Neuronale Netze

Recurrent Networks und Radial Basis Function Networks

Christian Mohr

20.12.2011

Recurrent Networks

- Networks in which units may have connections to units in the same or preceding layers
- Also connections to the unit itself possible
- Already covered:
  - Hopfield Nets (no general RNN)
  - Boltzmann Machines

Recurrent Networks

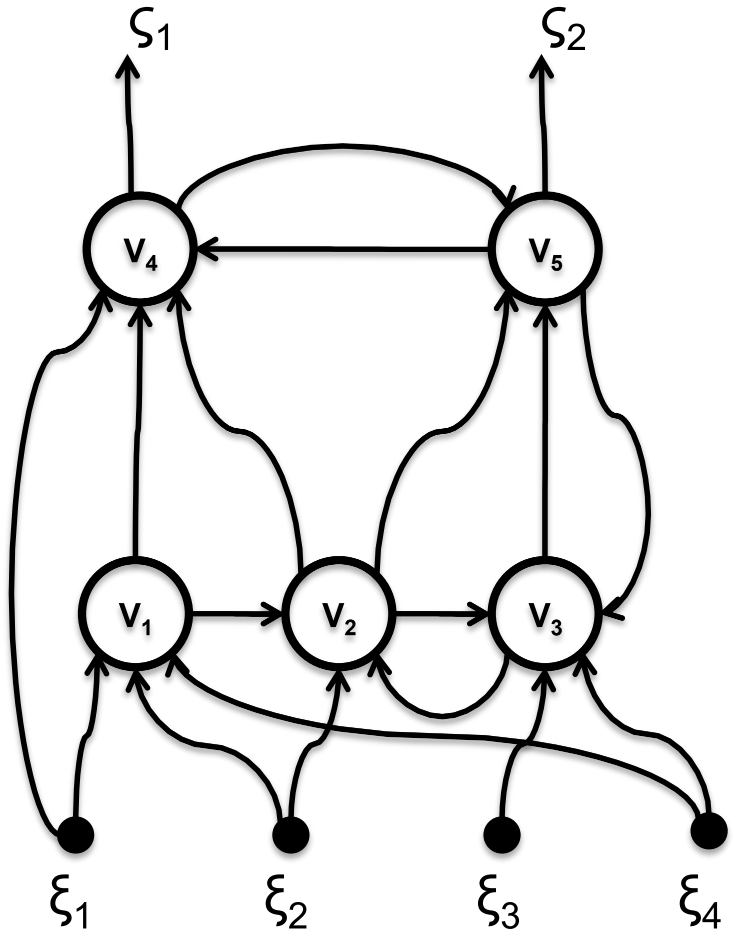
- Arbitrary connection weights
- As seen in Hopfield Nets / Boltzmann:
    - Symmetric weights: Networks will settle down to a stable state
- But nets with non symmetric weights are not necessarily unstable
- Unstable nets can be used to recognize or reproduce time sequences

Recurrent Back-Propagation

- Back-Propagation can be extended to arbitrary networks if they converge to stable states (we use continuous valued units, so states are called points)
- Use a modified version of the network itself to calculate the weights

Recurrent Back-Propagation

- Consider N continuous-valued units V_i
- With weights w_{ij} and activation function g(h)
- Some units are input units with input ξ^{μ}_{i} specified in patterns μ
- All other units input is 0
- Also some units are output units with output value denoted with ς^{μ}_{i}



Recurrent Back-Propagation

- Consider the net to apply the following evolution rule

$$\tau \frac{dV_i}{dt} = -V_i + g\left(\sum_j w_{ij} V_j + \xi_i\right)$$

which is the differential equation for continuous valued nets that also update continuously

- Then stable points are where $dV_i / dt = 0$:

$$V_i = g\left(\sum_j w_{ij} V_j + \xi_i\right)$$

Recurrent Back-Propagation

- We assume at least one stable point
- As error measure we use:

$$E = \frac{1}{2} \sum_k E_k^2$$

with

$$E_k = \begin{cases} \varsigma_k - V_k & \text{if } k \text{ is an output} \\ 0 & \text{otherwise} \end{cases}$$

Recurrent Back-Propagation

- The delta-rule for recurrent back-propagation is*

$$\Delta w_{ij} = \eta g'(h_i) Y_i V_j$$

where h_i is the net input to a unit I and g' is the derivative of h

- To find the Y's the following differential equation has to be solved

$$\tau \frac{dY_i}{dt} = -Y_i + g\left(\sum_j g'(h_j) w_{ji} V_j + E_i \right)$$

which can be done by evolution of a new **error-propagation network**

Recurrent Back-Propagation

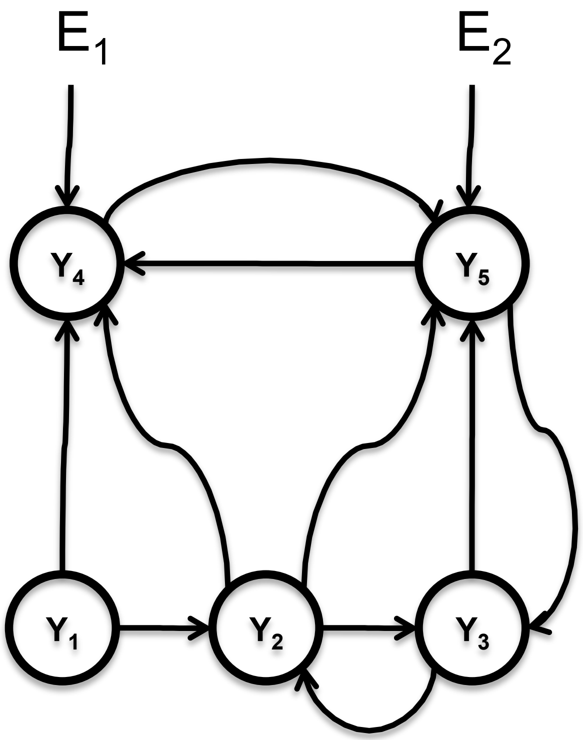
- The error-propagation has the same topology as the original net
- Weights:

$$\dot{w}_{ij} = g'(h_i) w_{ij}$$

- Transfer function:

$$\dot{g}(x) = x$$

- Inputs: Errors E_i of the units i in the original network



Recurrent Back-Propagation

- Training procedure:
  - Relax original net to find V_i's
  - Compute E_i's
  - Relax error-propagation network to find Y_i's
  - Update the weights using

$$\Delta w_{ij} = \eta g'(h_i) Y_i V_j$$

Recurrent Back-Propagation

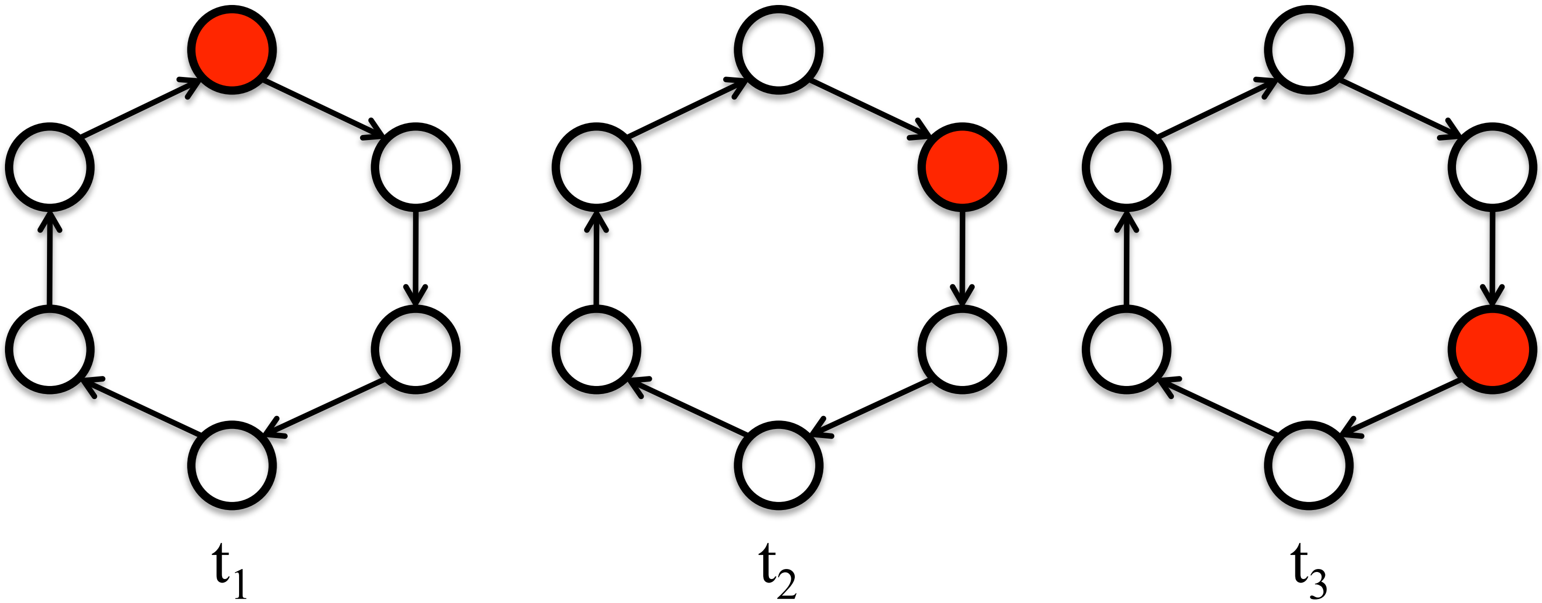
- Recurrent Back-Propagation scales with N^2 with N units in the net
- The use of recurrent nets gives a large improvement in performance over normal feed-forward for a number of problems as pattern completion

Temporal Sequences of Patterns

- So far only fixed patterns
- Extension: Sequence of patterns
    - No stable state, but go through a predetermined sequence of states.

