Long Short-Term Memory:

2003 Tutorial on LSTM Recurrent Nets

(there is a recent, much nicer one, with many new results!)

Jürgen Schmidhuber

Pronounce:

You_again Shmidhoobuh

IDSIA, Manno-Lugano, Switzerland

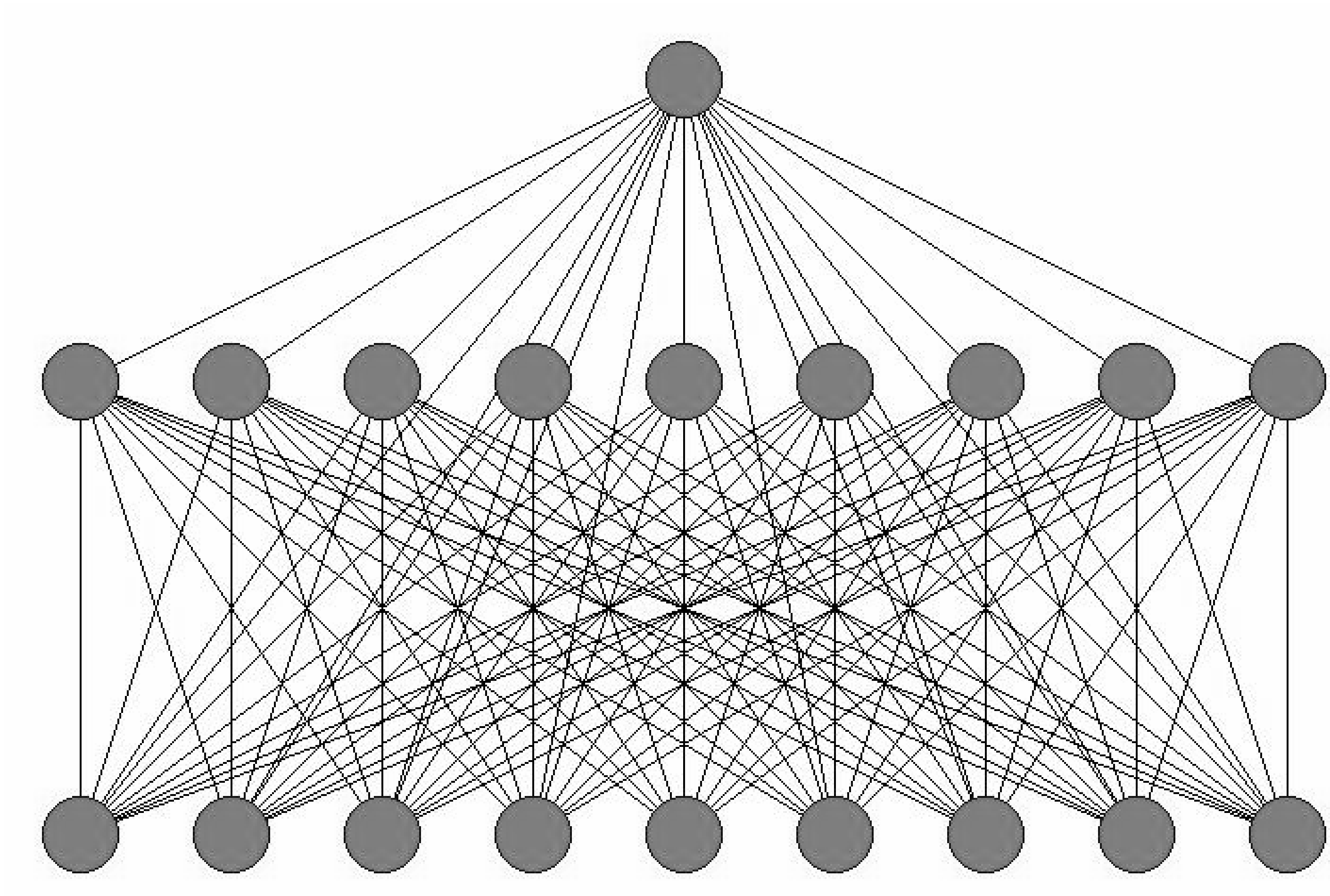
www.idsia.ch

Tutorial covers the following LSTM journal publications:

- *Neural Computation,* 9(8):1735-1780, 1997
- *Neural Computation,* 12(10):2451--2471, 2000
- *IEEE Transactions on NNs* 12(6):1333-1340, 2001
- *Neural Computation,* 2002
- *Neural Networks,* in press, 2003
- *Journal of Machine Learning Research,* in press, 2003

- Also many conference publications: *NIPS 1997, NIPS 2001, NNSP 2002, ICANN 1999, 2001, 2002, others*

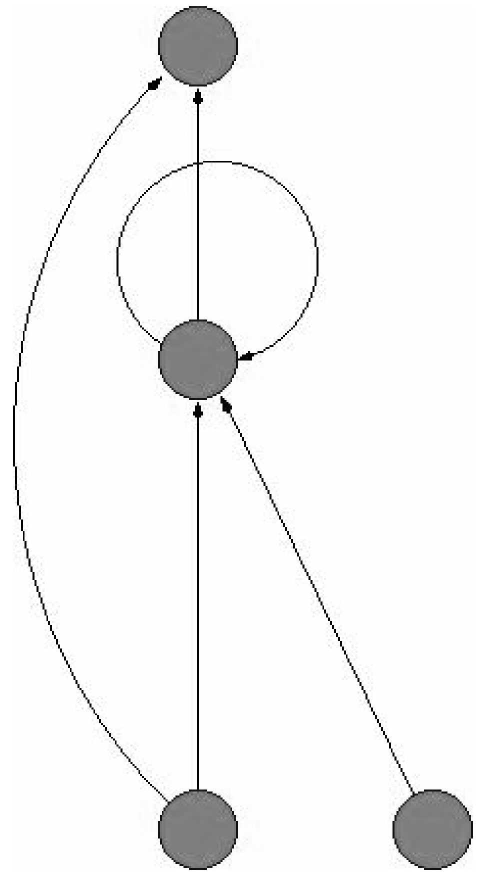
Even static problems may profit from recurrent neural networks (RNNs), e.g., parity problem: number of 1 bits odd? 9 bit feedforward NN:



Parity problem, sequential: 1 bit at a time

- **Recurrent net** learns much faster - even with random weight search: only 1000 trials!

- many fewer parameters
- much better generalization
- the natural solution



Other sequential problems

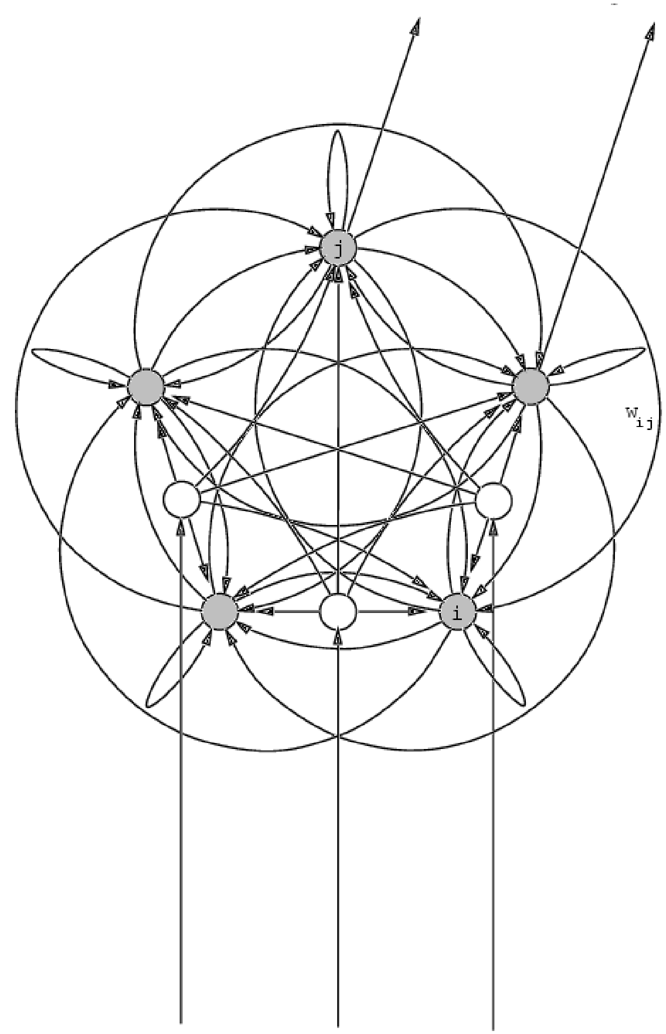
- **Control of attention:** human pattern recognition is sequential
- **Sequence recognition:** speech, time series….
- **Motor control** (memory for partially observable worlds)
- **Almost every real world task**
- Strangely, many researchers still content with reactive devices (FNNs & SVMs etc)

Other sequence learners?

- **Hidden Markov Models:** useful for speech etc. But discrete, cannot store real values, no good algorithms for learning appropriate topologies
- **Symbolic approaches:** useful for grammar learning. Not for real-valued noisy sequences.
- **Heuristic program search** (e.g., Genetic Programming, Cramer 1985): no direction for search in algorithm space.
- **Universal Search** (Levin 1973): asymptotically optimal, but huge constant slowdown factor
- **Fastest algorithm** for all well-defined problems (Hutter, 2001): asymptotically optimal, but huge *additive* constant.
- **Optimal ordered problem solver** (Schmidhuber, 2002)

Gradient-based RNNs: *∂ wish / ∂ program*

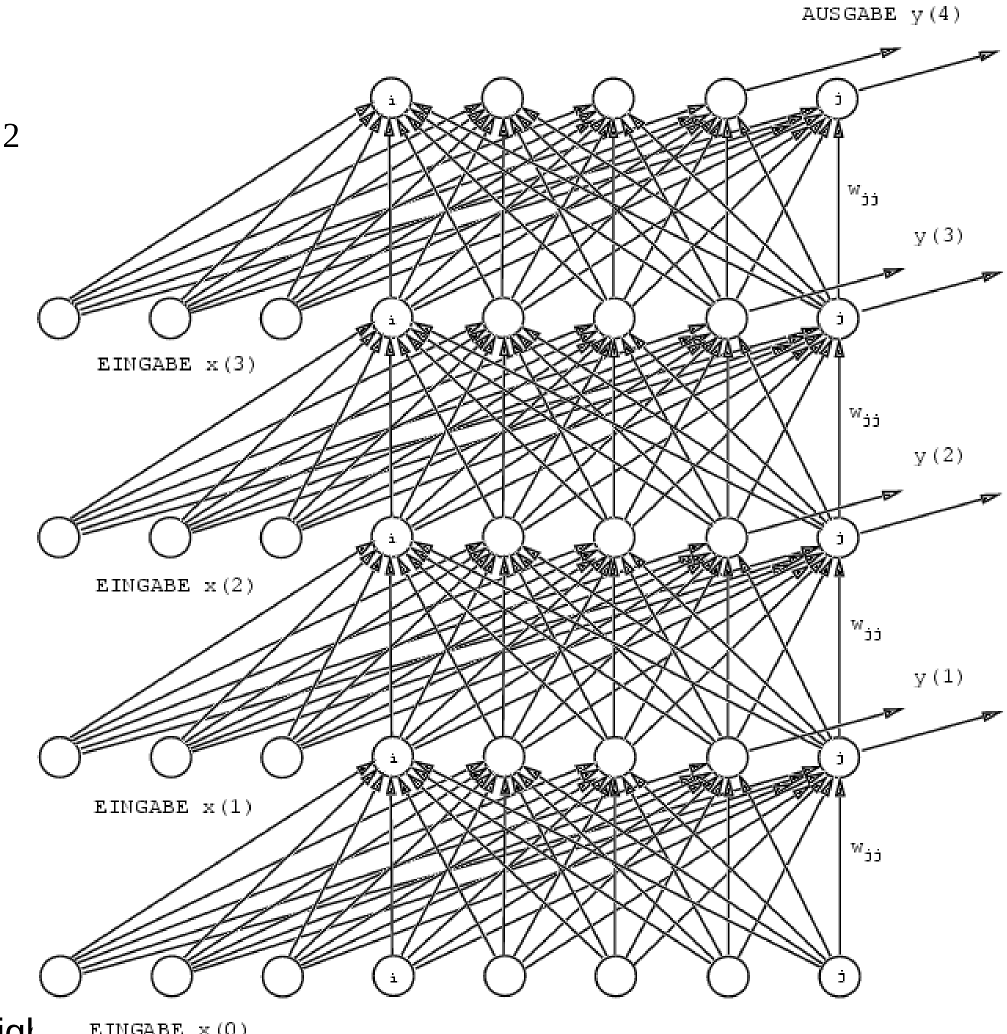
- RNN weight matrix embodies general algorithm space
- Differentiate objective with respect to program
- Obtain gradient or search direction in program space



1980s: BPTT, RTRL - gradients based on "unfolding" etc. (Williams, Werbos, Robinson)

$$E = \sum_{seq\ s}\sum_{t}\sum_{o_i}(o_i^s(t) - d_i^s(t))^2$$

$$\Delta w \propto \frac{\partial E}{\partial w}$$



1990s: Time Lags!

- 1990: RNNs great in principle but don't work?
- Standard RNNs: Error path integral decays exponentially! *(first rigorous analysis due to Schmidhuber's former PhD student Sepp Hochreiter 1991; compare Bengio et al 1994, and Hochreiter & Bengio & Frasconi & Schmidhuber, 2001)*
- $net_k(t) = S_i w_{ki}\, y_i(t-1)$
- Forward: $y_k(t)=f_k\,(net_k(t))$
- Error: $e_k(t)=f_k'(net_k(t))\; S_i\, w_{ik}\, e_i(t+1)$

Exponential Error Decay

- Lag q: $$\frac{\partial e_v(t-q)}{\partial e_u(t)} = f_v'(net_v(t-1))w_{uv} \quad if \;\; q=1$$

  $$otherwise \quad f_v'(net_v(t-q))\sum_{l=1}^{n} \frac{\partial e_l(t-q+1)}{\partial e_u(t)} w_{lv}$$

- Decay: $$\| \frac{\partial e(t-q)}{\partial e(t)} \| = \| \prod_{m=1}^{q} WF'(Net(t-m)) \| \quad \leq$$

  $$(\| W \| \max_{Net}\{\| F'(Net) \|\})^q$$

- Sigmoid: *max f'=0.25; |weights|<4.0;* vanish!
  (higher weights useless - derivatives disappear)

Training: forget minimal time lags > 10!

