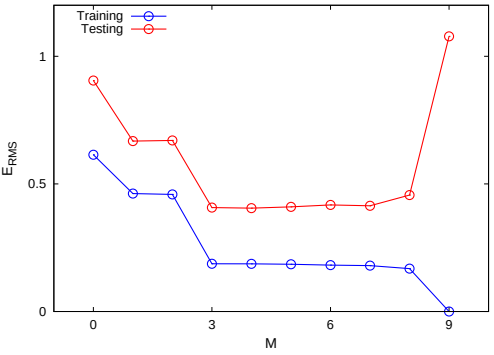


$M = 8$

Weight decay and Overfitting

Example





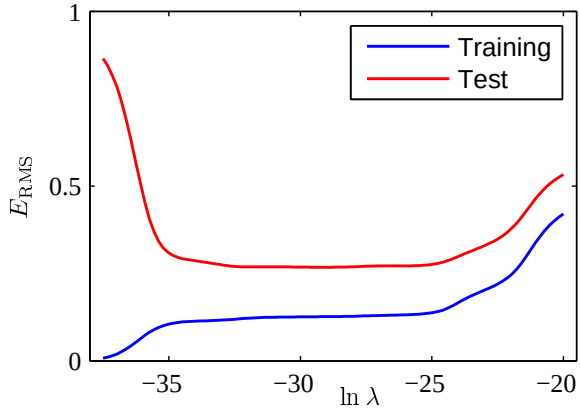
$M = 9$

Weights and overfitting

	$M = 0$	$M = 1$	$M = 3$	$M = 6$	$M = 9$
w_0	0.19	0.82	0.31	0.35	0.35
w_1		-1.27	7.99	2.62	232.37
w_2			-25.43	32.10	-5321.79
w_3			17.37	-206.27	48568.00
w_4				399.00	-231637.92
w_5				-332.71	640038.66
w_6				105.16	-1061794.80
w_7					1042394.73
w_8					-557680.13
w_9					125200.80

Parameter shrinkage

Example: Polynomial curve fitting



Weight decay

Again, the traditional technique: penalise large weights

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^\top \mathbf{w},$$

which can be interpreted as the negative logarithm of a zero-mean Gaussian prior over the weights

Problem: if we should do a linear transformation of the data and train a new network on the transformed data, we should obtain an equivalent network (with linearly transformed input weights)

- Weight decay treats all weights equally (biases included)
- It does therefore not satisfy this property

Solution: Treat the weights of each layer separately, and do not constrain the biases

$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda_1}{2} \sum_{w \in \mathcal{W}_1} w^2 + \frac{\lambda_2}{2} \sum_{w \in \mathcal{W}_2} w^2$$

Weight symmetries and comparing models

Multiple parameter values result in equivalent networks:

- If $h(a)$ is odd (*e.g.* hyperbolic tangent tanh, ...)

$$h(-a) = -h(a),$$

  changing the sign of all weights leading into a node and all weights leading out of that node
- Exchanging all weights of a hidden node with all weights of another node in the same layer
- In total: $M!2^M$ symmetries
- Little importance in practice (but see later)
- Complex, non-linear function — local optima

Improper priors

The split regularisation term also corresponds to a prior over the weights:

$$p(\mathbf{w}|\lambda_1, \lambda_2) \propto \exp\left(\frac{\lambda_1}{2}\sum_{w\in\mathcal{W}_1} w^2 + \frac{\lambda_2}{2}\sum_{w\in\mathcal{W}_2} w^2\right)$$

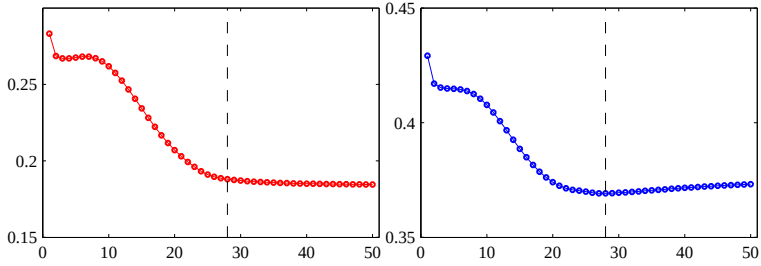
but these are *improper* because the bias parameters are unconstrained.