Temporal Sequences of Patterns

- Simple Example of sequence generation:
  - Synchronous updating and equal weights
  - Just turn on the first unit
  - Only simple sequences and not very robust



Temporal Sequences of Patterns

- For more arbitrary sequences using asynchronous updating we need asymmetric connections
- Instead of using:

$$w_{ij} = \frac{1}{N}\sum_{\mu} \xi_i \xi_j$$

add an additional term:

$$w_{ij} = \frac{1}{N}\sum_{\mu} \xi_i^{\mu} \xi_j^{\mu} + \frac{\lambda}{N}\sum_{\mu} \xi_i^{\mu+1} \xi_j^{\mu}$$

- Uses information from the next pattern

Temporal Sequences of Patterns

- Asynchronous updating tends to dephase the system
- Net reaches states that overlap several consecutive patterns
- Method only usable for short sequences
- Possible solution:
    - Fast and slow connections

$$h_i(t) = \sum_{\mu} w_{ij}^{S} V_j(t) + \lambda w_{ij}^{L} \overline{V}_j(t)$$

Temporal Sequences of Patterns

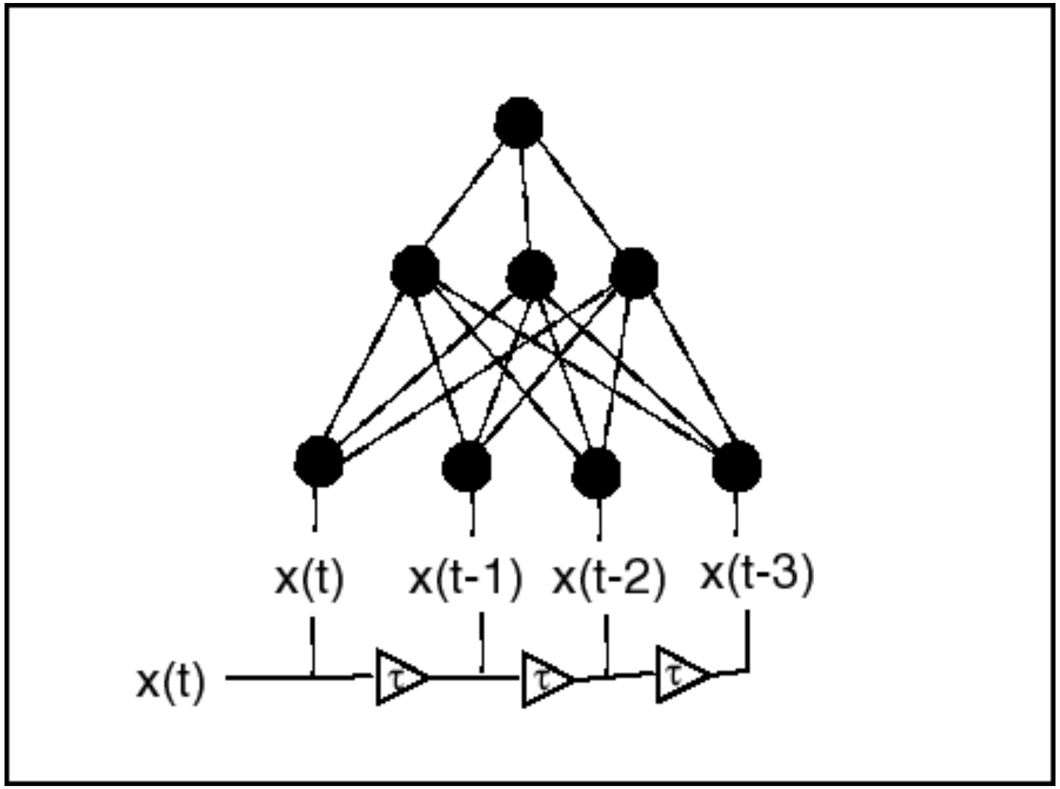
- Conclusion:
  - No feed forward net nor nets with symmetric weights are capable of pattern sequences
  - Sequences are calculated not learned
  - How can a recurrent net learn a pattern sequence?

Learning Time Sequences

- 3 distinct tasks
  - Sequence Recognition: Produce output pattern from input pattern sequence
  - Sequence Reproduction: ≈ Pattern completion with dynamic patterns
  - Temporal Association: Produce output pattern sequence from input pattern sequence

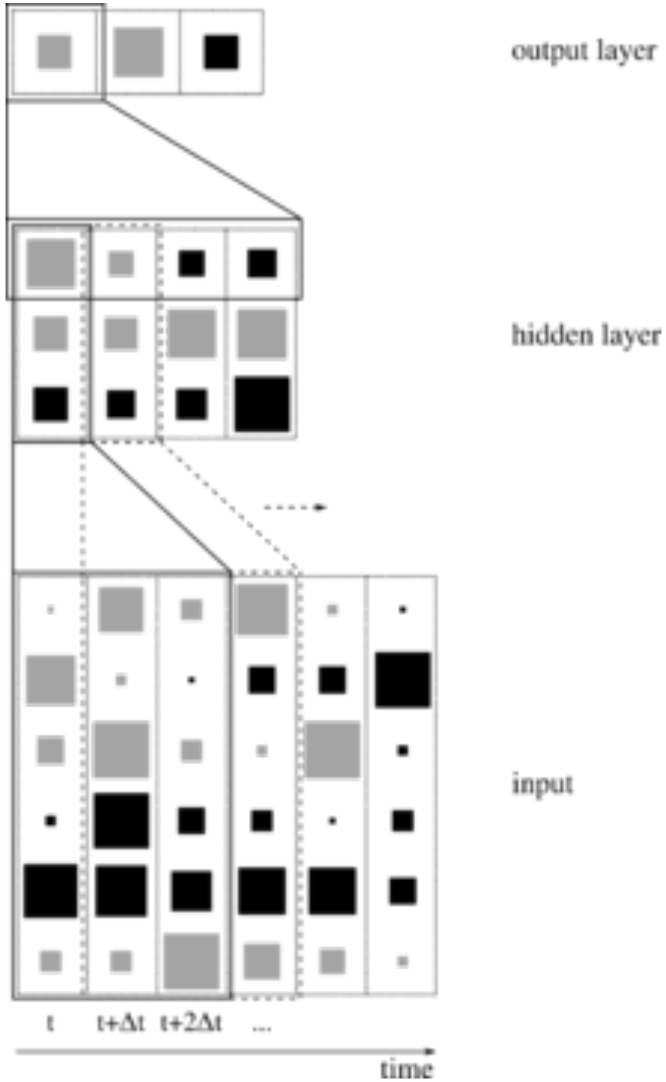
Tapped Delay Lines

- Easiest way of sequence recognition (not recurrent)
- Turn time pattern into spatial pattern
- So several time steps are presented to the net simultaneously



Tapped Delay Lines

- Widely applied to speech recognition task
  - Time-Delay Neural Networks [Waibel et. al. '89]
  - Vectors of spectral coefficients as time signal (2 dimensional time-frequency plane)
  - Train network with spectra of phonemes

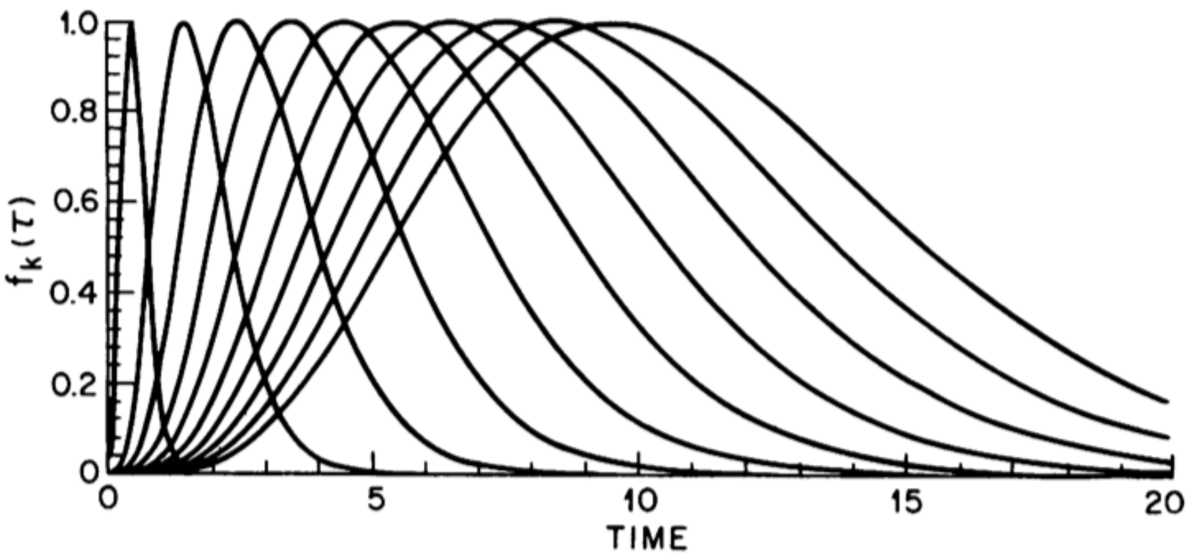


Tapped Delay Lines

- General approach yields several drawbacks to sequence recognition
  - Maximum length of possible sequence has to be chosen in advance
  - High number of units = slow computation
  - Timing has to be very accurate