- **So why study RNNs at all?**
- Hope for generalizing from short exemplars? Sometimes justified, often not.

- To overcome long time lag problem: history compression in RNN hierarchy - level *n* gets unpredictable inputs from level *n-1* *(Schmidhuber, NIPS 91, Neural Computation 1992)*
- Other 1990s ideas: *Mozer, Ring, Bengio, Frasconi, Giles, Omlin, Sun, ...*

Constant Error Flow!

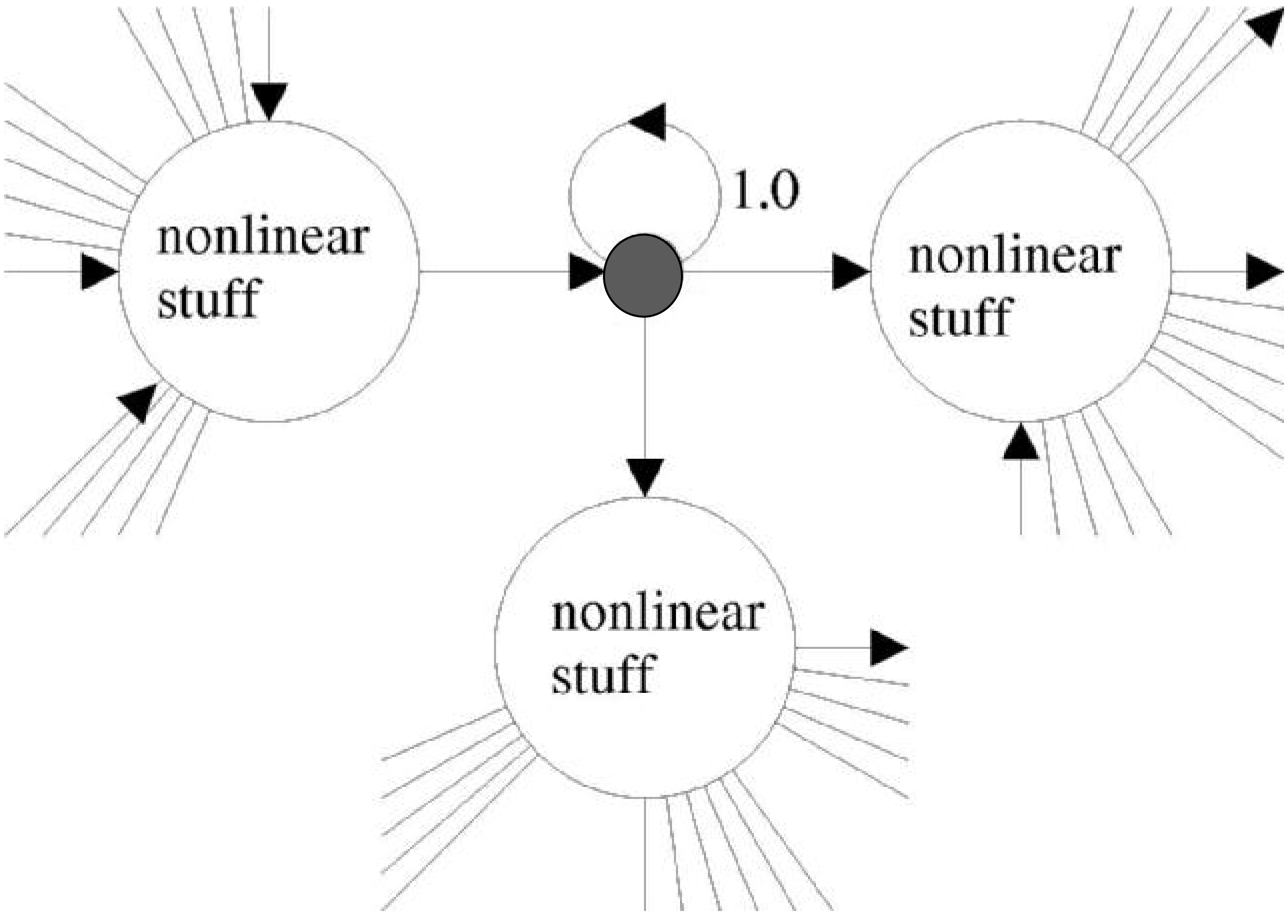
- Best 90s idea
  Hochreiter *(back then an undergrad student on Schmidhuber's long time lag recurrent net project, since 2002 assistant professor in Berlin)*

- Led to Long Short-Term Memory (LSTM):
- Time lags > 1000
- No loss of short time lag capability
- O(1) update complexity per time step and weight

Basic LSTM unit: linear integrator

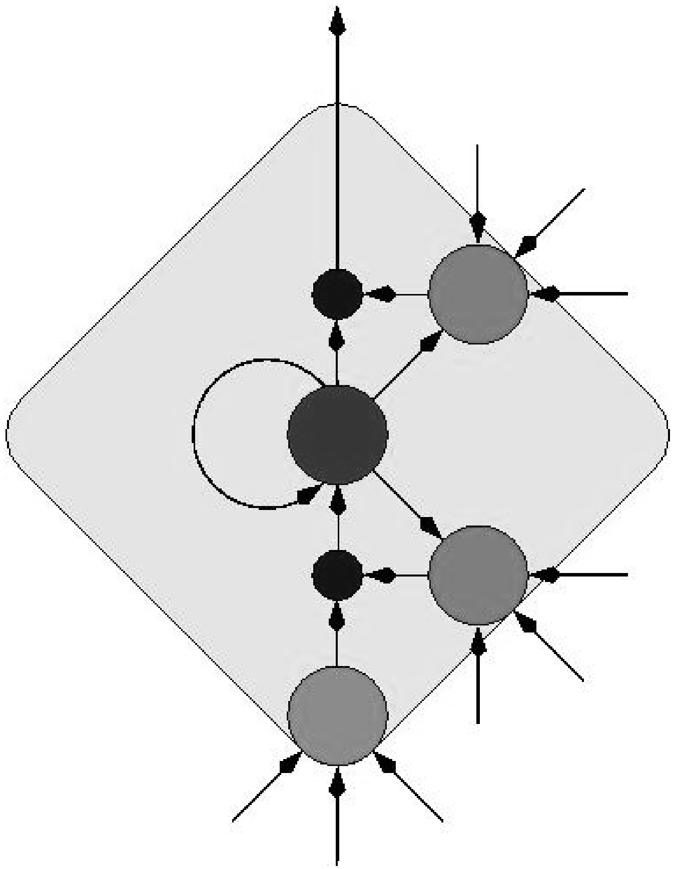
- Very simple self-connected linear unit called the error carousel.
- Constant error flow:
  $e(t) = f'(net(t))\ w\ e(t+1) = 1.0$
- Most natural: f linear, $w = 1.0$ fixed.
- Purpose: Just deliver errors, leave learning to other weights.

Long Short-Term Memory (LSTM)



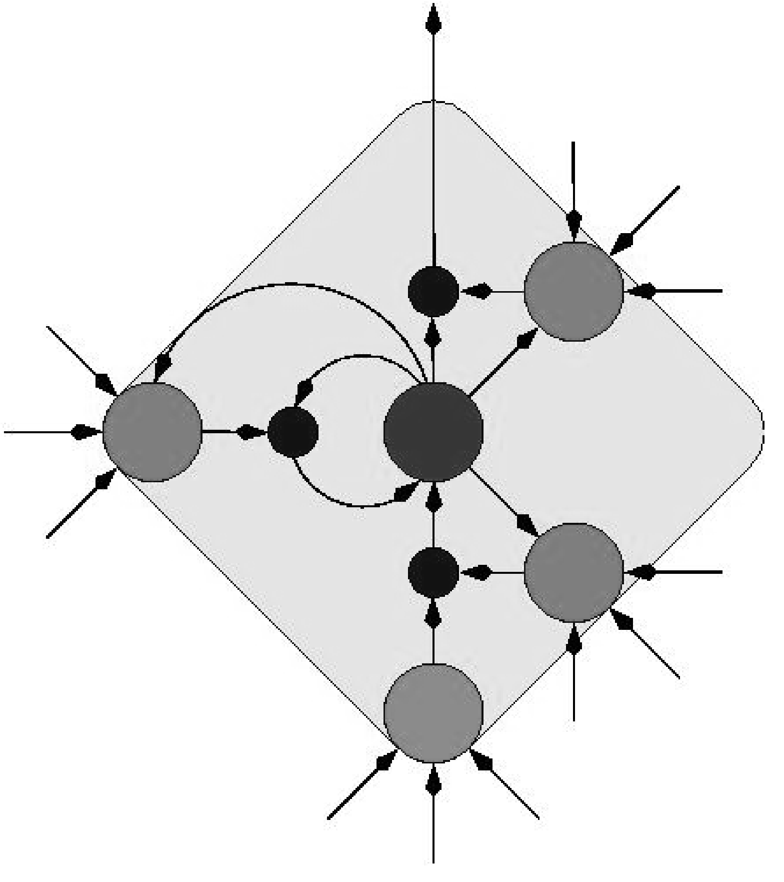
Possible LSTM cell (original)

- Red: linear unit, self-weight 1.0 - the error carousel
- Green: sigmoid gates open / protect access to error flow
- Blue: multiplicative openings or shut-downs

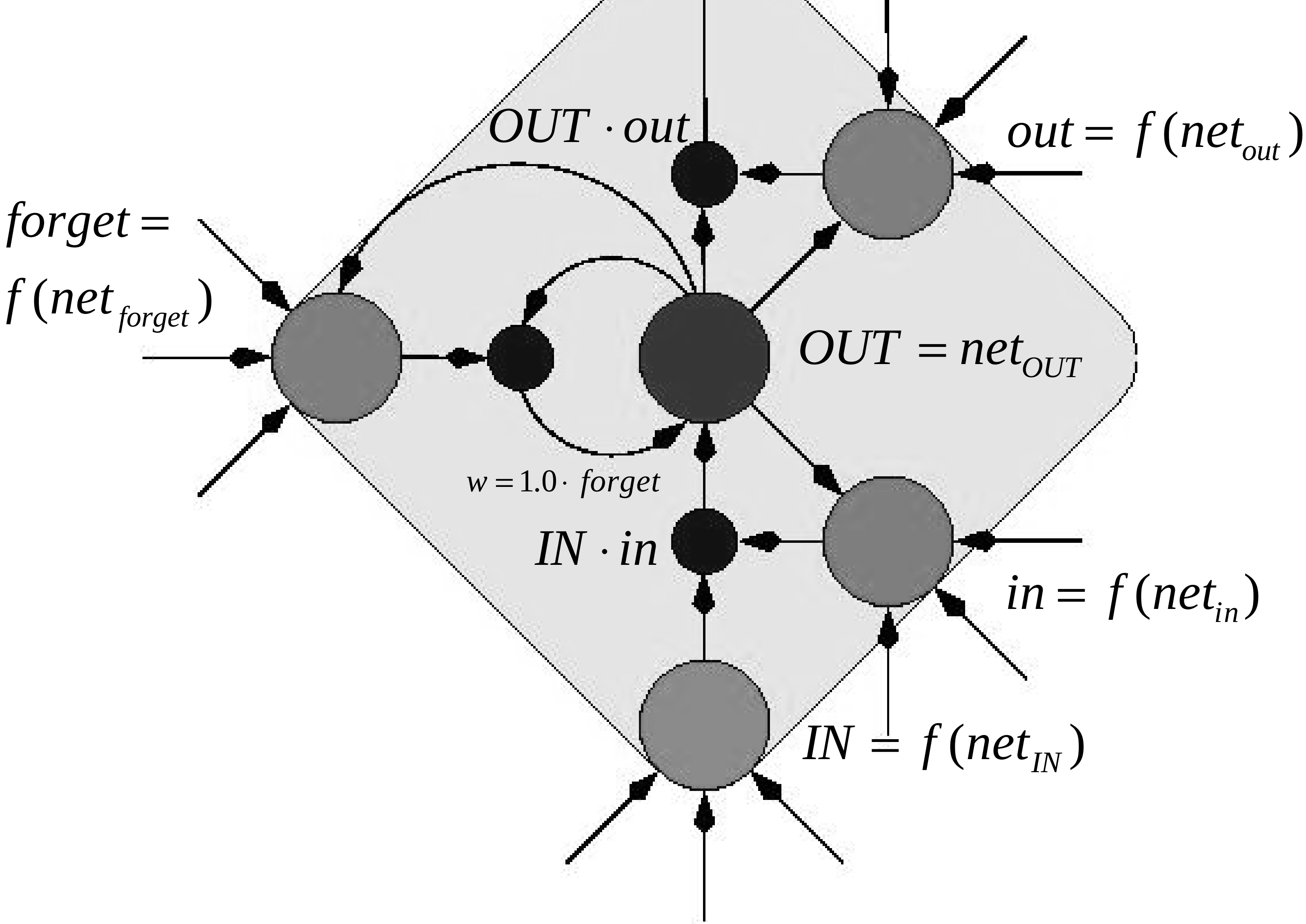


LSTM cell (current standard)