- It is therefore customary to add separate priors over the bias parameters
- We can generalise this and consider priors over arbitrary groups of parameters

Early Stopping

An alternative is to stop training when things get worse

Example



This is similar to weight decay: if we start from the origin, stopping early restricts the weights to small values

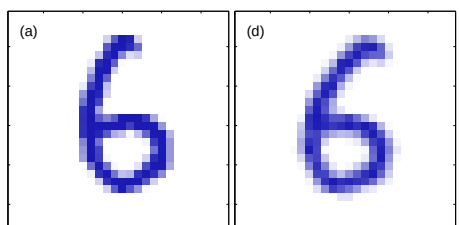
Input Invariance

MLP are extremely flexible

- In a way, we're doing automatic feature extraction and regression/classification at the same time
- Overfitting is a problem

Often, however, we know what aspects of the data do not matter

Digit example: Translation/Rotation



We would like to find ways to force the MLP to be invariant to those variations, without discarding valuable information

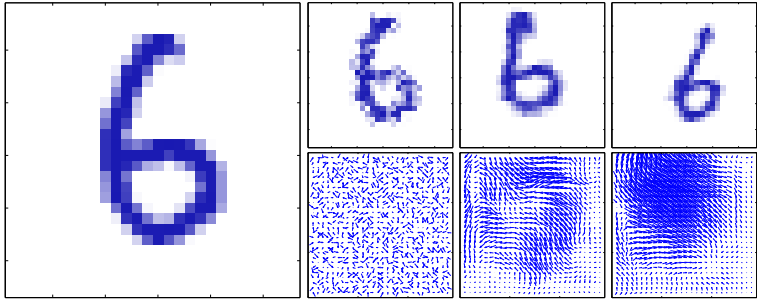
Encouraging invariance

Approaches to encourage the model to be invariant to certain transformations

1. Augment training set with modified patterns with desired invariances
2. Penalise changes in error function due to invariances (Tangent propagation)
3. Pre-process data: extract transformation-insensitive features
4. Build invariances into network structure

Augmenting the training set

Example

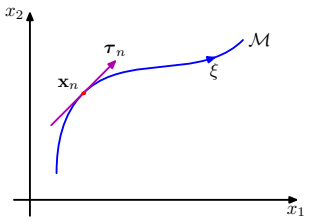


- Easy to implement
- Particularly appropriate for on-line (SGD) learning
  - Apply random transformation as we cycle through the data
- In the limit for infinite set of variations: equivalent with tangent propagation

Tangent propagation

In the case of continuous transformation, a transformed input pattern will result in a manifold $\mathcal{M}$ in the input space

Example



Suppose the transformation $\mathbf{s}$ is controlled by a single parameter ξ, and $\mathbf{s}(\mathbf{x}, 0) = \mathbf{x}$

We are interested in small variations $\Rightarrow$ approximate manifold with tangent vector

We want the error to be invariant to changes in ξ around the training data

Regularised error $\tilde{E} = E + \lambda\Omega$, where

$$\Omega = \frac{1}{2} \sum_n \sum_k \left(\frac{\partial y_k}{\partial \xi} \bigg|_{\xi=0} \right)^2$$