Tapped Delay Lines

- Solution to timing problem:
  - Use filters that broaden the time signal instead of fixed delay
  - The longer the delay, the broader the filter



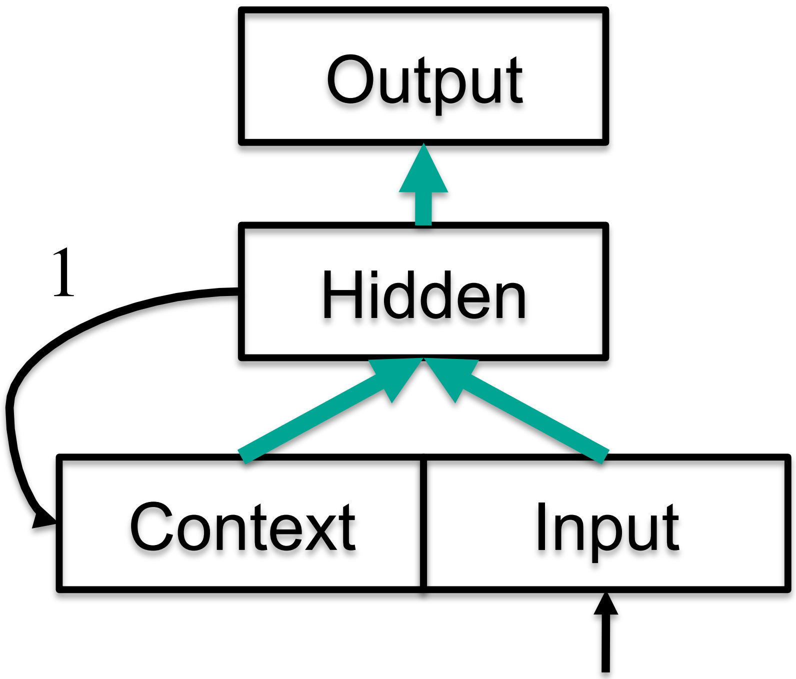
Context Units

- Partially recurrent networks
  - Mainly feed-forward, but carefully chosen set of feedback connections
- Mostly fixed feedback weights
  - Does not complicate training
- Synchronous updating (one update for all units at a discrete time step)
- Also referred to as **Sequential Units**

Context Units

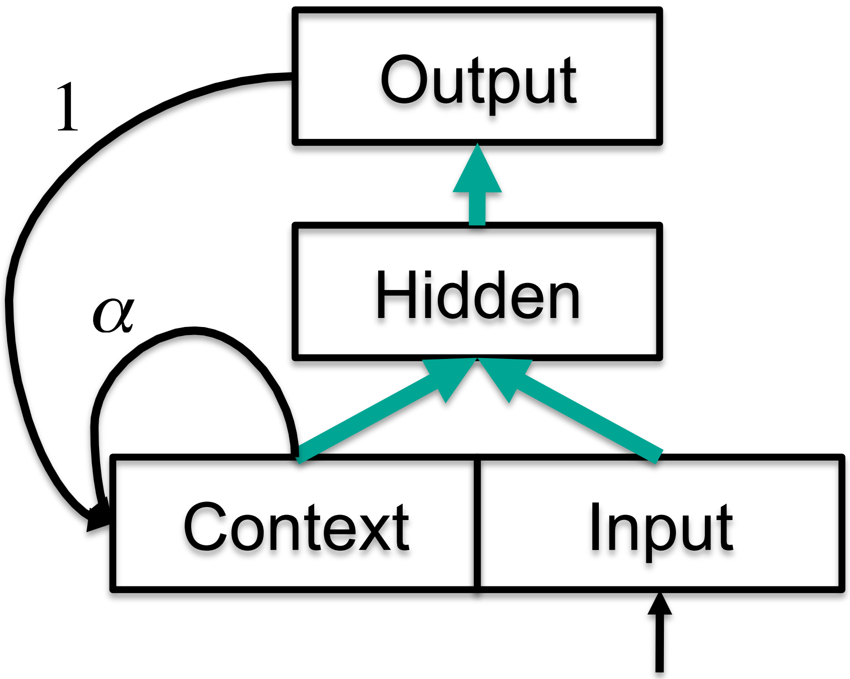
- Different architectures with a whole or part of a layer being **Context Units**
- Context units receive some signals from the net at a time t and forward them at t+1
- Net remembers some aspects of previous time steps
- State of the net depends on past states and current input
- Net can recognize sequences based on its state at the end of a sequence (and generate in some cases)

Context Units
Elman Nets



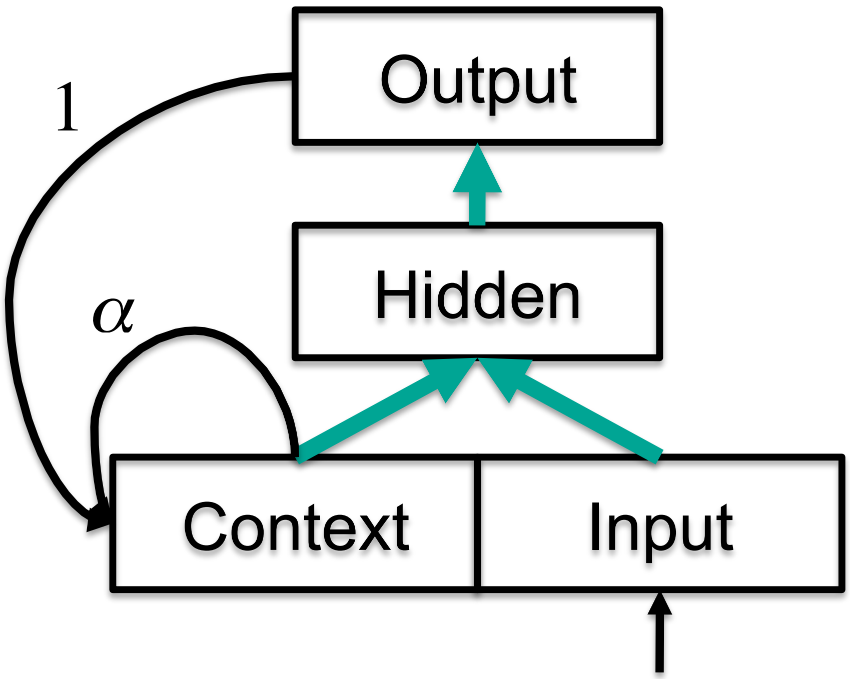
- Hidden Units hold a copy of the output of the hidden layer
- Modifiable connections all feed-forward (Backpropagation)
- Usable for recognition and short continuations
- Also can mimic finite state machines

Context Units
Jordan Nets



- Context Units hold a copy of the output layer and themselves
- Self connection gives individual memory or inertia
- With fixed outputs context units would decay exponentially (**Decay Units**)