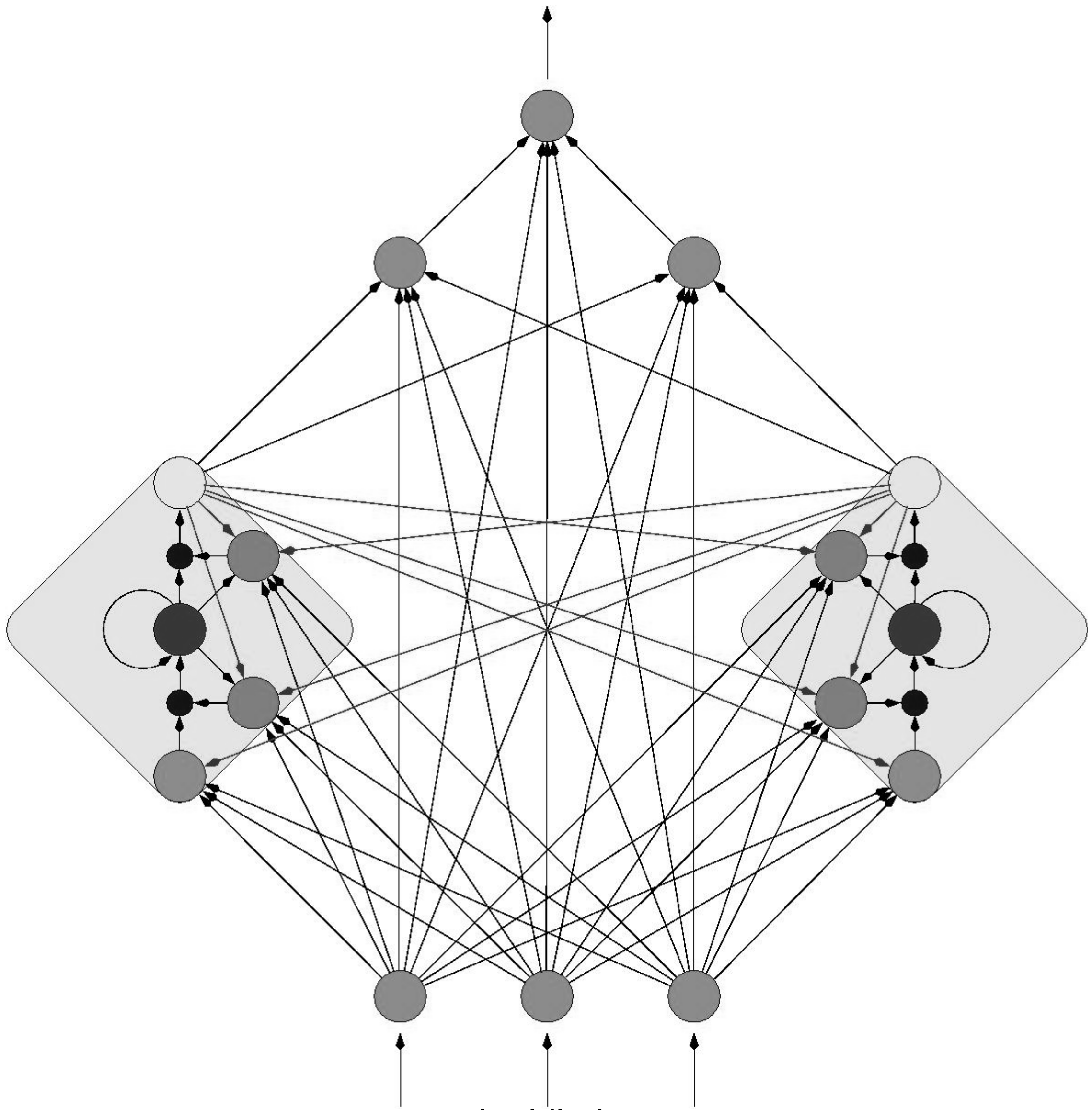
- Red: linear unit, self-weight 1.0 - the error carousel
- Green: sigmoid gates open / protect access to error flow;
  forget gate (left) resets
- Blue: multiplications



