[Spring, 2017]

(Artificial) Neural Networks

Pattern Recognition (BRI623)

Heung-Il Suk

hisuk@korea.ac.kr

http://www.ku-milab.org

Department of Brain and Cognitive Engineering,
Korea University

Contents

Introduction

- Linear combination of fixed basis functions:
  regression (Ch. 3) and classification (Ch. 4)
  - Useful analytical and computational properties
  - Limited to practical applicability due to 'curse of dimensionality'

For their application to large-scale problems,
it is necessary *to adapt the basis functions to the data*

⇓

Neural Networks and Support Vector Machine

NN vs. SVM

Neural Network (NN)

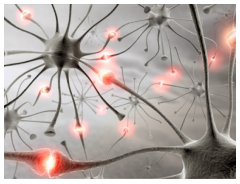
- Fixing the number of basis functions in advance
- Allow them to be adaptive in parametric forms
  - MLP: layers of continuous nonlinear models
  - Involves non-convex optimization during training (many minima)

Support Vector Machine (SVM)

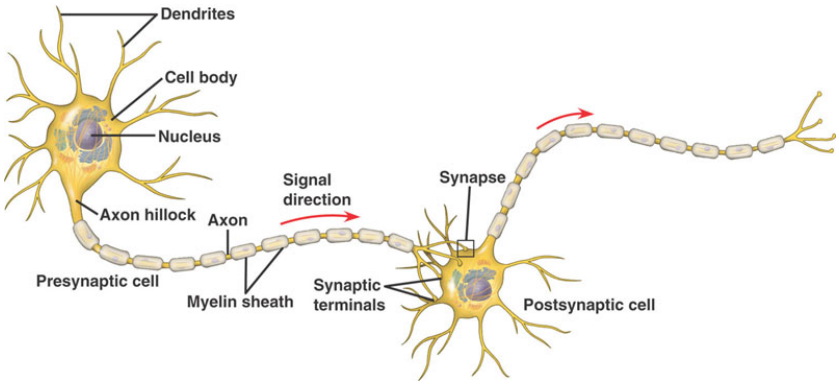
- Defining basis functions that are centered on the training data points
- Selecting a subset of these during training
  - Convex objective function via non-linear optimization

Biologically Motivated

- Biological learning systems are built of very complex webs of inter-connected neurons
  - 10^{11} (100 billion) neurons
  - Each neuron is connected to 10^4 others



- Each unit takes real-valued inputs (possibly from other units)
- Produces a single real-valued output (which further becomes the input to many other units)



[Image source: 'http://biology.stackexchange.com/']

Research Groups in NN

1. (Biological NN) Study and model biological learning
   - Network of neurons in the brain provide people with ability assimilate information
   - Will simulations of such networks reveal the underlying mechanism of learning?

2. (Artificial NN) Obtain highly effective learning machines
   - Biological realism imposes unnecessary constraints
   - Primarily multilayer perceptron (OUR INTEREST!!!)

History in Neural Networks

Neural networks have had a unique history in the realm of technology. Unlike many technologies today which either immediately fail or are immediately popular, neural networks for popular for a short time, took a two-decade hiatus, and have been popular every since. [Source: E. Roberts]

- Origins: algorithms that try to mimic the brain
- Widely used in 80s and the early 90s
- Popularity diminished in the late 90s
- Resurged recently

1. Creation of a computational model for neural networks based on mathematics and algorithms called *threshold logic*

BULLETIN OF
MATHEMATICAL BIOPHYSICS
VOLUME 5, 1943

A LOGICAL CALCULUS OF THE
IDEAS IMMANENT IN NERVOUS ACTIVITY

WARREN S. MCCULLOCH AND WALTER PITTS

FROM THE UNIVERSITY OF ILLINOIS, COLLEGE OF MEDICINE,
DEPARTMENT OF PSYCHIATRY AT THE ILLINOIS NEUROPSYCHIATRIC INSTITU
AND THE UNIVERSITY OF CHICAGO

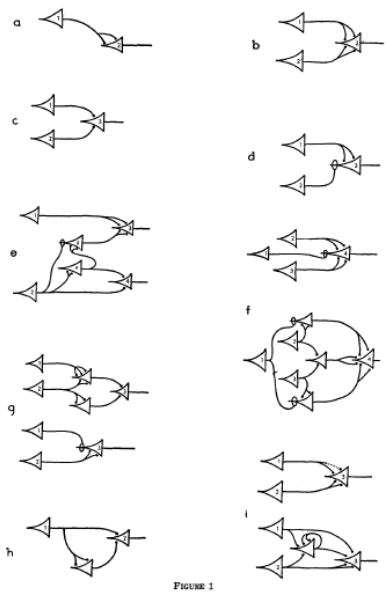


FIGURE 1

2. Creation of the *perceptron*, an algorithm for pattern recognition based on a two-layer learning computer network using simple addition and subtraction

Psychological Review
Vol. 65, No. 6, 1958

THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN [1]

F. ROSENBLATT

Cornell Aeronautical Laboratory

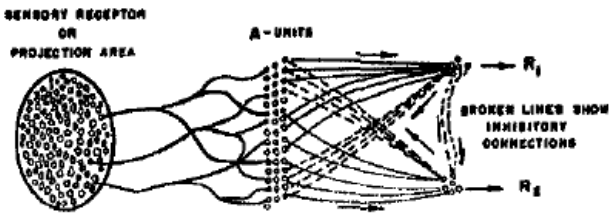
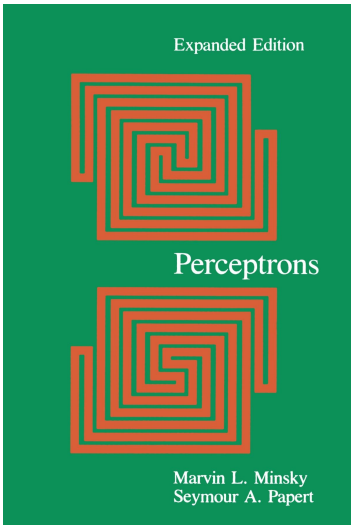


FIG. 2A. Schematic representation of connections in a simple perceptron.

3. Discovery of two key issues with the computational machines that processed neural networks
   - Single-layer neural networks were incapable of processing the exclusive-or circuit
   - Computers were not sophisticated enough to effectively handle the long run time required by large neural networks



M. Minsky, and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, MIT Press, 1969

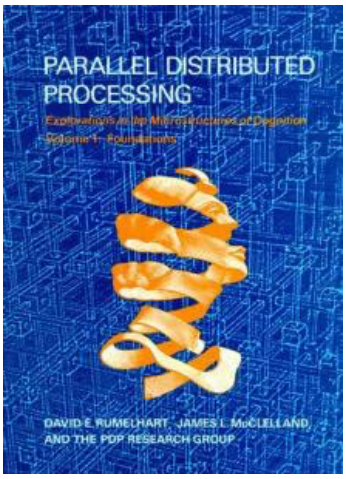
4. Revival: Backpropagation
   - First invented in 1969 by Bryson and Ho (Russell and Norvig, 2010)
   - Reinvented in the mid-1980s by at least four different groups
   - Widespread disseminatioin via the collection *Parallel Distributed Processing* (Rumelhar and McClelland, 1986)

NATURE VOL. 323 9 OCTOBER 1986

Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA



5. Neurocognition

Biol. Cybernetics 36, 193–202 (1980)

Biological Cybernetics

Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position

Kunihiko Fukushima

NHK Broadcasting Science Research Laboratories, Kinuta, Setagaya, Tokyo, Japan

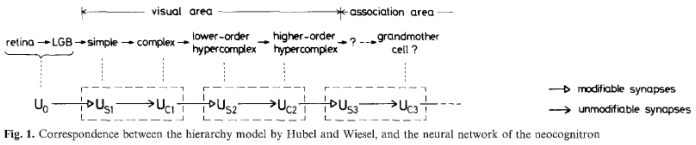


Fig. 1. Correspondence between the hierarchy model by Hubel and Wiesel, and the neural network of the neocognitron

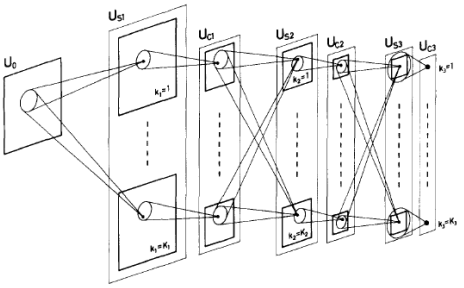


Fig. 2. Schematic diagram illustrating the interconnections between layers in the neocognitron

6 Convolutional Neural Networks

Gradient-Based Learning Applied to Document Recognition

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner

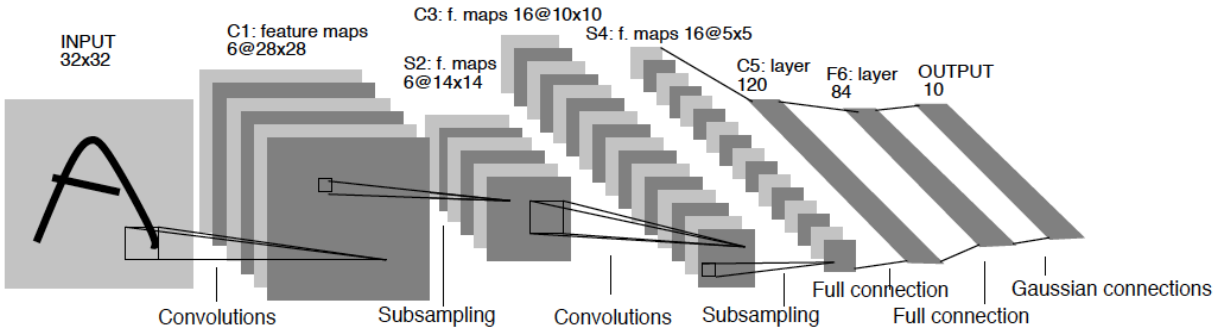


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

7 Resurgence: Deep Learning (DL)

LETTER Communicated by Yann Le Cun

A Fast Learning Algorithm for Deep Belief Nets

Geoffrey E. Hinton
hinton@cs.toronto.edu
Simon Osindero
osindero@cs.toronto.edu
Department of Computer Science, University of Toronto, Toronto, Canada M5S 3G4

Yee-Whye Teh
tehyw@comp.nus.edu.sg
Department of Computer Science, National University of Singapore, Singapore 117543

Reducing the Dimensionality of Data with Neural Networks

G. E. Hinton* and R. R. Salakhutdinov

High-dimensional data can be converted to low-dimensional codes by training a multilayer neural network with a small central layer to reconstruct high-dimensional input vectors. Gradient descent can be used for fine-tuning the weights in such "autoencoder" networks, but this works well only if the initial weights are close to a good solution. We describe an effective way of initializing the weights that allows deep autoencoder networks to learn low-dimensional codes that work much better than principal components analysis as a tool to reduce the dimensionality of data.