Context Units
Jordan Nets



- Weighted moving average or trace

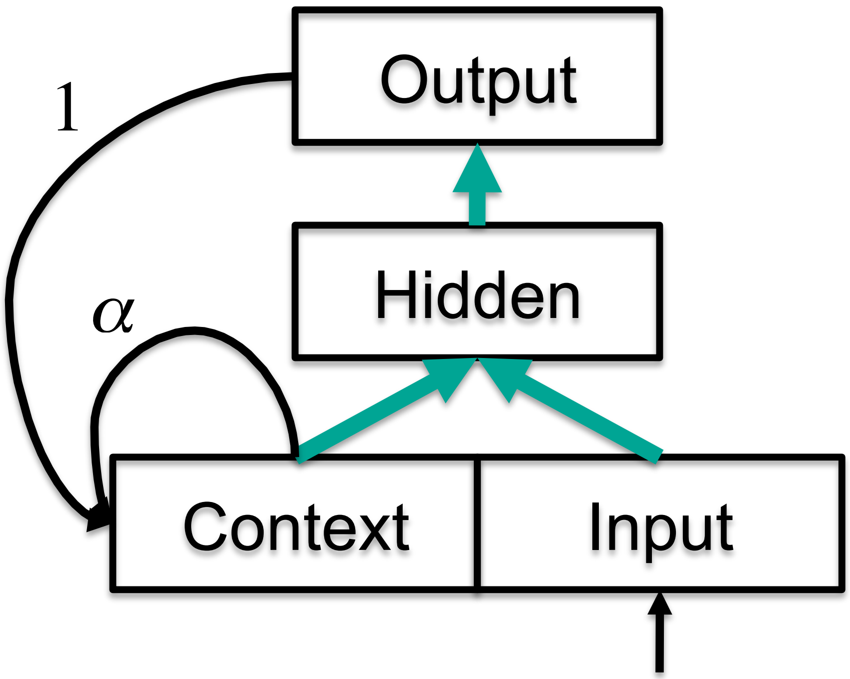
$$C_i(t+1) = \alpha C_i(t) + O_i(t)$$

$$= O_i(t) + \alpha O_i(t-1) + \dots$$

$$= \sum_{t'=0}^{t} \alpha^{t-t'} O_i(t')$$

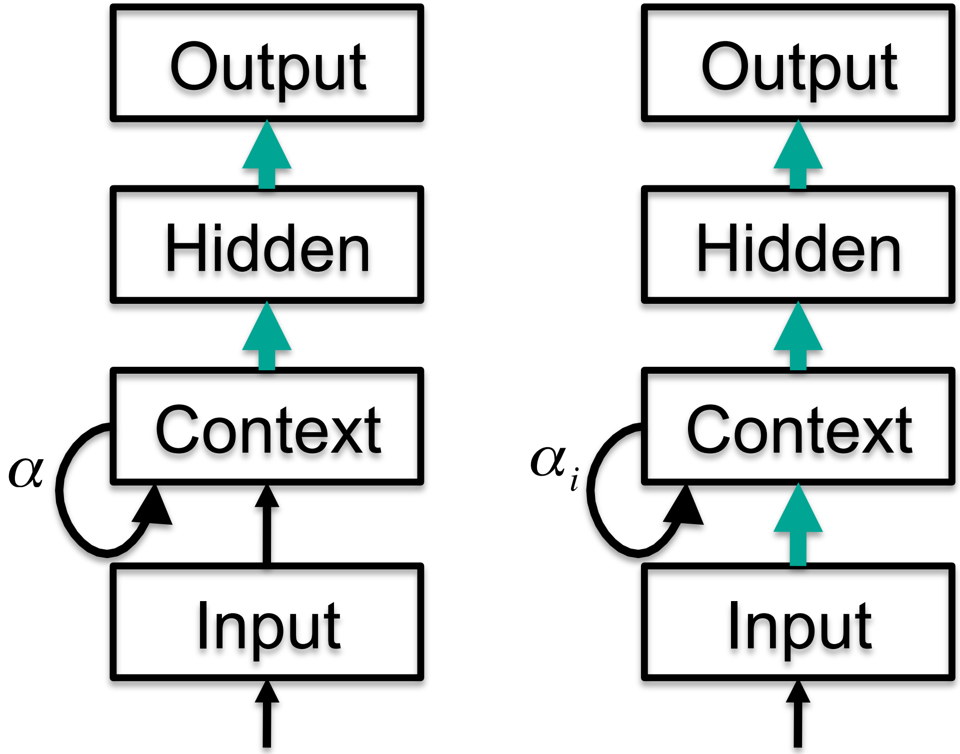
$$cont. \quad = \int_0^t e^{-|\log\alpha|(t-t')} O_i(t')\,dt'$$

Context Units
Jordan Nets



- Usable for:
  - Generating a set sequences with different fixed inputs
  - Recognizing different input sequences

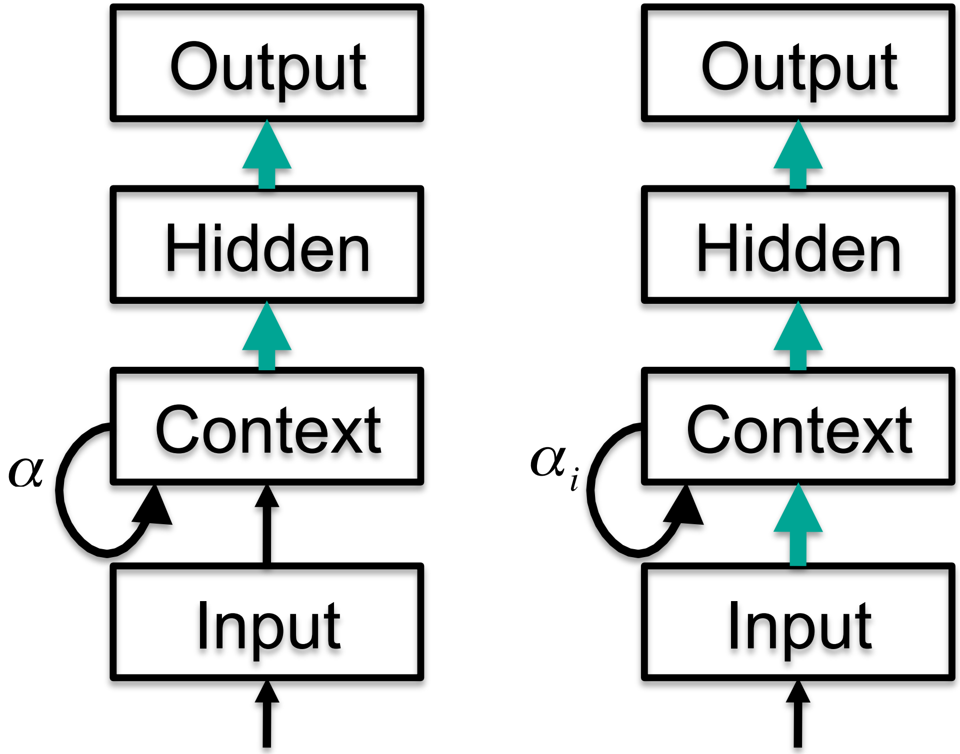
Context Units
More Architectures



- Input gets to the net only via context units
- Acts as an IIR Filter with transfer function

$$\frac{1}{1-\alpha z^{-1}}$$

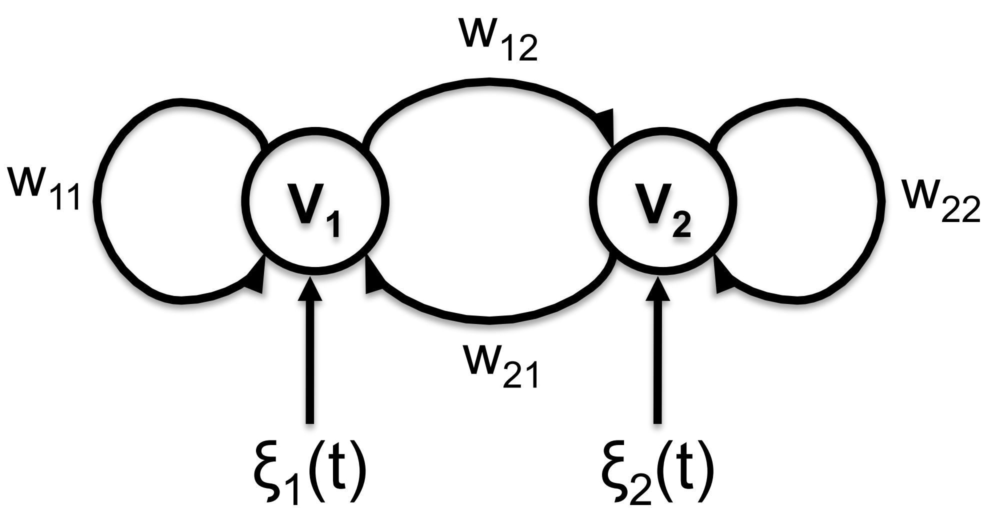
Context Units
More Architectures



- Right: modifiable feedback weights
- Comparable to "real-time recurrent networks"
- Both work better on recognizing than on generating or reconstruction

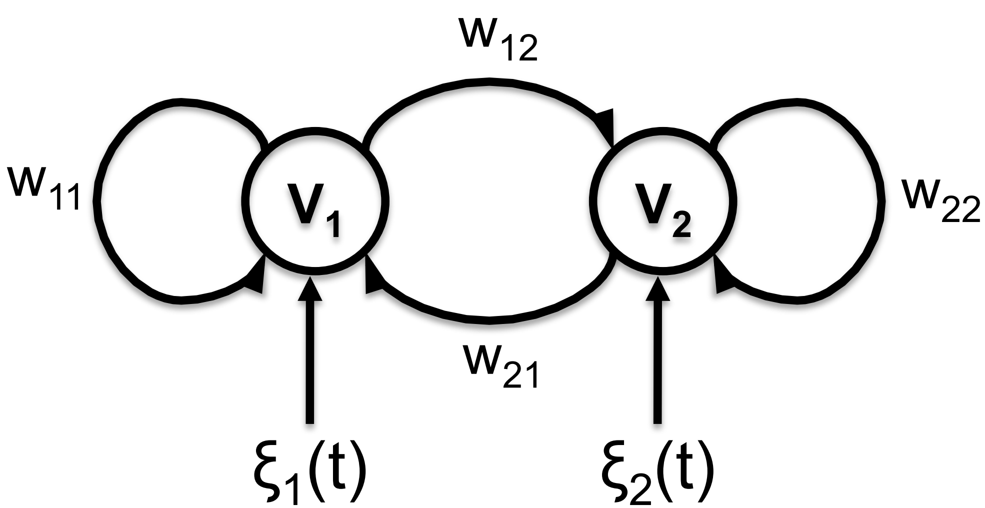
Back-Propagation Through Time

- Use fully connected units (also each to itself)
- Units are updated synchronous
- Every unit may be input, output, both or neither