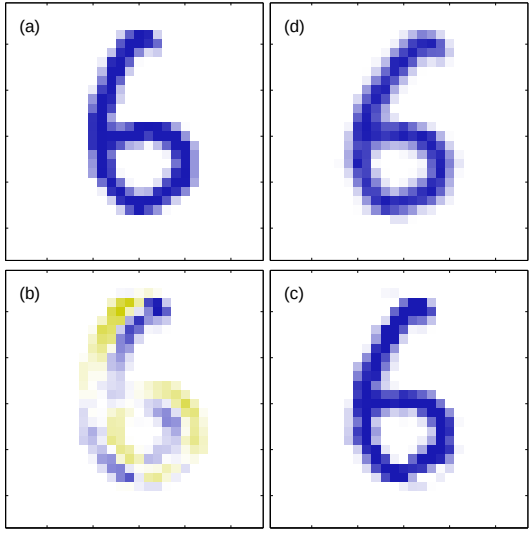
From the chain rule, we have

$$\left.\frac{\partial y_k}{\partial \xi}\right|_{\xi=0} = \sum_{i=1}^{D} \frac{\partial y_k}{\partial x_i} \left.\frac{\partial x_i}{\partial \xi}\right|_{\xi=0}$$

where

- $\frac{\partial y_k}{\partial x_i}$ is the so-called Jacobian and can easily be computed using back-propagation
- $\frac{\partial x_i}{\partial \xi}$ is often obtained numerically using finite differences

Example

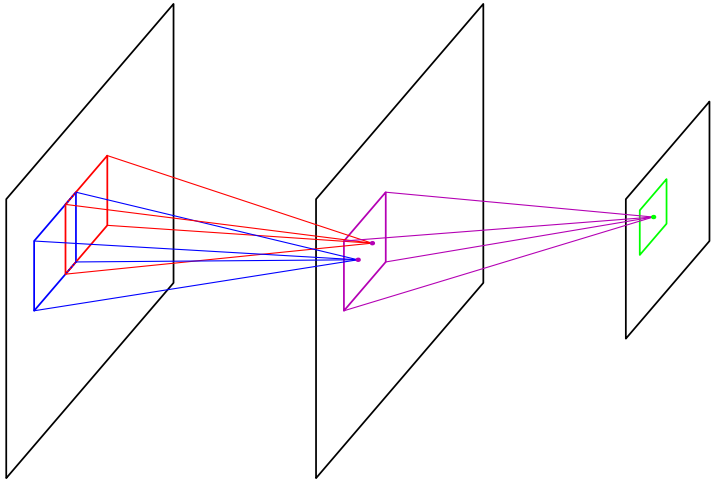


Fully connected neural networks can learn the right invariances given enough training data, however this still disregards aspects of the data

- Specifically, in images: nearby pixels are more strongly correlated
- In computer vision, this is often leveraged to extract local features from the image
- Features that are useful in one location are likely to be useful elsewhere, *e.g.* if an object was translated

These aspects are included in CNN through:

- Local receptive fields
- Weight sharing
- Subsampling



Input image

Convolutional layer

Sub-sampling layer

Convolutional Neural Networks

Local receptive fields:

- Only specific weights are non-zero

Weight Sharing:

- Force the weights to be identical over different fields
- Requires a simple adaptation of backpropagation

Subsampling:

- Combine 2×2 node grid from convolutional layer into a single node in subsampling layer
- Non-overlapping grids
- Introduces a degree of translation invariance

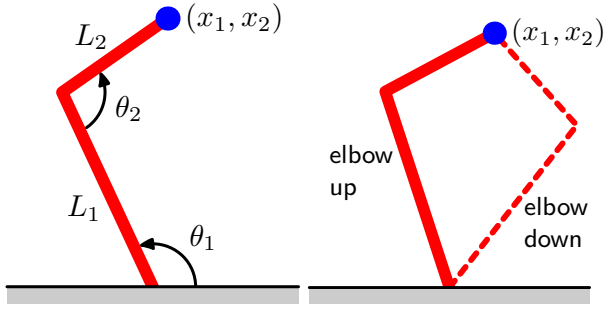
In practice:

- multiple iterations of convolution and subsampling
- End layer typically fully connected with softmax output

Minimising the sum-squared-error is equivalent with assuming Gaussian noise on the output