Overview

- Begin by considering the functional form of the network model
  - Specific parameterization of the basis functions
- *Error backpropagation*
  - Determining the network parameters within *maximum likelihood* framework
  - Involves the solution of non-linear optimization problem
- Regularization of NN training
- Extension to other derivatives: Jacobian and Hessian Matrix
- Mixture density network
  - General framework for modelling conditional probability distributions
- Bayesian treatments of NNs

Feed-Forward Network Functions

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

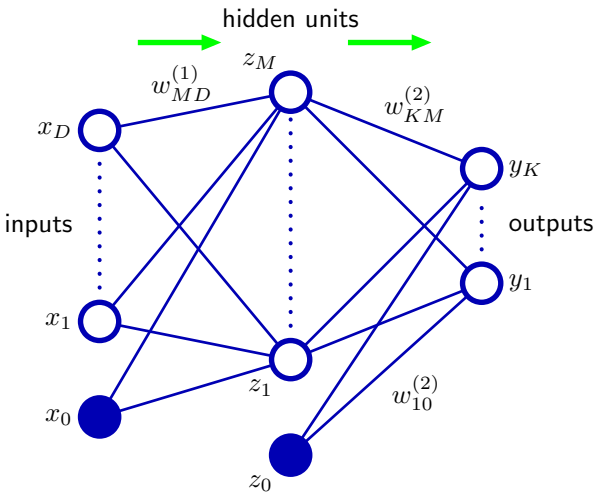
- Function $f(\cdot)$: continuous output (c.f., perceptron: discontinuous)
  - Linear regression: identify
  - Linear classification: nonlinear activation

$$\Downarrow$$

- Basis functions $\phi_j(\mathbf{x})$: a parametric form
- These parameters to be adjusted along with the coefficients $\{w_j\}$

- In NN, each basis function itself is a nonlinear function of a linear combination of the inputs
- Coefficients in the linear combination are adaptive parameters

Feed-Forward Network Functions



Network diagram for the two-layer neural network

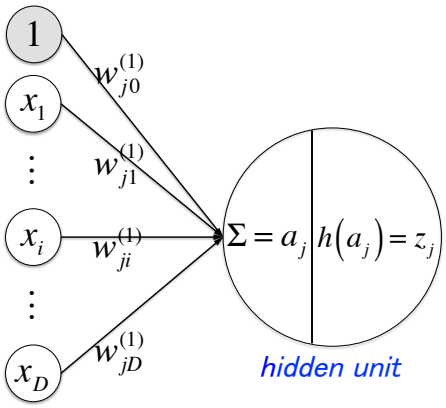
- Activations: $\{a_j\}_{j=1}^{M}$

$$a_j = \sum_{i=1}^{D} \underbrace{w_{ji}^{(1)}}_{\text{weights}} x_i + \underbrace{w_{j0}^{(1)}}_{\text{biases}}$$

- Outputs of the basis functions ('*hidden units*')

$$z_j = h\left(a_j\right)$$

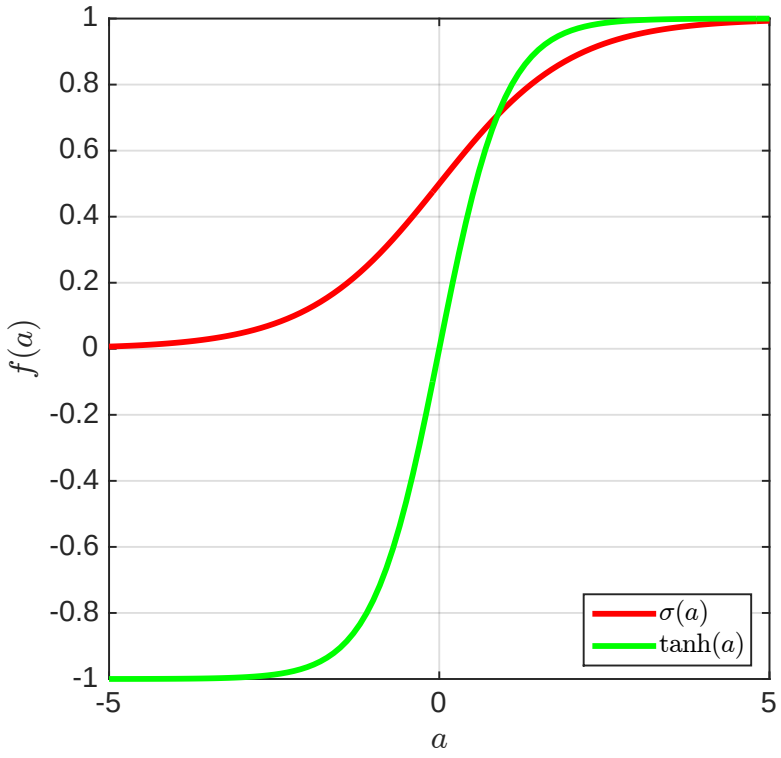
  - $h\left(\cdot\right)$: differentiable, nonlinear activation function
  - sigmoidal functions such as 'logistic sigmoid' or 'tanh'
  - or others (discussed in deep learning)



Connection between input and hidden units

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

$$\tanh(a) = \frac{\exp[a] - \exp[-a]}{\exp[a] + \exp[-a]}$$

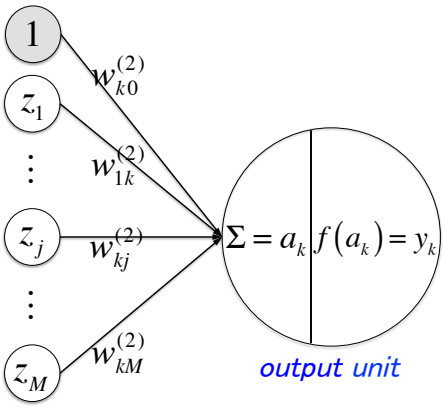


- Activations: $\{a_k\}_{k=1}^{K}$

$$a_k = \sum_{j=1}^{M} \underbrace{w_{kj}^{(2)}}_{\text{weights}} z_j + \underbrace{w_{k0}^{(2)}}_{\text{biases}}$$

- Outputs of the basis functions ('*output units*')

$$y_k = f(a_k)$$



Connection between hidden and output units

- Choice of the output activation function $f(\cdot)$ is determined by the nature of the data and the assumed distribution of target variables
  - Regression: identity

$$y_k = a_k$$

  - Binary classification: logistic sigmoid

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

  - Multiclass classification: softmax

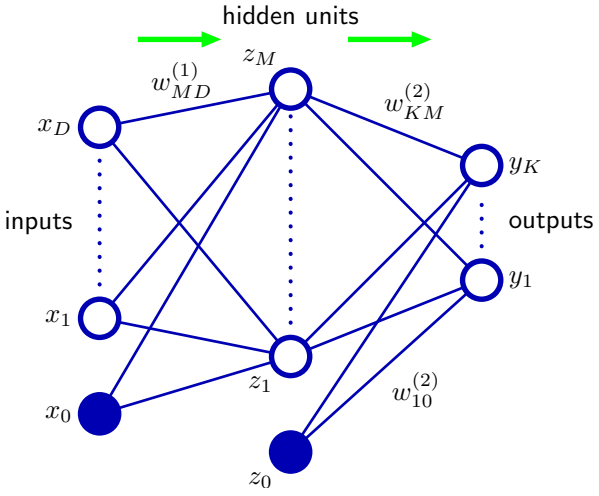
$$y_k = s(a_k) = \frac{\exp(a_k)}{\sum_{l=1}^{K} \exp(a_l)}$$

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

equivalently

$$y_k(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=0}^{M} w_{kj}^{(2)} \underbrace{h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)}_{\phi_j(\mathbf{x})}\right)$$

where $\mathbf{w} = [w_{j0}, w_{j1}, \ldots, w_{jD}, w_{k0}, w_{k1}, \ldots, w_{kM}]^\top$



Each step of input-hidden and hidden-output

- resembles the perceptron model: weighted sum + step-function nonlinearities
- 'Multi-Layer Perceptron' (MLP)
  - Continuous sigmoidal-function nonlinearities in the hidden units
  - Differentiable w.r.t. network parameters $\mathbf{w}$, which plays a central role in network training

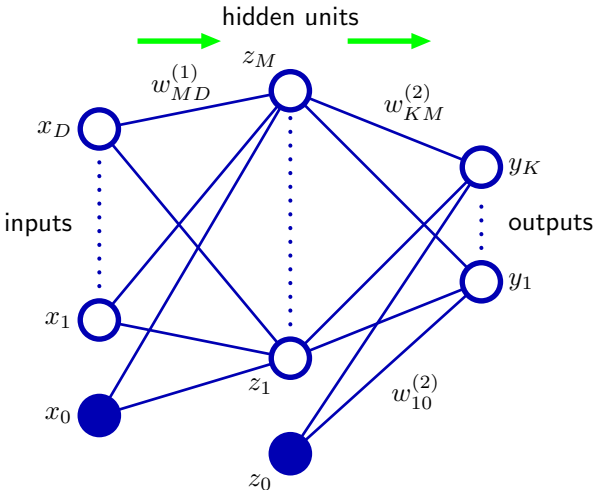
$$\mathbf{x} \in \mathbb{R}^D \xrightarrow{W^{(1)} \in \mathbb{R}^{D \times M}} \mathbf{h} \in \mathbb{R}^M \xrightarrow{W^{(2)} \in \mathbb{R}^{M \times K}} \mathbf{o} \in \mathbb{R}^K$$

- If the activation functions of all the hidden units in a network are linear,
  - we can always find the equivalent network without hidden units

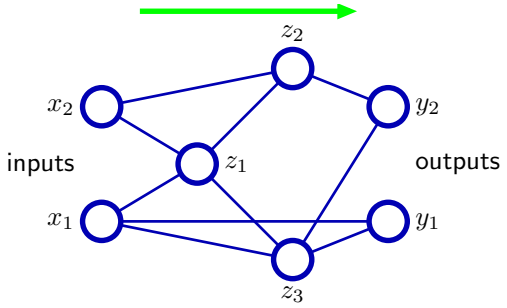
$$W = W^{(1)} W^{(2)}$$

- When $M < D$ with linear hidden units, it has the effect of dimensionality reduction
  - Networks of linear units give rise to Principal Component Analysis (PCA).

