

Neural Networks

Alberto Paccanaro

EMAp – FGV

www.paccanarolab.org

Material and images in these slides are from (or adapted from):

C. Bishop, Pattern Recognition and Machine Learning, Springer, 2006

We saw linear combinations of **FIXED** basis functions

Linear regression

- linear combinations of **fixed** nonlinear Basis Functions, $\phi_j(\mathbf{x})$:

$$y(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{j=1}^{M-1} w_j \phi_j(\mathbf{x})$$

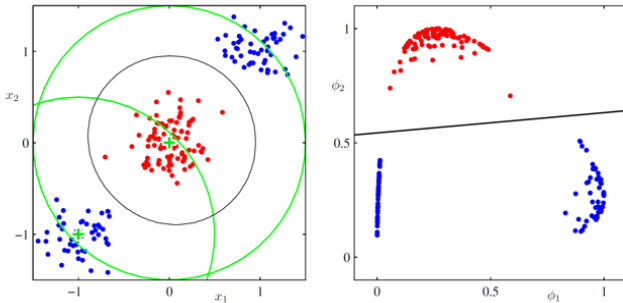
Nonlinear basis functions
allow $y(\mathbf{x}, \mathbf{w})$ to be nonlinear !

DO NOT DISTRIBUTE

Linear classification

All these algorithms are equally applicable if we first apply a nonlinear transformation of the inputs $\phi(x)$

Decision boundaries will be linear in the feature space $\phi(x)$ and nonlinear in the original x space



Logistic Regression (2 classes)

$$p(C_1|\phi) = y(\phi) = \sigma(\mathbf{w}^T \phi)$$

$$p(C_2|\phi) = 1 - p(C_1|\phi)$$

A logistic sigmoid
acting on a linear
function of the feature
vector ϕ

How about
adapting the
basis function
to the data?

- FIX THE FORM, but not the number:

Support vector machines: define basis functions centred on the training datapoints and then select a subset of these during training

- FIX THE NUMBER, but not the form:

Neural Networks: the number of basis function is fixed, but they adapt

- use parametric forms for the basis functions, and the parameter values are adapted during training
- multilayer perceptron: multiple layers of logistic regression models ☺

Roadmap



1. The idea, the networks
2. Error functions and Activation functions
3. Neural Network training
4. The backpropagation algorithm
5. Regularization

Feed-forward Networks

Linear models for regression and classification, so far:

$$y(\mathbf{x}, \mathbf{w}) = f \left(\sum_{j=1}^M w_j \phi_j(\mathbf{x}) \right)$$

Regression: f is identity

Classification: f is nonlinear (e.g. σ)

Idea: make the basis functions $\phi_j(\mathbf{x})$ depend on parameters and allow these parameters to be adjusted during training (along with the coefficients w_j).

Neural networks use basis functions that follow the same form – each basis function is itself a nonlinear function of a linear combination of the inputs, where the coefficients in the linear combination are adaptive parameters.

A series of transformations of the inputs x_1, \dots, x_D

1. Construct M linear combinations of the input variables x_1, \dots, x_D :

$$a_j = \sum_{i=1}^D w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

2. Each combination is transformed using a differentiable, nonlinear activation function $h(\cdot)$:

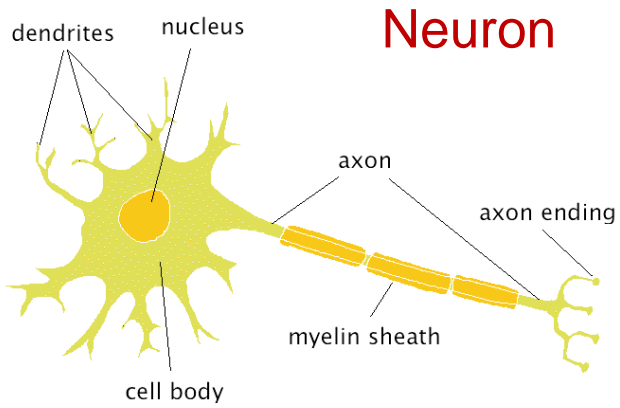
$$z_j = h(a_j)$$

3. These values are again linearly combined:

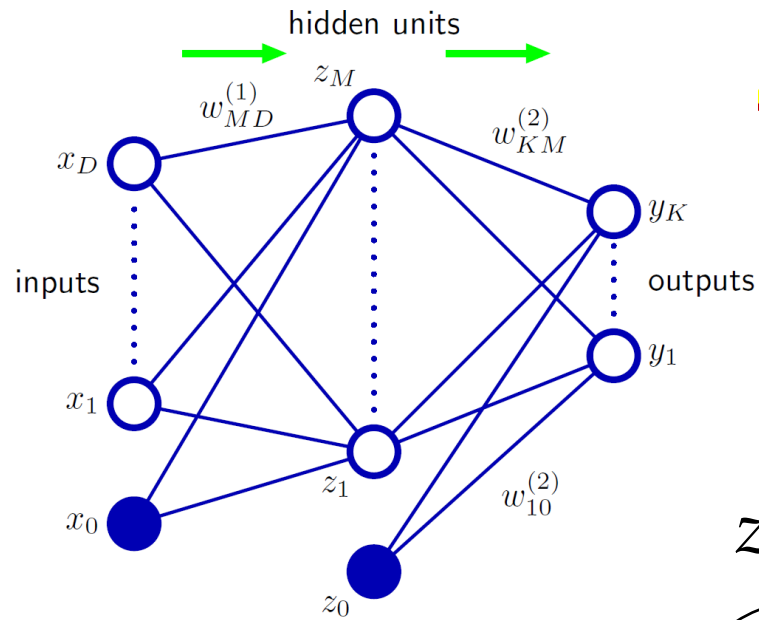
$$a_k = \sum_{j=1}^M w_{kj}^{(2)} z_j + w_{k0}^{(2)}$$

4. ... and so on, until the last activation function.