$$net_k = \sum_i w_{ki} y_i$$

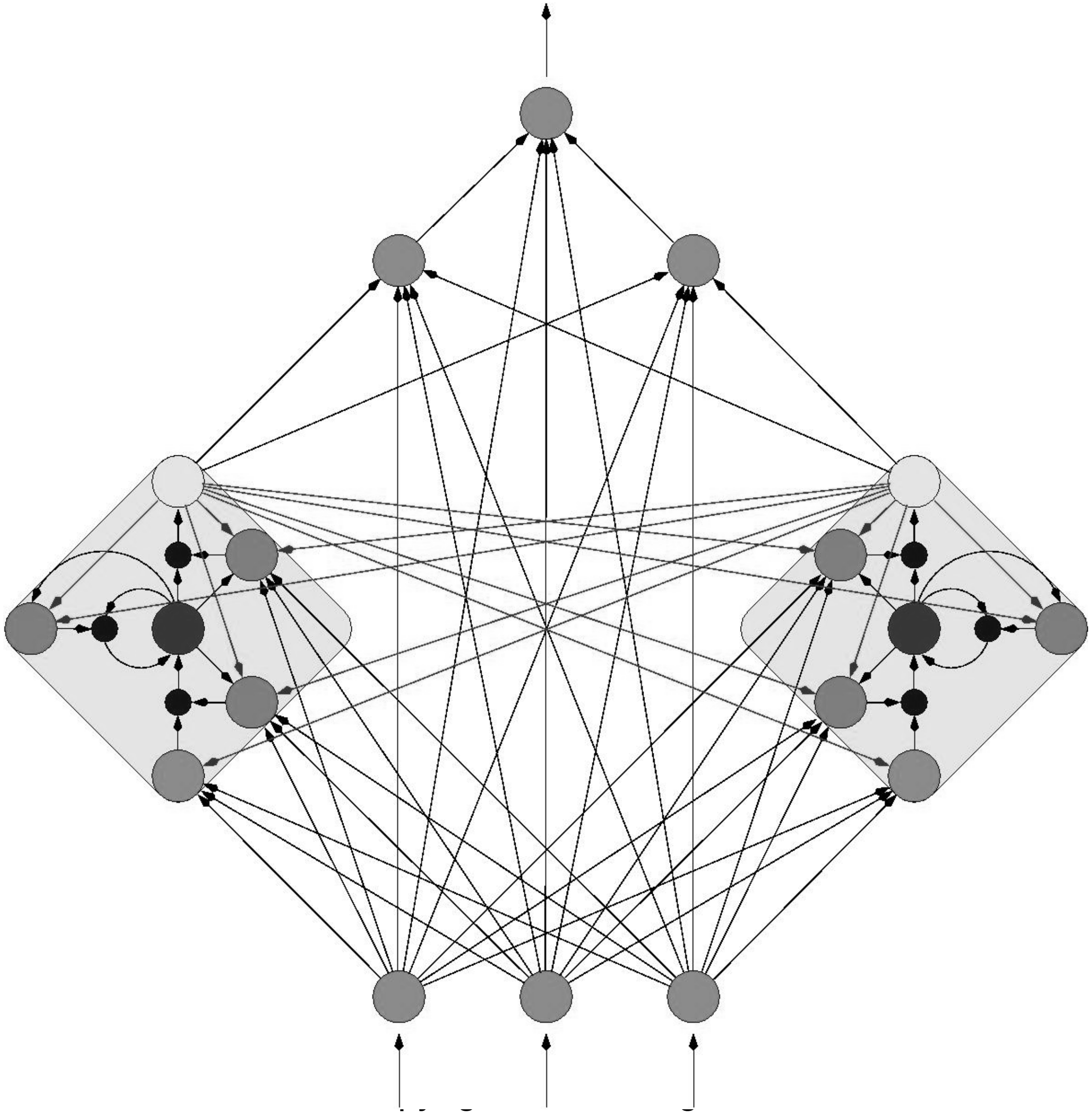


Mix LSTM cells and others



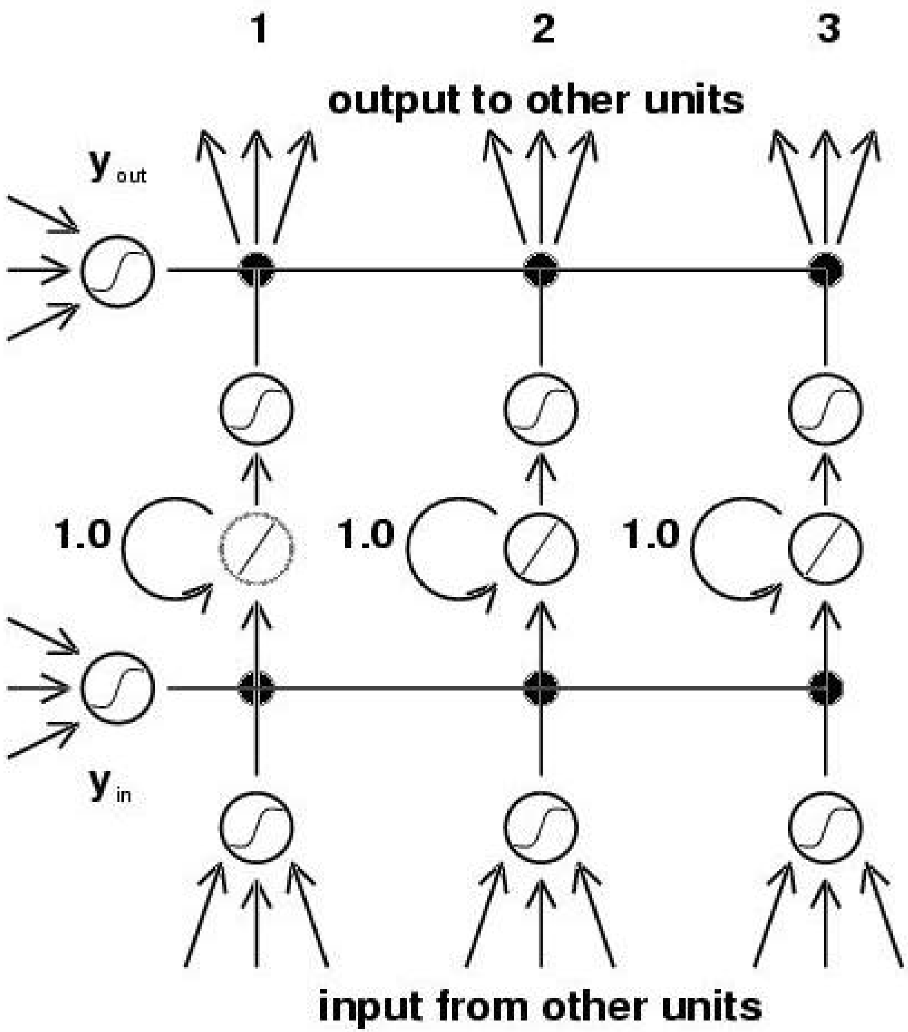
Schmidhuber

Mix LSTM cells and others

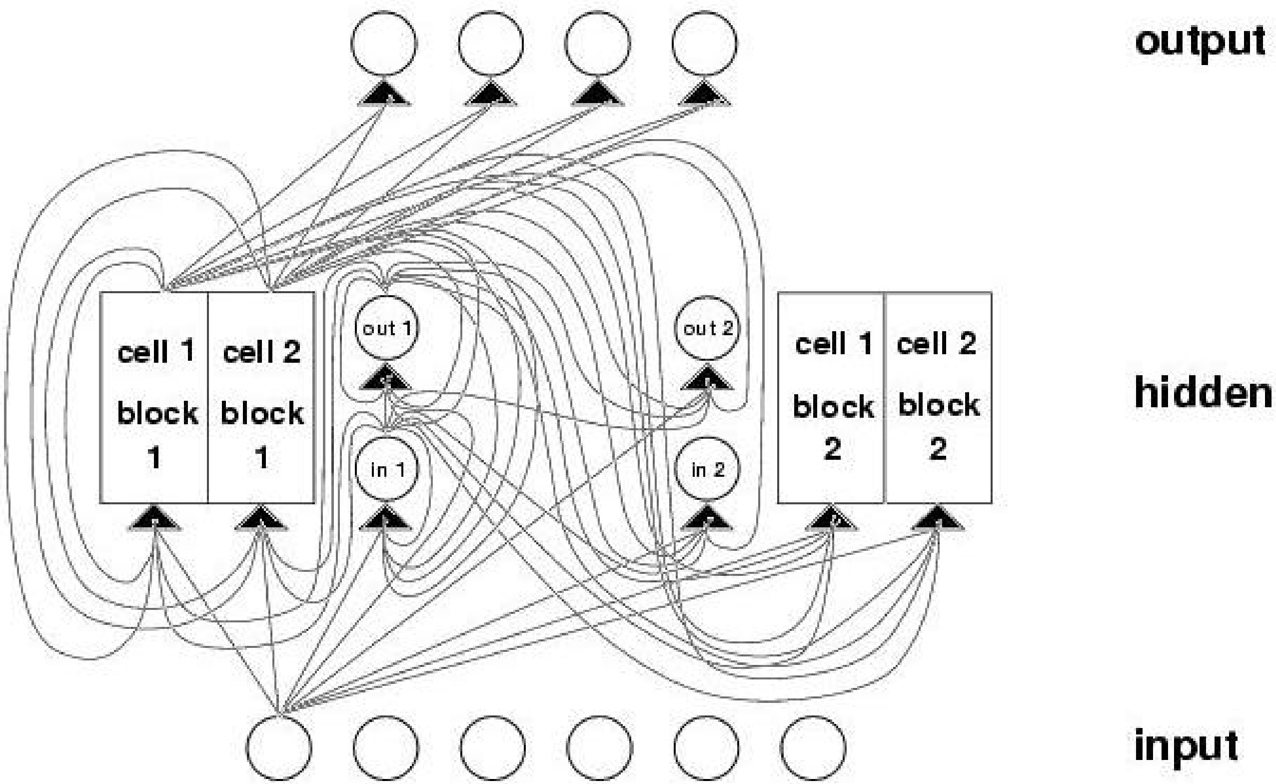


Schmidhuber

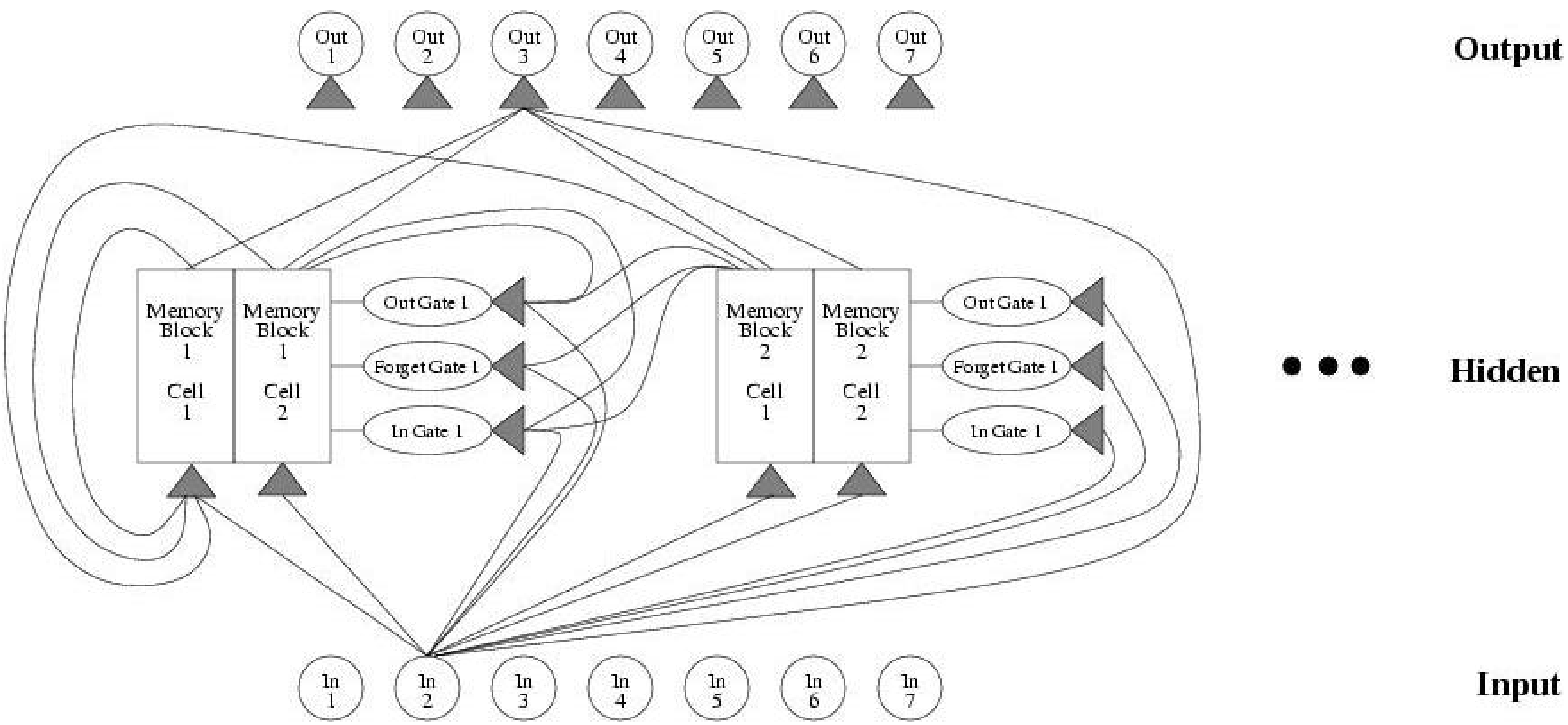
Also possible: LSTM memory blocks: error carousels may share gates



Example: no forget gates; 2 connected blocks, 2 cells each



Example with forget gates



Next: LSTM Pseudocode

- Typically: truncate errors once they have changed incoming weights

- Local in space and time: O(1) updates per weight and time step

- Download: *www.idsia.ch*

Download LSTM code: www.idsia.ch/~juergen/rnn.html

init network:

reset: CECs: $s_{c_j^v} = \hat{s}_{c_j^v} = 0$; **partials:** $dS = 0$; **activations:** $y = \hat{y} = 0$;

forward pass:

input units: $y =$ `current external input;`

roll over: activations: $\hat{y} = y$; **cell states:** $\hat{s}_{c_j^v} = s_{c_j^v}$;

`loop over memory blocks, indexed` j `{`

Step 1a: input gates (5.1):

$net_{in_j} = \sum_m w_{in_j m} \ \hat{y}^m + \sum_{v=1}^{S_j} w_{in_j c_j^v} \ \hat{s}_{c_j^v}$; $\quad y^{in_j} = f_{in_j}(net_{in_j})$;

Step 1b: forget gates (5.2):

$net_{\varphi_j} = \sum_m w_{\varphi_j m} \ \hat{y}^m + \sum_{v=1}^{S_j} w_{\varphi_j c_j^v} \ \hat{s}_{c_j^v}$; $\quad y^{\varphi_j} = f_{\varphi_j}(net_{\varphi_j})$;

Step 1c: CECs, i.e the cell states (5.3):

`loop over the` S_j `cells in block` j`, indexed` v `{`

$net_{c_j^v} = \sum_m w_{c_j^v m} \ \hat{y}^m$; $\quad s_{c_j^v} = y^{\varphi_j} \ \hat{s}_{c_j^v} + y^{in_j} \ g(net_{c_j^v})$; `}`

Step 2:

output gate activation: (5.4):

$net_{out_j} = \sum_m w_{out_j m} \ \hat{y}^m + \sum_{v=1}^{S_j} w_{out_j c_j^v} \ s_{c_j^v}$; $\quad y^{out_j} = f_{out_j}(net_{out_j})$;

cell outputs (5.5):

`loop over the` S_j `cells in block` j`, indexed` v `{` $y^{c_j^v} = y^{out_j} \ s_{c_j^v}$; `}`

`} end loop over memory blocks`

output units (2.9): $net_k = \sum_m w_{km} \ y^m$; $\quad y^k = f_k(net_k)$;

partial derivatives:

`loop over memory blocks, indexed` j `{`

`loop over the` S_j `cells in block` j`, indexed` v `{`

cells (5.6), $(dS_{cm}^{jv} := \frac{\partial s_{c_j^v}}{\partial w_{c_j^v m}})$:

$dS_{cm}^{jv} = dS_{cm}^{jv} \ y^{\varphi_j} + g'(net_{c_j^v}) \ y^{in_j} \ \hat{y}^m$;

input gates (5.7), (5.7b), $(dS_{in,m}^{jv} := \frac{\partial s_{c_j^v}}{\partial w_{in_j m}}, dS_{in,c_j^{v'}}^{jv} := \frac{\partial s_{c_j^v}}{\partial w_{in_j c_j^{v'}}})$:

$dS_{in,m}^{jv} = dS_{in,m}^{jv} \ y^{\varphi_j} + g(net_{c_j^v}) \ f'_{in_j}(net_{in_j}) \ \hat{y}^m$;

`loop over peephole connections from all cells, indexed` v' `{`

$dS_{in,c_j^{v'}}^{jv} = dS_{in,c_j^{v'}}^{jv} \ y^{\varphi_j} + g(net_{c_j^v}) \ f'_{in_j}(net_{in_j}) \ \hat{s}_c^{v'}$; `}`

forget gates (5.8), (5.8b), $(dS_{\varphi m}^{jv} := \frac{\partial s_{c_j^v}}{\partial w_{\varphi_j m}}, dS_{\varphi c_j^{v'}}^{jv} := \frac{\partial s_{c_j^v}}{\partial w_{\varphi_j c_j^{v'}}})$:

$dS_{\varphi m}^{jv} = dS_{\varphi m}^{jv} \ y^{\varphi_j} + \hat{s}_{c_j^v} \ f'_{\varphi_j}(net_{\varphi_j}) \ \hat{y}^m$;

`loop over peephole connections from all cells, indexed` v' `{`

$dS_{\varphi c_j^{v'}}^{jv} = dS_{\varphi c_j^{v'}}^{jv} \ y^{\varphi_j} + \hat{s}_{c_j^v} \ f'_{\varphi_j}(net_{\varphi_j}) \ \hat{s}_c^{v'}$; `}`

`} } end loops over cells and memory blocks`

200
nidh

backward pass (if error injected):

errors and δ**s:**

injection error: $e_k = t^k - y^k$;

δ**s of output units (5.10):** $\delta_k = f'_k(net_k) \ e_k$;

`loop over memory blocks, indexed` j `{`

δ**s of output gates (5.11b):**

$\delta_{out_j} = f'_{out_j}(net_{out_j}) \ \left(\sum_{v=1}^{S_j} \ s_{c_j^v} \sum_k w_{kc_j^v} \ \delta_k\right)$;

internal state error (5.15):

`loop over the` S_j `cells in block` j`, indexed` v `{`

$e_{s_{c_j^v}} = y^{out_j} \ \left(\sum_k w_{kc_j^v} \ \delta_k\right)$; `}`

`} end loop over memory blocks`

weight updates:

output units (5.9): $\Delta w_{km} = \alpha \ \delta_k \ y^m$;

`loop over memory blocks, indexed` j `{`

output gates (5.11a):

$\Delta w_{out,m} = \alpha \ \delta_{out} \ \hat{y}^m$; $\quad \Delta w_{out,c_j^v} = \alpha \ \delta_{out} \ s_{c_j^v}$;

input gates (5.13):

$\Delta w_{in,m} = \alpha \sum_{v=1}^{S_j} \ e_{s_{c_j^v}} \ dS_{in,m}^{jv}$;

`loop over peephole connections from all cells, indexed` v' `{`

$\Delta w_{in,c_j^{v'}} = \alpha \sum_{v=1}^{S_j} \ e_{s_{c_j^v}} \ dS_{in,c_j^{v'}}^{jv}$; `}`

forget gates (5.14):

$\Delta w_{\varphi m} = \alpha \sum_{v=1}^{S_j} \ e_{s_{c_j^v}} \ dS_{\varphi m}^{jv}$;

`loop over peephole connections from all cells, indexed` v' `{`

$\Delta w_{\varphi c_j^{v'}} = \alpha \sum_{v=1}^{S_j} \ e_{s_{c_j^v}} \ dS_{\varphi c_j^{v'}}^{jv}$; `}`

cells (5.12):

`loop over the` S_j `cells in block` j`, indexed` v `{`

$\Delta w_{c_j^v m} = \alpha \ e_{s_{c_j^v}} \ dS_{cm}^{jv}$; `};`

`} end loop over memory blocks`

Experiments: first some LSTM limitations

- Was tested on classical time series that feedforward nets learn well when tuned (MackeyGlass...)
- LSTM: 1 input unit, 1 input at a time (memory overhead)
  FNN: 6 input units (no need to learn what to store)
- LSTM extracts basic wave; but best FNN better!
- Parity: random weight search outperforms all!
- So: use LSTM only when simpler approaches fail! Do not shoot sparrows with cannons.
- Experience: LSTM likes sparse coding.

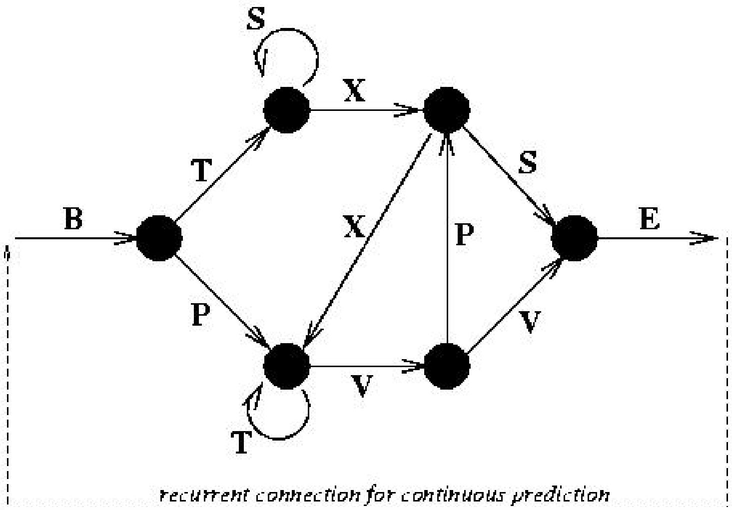
“True” Sequence Experiments

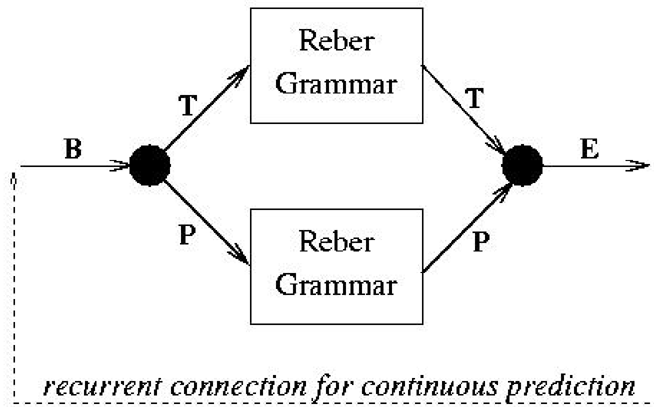
LSTM in a league by itself

- Noisy extended sequences
- Long-term storage of real numbers
- Temporal order of distant events
- Info conveyed by event distances
- Stable smooth and nonsmooth trajectories, rhythms
- Simple regular, context free, context sensitive grammars (Gers, 2000)
- Music composition (Eck, 2002)
- Reinforcement Learning (Bakker, 2001)
- Metalearning (Hochreiter, 2001)
- Speech (vs HMMs)? One should try it….

Regular Grammars: LSTM vs Simple RNNs

(Elman 1988) & RTRL / BPTT *(Zipser & Smith)*





method	hidden units	# weights	learning rate	% of success	success after
RTRL	3	≈ 170	0.05	"some fraction"	173,000
RTRL	12	≈ 494	0.1	"some fraction"	25,000
ELM	15	≈ 435		0	>200,000
RCC	7-9	≈ 119-198		50	182,000
LSTM	4 blocks, size 1	264	0.1	100	39,740
LSTM	3 blocks, size 2	276	0.1	100	21,730
LSTM	3 blocks, size 2	276	0.2	97	14,060
LSTM	4 blocks, size 1	264	0.5	97	9,500
LSTM	3 blocks, size 2	276	0.5	100	8,440

Contextfree / Contextsensitive Languages

	Train[n]	% Sol.	Test[n]
A^nB^n			
Wiles & Elman 95	1…11	20%	1…18
LSTM	1…10	100%	1…1000
$A^nB^nC^n$			
LSTM	1…50	100%	1…500

What this means:

- --------------------***LEGAL:***--------------------
- aaaaa…..aaabbbbb…..bbbccccc…..ccc