- In general, there is little interest in multilayer networks of linear units



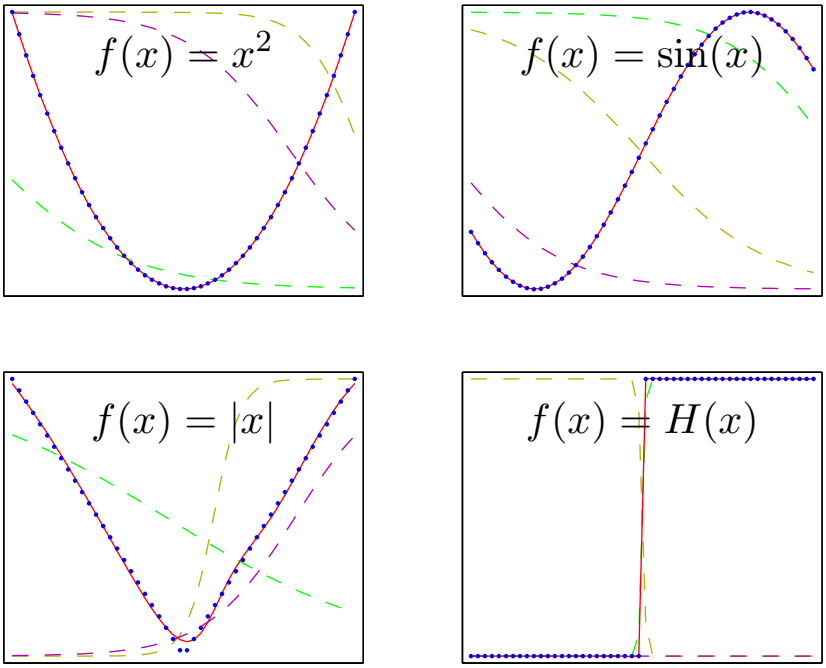
Most common in practice



General feed-forward topology

- No closed directed cycles
- Outputs are deterministic functions of the inputs

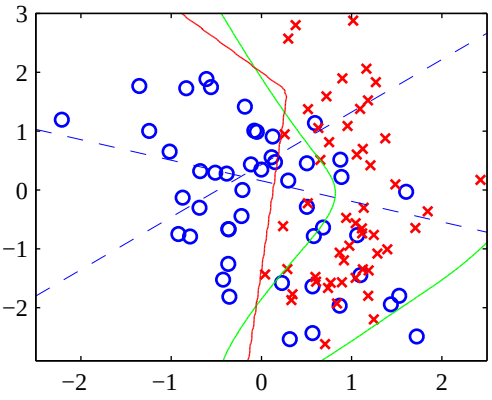
MLP: *universal approximator*



- Blue dots: 50 data points uniformly $(-1, 1)$; 3 hidden units (tanh), linear output units
- Network output (red curve), outputs of thee hidden units

Machine Intelligence Lab

- Binary classification with synthetic data
- Two inputs, two hidden units (tanh), single output (logistic sigmoid)
- Dashed blue lines: $z = 0.5$ contours for each of the hidden units
- Red line: $y = 0.5$ decision surface for the network
- Green lines: optimal decision boundary computed from the distributions used to generate the data



Machine Intelligence Lab

Multiple (L) hidden layers

$$y_k = f\left(\sum_l W_{kl}^{(L)} h\left(\sum_m W_{ml}^{(L-1)} h\left(\cdots h\left(\sum_i W_{ij}^{(1)} x_i\right)\right)\right)\right)$$

e.g., 3-layer neural network

$$\begin{aligned} z_m^{(1)} &= h\left(\mathbf{W}_{:m}^{(1)T}\mathbf{x}\right) = h\left(\sum_{j=1}^{D} W_{mj}^{(1)} x_j + W_{h0}^{(1)}\right), \quad m = 1, \ldots, H_1 \\ z_l^{(2)} &= h\left(\mathbf{W}_{:l}^{(2)T}\mathbf{z}^{(1)}\right) = h\left(\sum_{m=0}^{H_1} W_{lm}^{(2)} z_m^{(1)} + W_{l0}^{(2)}\right), \quad l = 1, \ldots, H_2 \\ y_k &= f\left(\mathbf{W}_{:k}^{(3)T}\mathbf{z}^{(2)}\right) = f\left(\sum_{l=1}^{H_2} W_{kl}^{(3)} z_l^{(2)} + W_{k0}^{(3)}\right) \end{aligned}$$

Weight-Space Symmetries

- There are multiple distinct choices for the weight vector $\mathbf{w}$
  - All give rise to the same mapping function from inputs to outputs
- Easily shown in fully connected network
  - M hidden units having tanh activation function change sign of all weights and bias feeding to particular hidden node
  - Since $\tanh(-a) = -\tanh(a)$, this change can be compensated by changing sign of all weights leading out of that hidden unit
  - For M hidden units, M such sign-flip symmetries
  - Thus, any given weight vector will be one of 2^M equivalent weight vectors
  - Since the values of all weights and bias of a node can be interchanged for M hidden units, there are $M!$ equivalent weight vectors
- Network has overall weight space symmetry factor of $M!2^M$
- No practical consequence other than in Bayesian neural networks

Network Training

Overview

Neural networks perform a transformation

- vector $\mathbf{x}$ of input variables to vector $\mathbf{y}$ of output variables

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

To determine $\mathbf{w}$, make an analogy with polynomial curve fitting

- Minimize a sum-of-squared error function

Error Functions

Given a set of input vectors $\{\mathbf{x}_n\}_{n=1}^N$ and target vectors $\{\mathbf{t}_n\}_{n=1}^N$

- Regression
  - Identity activation function
  - Sum-of-squares error: $E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^N \|\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n\|^2$
- Binary classification
  - Logistic sigmoid activation function
  - Cross-entropy error function: $E(\mathbf{w}) = -\sum_{n=1}^N \{t_n \ln y_n + (1-t_n)\ln(1-t_n)\}$

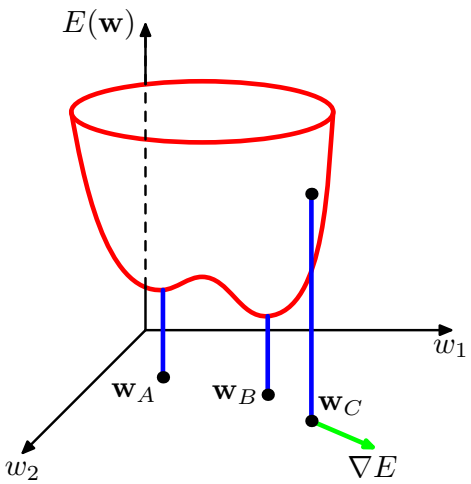
▸ Probabilistic View

- Multiclass classification
  - Softmax function
  - Cross-entropy error function: $E(\mathbf{w}) = -\sum_{n=1}^N \sum_{k=1}^K t_{kn} \ln y_{kn}$

▸ Probabilistic View

Parameter Optimization

- Finding a weight vector $\mathbf{w}$ which minimizes the chosen error function $E(\mathbf{w})$

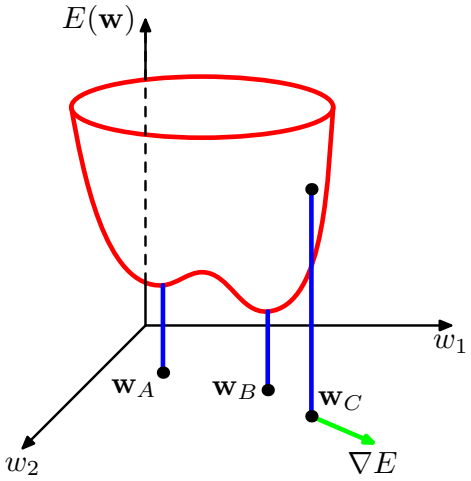


- Geometrical picture of error function
  - a surface sitting over the weight space
- Small step from $\mathbf{w}$ to $\mathbf{w} + \delta\mathbf{w}$ leads to change in error function

$$\delta E \approx \delta\mathbf{w}^\top \nabla E(\mathbf{w})$$

  - At any point $\mathbf{w}_C$, the local gradient of the error surface is given by the vector ∇E

$$\nabla E = \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \cdots, \frac{\partial E}{\partial w_d}\right]^\top$$



- ∇E: the direction of greatest rate of increase of the error function $E(\mathbf{w})$
- To reduce the error, make a small step in the direction of $-\nabla E(\mathbf{w})$
- A smooth continuous function of $\mathbf{w}$
  - its smallest value occurs at a point such that the gradient of the error function vanishes, *i.e.*, $\nabla E(\mathbf{w}) = 0$
  - Points at which the gradient vanishes: stationary points
    - minima, maxima, and saddle points
  - $\mathbf{w}_A$: local minimum
  - $\mathbf{w}_B$: global minimum

The error function typically has a highly nonlinear dependence on the weights and bias parameters.

- There will be many points in weight space at which the gradient vanishes (or numerically very small)
- Weight symmetry: each point in weight space is a member of a family of $M!2^M$ equivalent points

For a successful application of neural network,

- it may not be necessary to find the global minimum (and in general it will not be known whether the global minimum has been found)
- but, it may be necessary to compare several local minima in order to find a sufficiently good solution

- There is no analytical solution
- Resort to iterative numerical procedures
- Choose some initial value $\mathbf{w}^{(0)}$ and then update it

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

- Different algorithms involve different choices for $\Delta\mathbf{w}^{(\tau)}$.
- Weight vector update $\Delta\mathbf{w}^{(\tau)}$ is usually based on gradient $\nabla E(\mathbf{w})$ evaluated at the weight vector $\mathbf{w}^{(\tau)}$
- In order to understand the importance of gradient information, it is useful to consider a local approximation to the error function based on a Taylor expansion.