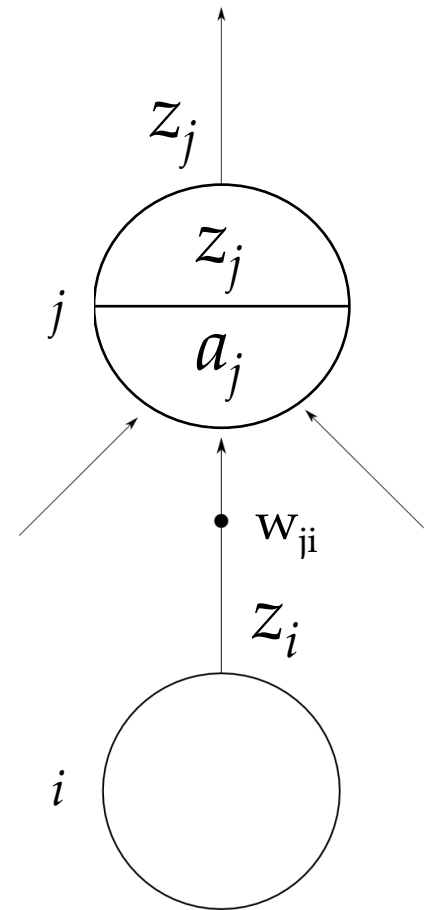
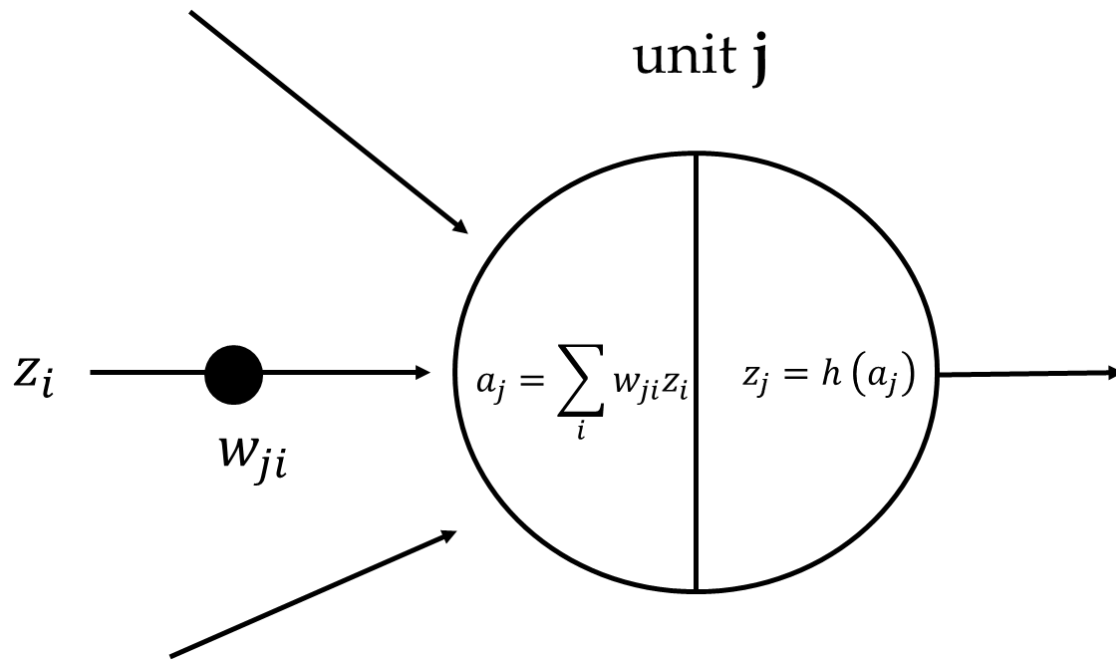
This complex function
can be represented in
the form of a network
diagram...



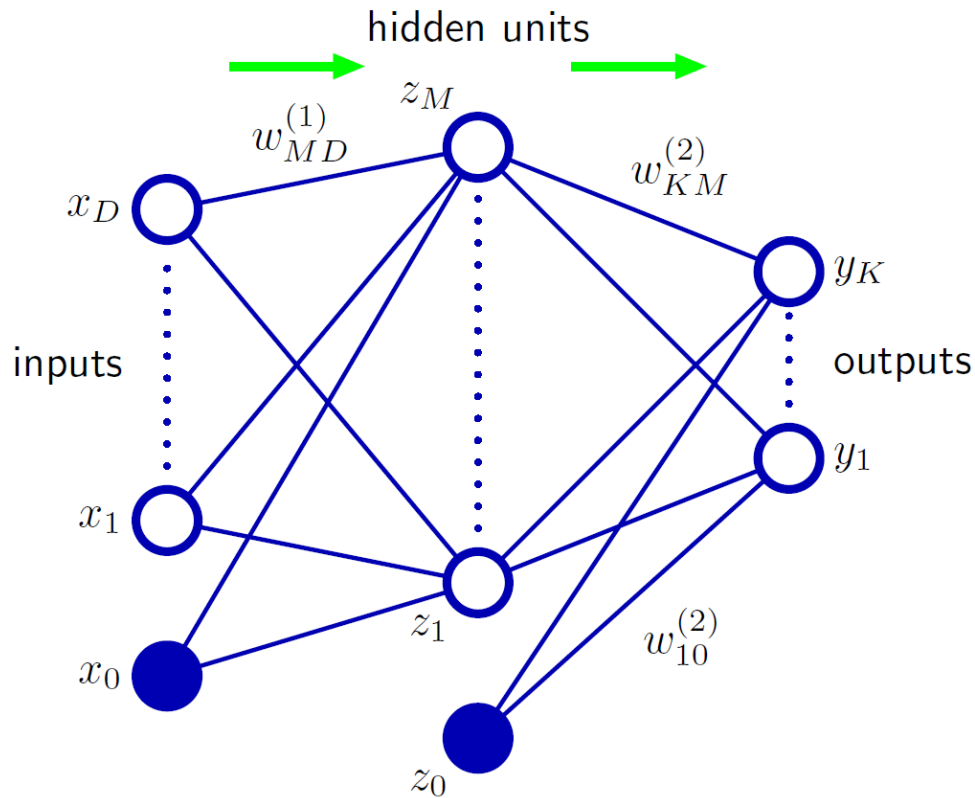
Neuron



**Multi-layer
perceptron**



For a 2 layer network



$$y_k(\mathbf{x}, \mathbf{w}) = \sigma \left(\sum_{j=1}^M w_{kj}^{(2)} h \left(\sum_{i=1}^D w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \right)$$

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma \left(\sum_{j=0}^M w_{kj}^{(2)} h \left(\sum_{i=0}^D w_{ji}^{(1)} x_i \right) \right) \quad (\text{absorbing the biases into the set of weights})$$

Important points

- If all the **activation functions of the hidden units are linear**, then we can always find an equivalent network without hidden units.