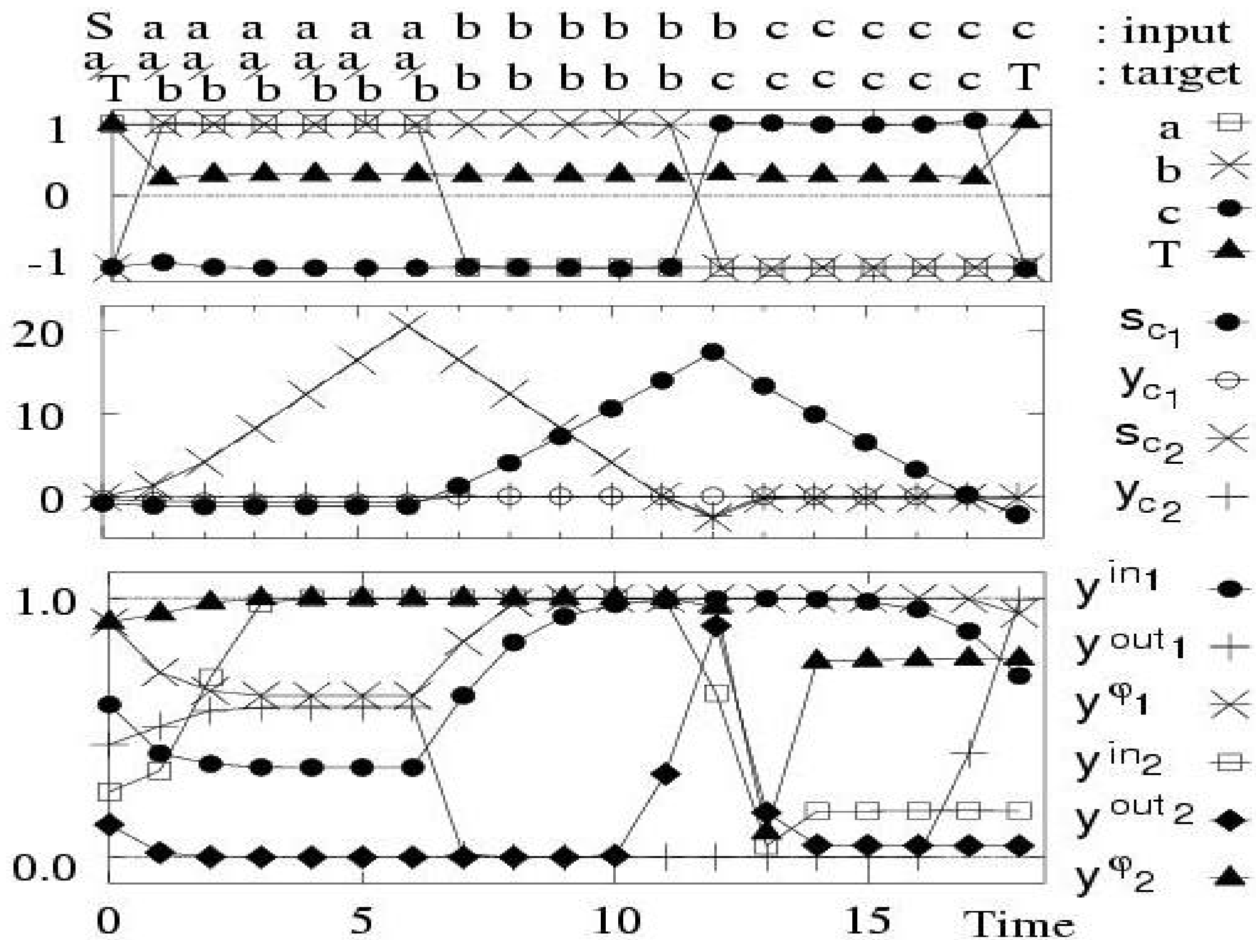
  500 500 500
- --------------------***ILLEGAL:***-------------------

  aaaaa…..aaabbbbb…..bbbccccc…..ccc

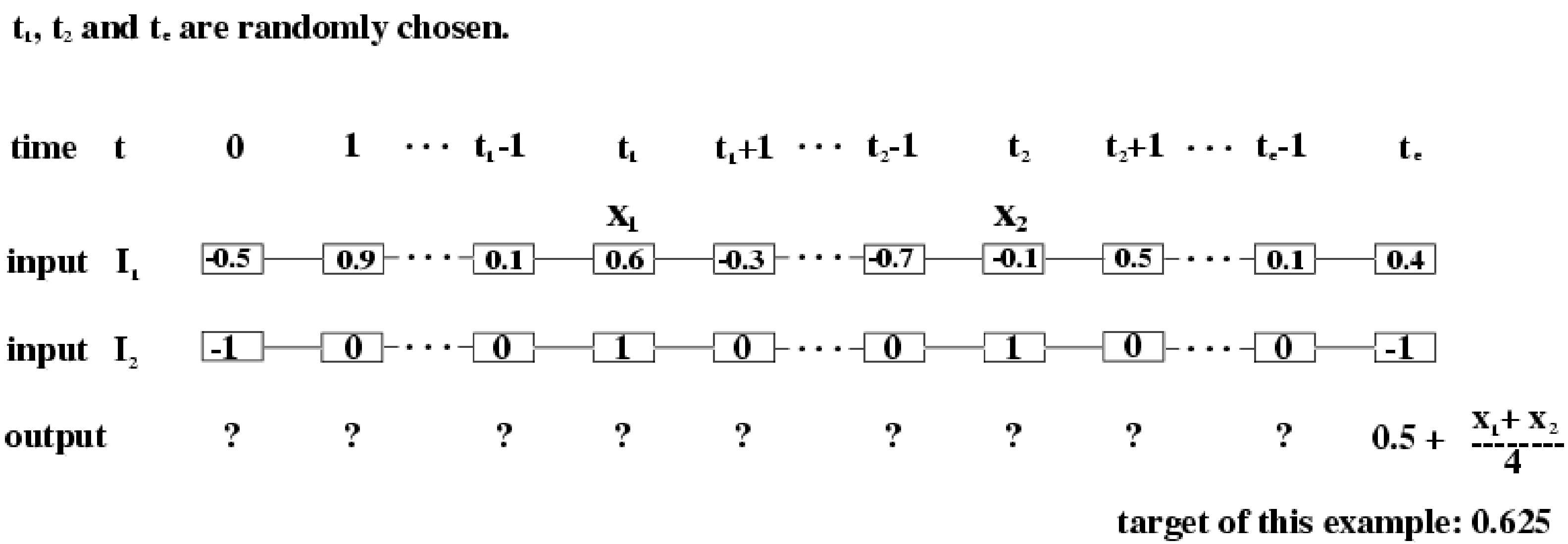
  500 499 500

- LSTM + Kalman: up to n=22,000,000 (Perez, 2002)!!!

Typical evolution of activations



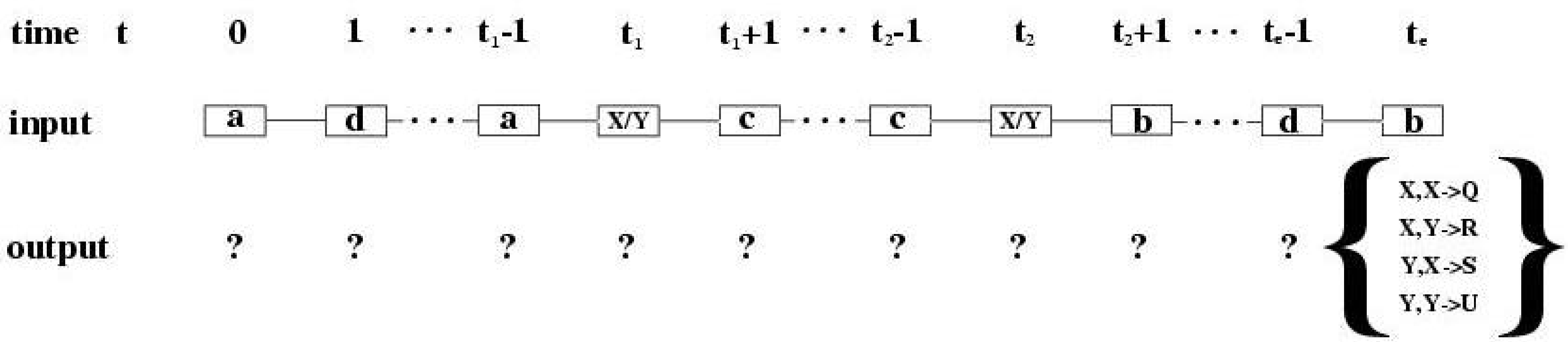
Storing & adding real values



- T=100: 2559/2560; 74,000 epochs
- T=1000: 2559/2560; 850,000 epochs

Noisy temporal order

t_1, t_2 and t_e are randomly chosen. At time t_1 and t_2 an input is randomly chosen from {X,Y}.



- T=100: 2559/2560 correct;
- 32,000 epochs on average

Noisy temporal order II

- Noisy sequences such as aabab...dcaXca...abYdaab...bcdXdb….
- 8 possible targets after 100 steps:
- X,X,X $\rightarrow$ 1; X,X,Y $\rightarrow$ 2; X,Y,X $\rightarrow$ 3; X,Y,Y $\rightarrow$ 4; Y,X,X $\rightarrow$ 5; Y,X,Y $\rightarrow$ 6; Y,Y,X $\rightarrow$ 7; Y,Y,Y $\rightarrow$ 8;
- 2558/2560 correct (error < 0.3)
- 570,000 epochs on average

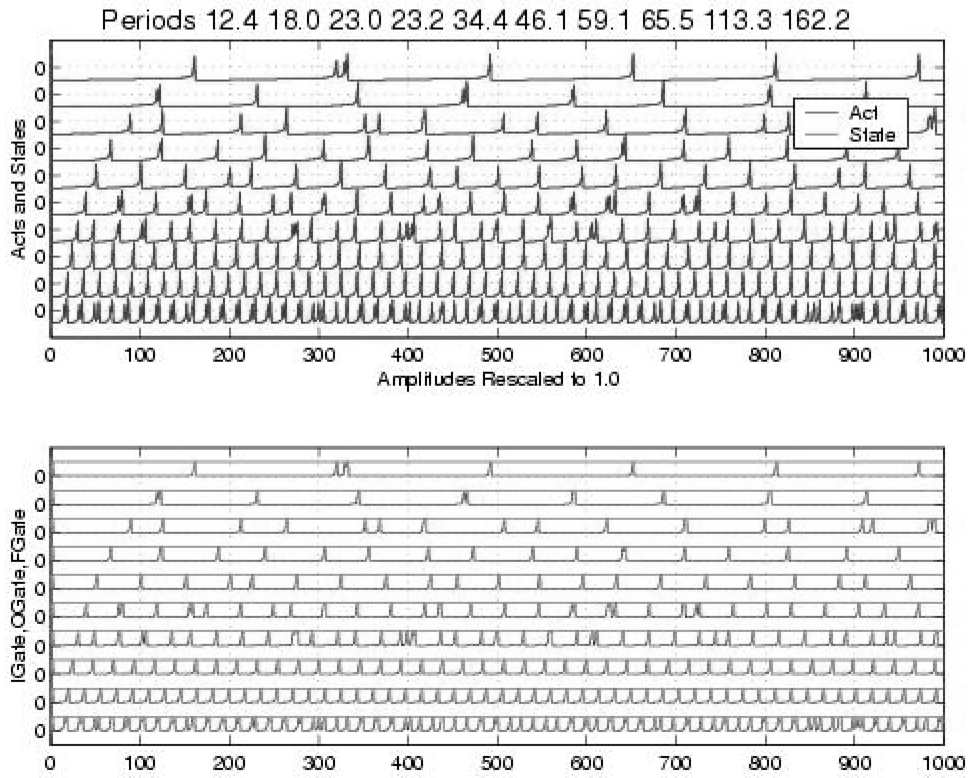
Learning to compose music with RNNs?

- Previous work by Mozer, Todd, others…
- Train net to produce probability distribution on next notes, given past
- Traditional RNNs do capture local structure, such as typical harmony sequences
- RNNs fail to extract global structure
- Result: “Bach Elevator Muzak” :-)
- Question: does LSTM find global structure?

Step 1: can LSTM learn precise timing?

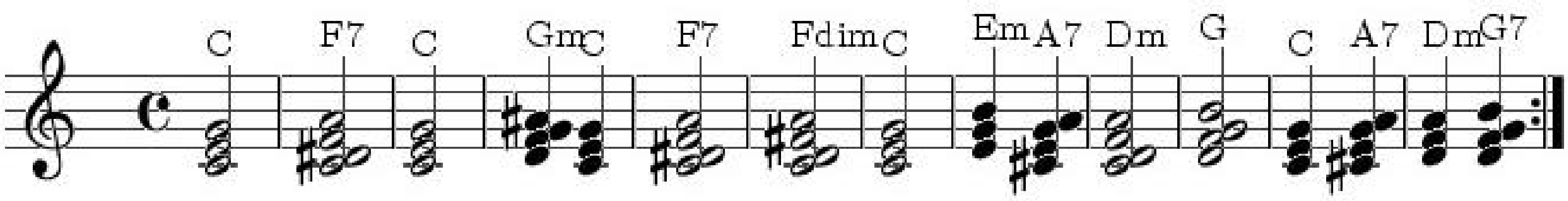
- Yes, can learn to make sharp nonlinear spikes every *n* steps (Gers, 2001)
- For instance: $n = 1,\dots,50,\dots.$ nonvariable
- Or: $n = 1\dots30...$ variable, depending on a special stationary input
- Can also extract info from time delays:
  Target = 1.0 if delay between spikes in input sequence = 20, else target = 0.0
- Compare HMMs which ignore delays

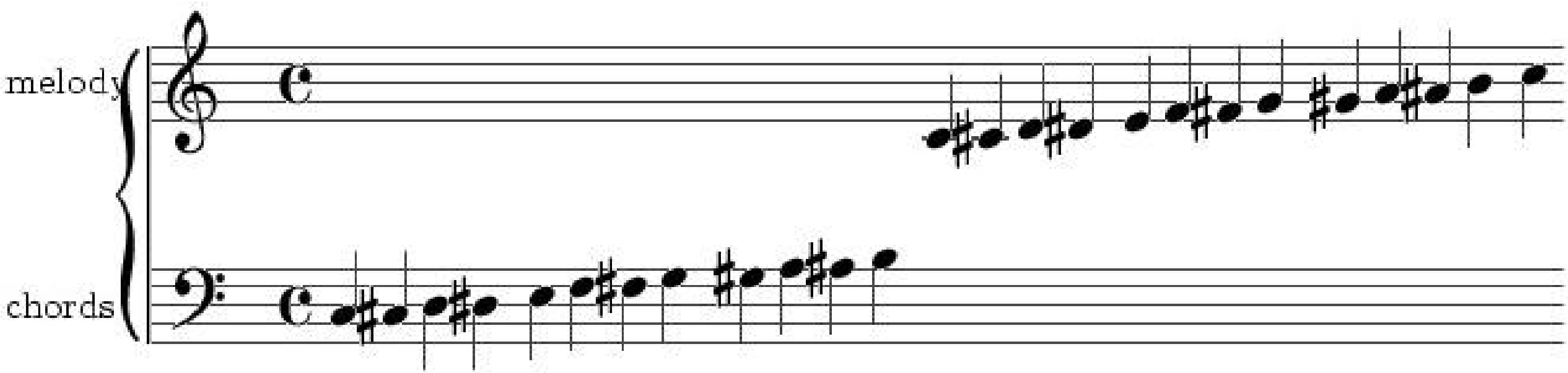
Self-sustaining Oscillation



Step 2: Learning the Blues (Eck, 2002)