- Consider a local quadratic approximation around $\mathbf{w}^\star$, a minimum of $E(\mathbf{w})$

$$E(\mathbf{w}) = E(\mathbf{w}^\star) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^\star)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^\star)$$

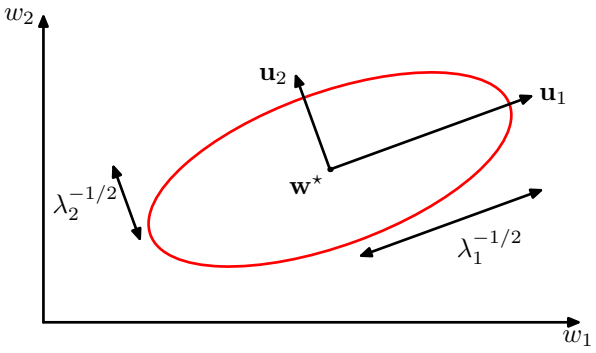
- Geometrical interpretation
  - Consider eigenvalue equation for $\mathbf{H}$

$$\mathbf{H}\mathbf{u}_i = \lambda_i \mathbf{u}_i$$

    - Eigenvectors $\mathbf{u}_i$ for a complete orthonormal set: $\mathbf{u}_i\mathbf{u}_j = \delta_{ij}$
  - Expand $(\mathbf{w} - \mathbf{w}^\star)$ as a linear combination of the eigenvectors

$$\mathbf{w} - \mathbf{w}^\star = \sum_i \alpha_i \mathbf{u}_i$$

    - can be regarded as a transformation of the coordinate system in which the origin is translated to the point $\mathbf{w}^\star$, and the axes are rotated to align with the eigenvectors



In the neighbourhood of a minimum $\mathbf{w}^\star$, the error function can be approximated by a quadratic. Contours of constant error are then ellipses whose axes are aligned with the eigenvectors $\mathbf{u}_i$ of the Hessian matrix, with lengths that are inversely proportional to the square roots of the correspond- ing eigenvectors λ_i.

$$
\begin{aligned}
E(\mathbf{w}) &= E(\mathbf{w}^\star) + \frac{1}{2}(\sum_i \alpha_i \mathbf{u}_i)^\top \mathbf{H} (\sum_i \alpha_i \mathbf{u}_i) \\
&= E(\mathbf{w}^\star) + \frac{1}{2}(\sum_i \alpha_i \mathbf{u}_i)^\top (\sum_i \alpha_i \mathbf{H}\mathbf{u}_i) \\
&= E(\mathbf{w}^\star) + \frac{1}{2}(\sum_i \alpha_i \mathbf{u}_i)^\top (\sum_i \alpha_i \lambda_i \mathbf{u}_i) \\
&= E(\mathbf{w}^\star) + \frac{1}{2}\sum_i \lambda_i \alpha_i^2
\end{aligned}
$$

- The stationary point $\mathbf{w}^\star$ will be a minimum if the Hessian matrix is positive definite, i.e., all its eigenvalues are positive.

Gradient Descent Optimization

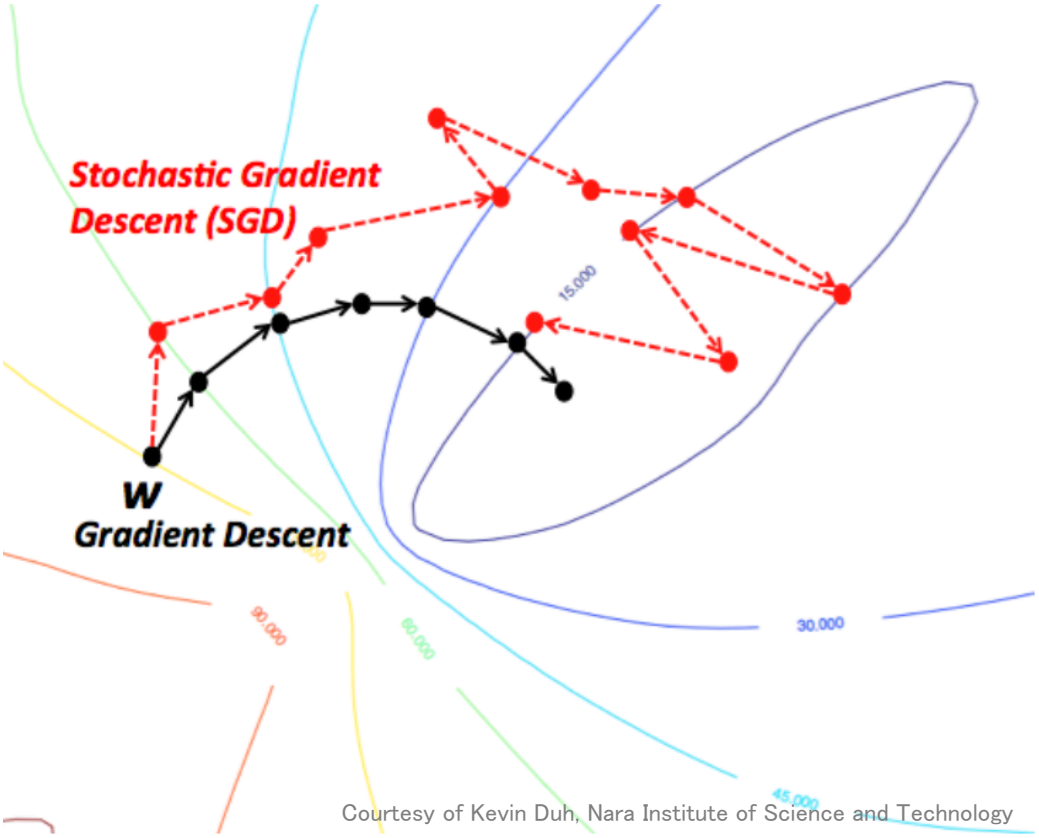
- Simplest approach to using gradient information
- Take a small step in the direction of the negative gradient

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

  - η: learning rate

[Different versions of gradient descent optimization]

- Batch: the entire training set to be processes to evaluate ∇E (a.k.a. *gradient descent* or *steepest descent*)
  - More efficient (robust and fast) methods: *conjugate gradients*, *quasi-Newton*
- On-line: one sample at a time (a.k.a. *stochastic gradient descent* or *sequential gradient descent*)
  - Repeated by cycling through the data either in sequence or by selecting at random with replacement
  - Efficiently handle redundancy in data
    - *e.g.*, duplicating every data point; simply multiplies the error function by a factor of 2 and so is equivalent to using the original error function
  - Possibility of escaping from local minima
    - Stationary point w.r.t. the error function for the whole data set will generally not be a stationary point for each data point individually
- Mini-batch stochastic gradient descent: a small set of samples at a time
  - good tradeoff between batch and on-line
  - approximation of the gradient of the loss function



Courtesy of Kevin Duh, Nara Institute of Science and Technology

Error Backpropagation

Overview

To find an efficient technique for evaluating the gradient of an error function $E(\mathbf{w})$ for a feed-forward neural network

- *Local message passing* scheme
- alternative information passing: forwards and backwards through the network

'Error backpropagation' or simply 'Backdrop'

Two states involved for minimization of an error function in an iterative procedure

1. Evaluation of the derivatives of the error function $\nabla E(\mathbf{w})$ w.r.t. the weights
   - Propagation of errors backwards through the network
   - Not limited to only multilayer perception, but applicable to many other kinds of network, including convolutional neural networks
   - Also applicable to error functions other that just the simple sum-of-squares and to the evaluation of other derivatives such as the Jacobian and Hessian matrices
2. Computing adjustments to the weights by using the derivatives

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

   - Can be tackled using a variety of optimization schemes, many of which are substantially more powerful than simple gradient descent

Evaluation of Error-Function Derivatives

- Derivation of backpropagation for a general network
  - Arbitrary feed-forward topology
  - Arbitrary differentiable nonlinear activation functions
  - A broad class of error function
- Error functions of practical interest comprise a sum of terms

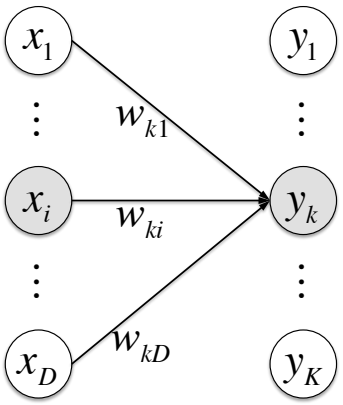
$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$

- In the following, we consider the problem of evaluating $\nabla E_n(\mathbf{w})$
  - Straightforward for stochastic gradient descent
  - Accumulate over whole training set for batch gradient descent

[Simple Linear Model]

$$\begin{aligned} y_k &= \sum_i w_{ki} x_i \\ y_{nk} &= y_k(\mathbf{x}_n, \mathbf{w}) \\ E_n &= \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2 \end{aligned}$$

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



- 'Local' computation
  - an 'error signal' $(y_{nk} - t_{nk})$ associated with the output end of the link w_{ki}
  - the variable x_{ni} associated with the input end of the link

[*Forward Propagation* in a General Feed-Forward Network]

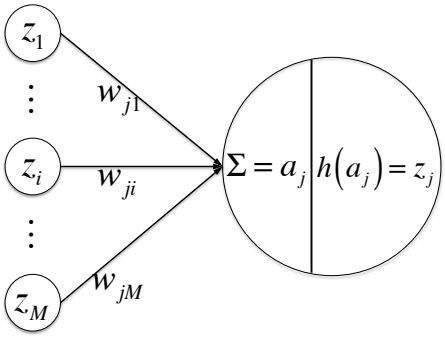
1. Each unit computes a weighted sum of its inputs

$$a_j = \sum_i w_{ji} z_i$$

   - z_i: activation of a unit (or input) that send a connection to unit j
   - w_{ji}: weight associated with that connection