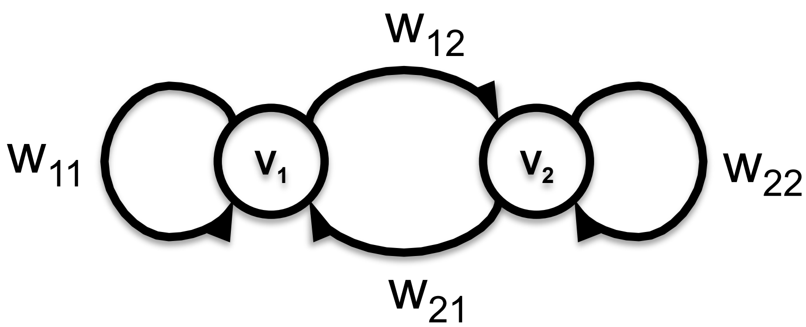
Back-Propagation Through Time

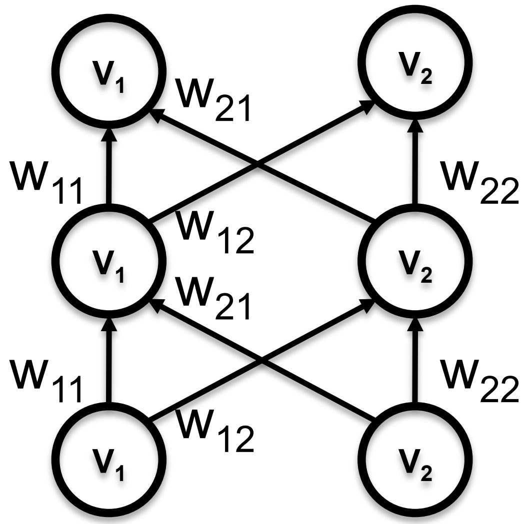
- For each time step update all units synchronous depending on the current input and state of the net
- Final output will be the classification result
- How to train the weights?



Back-Propagation Through Time

- For a time sequence of length T copy all units T times





- Net will behave identically for T time steps
- Weights need to the same (so time independent)

Back-Propagation Through Time

- Train weights on the equivalent feed-forward net and use them in the recurrent network
- Problem: Backpropagation would not get to equal weights for each time step
  - Add up all Δw_{ij}'s and change all copies of the same weight by the same amount

Back-Propagation Through Time

- Needs very much resources in training and also much training data
- Completely impractical for large or even unknown length of sequences
- Largely superseded by the other approaches

Real-Time Recurrent Learning

- Learning rule for pattern sequences in recurrent networks (Recurrent Back-Propagation)
- One version can be run online (while sequences are presented, not after they are finished)
    - Can deal with arbitrary length

Real-Time Recurrent Learning

- Assume same dynamics as in Back-Propagation through time

$$V_i(t) = g(h_i(t-1)) = g\left(\sum_j w_{ij} V_j(t-1) + \xi_i(t-1)\right)$$

- With target outputs $\zeta_k(t)$ for some units at some time steps, we get the following error measure

$$E_k(t) = \begin{cases} \varsigma_k(t) - V_k(t) & \text{if } k \text{ is an output at } t \\ 0 & \text{otherwise} \end{cases}$$

Real-Time Recurrent Learning

- The cost function is then the sum of the cost function per time step over all time steps

$$E = \sum_{t=0}^{T} E(t) = \frac{1}{2} \sum_{t=0}^{T} \sum_{k} E_k(t)^2$$

- Due to the time dependency we get a time dependent Δw_{ij}

$$\Delta w_{ij}(t) = \eta \sum_{k} E_k(t) \frac{\partial V_k(t)}{\partial w_{ij}}$$

$$\frac{\partial V_k(t)}{\partial w_{ij}} = g'(h_k(t-1)) \left[\delta_{ki} V_j(t-1) + \sum_{p} w_{kp} \frac{\partial V_p(t-1)}{\partial w_{ij}} \right]$$

Real-Time Recurrent Learning

- No stable points in general so derivatives depend on the derivatives of the preceding time step

$$\frac{\partial V_k(t)}{\partial w_{ij}} = g'(h_k(t-1))\left[\delta_{ki} V_j(t-1) + \sum_p w_{kp} \frac{\partial V_p(t-1)}{\partial w_{ij}}\right]$$

- But net is time discrete so we can calculate the derivatives iteratively
- Just need to the initial condition

$$\frac{\partial V_k(0)}{\partial w_{ij}} = 0$$

Real-Time Recurrent Learning

- Since all derivatives can be computed iteratively the time dependent $\Delta w_{ij}(t)$ can be found
- Just iterate through all time steps
- Sum up all partial weight changes to get the total changes
- Repeat until net remembers the correct sequence

Real-Time Recurrent Learning

- Algorithm needs very much computation time and memory
  - For N fully recurrent units there are N^3 derivatives to be maintained
  - Updating is proportional to N
  - So algorithm's complexity is N^4
- But updating weights can be done after each time step if η is small
  - =Real-Time Recurrent Learning

Real-Time Recurrent Learning

- Works well for sequence recognition but simpler nets can do that as well
- Can learn a flip-flop net
  - Output a signal only after a symbol A has occurred until another symbol B has occurred
- Can learn Finite State Machine
- With some modifications (teacher forcing) algorithm can be used to train a square wave or sine wave oscillator

Time-Dependent Recurrent Back-Propagation

- Related algorithm for time-continuous recurrent nets

$$\tau_i \frac{dV_i}{dt} = -V_i + g\left(\sum_j w_{ij} V_j\right) + \xi_i(t)$$

- Sum over time steps in error function becomes an integral

$$E = \frac{1}{2}\int_0^T \sum_{k \in O} \left[V_k(t) - \xi_k(t)\right]^2 dt$$

- Again a second DGL

$$\frac{dY_i}{dt} = -\frac{1}{\tau_i} Y_i + \sum_j \frac{1}{\tau_j} w_{ji} g'(h_j) Y_j + E_i(t)$$

Time-Dependent Recurrent Back-Propagation

Karlsruher Institut für Technologie

- Integrate DGL of the original net from t=0 to T to get the V_i's
- Integrate the second DGL from t=T to 0 to get the Y_i's
- Get weight changes with

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

since

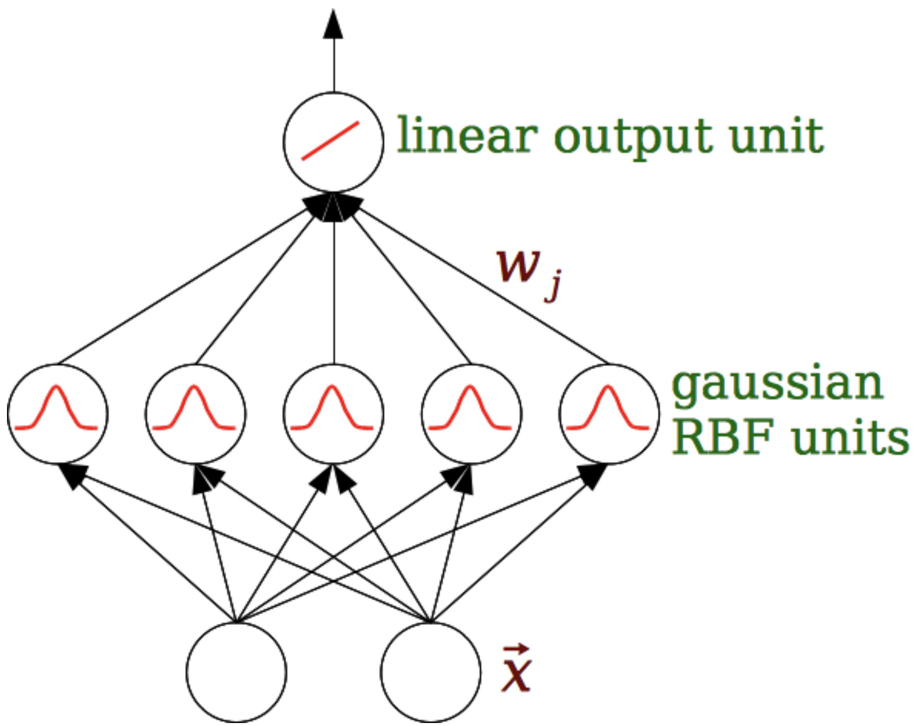
$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{\tau_i} \int_0^T Y_i g'(h_i) V_j \, dt$$

Time-Dependent Recurrent Back-Propagation

- Was used to train a net with 2 outputs to follow a 2 dimensional trajectory, including circles and figure eights
- Net was capable of returning to the trajectory even after disturbance
- Best approach unless online learning is needed

Radial Basis Function Networks

- Networks that use Radial Basis Functions as activation functions
- Used for function approximation, time series prediction, and control
- Typically have a hidden layer with Radial Basis Functions as activation functions and a linear output layer



Models for Function Approximation

- Train a model to approximate a function f(x) by a linear combination of a set of fixed functions (basis functions)

$$f(x) = \sum_{j=1}^{m} w_j h_j(x)$$

- Model is linear if parameters of basis functions are fixed and only linear parameter w (weights) are trained