- Multiple distinct choices for the weight vector \mathbf{w} can all give the same function.

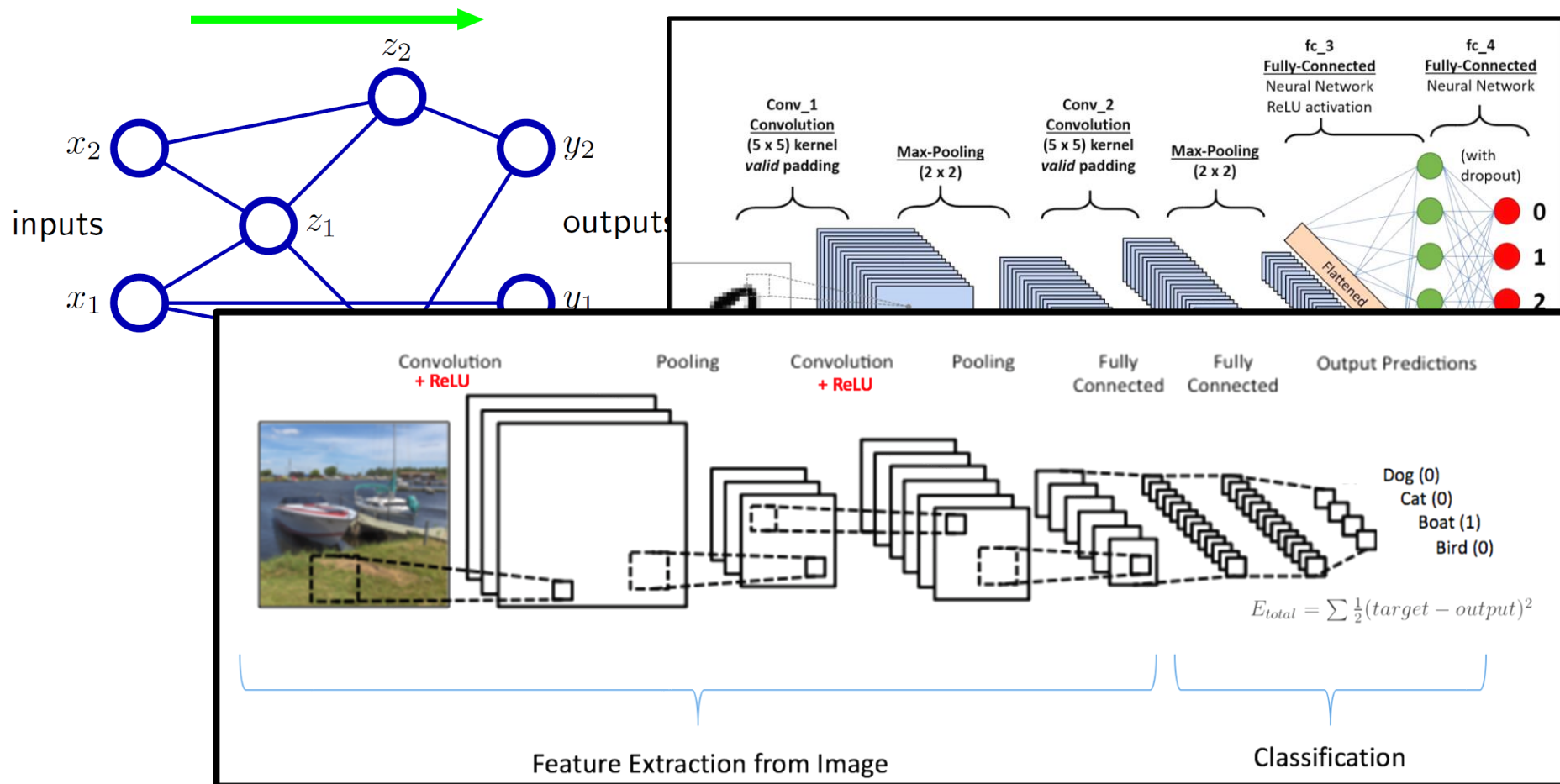
For a network with one hidden layer, M units:

- a change the sign of all of the weights and the bias feeding into a particular hidden unit, can be compensated by a change in the sign of the outputs (2^M ways to do that)
- we can interchange the values of all of the weights leading both into and out of a particular hidden unit with the corresponding values of the weights associated with a different hidden unit ($M!$ ways to do that)

The network will therefore have an overall weight-space symmetry factor of $M! * 2^M$.

For networks with more than two layers of weights, the total level of symmetry will be given by the product of such factors, one for each layer of hidden units.

- Because there is a direct correspondence between a network diagram and its mathematical function, **we can develop more general network mappings by considering more complex network diagrams.**


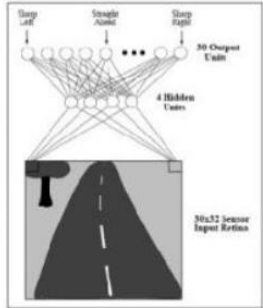




Uber ATG 2015

Autonomous driving

- ALVINN – Drives 70mph on highways

ALVINN CMU, 1989

- **Neural networks are *universal approximators*. A two-layer network** with linear outputs can uniformly approximate any continuous function on a compact input domain to arbitrary accuracy provided the network has a sufficiently large number of hidden units.

Roadmap

1. The idea, the networks



2. Error functions and Activation functions

3. Neural Network training

4. The backpropagation algorithm

5. Regularization

A probabilistic interpretation to the network outputs motivates:

1. choice of **output unit activation function**
2. the choice of **error function**.

