Neuronale Netze (SS 2002), 24.6.

Partially recurrent neural networks

- **Definition:**

  $(N, \rightarrow, \vec{w}, \vec{\theta}, \vec{f}, I, O)$ where $(N, \rightarrow)$ might be a cyclic graph.

  inputs at time step t: $x_i(t)$

  activations and outputs at time t:
  $net_i(t) = o_i(t) = x_i(t)$ for input neurons i
  $net_i(0) = o_i(0) =$ some fixed value for other neurons at time step o
  $net_i(t+1) = \sum_{j \rightarrow i} w_{ji} o_j(t) - \theta_i$ for other neurons and later time steps
  $o_i(t+1) = f_i(net_i(t+1))$

- **Applications** for time series/sequences:
  - time series prediction for weather, dynamic systems, financial data
  - filtering, noise reduction, tracking, optimal control
  - robotics, gait generation, planing
  - vision, grouping and binding, tracking
  - speech recognition/processing/production, simulation of automata
  - possbile interface to symbolic mechanisms via recursive networks

- **Training:**
  - given input sequenz $x_i(t)$ and output sequence $y_i(t)$, minimize the quadratic error

    $$E = \sum_t \sum_i \frac{1}{2} \left(o_i(t) - y_i(t)\right)^2$$

    e.g. with a gradient descent.

    Unfolding in time transfers a recurrent network to a feedforward network with shared weights. Gradient computations can be done similar to backpropagation.

    Only difference: shared weights, i.e. the derivative of E with respect to a weight w_{ij} decomposes into the sum of the derivatives with respect

to the weight at all time steps $w_{ij}(t) :=$ copy of w_{ij} in the t th recursive step.

  - backpropagation $\rightarrow$ **backpropagation through time**
    time $\sim WT$, space $\sim NT$
    The entire sequence has to be available for training!
  - Direct (only feedforward) gradient computation $\rightarrow$ **real time recurrent learning**
    time $\sim W^2T$, space $\sim W$
    Can be done in an online fashion!
  - General problem: weight sharing prohibits to apply the nice tricks from FNNs. The problem of long term dependencies is still unsolved!

- **Training pipeline:** just as for FNNs ...
  - preprocessing
  - architecture selection
  - minimization of training error, possibly pruning, early stopping, weight decay, ...
  - interpretation of the result, computing the test error

- **Approximation ability**
  RNNs can approxiamte every continuous function on compact sets arbitrarily well.

- **Complexity of training**
  ... not easier than FNN training, in addition: prolem of long term dependencies

- **Generalization ability**
  The VC dimension depends on the number of weights and the maximum input length.
  - RNNs are not distribution independent PAC/UCED.
  - RNNs are distribution dependent PAC/UCED. Bounds can be rather bad ...

  - Posterior bounds depending on the concrete trainingset are possible as an alternative for unknown input distribution.

- **RNNs and automata**
  - RNNS can simulate automata. The proof is constructive, i.e. prior knowledge in the fomr of automata rules can be integrated.
  - There exist mechanisms which approximately extract automata rules from RNNs. These are heuristics.

- **RNNs and computability**
  - RNNs can count, this can be trained!
  - RNNs can simulate Turing machines in principle. Necessary: memory/stack, Boolean circuit, memory with the program; these parts can be simulated.
  - RNNs are equivalent to families of non-uniform Boolean circuits, thy are super-Turing universal.