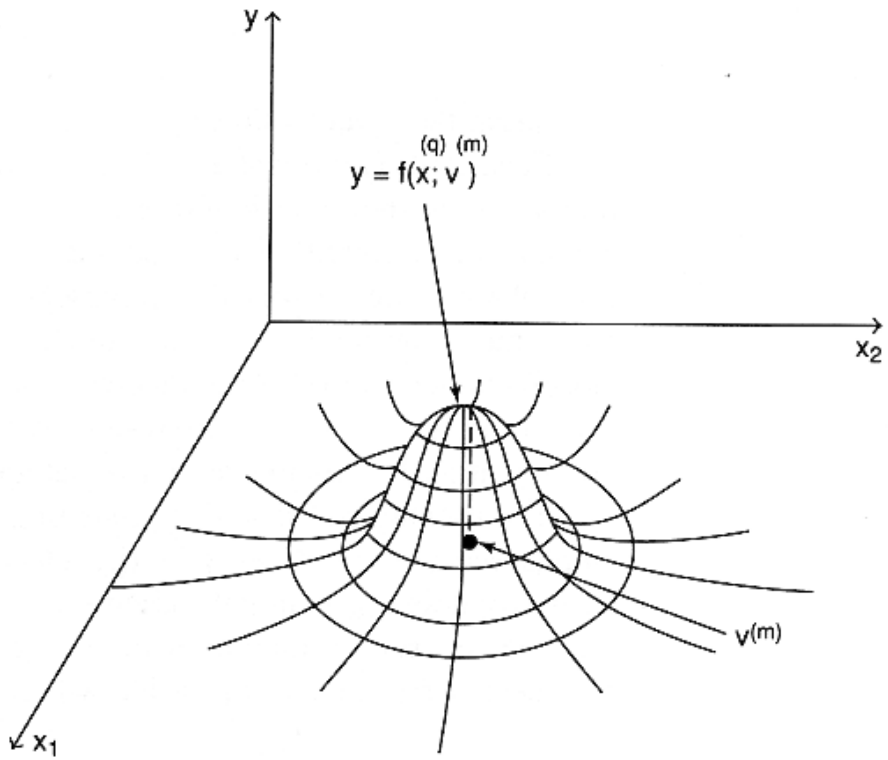
Radial Basis Functions

- Functions which monotonic decrease in response with the distance to a central point
- Most common: Gaussian

$$h(x) = e^{-\frac{\|\vec{x}-\vec{\mu}\|^2}{\sigma^2}}$$

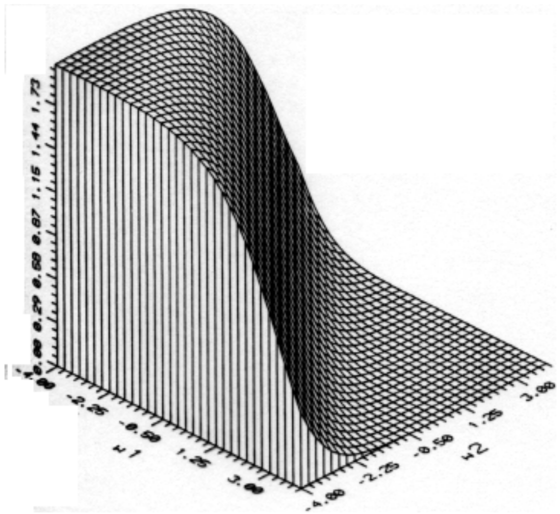
$\vec{\mu}$: Mean

σ: Variance

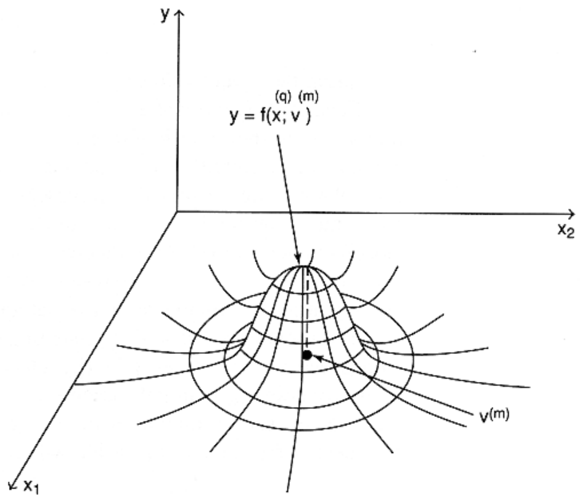


Radial Basis Functions

- Lead to a hyperelliptic decision surface instead of hyperplane
- So we get **local** units



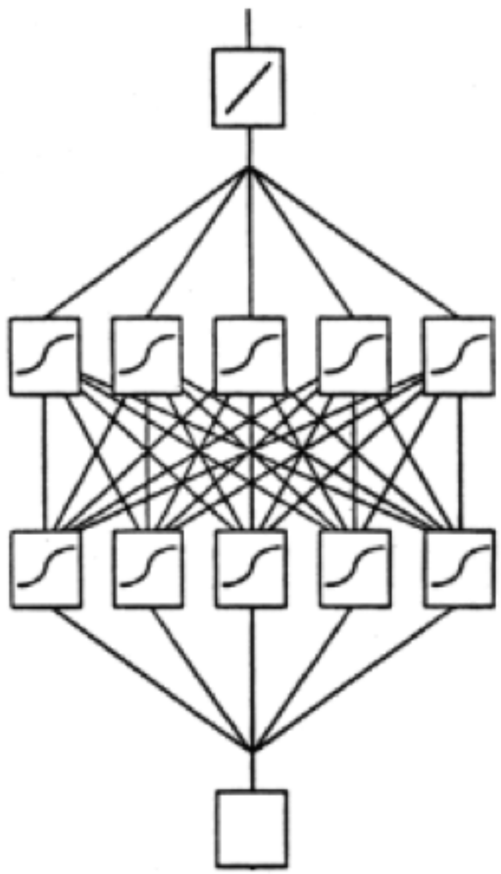
$$y_j = \tanh\left(\sum_i w_{ij} x_i\right)$$

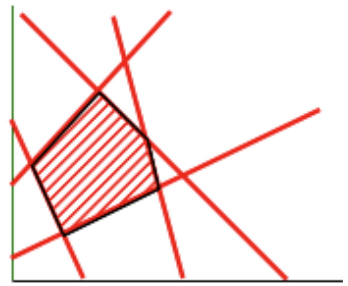


$$y_j = e^{-\frac{\left\|\vec{x} - \vec{\mu}_j\right\|^2}{\sigma_j^2}}$$

Local Activation with linear units

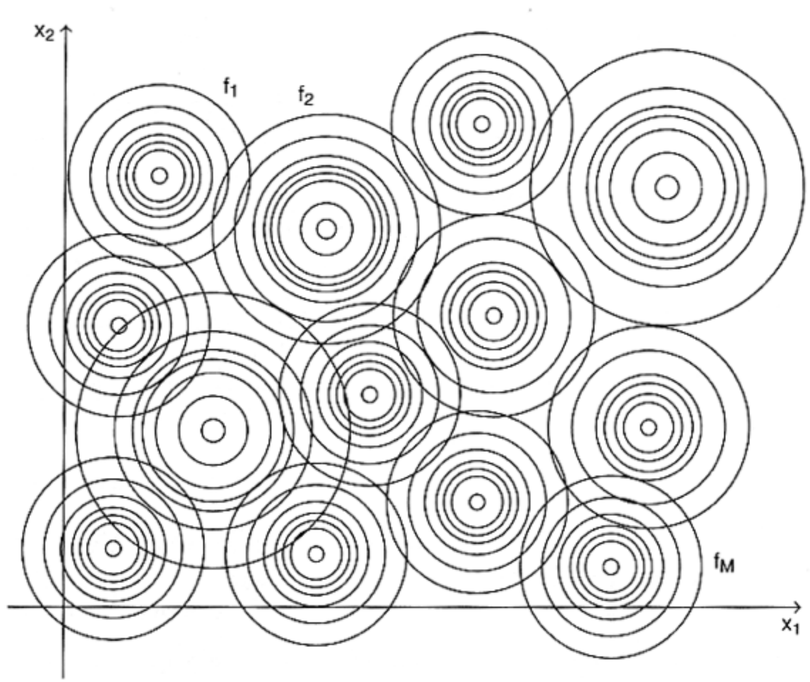
- Possible but needs multi layer perceptron