2. Transformation by a nonlinear activation function $h(\cdot)$ to give the activation z_j of unit j

$$z_j = h(a_j)$$



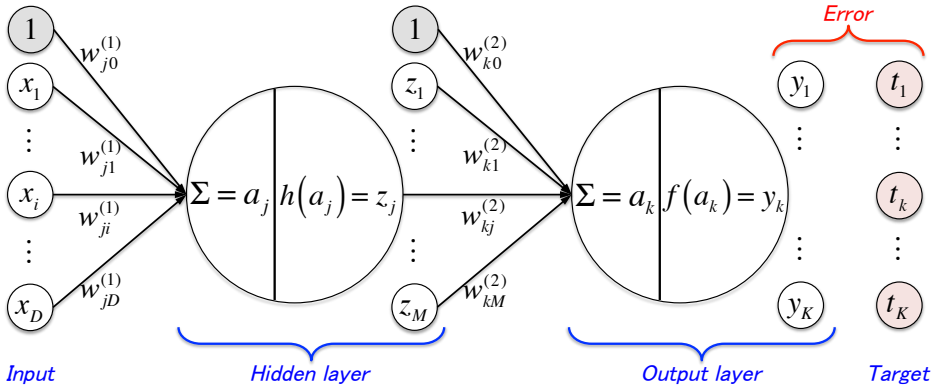
Forward flow of information through the network

Evaluation of the derivative of E_n w.r.t. w_{ji}

$$\frac{\partial E_n}{\partial w_{ji}} = \underbrace{\frac{\partial E_n}{\partial a_j}}_{\equiv \delta_j} \underbrace{\frac{\partial a_j}{\partial w_{ji}}}_{z_i} \quad \text{(by chain rule)}$$
$$= \delta_j z_i$$

δ_j: gradient of the error function w.r.t. the activation a_j

- Required derivative $\frac{\partial E_n}{\partial w_{ji}}$ is obtained by multiplying the value of δ_j for the unit at the output end of the weight by the value of z_i for the unit at the input end of the weight
- Note that this takes the same form as for the simple linear model
- Need only to calculate the value of δ_j for each hidden and output unit in the network



- For output unit k,

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

$$\frac{\partial E_n}{\partial a_k} \equiv \delta_k = f'(a_k)(y_{nk} - t_{nk}) \qquad f'(a_k) = \begin{cases} 1 & \text{identity} \\ y_{nk}(1 - y_{nk}) & \text{logistic sigmoid} \\ 1 - y_{nk}^2 & \tanh \\ \vdots & \vdots \end{cases}$$

- For hidden unit j,

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

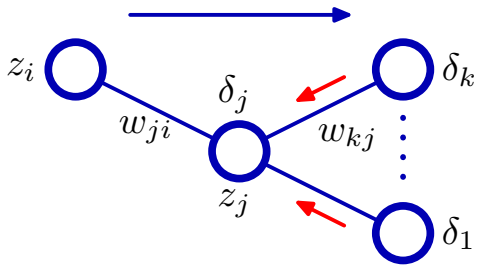
  - We are making use of the fact that variations in a_j give rise to variations in the error function only through variations in the variable a_k
  - By definition

$$\frac{\partial E_n}{\partial a_k} \equiv \delta_k$$

$$\frac{\partial a_k}{\partial a_j} = \frac{\partial\left(\sum_l w_{kl} z_l\right)}{\partial a_j} = \frac{\partial\left(\sum_l w_{kl} h(a_l)\right)}{\partial a_j} = h'(a_j) w_{kj}$$

$$\text{or} \quad \frac{\partial a_k}{\partial a_j} = \frac{\partial a_k}{\partial z_j}\frac{\partial z_j}{\partial a_j} = w_{kj} h'(a_j)$$

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



Blue arrow: information flow during forward propagation

Red arrow: backward propagation of error information

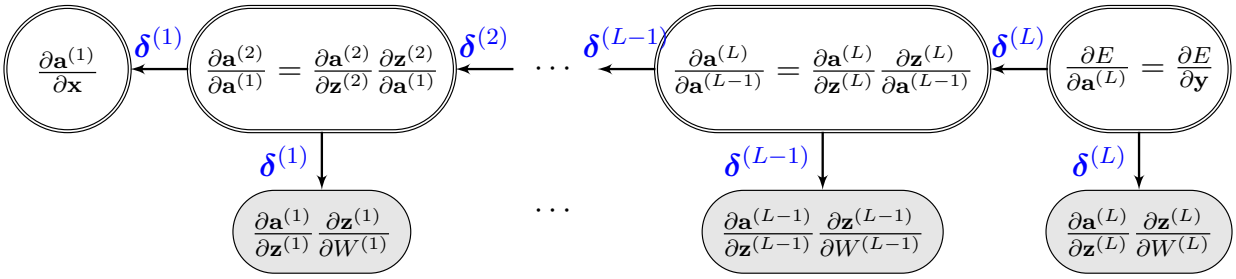
The value of δ for a particular hidden unit can be obtained by propagating the δ's backwards from units higher up in the network

- Forward propagation

$$\begin{aligned} a_j &= \sum_i w_{ji} z_i \\ z_j &= h(a_j) \end{aligned}$$

- Backward propagation

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



Graphical representation of a backpropagation algorithm in an L-layer network

[Error Backpropagation Algorithm]

1. Apply an input vector $\mathbf{x}_n$ to the network and forward propagate through the network to find the activations of all the hidden and output units

$$\begin{aligned} a_j &= \sum_i w_{ji} z_i \\ z_j &= h\left(a_j\right) \end{aligned}$$

2. Evaluate the δ_k for all the output units

$$\delta_k = f'\left(a_k\right)\left(y_k - t_k\right) \qquad f'\left(a_k\right) = \begin{cases} 1 & \text{identity} \\ y_k\left(1 - y_k\right) & \text{logistic sigmoid} \\ 1 - y_k^2 & \tanh \end{cases}$$

3. Backpropagate the δ's to obtain δ_j for each hidden unit in the network

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

4. Evaluate the required derivative

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

Further comments:

- In batch learning: repeat the steps for each pattern in the training set and then sum over all patterns

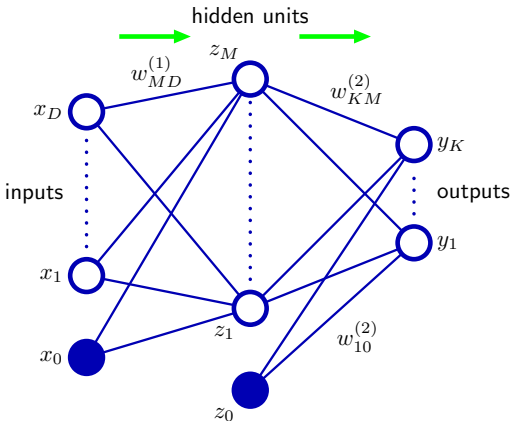
$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E_n}{\partial w_{ji}}$$

- Generalization: allowing different hidden/output units to have individual activation functions
  - Simply by keeping track of which form of an activation function goes with which unit

A Simple Example

- Topology: two-layer network
- Error function: sum-of-squared error

$$E_n = \frac{1}{2}\sum_{k=1}^{K}(y_k - t_k)^2$$



- Activation functions
  - Output units: linear function (*i.e.*, multiple regression)

$$y_k = a_k$$

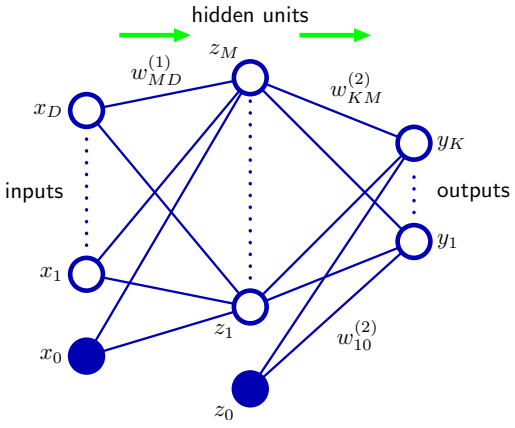
  - Hidden units: sigmoidal function

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

$$h'(a) = 1 - h(a)^2$$

Step 1: Forward propagation for each input in the training set

$$
\begin{aligned}
a_j &= \sum_{i=0}^{D} w_{ji}^{(1)} x_i \\
z_j &= \tanh(a_j) \\
y_k &= \sum_{j=0}^{M} w_{kj}^{(2)} z_j
\end{aligned}
$$



Step 2: Compute the error for each output unit

$$\delta_k = y_k - t_k$$

Step 3: Backpropagate the errors to the hidden units

$$\delta_j = (1 - z_j^2) \sum_{k=1}^{K} w_{kj} \delta_k$$

Step 4: Obtain the derivatives of weights w.r.t. the first- and second-layers

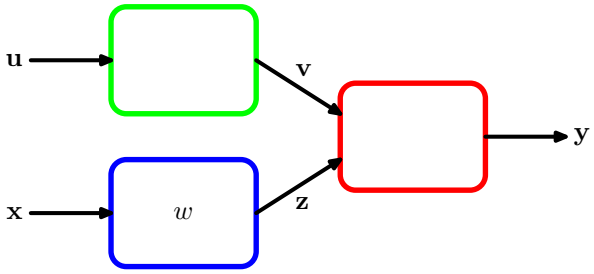
$$\frac{\partial E_n}{\partial w_{ji}^{(1)}} = \delta_j x_i, \qquad \frac{\partial E_n}{\partial w_{ji}^{(2)}} = \delta_k z_j$$

Step 5: Update weights by using gradient descent

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

$$\nabla E\left(\mathbf{w}\right) = \frac{\partial}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial E}{\partial w_{11}^{(1)}} \\ \vdots \\ \frac{\partial E}{\partial w_{M(D+1)}^{(1)}} \\ \frac{\partial E}{\partial w_{11}^{(2)}} \\ \vdots \\ \frac{\partial E}{\partial w_{K(M+1)}^{(2)}} \end{bmatrix}$$

- Play a useful role in systems built from a number of distinct modules



- Suppose we wish to minimize an error function E with respect to the parameter w
- Derivative of the error function

$$\frac{\partial E}{\partial w} = \sum_{k,j} \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial z_j} \frac{\partial z_j}{\partial w}$$