... so we start with that ... 😊

(1) Regression

Output unit activation function: **identity**

$$p(t|\mathbf{x}, \mathbf{w}) = \mathcal{N}(t|y(\mathbf{x}, \mathbf{w}), \beta^{-1})$$

t has a Gaussian distr. with mean given by the output of the neural network

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^N p(t_n|\mathbf{x}_n, \mathbf{w}, \beta)$$

Likelihood function

$$\frac{\beta}{2} \sum_{n=1}^N \{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2 - \frac{N}{2} \ln \beta + \frac{N}{2} \ln(2\pi)$$

Negative log likelihood

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N \{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2$$

Sum-of-squares error function

(2a) Classification, binary case

Output unit activation function: **sigmoid**

$$p(t|\mathbf{x}, \mathbf{w}) = y(\mathbf{x}, \mathbf{w})^t \{1 - y(\mathbf{x}, \mathbf{w})\}^{1-t} \quad \text{conditional distribution of targets}$$

$$E(\mathbf{w}) = - \sum_{n=1}^N \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\} \quad \text{Cross-entropy error function}$$

(2b) Classification, k separate classifications (non mutually exclusive classes)

Output unit activation function: **sigmoid**

$$p(\mathbf{t}|\mathbf{x}, \mathbf{w}) = \prod_{k=1}^K y_k(\mathbf{x}, \mathbf{w})^{t_k} [1 - y_k(\mathbf{x}, \mathbf{w})]^{1-t_k}$$

$$E(\mathbf{w}) = - \sum_{n=1}^N \sum_{k=1}^K \{t_{nk} \ln y_{nk} + (1 - t_{nk}) \ln(1 - y_{nk})\}$$

(2c) Classification, k classes, mutually exclusive

Output unit activation function: **softmax**

$$y_k(\mathbf{x}, \mathbf{w}) = \frac{\exp(a_k(\mathbf{x}, \mathbf{w}))}{\sum_j \exp(a_j(\mathbf{x}, \mathbf{w}))}$$

$$E(\mathbf{w}) = - \sum_{n=1}^N \sum_{k=1}^K t_{kn} \ln y_k(\mathbf{x}_n, \mathbf{w})$$


*multiclass
cross-entropy
error function*

a particular output unit takes the familiar form (5.18).

In summary, there is a natural choice of both output unit activation function and matching error function, according to the type of problem being solved. For regression we use linear outputs and a sum-of-squares error, for (multiple independent) binary classifications we use logistic sigmoid outputs and a cross-entropy error function, and for multiclass classification we use softmax outputs with the corresponding multiclass cross-entropy error function. For classification problems involving two classes, we can use a single logistic sigmoid output, or alternatively we can use a network with two outputs having a softmax output activation function.

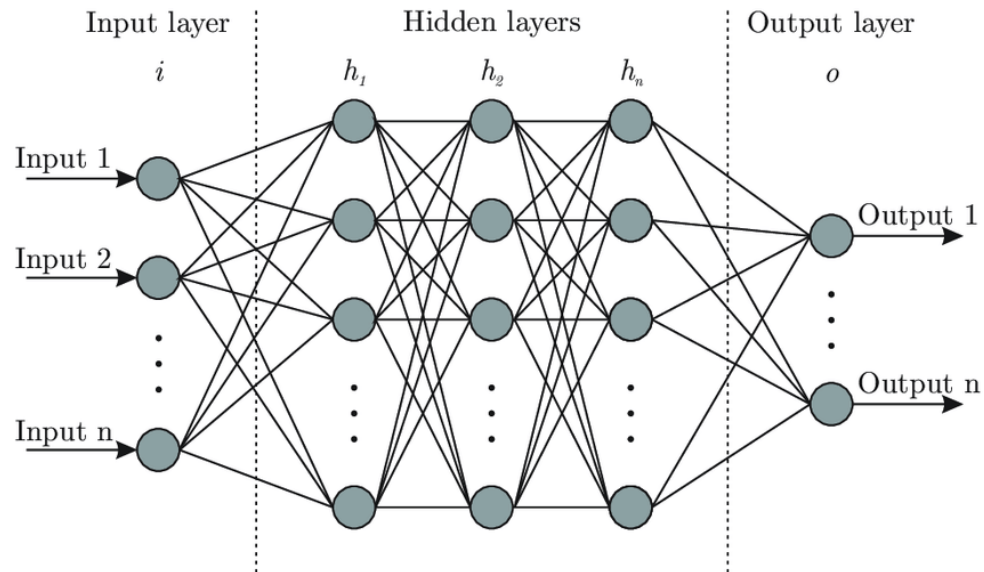
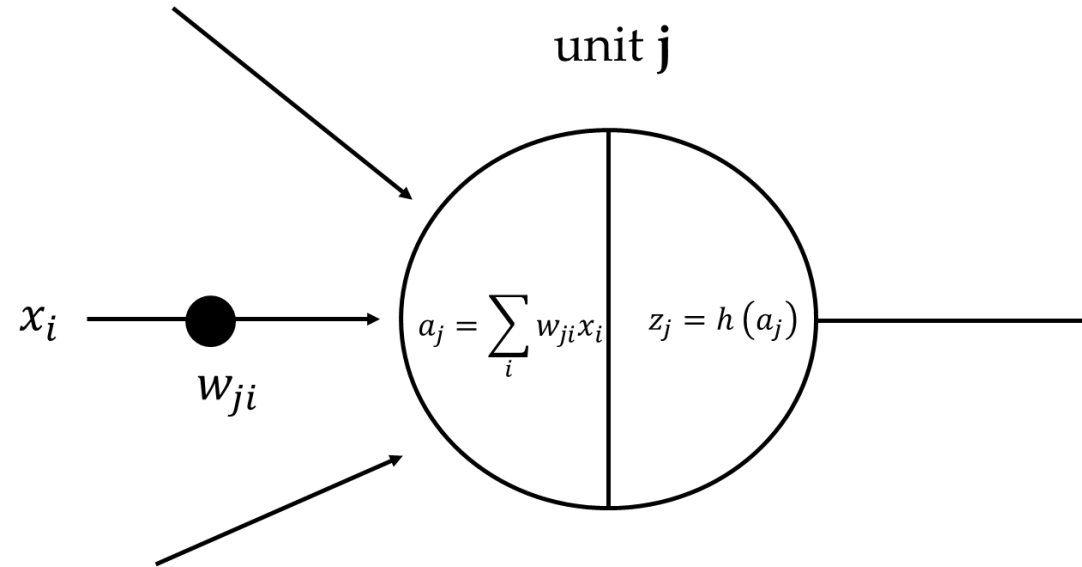
(*C. Bishop, page 236*)