Tile the Input Space

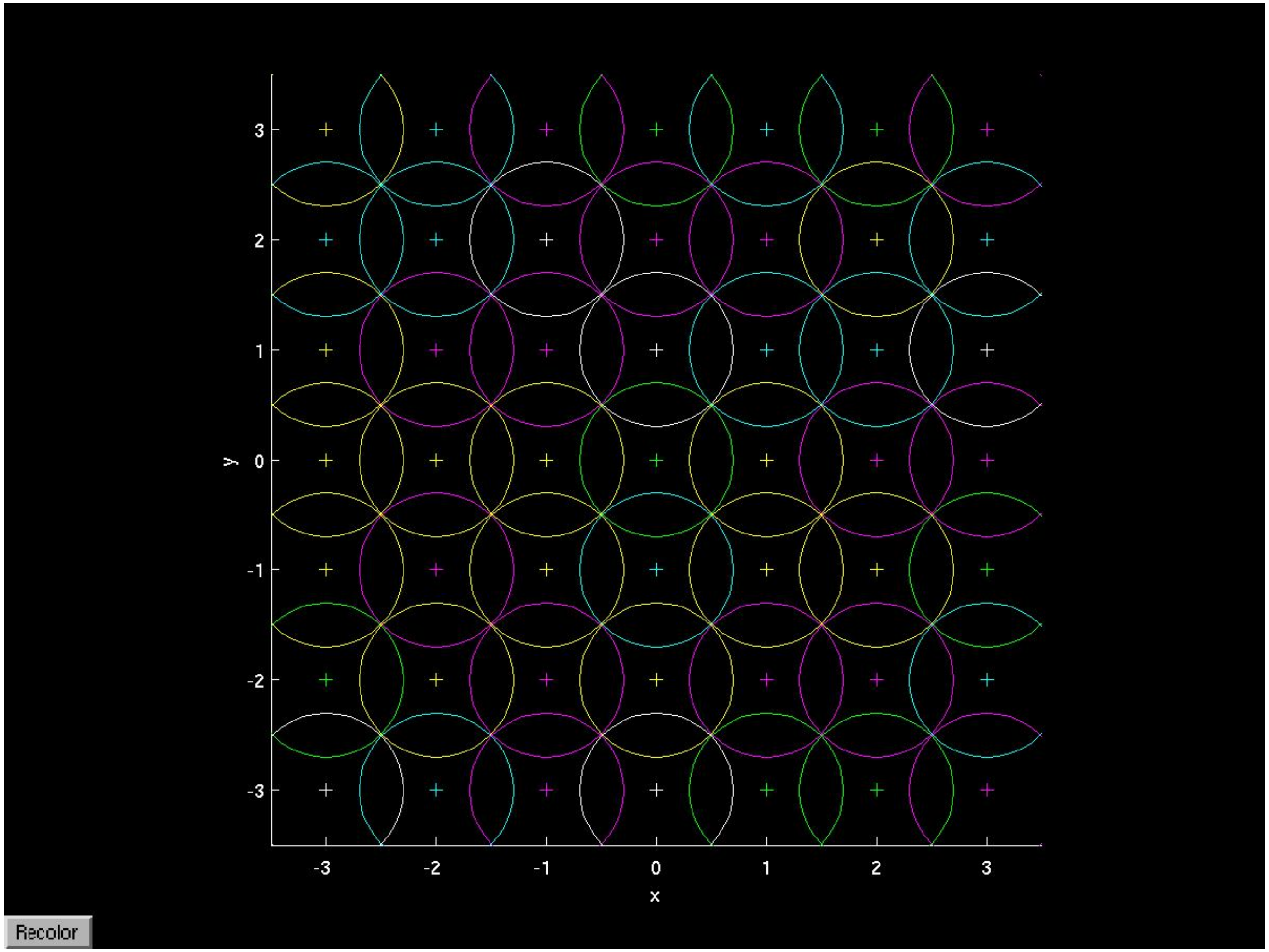
- Receptive fields overlap a bit, so there is usually more than one unit active.
- But for a given input, the total number of active units will be small.
- The locality property of RBFs makes them similar to Parzen windows.



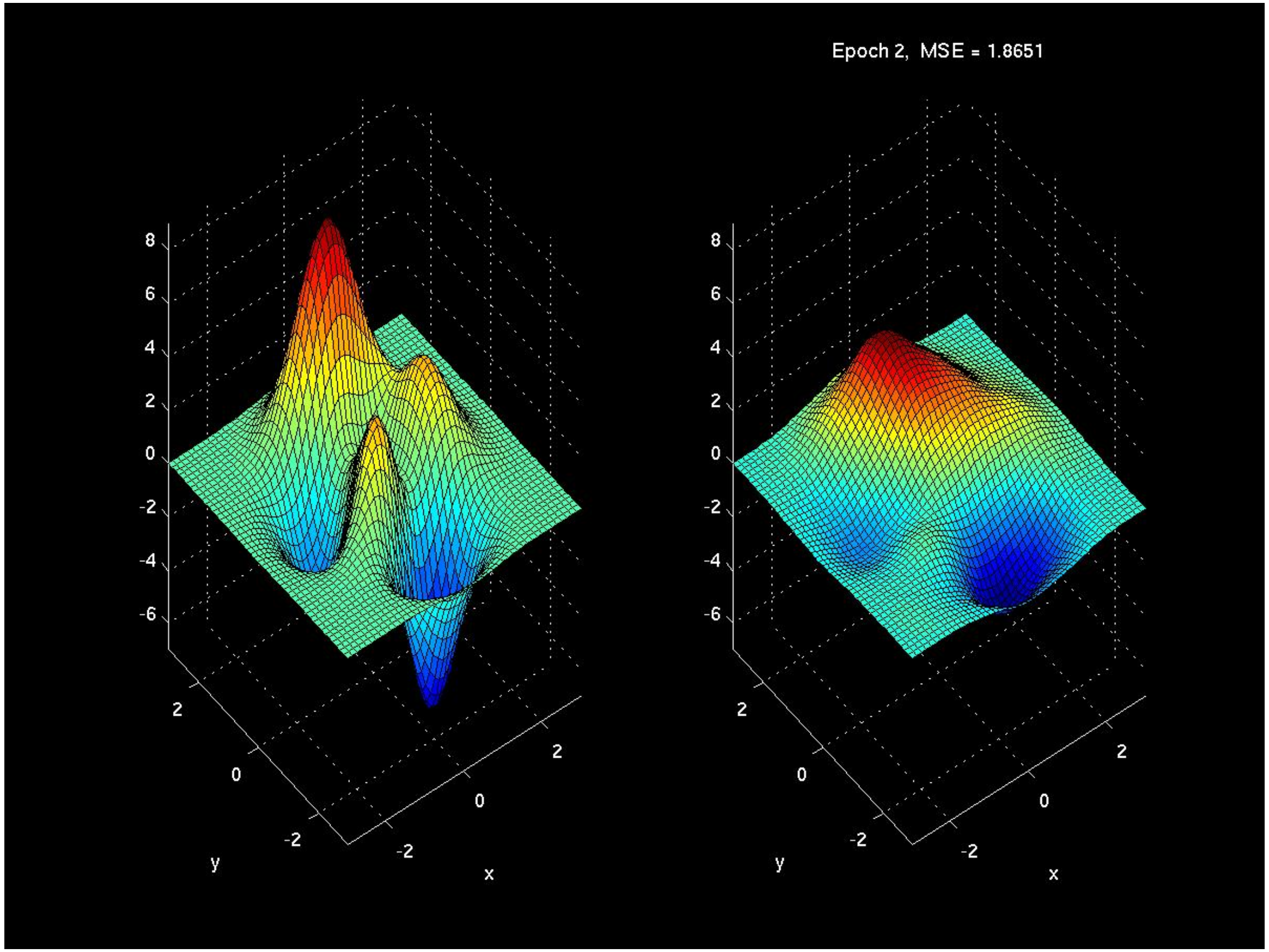
Training RBF Nets

- Training Scheme for RBF Nets is hybrid
  - Use unsupervised learning for the center points and perhaps also the variances
    - Use k-means algorithm, intialized from randomly chosen points from the training set.
    - Use a Kohonen SOFM (Self-Organizing Feature Map) to map the space. Then take selected units' weight vectors as our RBF centers
  - Least Mean Square algorithm to train the output weights
- First step can be skipped if input space is split equidistant and variances are fixed
  - Maybe the number of units is unnecessarily high