- Jacobian matrix provides a measure of the local sensitivity of the outputs to changes in each of the input variables
- Allows any known errors Δx_i associated with the inputs to be propagated through the trained network in order to estimate their contribution Δy_k to the errors at the outputs, through the relation

$$\Delta y_k \simeq \sum_i \frac{\partial y_k}{\partial x_i} \Delta x_i$$

  - Valid provided the $|\Delta x_i|$ are small

[Estimation of the Jacobian matrix using a backpropagation]

$$J_{ki} = \frac{\partial y_k}{\partial x_i} = \sum_j \frac{\partial y_k}{\partial a_j} \frac{\partial a_j}{\partial x_i} = \sum_j w_{ji} \frac{\partial y_k}{\partial a_j}$$

- Summation runs over all units j to which the input unit i sends connections
- Recursive backpropagation formula to determine the derivative $\frac{\partial y_k}{\partial a_j}$

$$\frac{\partial y_k}{\partial a_j} = \sum_l \frac{\partial y_k}{\partial a_l} \frac{\partial a_l}{\partial a_j} = h'(a_j) \sum_l w_{lj} \frac{\partial y_k}{\partial a_l}$$

- Starting at the output units
  - The required derivatives can be found directly from the functional form of the output-unit activation function

$$\underbrace{\frac{\partial y_k}{\partial a_l} = \delta_{kl} \sigma'(a_l)}_{K=2} \quad \text{or} \quad \underbrace{\frac{\partial y_k}{\partial a_l} = \delta_{kl} y_k - y_k y_l}_{K>2}$$

1. Apply the input vector corresponding to the point in input space at which the Jacobian matrix is to be found
2. Forward propagate in the usual way to obtain the activations of all of the hidden and output units in the network
3. For each row k of the Jacobian matrix, corresponding to the output unit k, backpropagate using the recursive relation

$$\frac{\partial y_k}{\partial a_j} = h'(a_j) \sum_l w_{lj} \frac{\partial y_k}{\partial a_l} \quad \text{starting with} \quad \begin{cases} K = 2: & \frac{\partial y_k}{\partial a_l} = \delta_{kl}\sigma'(a_l) \\ K > 2: & \frac{\partial y_k}{\partial a_l} = \delta_{kl} y_k - y_k y_l \end{cases}$$

4. Finally obtain elements of the Jacobian

$$J_{ki} = \frac{\partial y_k}{\partial x_i} = \sum_j w_{ji} \frac{\partial y_k}{\partial a_j}$$

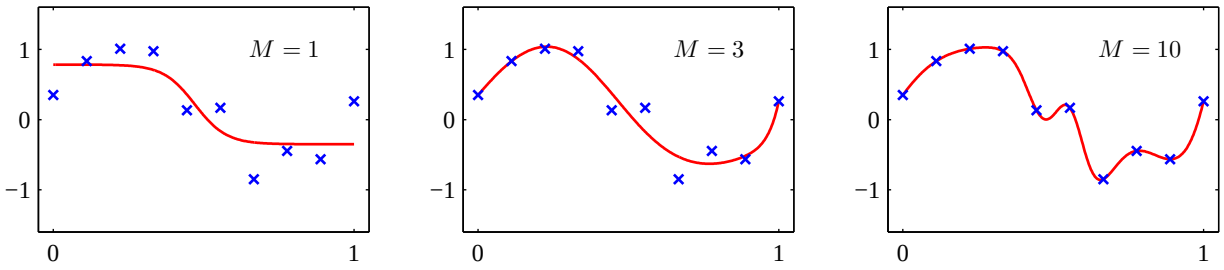
Regularization in Neural Networks

Regularization

- Introducing additional information to prevent over-fitting or to solve ill-posed problem
- In different forms as a penalty to complexity
  - Restrictions for smoothness
  - Bounds on the vector space norm
- Theoretical justification for regularization
- From a Bayesian perspective
  - Corresponds to the imposition of a prior distribution on model parameters
  - Attemps to impose Occam's razor on the solution

In neural network, regularization by determining the number of hidden units

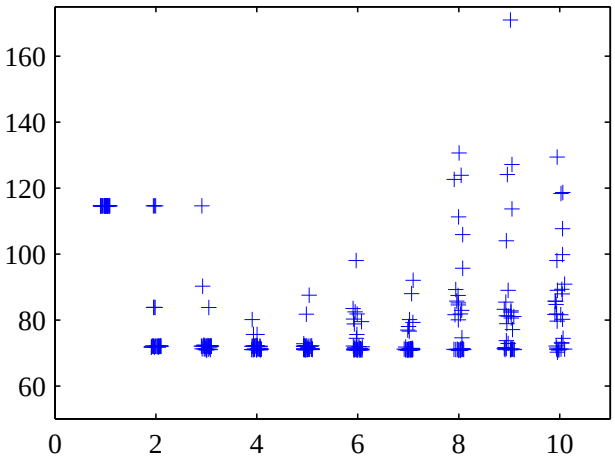
- The number of input and output units is determined by the dimensionality of the data set and the number of target values or classes, respectively.
- The number M of hidden units is a free parameter that can be adjusted to give the best predictive performance.
  - In a maximum likelihood setting, there will be an optimum value of M that gives the best generalization performance, corresponding to the optimum balance between under-fitting and over-fitting.



Examples of two-layer networks trained on 10 data points drawn from the sinusoidal data set.

The generalization error is not a simple function of M due to the presence of local minima in the error function.

- x-axis: number of hidden units in the network
- y-axis: sum-of-squares test-set error for the polynomial data set
- 30 random starts for each network, showing the effect of local minima
- The weight vector was initialized by sampling from $\mathcal{N}(\mathbf{0}, 10\mathbf{I})$



In practice, we can plot a graph of validation set error vs. M, and then choose the specific solution having the smallest validation set error.

Weight Decay

Choose a relatively large M and control complexity by the addition of a regularization term to the error function

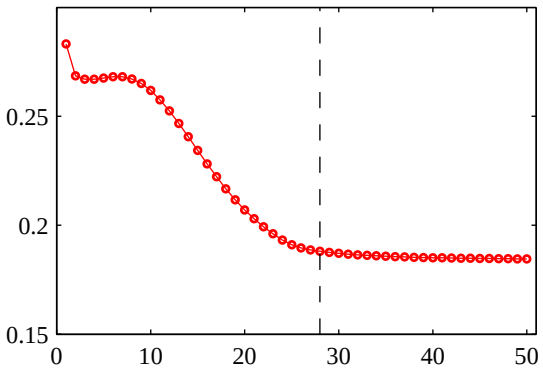
$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^\top \mathbf{w}$$

- λ: regularization coefficient
- Can be interpreted as the negative logarithm of a zero-mean Gaussian prior distribution over the weight vector $\mathbf{w}$

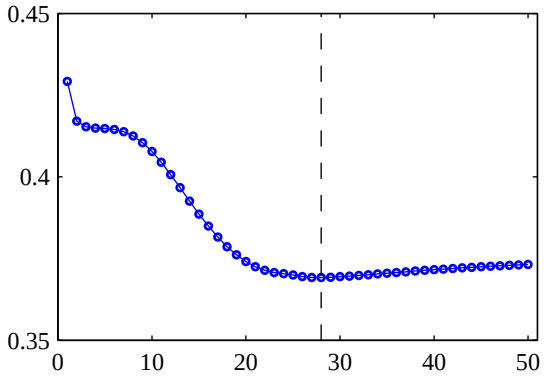
Early Stopping

- Alternative to regularization in controllong the effective complexity of a network
- For many of the optimization algorithms used for network training, such as conjugate gradients, the error is a non-increasing function of the iteration index
- However, the error measured w.r.t. independent data, generally called a validation set, often shows a decrease at first, followed by an increase as the network starts to overfit.
- Training can therefore be stopped at the point of smallest error w.r.t. the validation data set in order to obtain a network having good generalization performance.

Training data set error



Validation set error

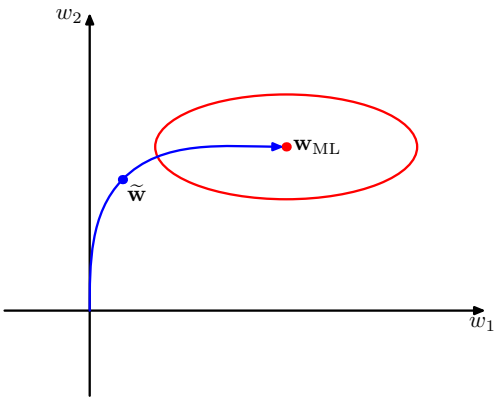


Halting training before a minimum of the training error has been reached represents a way of limiting the effective network complexity.

[Interpreting the effect of Early Stopping]

- Consider a quadratic error function
- Axes in weight space have been rotated to be parallel to eigenvectors of the Hessian matrix
- In the absence of weight decay, the weight vectors starts at the origin and proceeds to $\mathbf{w}_{ML}$
- Stopping at $\tilde{\mathbf{w}}$ is similar to weight decay

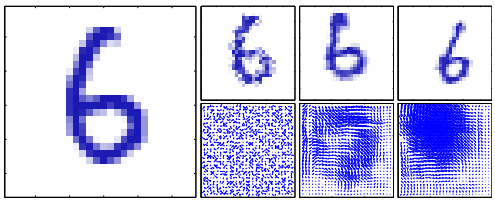
$$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^{\top}\mathbf{w}$$



Contour: constant error

Invariances

- In many applications of PR, predictions should be unchanged, or *invariant*, under one or more transformations of the input variables.
  - *e.g.*, handwritten digits recognition: position within the image (*translation invariance*), size (*scale invariance*), rotation (*rotation invariance*)



- Sufficiently large number of examples of the effects of the various transformations
  - adaptive model, *e.g.*, neural networks, that can learn the invariance, at least approximately
  - impractical: exponential growing number according to the number of combinations of transformations