Roadmap

1. The idea, the networks
2. Error functions and Activation functions
-  3. Neural Network training
4. The backpropagation algorithm
5. Regularization

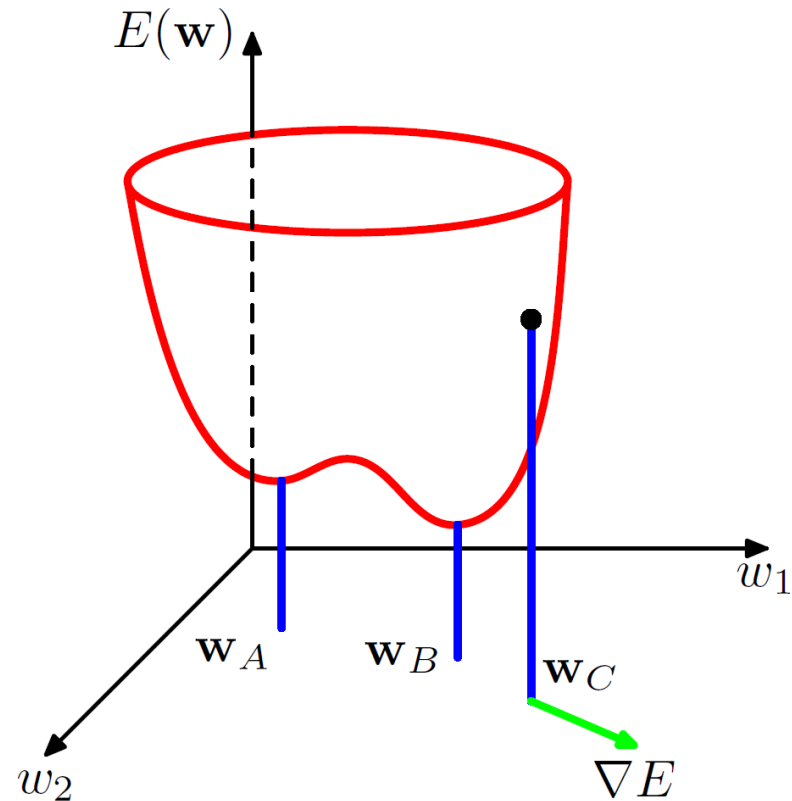
Task: learn the weights w_{ji} to minimize an error function.

Neural network training



*The error function
is a surface sitting
over weight space.*

Global minimum
Local minima



No closed form solution for $\nabla E(\mathbf{w}) = 0$

We use **iterative numerical procedures**.

Choose some initial value $\mathbf{w}^{(0)}$ for the weight vector and then move through weight space using:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta \mathbf{w}^{(\tau)}$$

- Different algorithms involve different choices for the weight vector update $\Delta \mathbf{w}(\tau)$.
- Many algorithms make use of **gradient information**.

Gradient descent (or steepest descent) optimization

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)})$$

η is called the *learning rate*.

Batch method: uses the whole data set at once

For batch optimization, there are more efficient methods (e.g. conjugate gradients, quasi-Newton methods)

Sequential (or stochastic) gradient descent optimization

$$E(\mathbf{w}) = \sum_{n=1}^N E_n(\mathbf{w})$$

where $E_n(\mathbf{w})$ is the error due to datapoint n . Then:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n(\mathbf{w}^{(\tau)})$$

makes an update based on one datapoint at a time.

Intermediate scenarios: updates are based on batches of data points.

It may be necessary to run a gradient-based algorithm multiple times, with different starting points

Iterative procedures that use gradient descent to minimize an error function

Each step has 2 stages:

1. Calculate the derivatives of the error function with respect to the weights
2. Use the derivatives to compute the adjustments to the weights


Stage 1 – backprop:

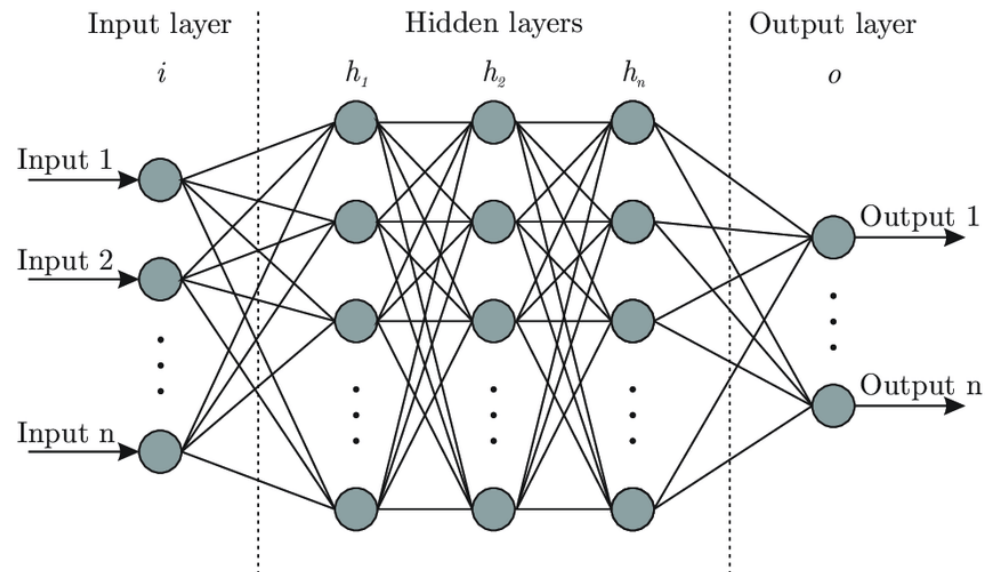
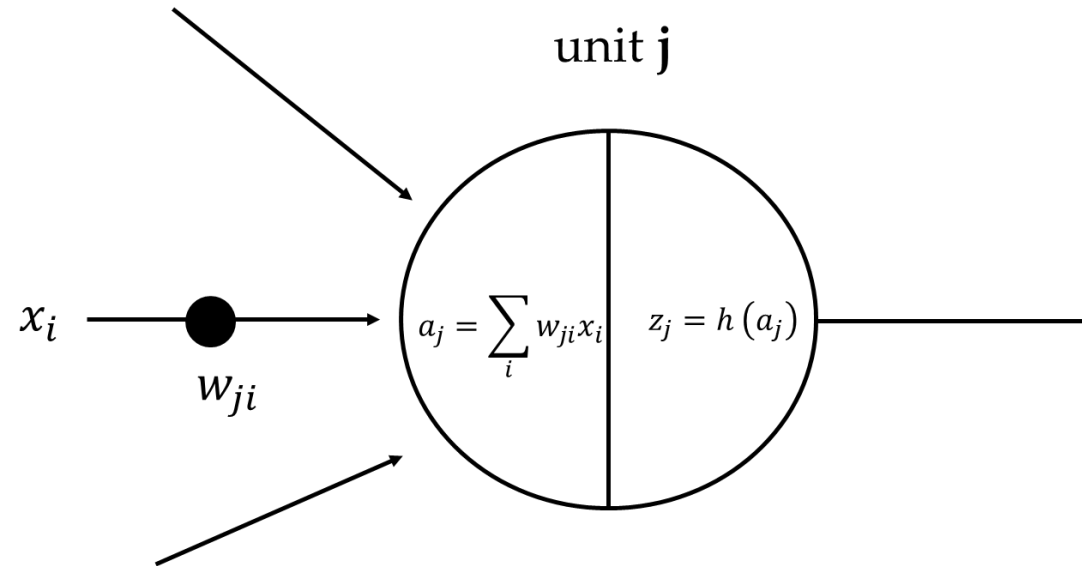
- can be applied to many kinds of network.
- can be applied to different error functions
- can be used to calculate other derivatives such as the Jacobian and Hessian matrices.