- Training form (each bar = 8 steps, 96 steps in total)





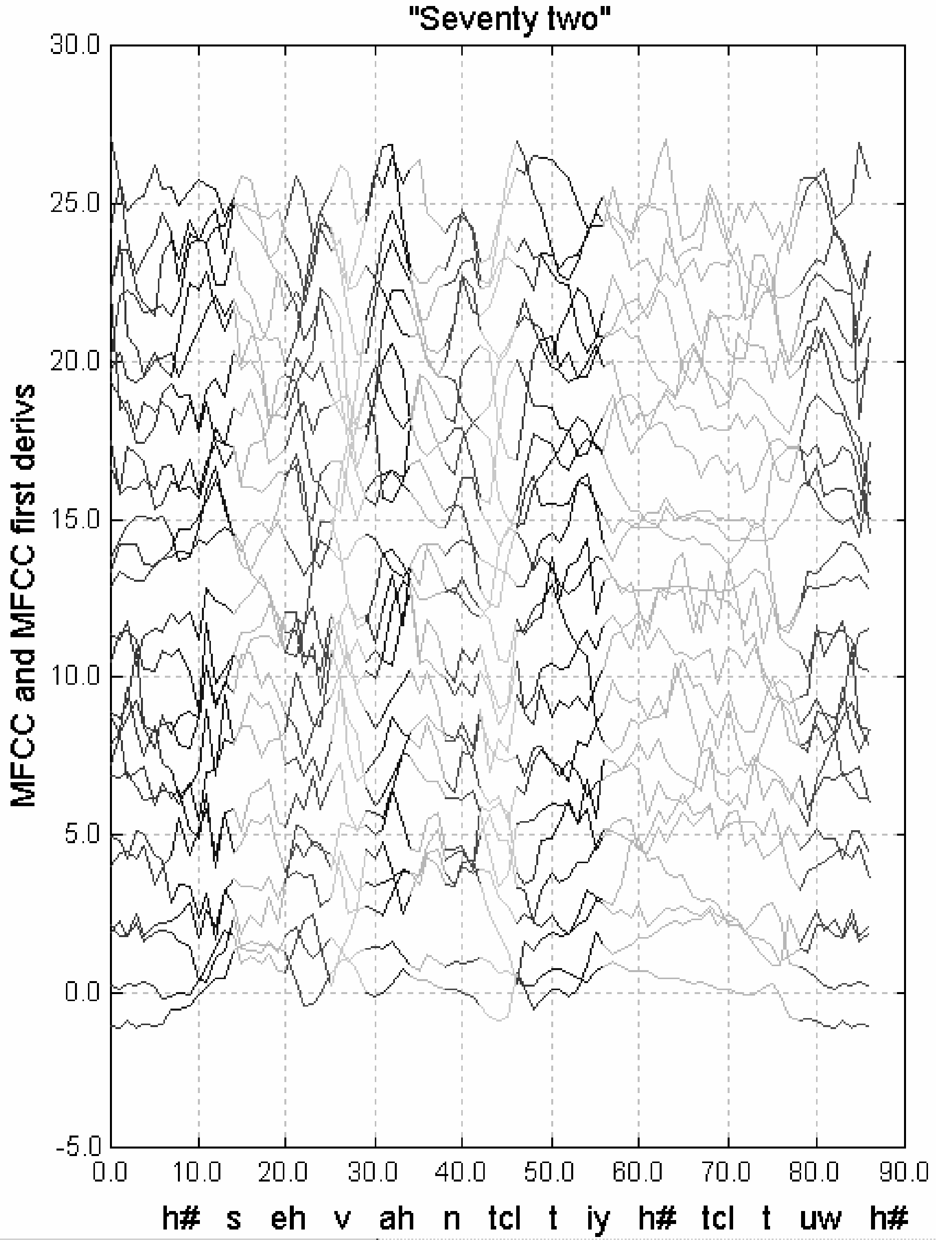
- Representative LSTM composition: 0:00 start; 0:28 -1:12: freer improvisation; 1:12: example of the network repeating a motif not found in the training set.

Speech Recognition

- NNs already show promise (Boulard, Robinson, Bengio)
- LSTM may offer a better solution by finding long-timescale structure
- At least two areas where this may help:
  - Time warping (rate invariance)
  - Dynamic, learned model of phoneme segmentation (with little apriori knowledge)

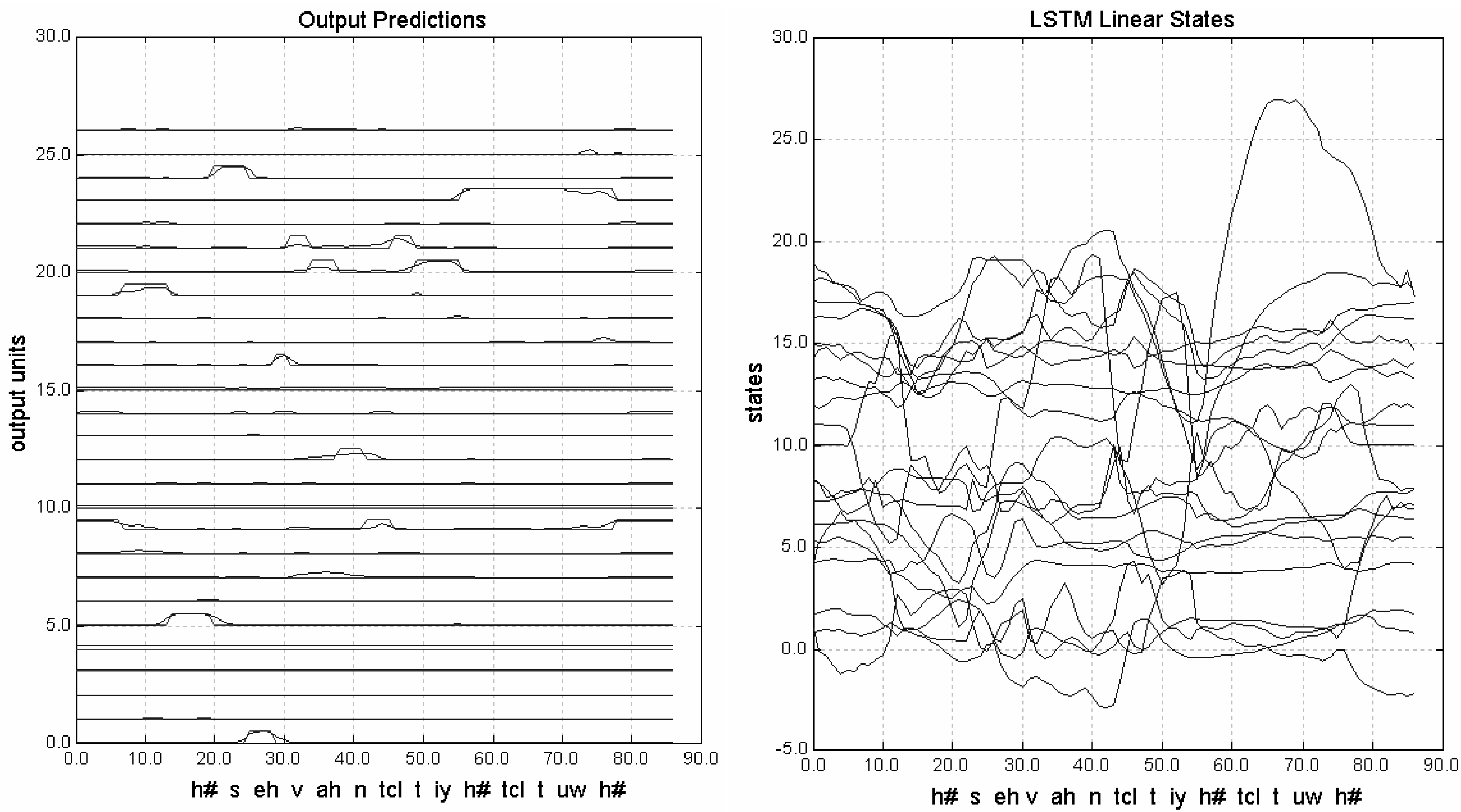
Speech Set 2: Phoneme Identification

- “Numbers 95” database. Numeric street addresses and zip codes (from Bengio)
- 13 MFCC values plus first derivative = 26 inputs
- 27 possible phonemes
- ~=4500 sentences
  ~=77000 phonemes
  ~= 666,000 10ms frames


"Seventy two"
MFCC and MFCC first derivs
30.0
25.0
20.0
15.0
10.0
5.0
0.0
-5.0
0.0 10.0 20.0 30.0 40.0 50.0 60.0 70.0 80.0 90.0
h# s eh v ah n tcl t iy h# tcl t uw h#

Task B: frame-level phoneme recognition

- Assign all frames to one of 27 phonemes.
- Use entire sentence
- For later phonemes, history can be exploited
- Benchmark ~= 80%
- LSTM ~= 78%*
- Nearly as good, despite early stage of LSTM-based speech processing - compare to many man-years of HMM-based speech research.

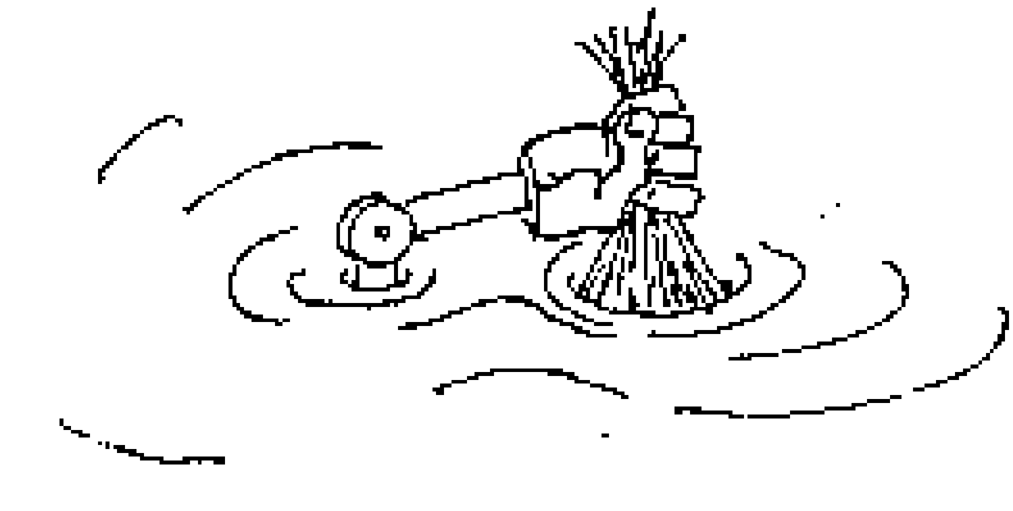


State trajectories suggest a use of history.

Discussion

- Anecdotal evidence suggests that LSTM learns a dynamic representation of phoneme segmentation
- Performance already close to state-of-art HMMs, but very preliminary results
- Much more analysis and simulation required - ongoing work!

Learning to Learn?



Learning to learn

- Schmidhuber (*1993):* a self-referential weight matrix. RNN can read and actively change its own weights; runs weight change algorithm on itself; uses gradient-based metalearning algorithm to compute better weight change algorithm.
- Did not work well in practice, because standard RNNs were used instead of LSTM.
- But Hochreiter recently used LSTM for metalearning (2001) and obtained astonishing results.

LSTM metalearner (Hochreiter, 2001)

- LSTM, 5000 weights, 5 months training: metalearns fast online learning algorithm for quadratic functions $f(x,y)=a_1x^2+a_2y^2+a_3xy+a_4x+a_5y+a_6$
  Huge time lags.
- After metalearning, freeze weights.
- Now use net: Select new f, feed training exemplars ...data/target/data/target/data... into input units, one at a time. After 30 exemplars the net predicts target inputs before it sees them.
  No weight changes! How?

LSTM metalearner: How?

- On the frozen net runs a sequential learning algorithm which computes something like error signals from inputs recognized as data and targets.

- Parameters of *f*, errors, temporary variables, counters, computations of *f* and of parameter updates are all somehow represented in form of circulating activations.