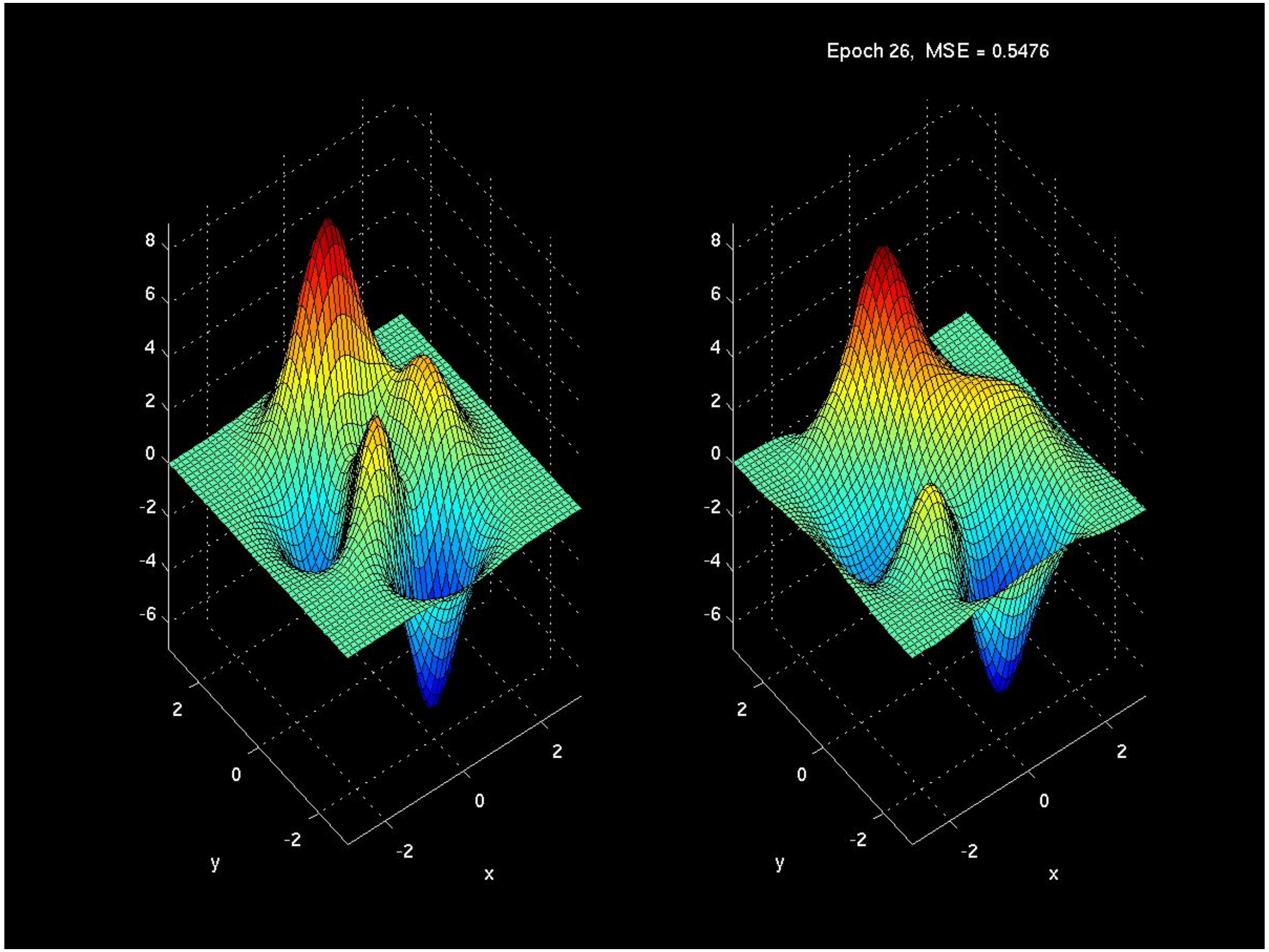
Example I: Input Space



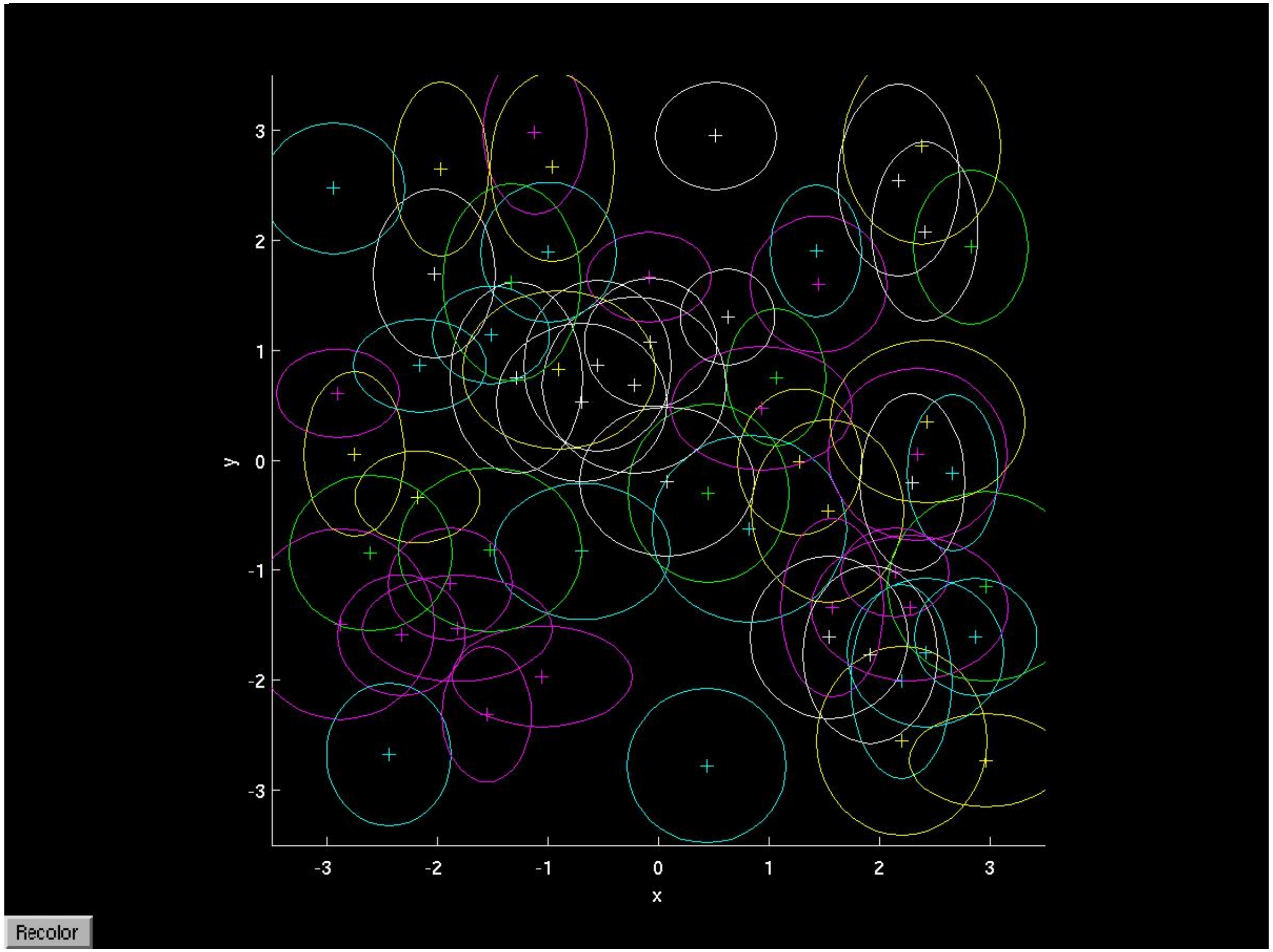
Example I: While Training



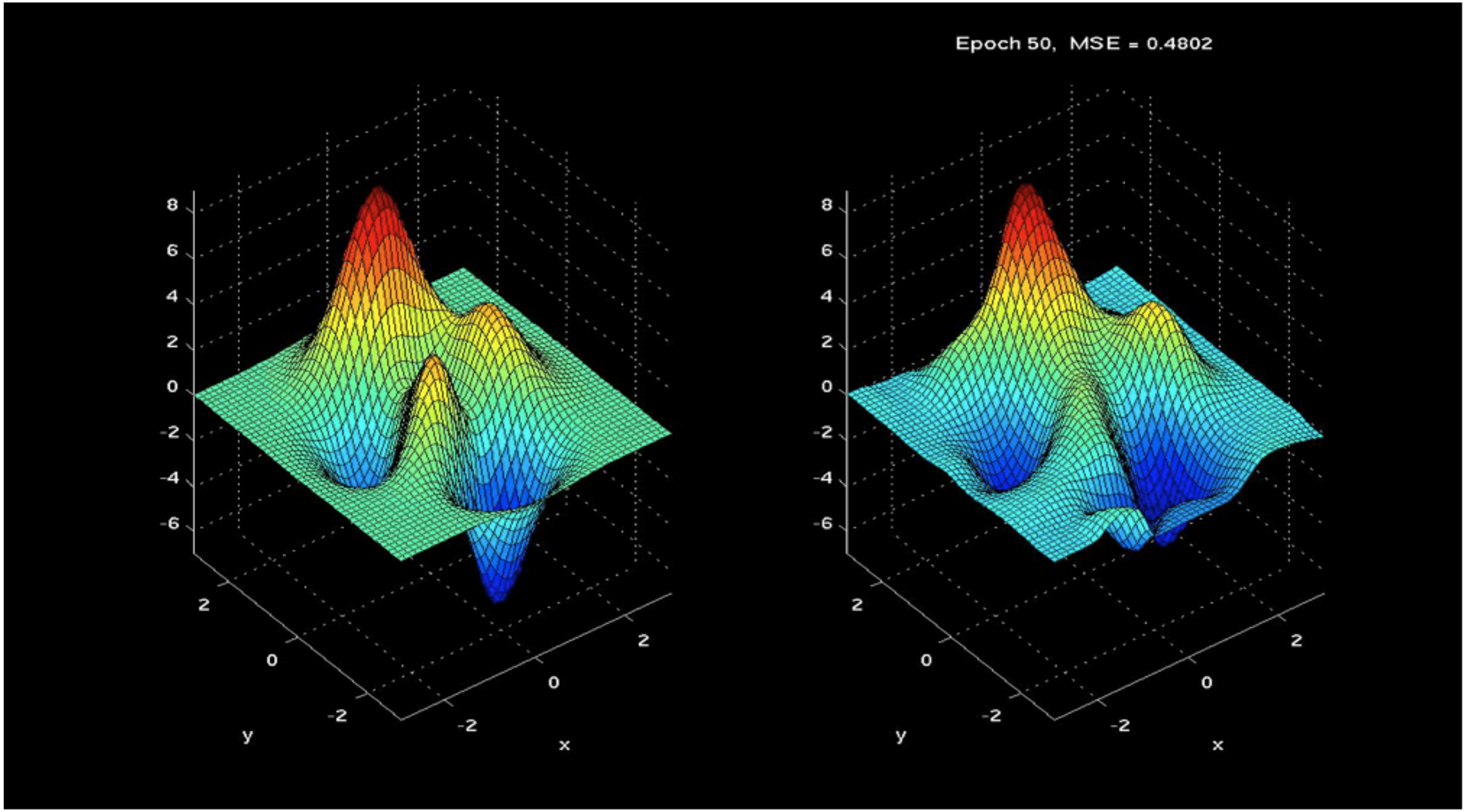
Example I: After Training



Example II: Input Space



Example II: After Training



RBF Nets

- In low dimensions we can also use a Parzen window (for classification) or a table-lookup interpolation scheme
- But in higher dimensions RBF Nets are much better since units can be placed only where they are needed