Backpropagation: an efficient method for calculating derivatives.

Stage 2: can use a variety of optimization schemes (also more powerful than gradient descent).

Roadmap

1. The idea, the networks
2. Error functions and Activation functions
3. Neural Network training
-  4. The backpropagation algorithm
5. Regularization



The backpropagation algorithm

$$E(\mathbf{w}) = \sum_{n=1}^N E_n(\mathbf{w})$$

Error function: a sum of terms, one for each data point in the training set

$$y_k = \sum_i w_{ki} x_i$$

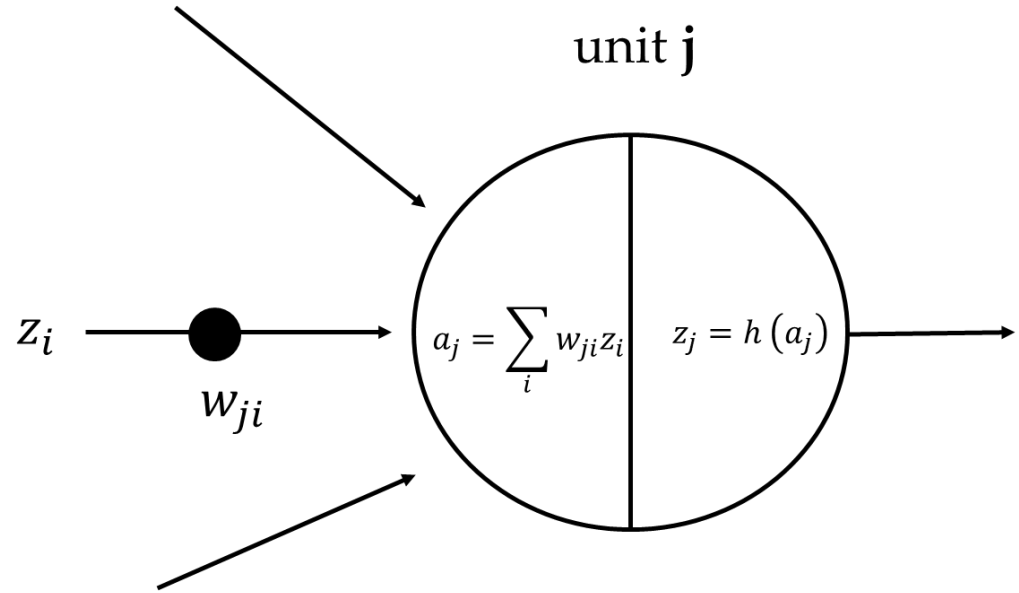
(linear activation)

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

$$\frac{\partial E_n}{\partial w_{ji}} = (y_{nj} - t_{nj}) x_{ni}$$

Note: a “local” computation involving the product of an *error signal* $(y_{nj} - t_{nj})$ and the variable x_{ni} .

In general...
$$\begin{cases} a_j = \sum_i w_{ji} z_i \\ z_j = h(a_j) \end{cases}$$



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

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j}$$

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

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

the derivative is obtained simply by multiplying δ for the unit at the output end of the weight by the value of z for the unit at the input end of the weight !!!

For the output units: $\delta_k = y_k - t_k$

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j}$$

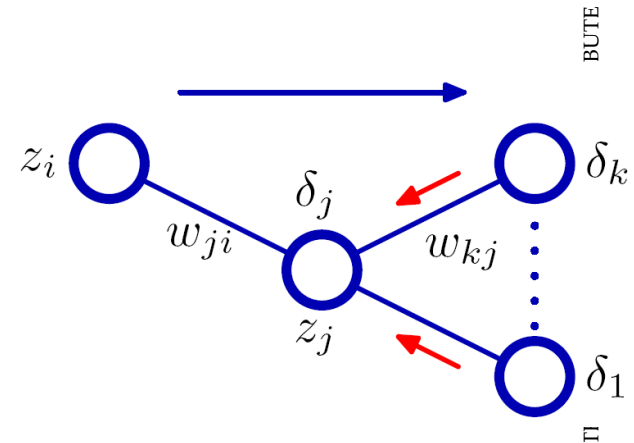
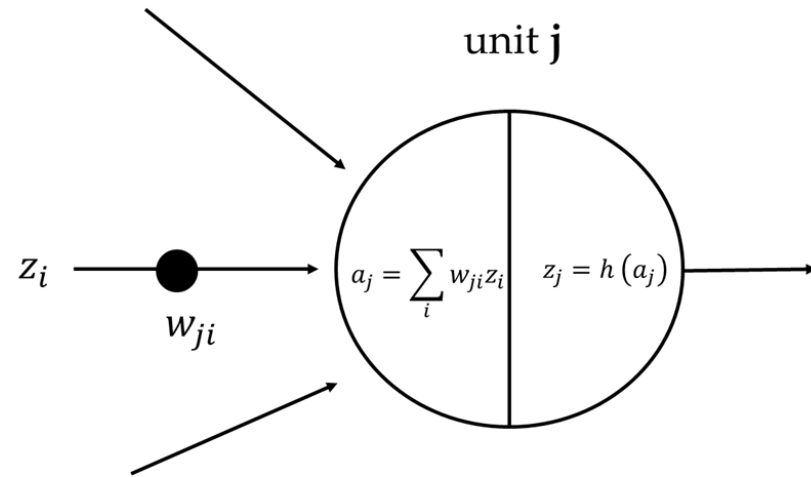
For the hidden units:

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

where the sum runs over all units k to which unit j sends connections !

backpropagation formula:

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$




δ for a particular hidden unit can be obtained by propagating the δ 's backwards from units higher up in the network !

Error Backpropagation

1. Apply an input vector \mathbf{x}_n to the network and forward propagate through the network
2. Evaluate the δ_k for all the output units
3. Use the backpropagation formula to backpropagate the δ 's to obtain δ_j for each hidden unit
4. The required derivatives are given by $\delta_j z_i$ where z_i is the i^{th} input to node j .

Important: go through the example on page 245 of Bishop's book (section 5.3.2)

Roadmap

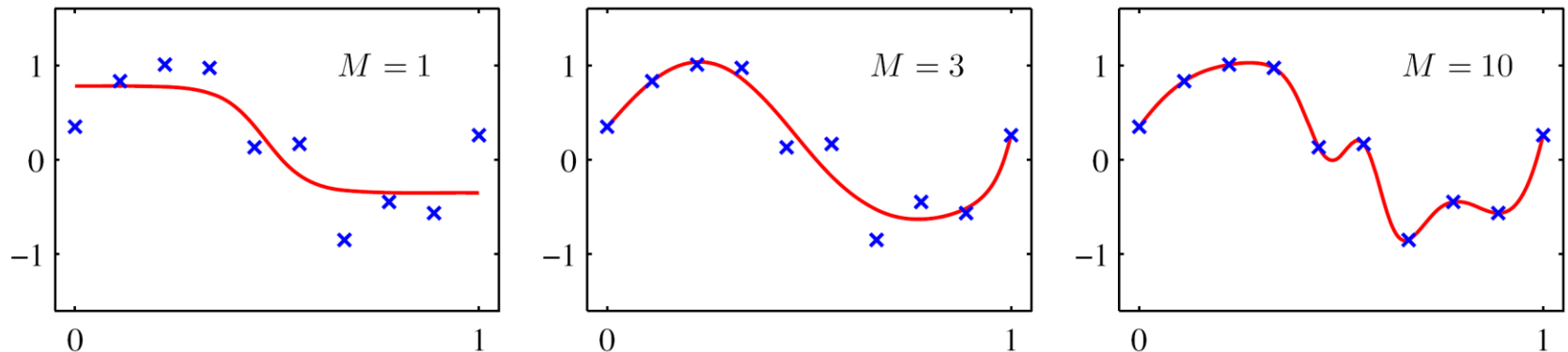
1. The idea, the networks
2. Error functions and Activation functions
3. Neural Network training
4. The backpropagation algorithm
-  5. Regularization

Number of free parameters

Number of units control the number of parameters (weights)

Number of input and output units: determined by the dimensionality of the data set

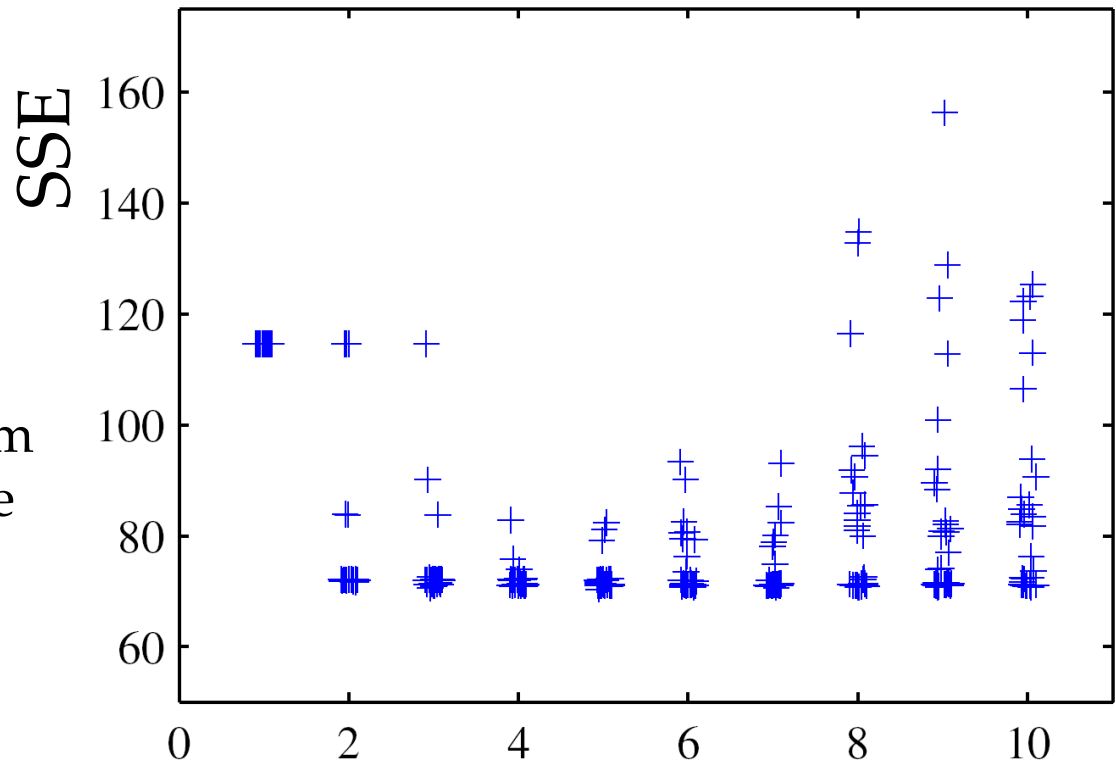
Number M of hidden units: free parameter



Local minima

Generalization error is not a simple function of M due to the local minima in the error function

Performance for the polynomial approximation example, on the **validation set** (*one way to choose M is to plot these graphs*).