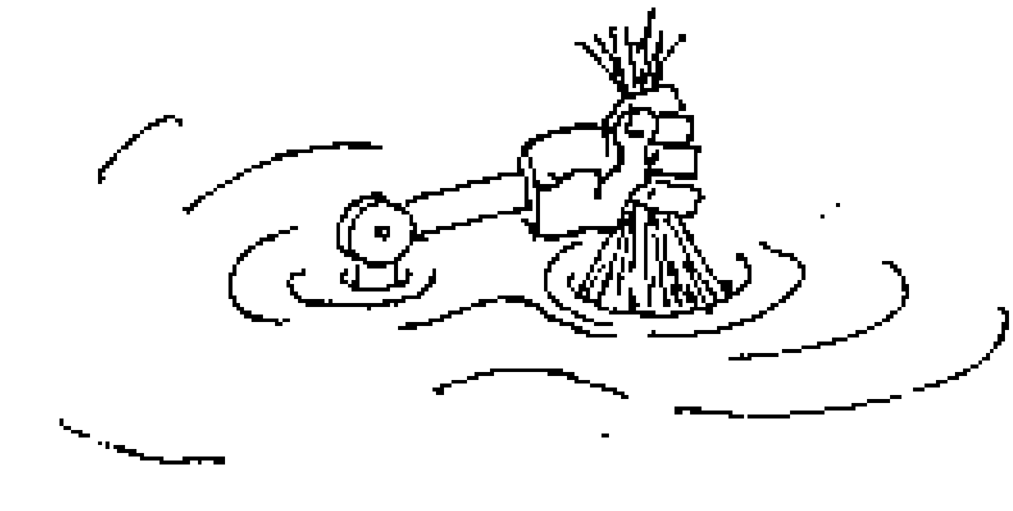
LSTM metalearner

- New learning algorithm much faster than standard backprop with optimal learning rate: *O(30) : O(1000)*

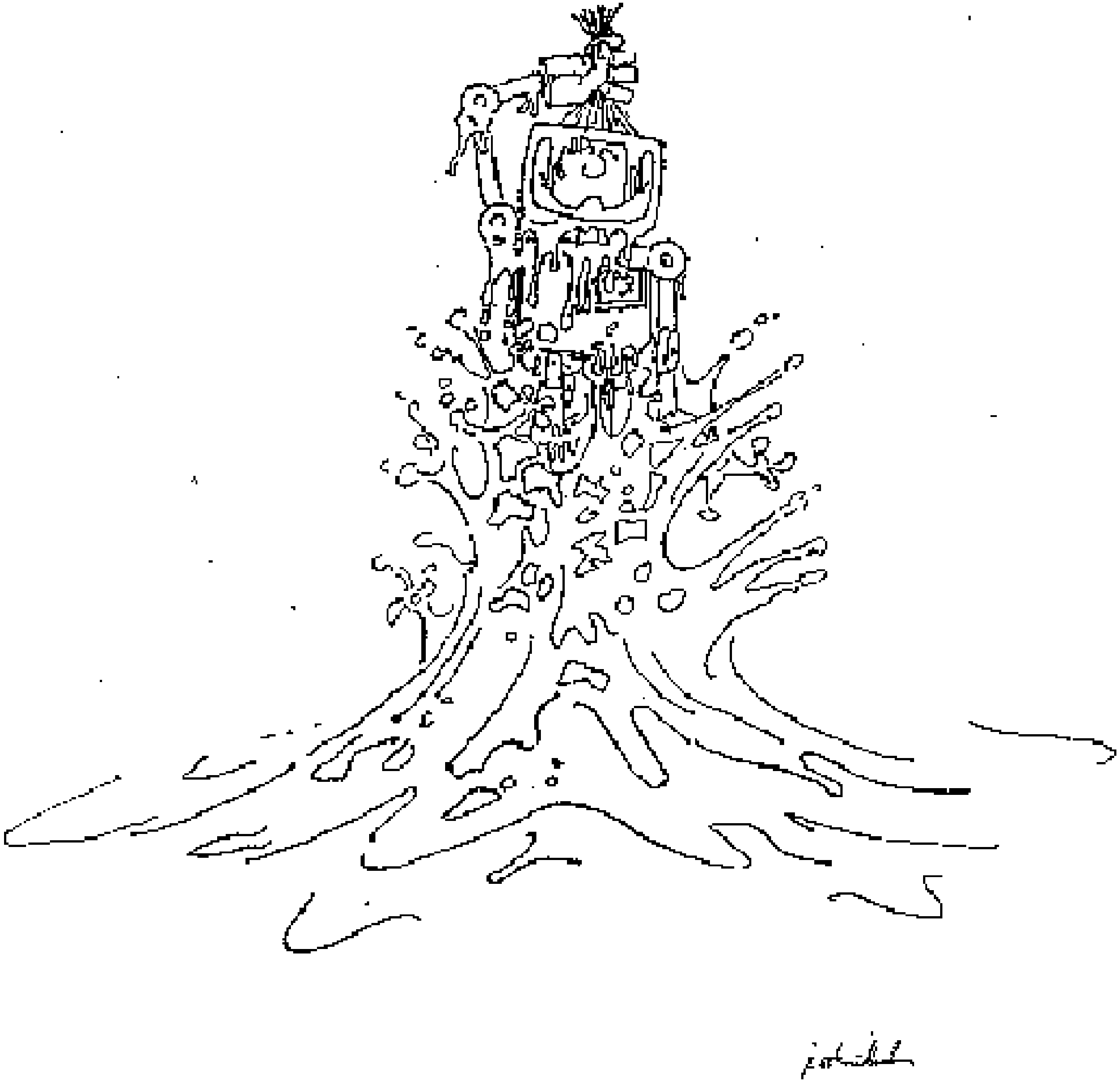
- Gradient descent metalearns online learning algorithm that outperforms gradient descent.

- Metalearning automatically avoids overfitting, since it punishes overfitting online learners just like slow ones: more cumulative errors!

Learning to Learn?



Some day

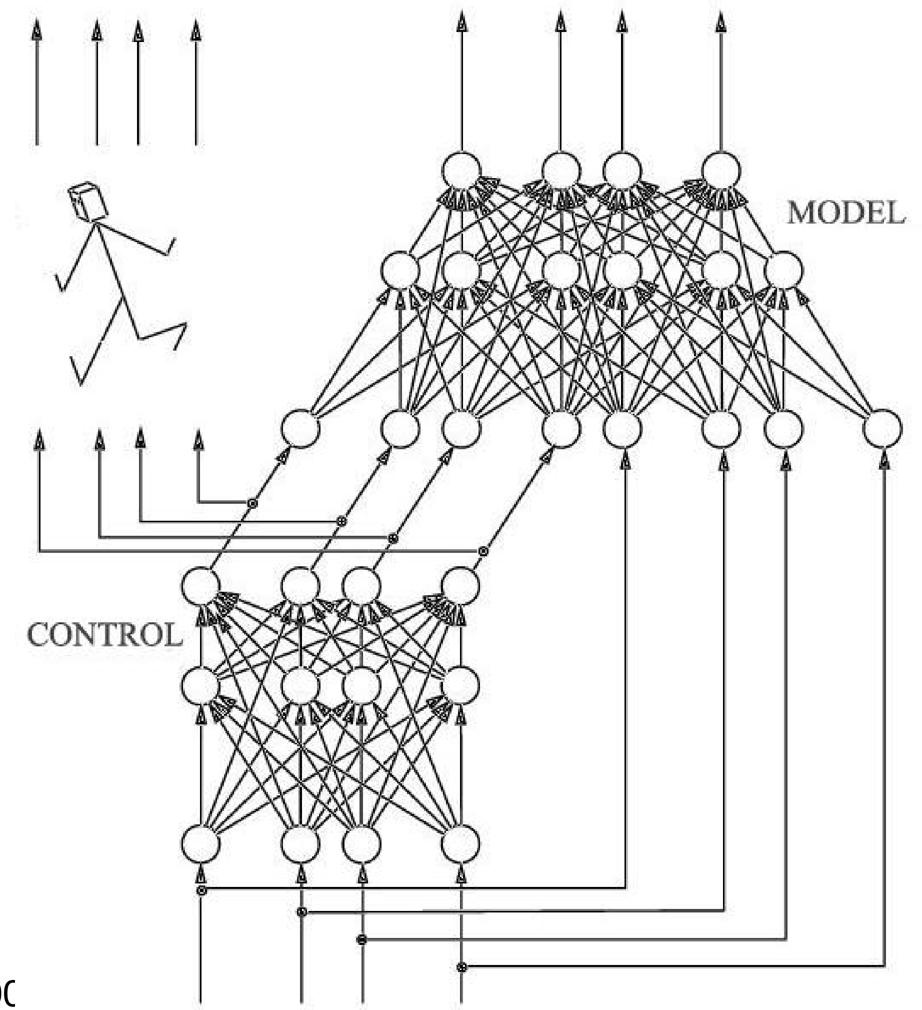


Reinforcement Learning with RNNs

- Forward model
  (Werbos, Jordan & Rumelhart, Nguyen & Widrow)

- Train model, freeze it, use it to compute gradient for controller

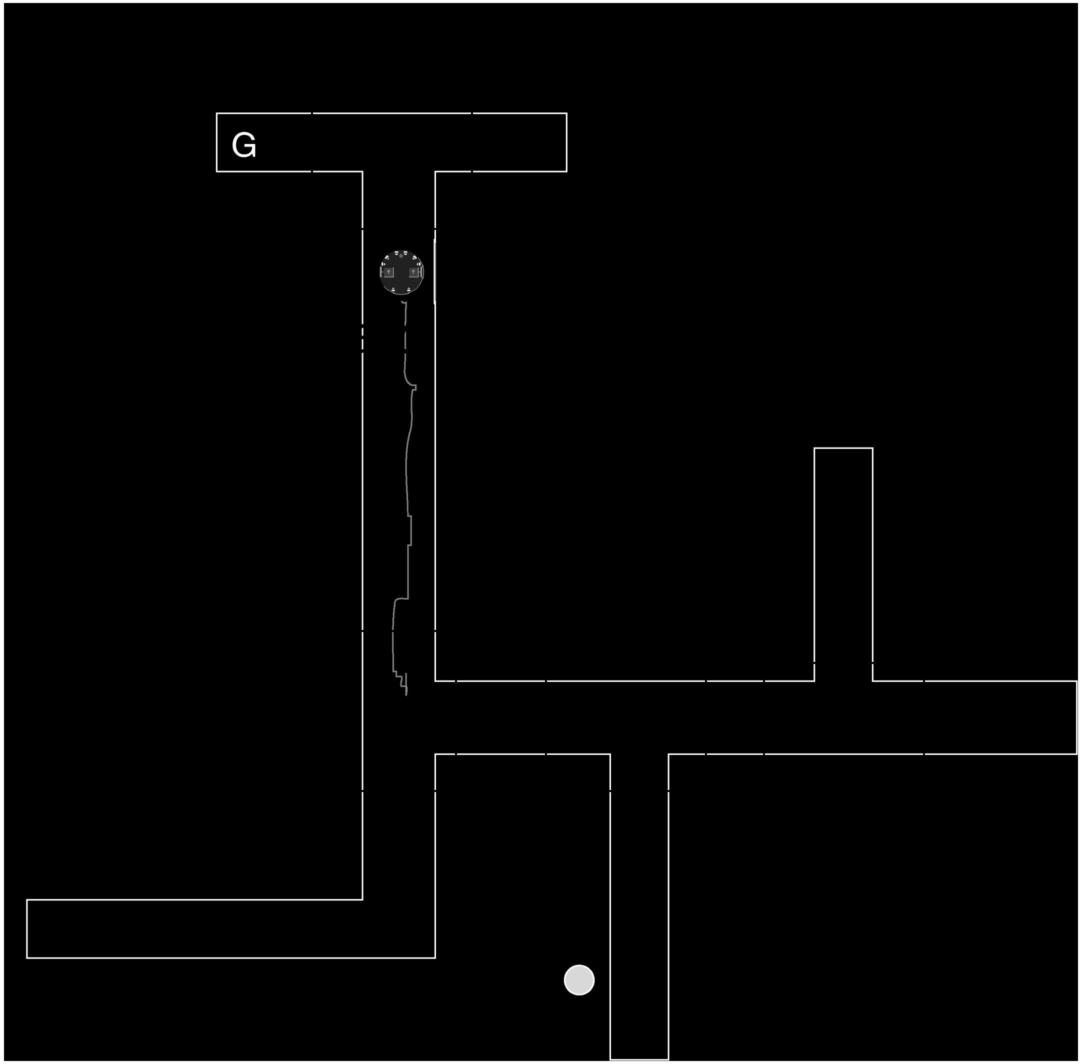
- Recurrent Controller & Model (Schmidhuber 1990)



Reinforcement Learning RNNs II

- Use RNN as function approximator for standard RL algorithms
(Schmidhuber, IJCNN 1990, NIPS 1991, Lin, 1993)
- Use LSTM as function approximator for standard RL (Bakker, NIPS 2002)
- Fine results

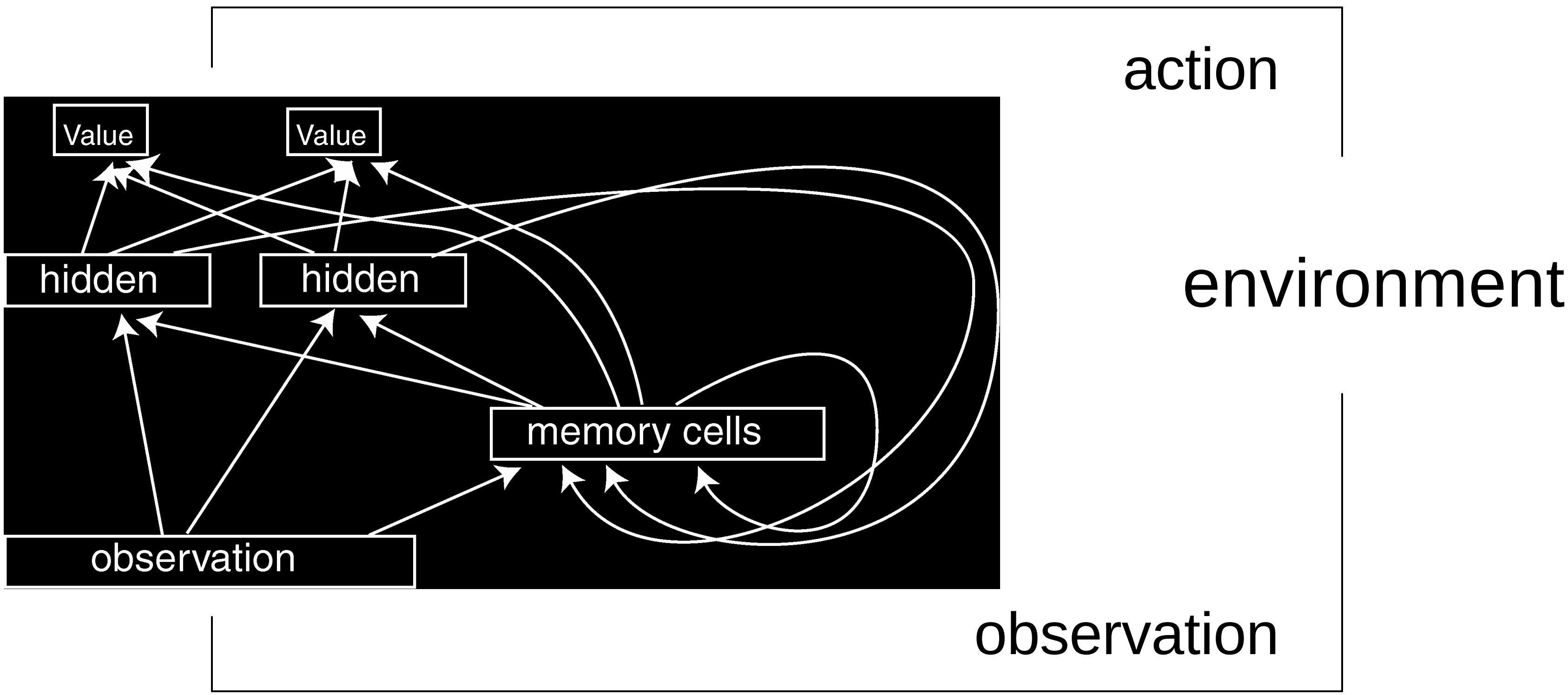
Using LSTM for POMDPs (Bakker, 2001)



*To the the robot, all T-junctions look the same. Needs **short-term memory** to disambiguate them!*

LSTM to approximate value function of reinforcement learning (RL) algorithm

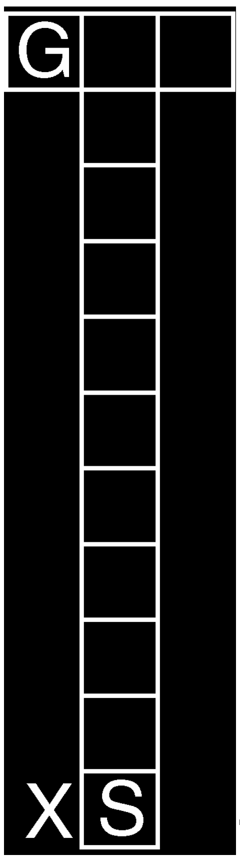
*Network outputs correspond to **values** of various **actions**, learned through **Advantage Learning** RL algorithm*



In contrast with supervised learning tasks, now LSTM determines its own subsequent inputs, by means of its outputs!