- This is not always a valid assumption
- In particular, we often want to solve "inverse problems"

Example



Mixture of Density Networks

We therefore assume a mixture of Gaussians for the output noise, and let the network learn the parameters of the mixture

$$p(\mathbf{t}|\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{t} \mid \mu_k(\mathbf{x}), \sigma_k^2(\mathbf{x}))$$

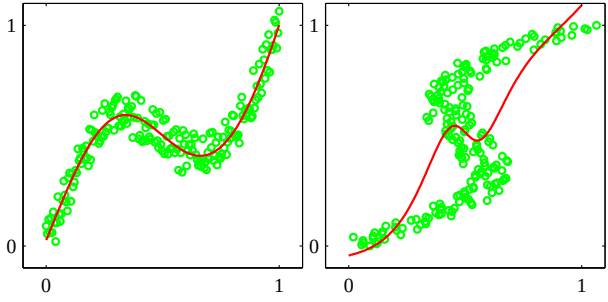
We enforce the constraints with our selection of output activation functions:

- $\sum_k \pi_k = 1$: use softmax

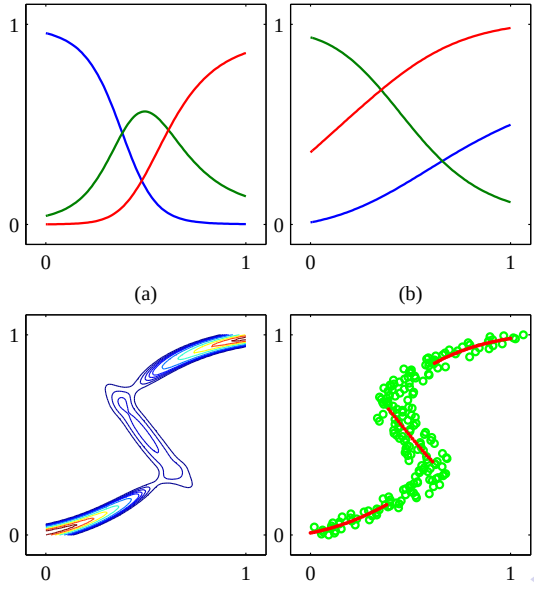
$$\pi_k(\mathbf{x}) = \frac{\exp(a_k^\pi)}{\sum_{l=1}^{K} \exp(a_l^\pi)}$$

- $\sigma_k(\mathbf{x}) \geqslant 0$: use exponentials
- $\mu_k(\mathbf{x})$ can have any real value: use linear activation function

Example



Example



Wrap up

Today, we've seen MLPs:

- General description and uses (Bishop, p. 225-232)
- Backpropagation (Bishop, p. 241-245)
- Regularisation and input invariance (Bishop, p. 256-269)
- Mixtures of density networks (Bishop, p. 272-275)

Exercise:

- Simple application of backpropagation

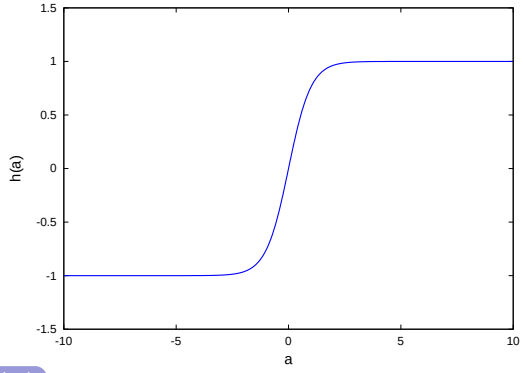
Lab:

- Exercise on neural networks

Activation functions

The hyperbolic tangent

Hyperbolic tangent



back

$$h(a) \equiv \tanh(a) = \frac{e^{a} - e^{-a}}{e^{a} + e^{-a}} \quad (4)$$

$$\frac{dh(a)}{da} = 1 - h^{2}(a) \quad (5)$$