Effect of multiple random initializations for a range of values of M .

Regularization – *weight decay*

We can control complexity by the addition of a regularization term to the error function.

The simplest regularizer is the quadratic:

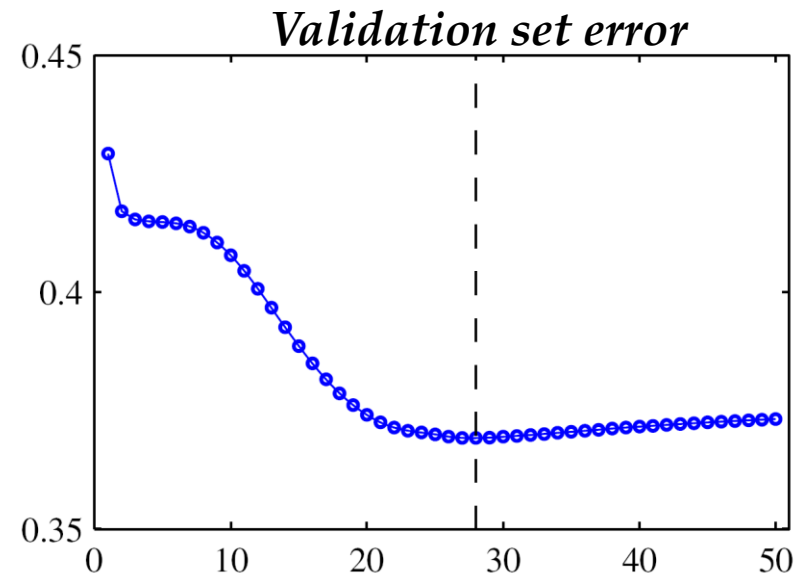
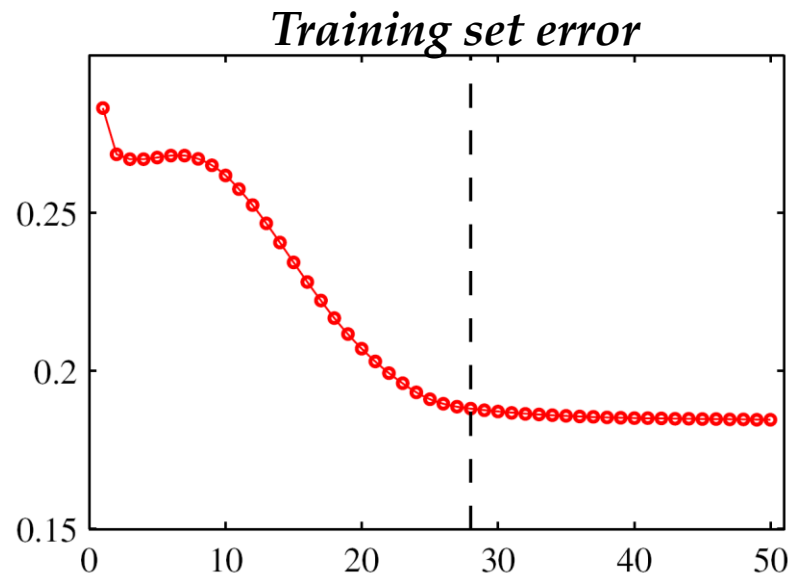
$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2} \mathbf{w}^T \mathbf{w}$$

This regularizer can be interpreted as the negative logarithm of a zero-mean Gaussian prior distribution over the weight vector w .

Regularization – *early stopping*

The error measured with respect to a validation set, often shows a decrease at first, followed by an increase as the network starts to over-fit.

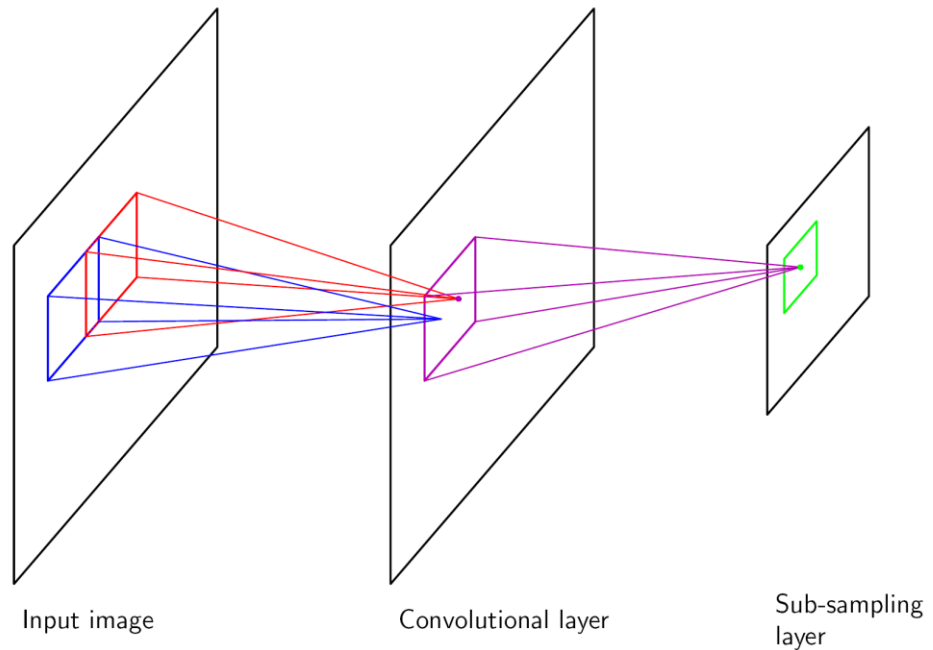
Why?



Training can therefore be stopped at the point of smallest error with respect to the validation data set

Convolutional networks

In the convolutional layer units are organized into *feature maps*.



Units in a feature map take inputs from a small subregion
All of the units in a feature map are constrained to share the same weight values.