Machine Intelligence Lab

Alternative approaches

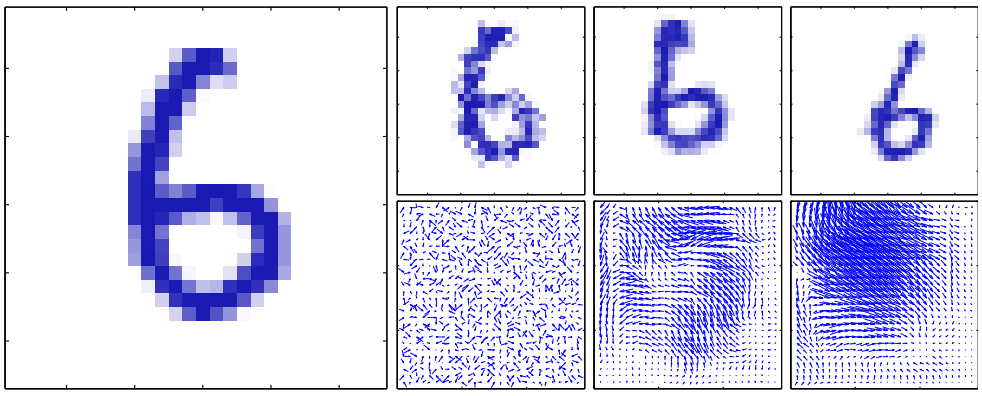
- *Preprocessing* by extracting features that are invariance under the required transformations
- *Augmentation* by using replicas of the training patterns, transformed according to the desired invariances
- Regularization that penalizes changes in the model output when the input is transformed $\rightarrow$ *tangent propagation*
- Building the invariance properties into the structure of a neural network $\rightarrow$ *convolutional neural networks*

Machine Intelligence Lab

Convolutional Neural Networks

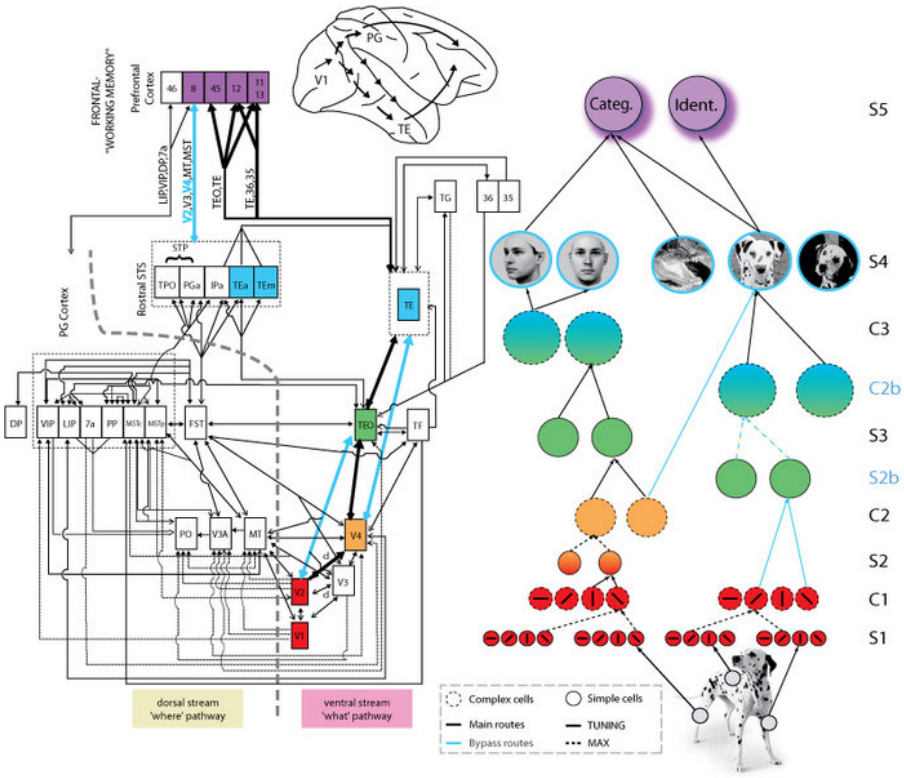
e.g.) Handwritten digits recognition

- The identify of the digit is invariant under translations and scaling as well as (small) rotations
- Network must also exhibit invariance to more subtle transformations such as elastic deformation



Fully connected network

- Given a sufficiently large training set, such a network in principle yield a good solution to this problem and would learn the appropriate invariances by example
- Ignores a key property of images
  - Nearby pixels are more strongly correlated than more distant pixels
  - Modern approaches to computer vision exploit this property by extracting *local* features that depend only on small subregions of the image
  - Local features that are useful in one region of the image are likely to be useful in other regions of the image, for instance if the object of interest is translated
  - Information can be merged at later stages to get higher order features and ultimately to yield information about the image as whole



Serre et al., PNAS 2007

Convolutional Neural Networks (CNNs)

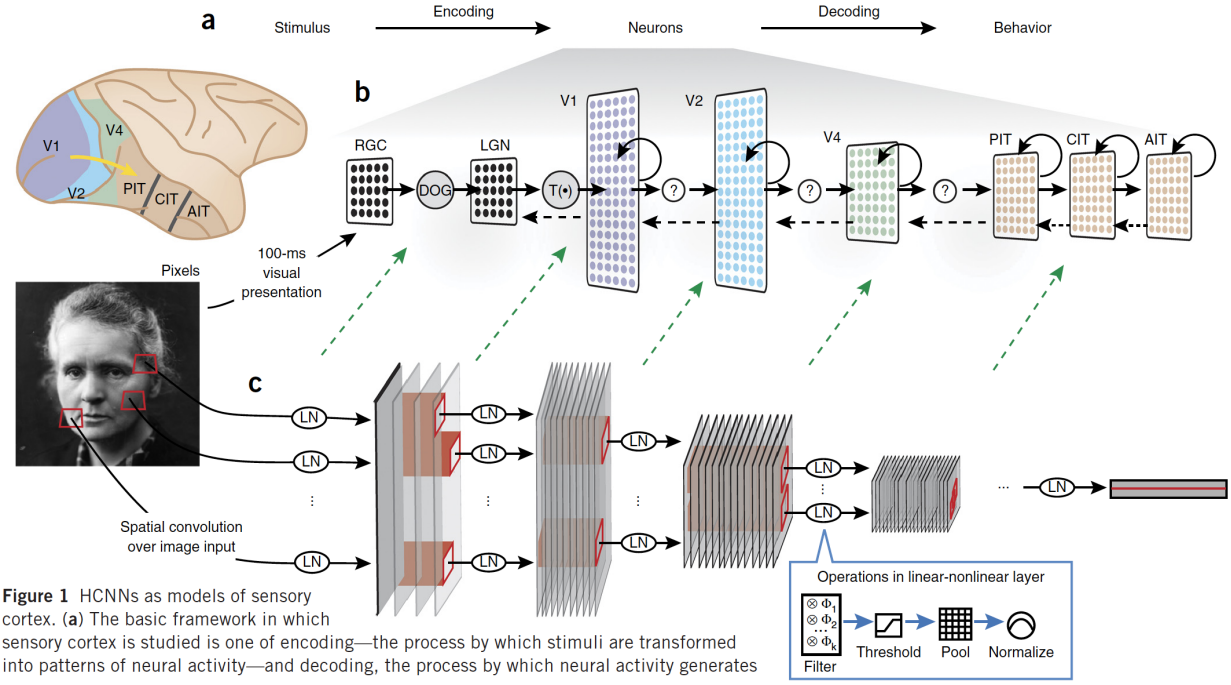
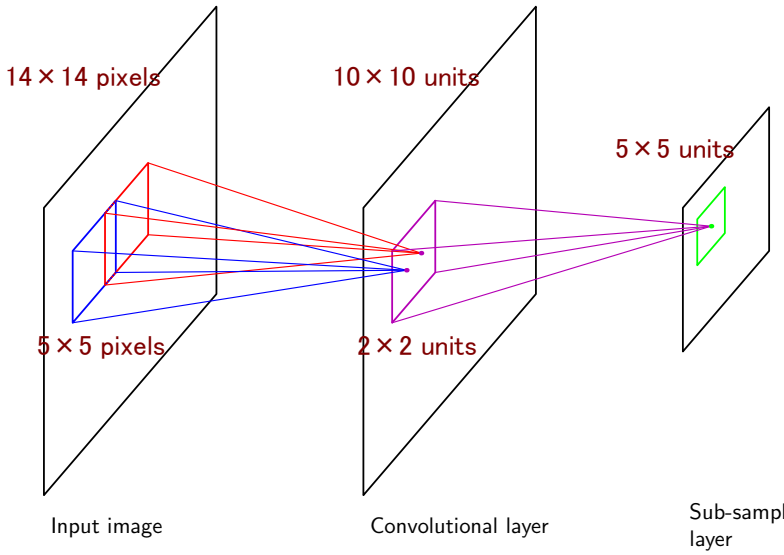


Figure 1 HCNNs as models of sensory cortex. (**a**) The basic framework in which sensory cortex is studied is one of encoding—the process by which stimuli are transformed into patterns of neural activity—and decoding, the process by which neural activity generates

Yamins & DiCarlo, *Nature Neuroscience*, 2016

Three mechanisms

1. Local receptive fields
2. Weight sharing
3. Subsampling



[Convolutional Layer]

- Units are organized into planes ('*feature maps*')
- Units in a feature map take inputs only from a subregion of the input
- All units in a feature map are constrained to share the same weight values
  - Small # of weights compared to a fully connected network

- Input values from a patch (receptive field) are linearly combined using the weights and the bias, and the result is transformed by a nonlinear function

$$C = h\left(X * W + b\right)$$

where $*$: a convolution operator

  - Units can be regarded as *feature detectors*
  - All of the units in a feature map detect the same pattern but at different locations in the input

- If the input image is shifted, the activations of the feature map will be shifted by the same amount but will otherwise be unchanged
  - Basis for the (approximate) invariance of the network outputs to translations and distortions of the input