Test problem 1: Long-term dependency T-maze with noisy observations

observation

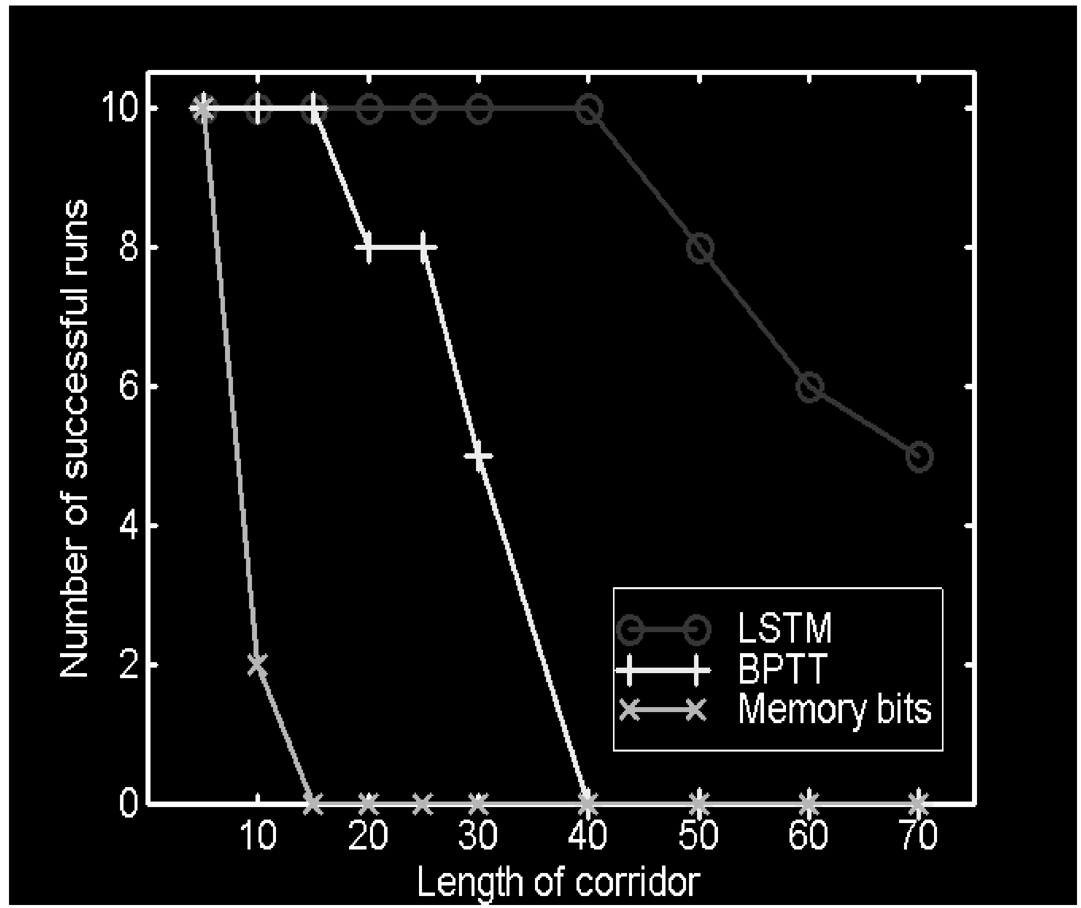


010

a0b

a and b random in [0,1]

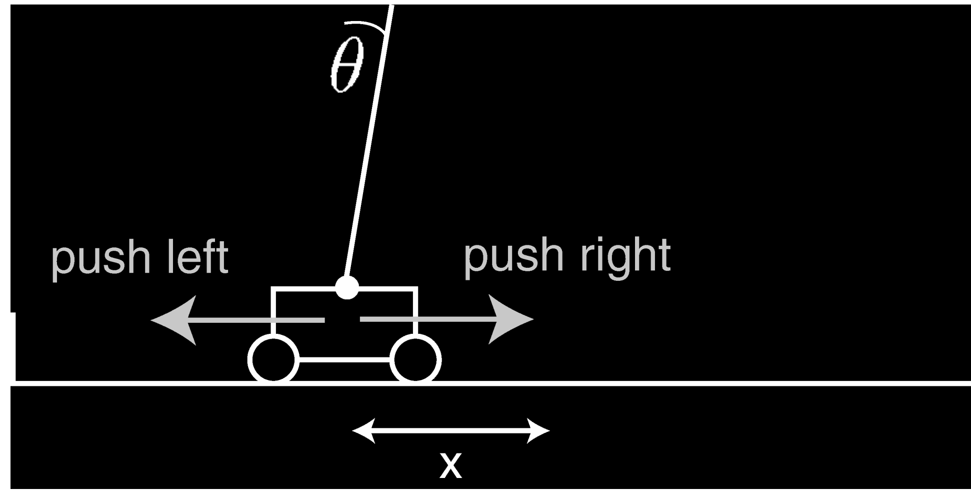
110 (011)



Test problem 2: partially observable, multi-mode pole balancing

- State of the environment:

  $x, \dot{x}, q, \dot{q}$

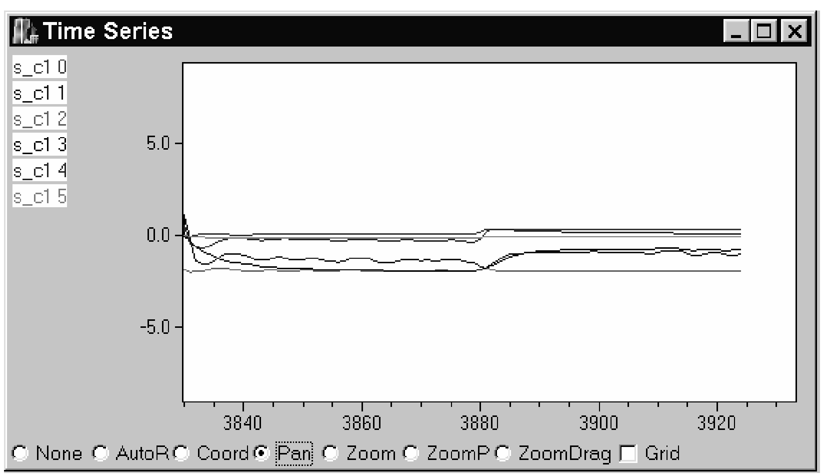


- Observation:
  - x, q : so $\dot{x}, \dot{q}$ *must be learned*
  - 1st second of episode (50 it.): “mode of operation”
    - mode A: action 1 is left, action 2 is right
    - mode B: action **2** is left, action **1** is right
- Requires combination of continuous & discrete internal state, and to remember “mode of operation” **indefinitely**

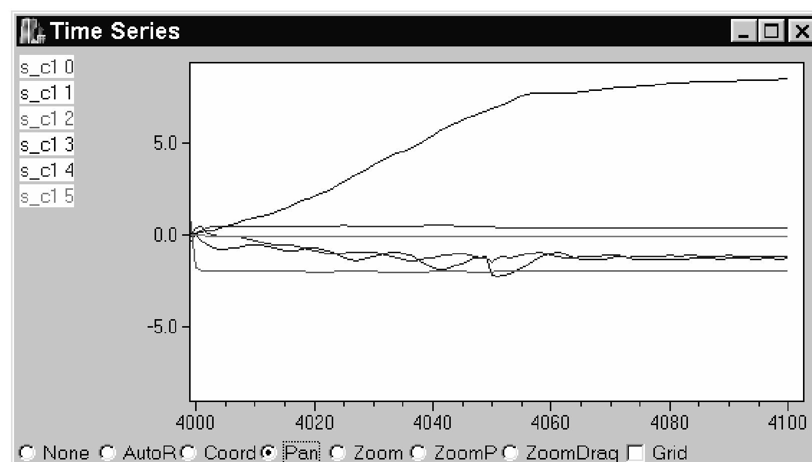
Results

- BPTT never reached satisfactory solution
- LSTM learned perfect solution in 2 out of 10 runs (after 6,250,000 it.). In 8 runs the pole balances in both modes for hundreds or thousands of timesteps (after 8,095,000 it.).

Internal state evolution of memory cells after learning



mode A



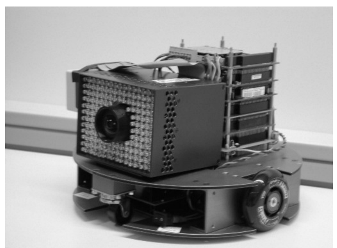
mode B

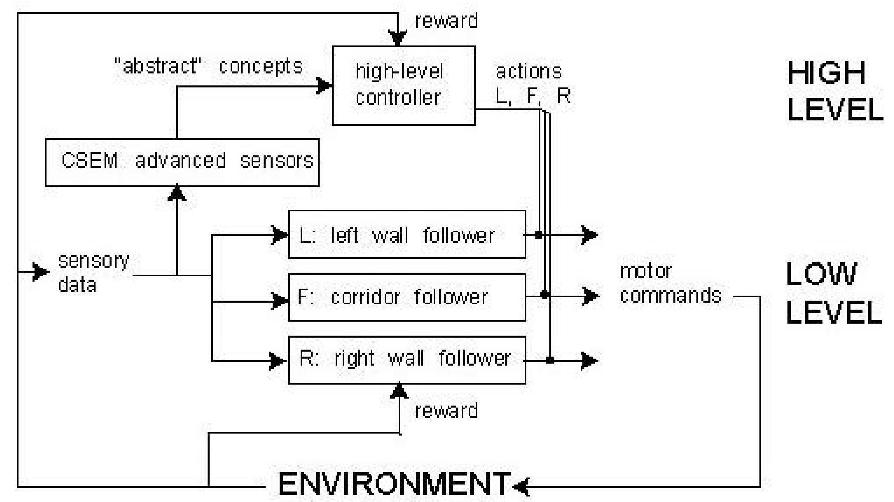
Ongoing: Reinforcement Learning Robots Using LSTM

Goal / Application

- Robots that *learn* complex behavior, based on *rewards*
- Behaviors that are hard to program, e.g. navigation in offices, object recognition and manipulation

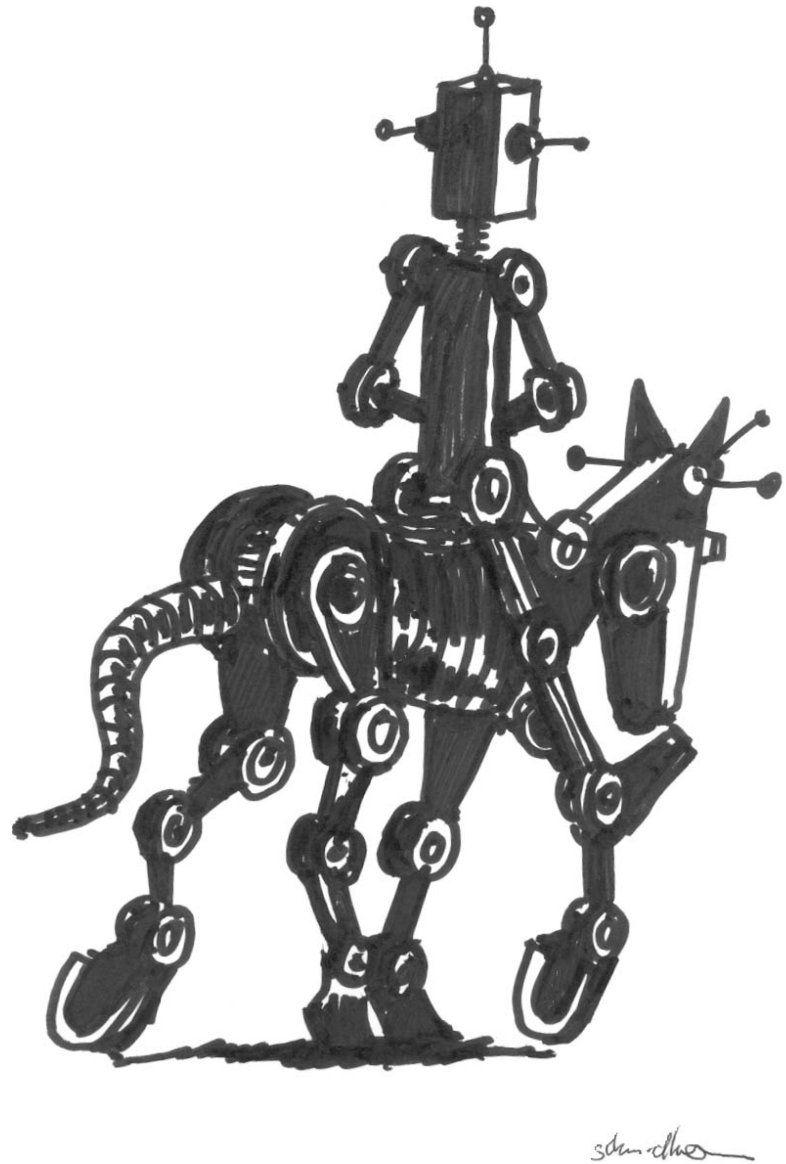
?





Approach

- Collect data from robot, learn controller in simulation, and fine tune again on real robot.
- Hierarchical control
- Exploit CSEM visual sensors




J. Schmidhuber, 2001