- Multiple feature maps: to detect multiple features in order to build an effective model

$$C_j = h\left(X * W_j + b_j\right)$$

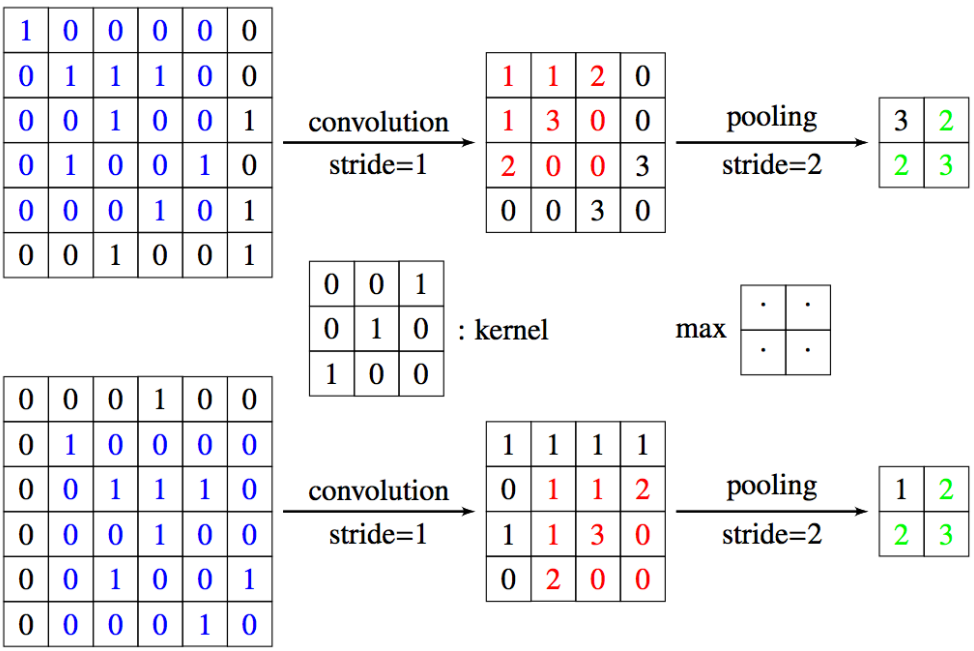
  - Each feature map has its own set of weights and bias parameters

[Subsampling Layer]

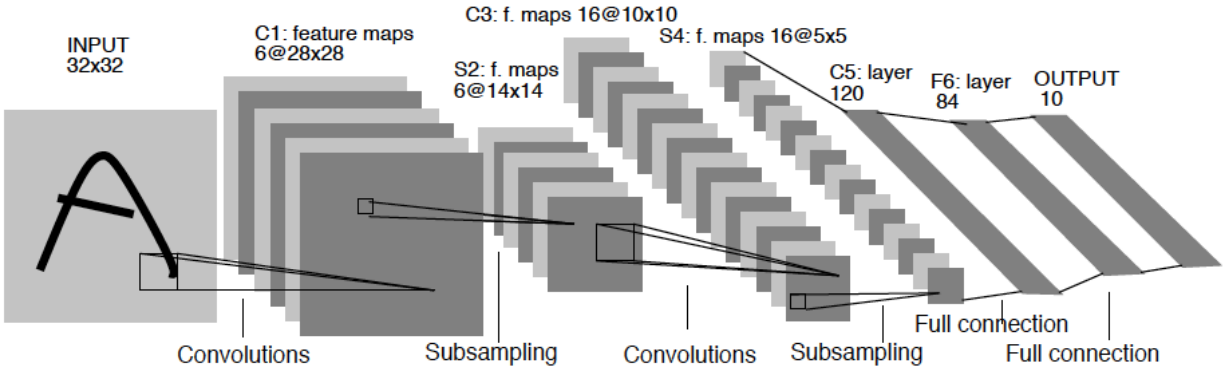
- The outputs of the convolutional units form the inputs to the subsampling layer of the network.
- For each feature map in the convolutional layer, there is a plane of units, each of which takes inputs from a small receptive field in the corresponding feature map of the convolutional layer.

$$S_j = \text{down}\left(C_j\right)$$

  - down $(\cdot)$: average or max function
  - The receptive fields are chosen to be continuous and non-overlapping.
- To reduce computation time and to gradually build up further *spatial* and *configural* invariance
  - Relatively insensitive to small shifts of the image in the corresponding regions of the input space
  - A small sub-sampling factor is desirable in order to maintain *specificity* at the same time.



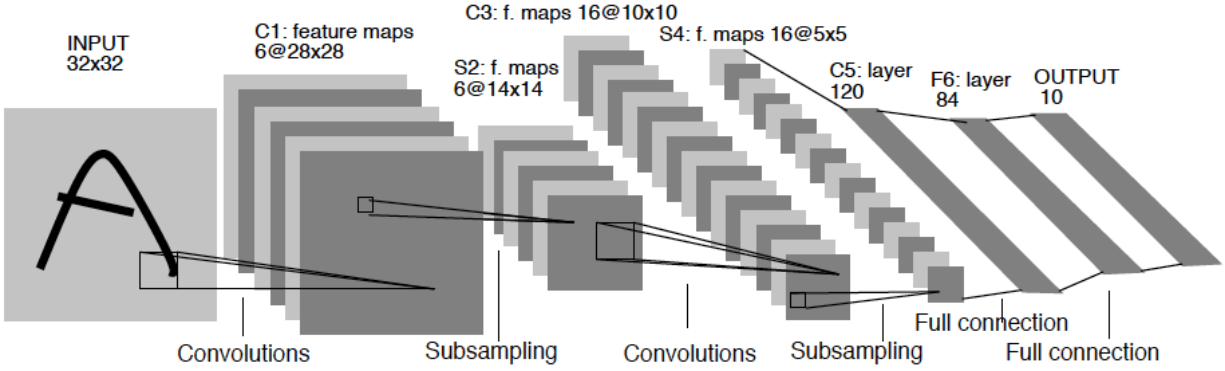
Practical Architecture



- Several pairs of convolutional and subsampling layers

$$C_j^{(l)} = h\left(\sum_i S_i^{(l-1)} * W_{ji}^{(l)} + b_j^{(l)}\right)$$

$$S_j^{(l)} = \text{down}\left(C_j^{(l-1)}\right)$$



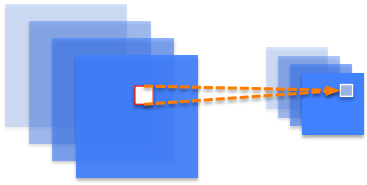
- At each stage, there is a larger degree of invariance to input transformations compared to the previous layer
- Several feature maps in a given convolutional layer for each plane of units in the previous subsampling layer
  - Gradual reduction in spatial resolution is compensated by an increasing number of features.
- The final layer of the network would typically be fully connected.

Convolutional Network Training

- Error minimization using backpropagation to evaluate the gradient of the error function
  - Needs a slight modification of the usual backpropagation algorithm to ensure that the shared-weight constraints are satisfied

[Computing the Gradients: Convolution Layers]

- A convolutional layer l followed by a subsampling layer $l+1$, one pixel in the next layer's associated sensitivity map Δ corresponds to a block of pixels in the convolutional layer's output map.



- Thus, each unit in a map at layer l connects to only one unit in the corresponding map at layer $l+1$.
- To compute the sensitivities at layer l efficiently, we can *upsample* the subsampling layer's sensitivity map to make it the same size as the convolutional layer's map,

$$\Delta_j^{(l)} = \left(h'\left(Z_j^{(l)}\right) \circ \text{up}\left(\Delta_j^{(l+1)}\right)\right)$$

- Finally, the gradients for the learnable weights are computed using backpropagation, except in this case the same weights are shared across many connections.
- We'll therefore sum the gradients for a given weight over all the connections that mention this weight:

$$\frac{\partial E}{\partial W_{ji}^{(l)}} = \sum_{u,v} \left(\Delta_j^{(l)}\right)_{uv} \left(P_i^{(l-1)}\right)_{uv}$$

  - $\left(P_i^{(l-1)}\right)_{uv}$: *patch* in $Z_i^{(l-1)}$ that was multiplied element wise by $W_j^{(l)}$ in the output convolution map $Z_j^{(l)}$

[Computing the Gradients: Subsampling Layers]

- Assume that the subsampling layers are surrounded above and below by convolution layers
- If the layer following the subsampling layer is a fully connected layer, then the feature maps for the subsampling layer can be computed with the backpropagation.
- We need to figure out which patch in the current layer's sensitivity map corresponds to a given pixel in the next layer's feature map in order to apply a delta recursion

Reducing the Effective Complexity of a Network

- Convolutional neural network (LeCun *et al.*, 1998)
  - Constrains weights within certain groups to be equal
  - Successful in various applications with the state-of-the-art performance
  - Applicable to particular problems in which the form of the constraints can be specified in advance

- **Soft weight sharing (Nolan and Hinton, 1992)**
  - Hard constraint of equal weights is replaced by a form of *regularization in which groups of weights are encouraged to have similar values*.
  - Division of weights into groups, the mean weight value for each group, and the spread of values within the groups are all determined as part of the learning process.

Bayesian Neural Networks

Introduction

Learning weights and biases in neural networks

- Focusing on the use of maximum likelihood
- Regularized maximum likelihood: interpreted as MAP approach
  - Regularizer: logarithm of prior parameter distribution
- Bayesian treatment: marginalize over the distribution of parameters in order to make predictions
  - Nonlinear dependence of the network functions on the parameter values
  - No exact evaluation possible
  - Log of the posterior distribution is non-convex implying multiple local minima in the error function

Approaches to Bayesian Treatment of NN

1. Variational inference
   - a factorized Gaussian approximation to the posterior distribution [Hinton and van Camp, 1993]
   - a full-covariance Gaussian [Barber and Bioshop, 1998]
2. Laplace approximation [MacKay, 1992]
   - Approximate the posterior distribution by a Gaussian, centered at a mode of the true posterior
   - Assume that the covariance of a Gaussian is small so that the network function is approximately linear w.r.t. the parameters over the region of parameter space for which the posterior probability is significantly nonzero.

With the approximations, *i.e.*, variational or Laplace, we can make use of the *evidence framework* (Chapter 3)

- to provide point estimates for the hyperparameters
- to compare alternative models (*e.g.*, networks having different numbers of hidden units)

Posterior Parameter Distribution

Predicting a single continuous target variable t from a vector $\mathbf{x}$

- Assume that the conditional distribution $p(t|\mathbf{x})$ is Gaussian

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

$$y(\mathbf{x}, \mathbf{w}) = \sum_l W_{:l}^{(L)} h\left(\sum_m W_{ml}^{(L-1)} h\left(\cdots h\left(\sum_i W_{ij}^{(1)} x_i\right)\right)\right)$$

c.f.,) $y(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\top \phi(\mathbf{x}_n)$ (linear regression)

- Prior distribution over the weights $\mathbf{w}$: Gaussian

$$p(\mathbf{w}|\alpha) = \mathcal{N}\left(\mathbf{w}|\mathbf{0}, \alpha^{-1}\mathbf{I}\right)$$