Recurrent Neural Networks
A Brief Overview

Douglas Eck

University of Montreal
RNNs versus FFNs

- Feed-Forward Networks (FFNs, left) can learn function mappings (FFN==>FIR filter)
- Recurrent Neural Networks (RNNs, right) use hidden layer as memory store to learn sequences (RNN==>IIR filter)
- RNNs can (in principle at least) exhibit virtually unlimited temporal dynamics
Several Methods

- SRN – Simple Recurrent Network (Elman, 1990)
- BPTT – Backpropagation Through Time (Rumelhart, Hinton & Williams, 1986)
- RTRL – Real Time Recurrent Learning (Williams & Zipser, 1989)
- LSTM – Long Short-Term Memory (Hochreiter & Schmidhuber, 1996)
SRN (Elman Net)

- Simple Recurrent Networks
- Hidden layer activations copied into a copy layer
- Cycles are eliminated, allowing use of standard backpropagation
Some Observations by Elman

- Some problems change nature when expressed in time
- Example: Temporal XOR (Elman, 1990) 
  \[101101110110000011000\]
- RNN learns Frequency Detectors
- Error can give information about temporal structure of input
- Increasing sequential dependencies does not necessarily make task harder
- Representation of time is task-dependent
SRN Strengths

- Easy to train
- Potential for complex and useful temporal dynamics
- Can induce hierarchical temporal structure
- Can learn, e.g., simple “natural language” grammar
SRN Shortcomings

• We need to compute weight changes at every timestep $t_0 < t \leq t_1$:
  \[ \sum_{t=t_0+1}^{t_1} \Delta w_{ij}(t) \]

• Thus we need to compute at every $t$:
  \[ - \frac{\partial E(t)}{\partial w_{ij}} = - \sum_{k \in U} \frac{\partial(t)}{\partial y_k(t)} \frac{\partial y_k(t)}{\partial w_{ij}} = \sum_{k \in U} e_k(t) \frac{\partial y_k(t)}{\partial w_{ij}} \]

• Since we know $e_k(t)$ we need only compute $\frac{\partial y_k(t)}{\partial w_{ij}}$

• SRN Truncates this derivative

• Long-timescale (and short-timescale) dependencies between error signals are lost
SRNs Generalized: BPTT

- Generalize SRN to remember deeper into past
- Trick: unfold network to represent time spatially (one layer per discrete timestep)
- Still no cycles, allowing use of standard backpropagation
BPTT(∞)

For each timestep t

- Current state of network and input pattern is added to history buffer (stores since time t=0)
- Error $e_k(t)$ is injected; $\epsilon$s and $\delta$s for times $(t_0 < \tau \leq t)$ are computed:
  \begin{align*}
  \epsilon_k(t) &= e_k(t) \\
  \delta_k(\tau) &= f'_k(s_k(\tau))\epsilon_k(\tau) \\
  \epsilon_k(\tau - 1) &= \sum_{l \in U} w_{lk}\delta_l(\tau)
  \end{align*}
- Weight changes are computed as in standard BP:
  \[
  \frac{\partial E(t)}{\partial w_{ij}} = \sum_{\tau=t_0+1}^{t} \delta_i(\tau)x_j(\tau - 1)
  \]
Truncated/Epochwise BPTT

- When training data is in epochs, can limit size of history buffer to $h$, length of epoch
- When training data not in epochs, can nonetheless truncate gradient after $h$ timesteps
- For $n$ units and $O(n^2)$ weights, epochwise BPTT has space complexity $O(nh)$ and time complexity $O(n^2h)$
- Compares favorably to BPTT$\infty$ (substitute $L$, length of input sequence, for $h$)
BPTT’s Gradient

- Recall that BPTT(∞) computes
  \[ \frac{\partial E(t)}{\partial w_{ij}} = \sum_{\tau = t_0 + 1}^{t} \delta_i(\tau) x_j(\tau - 1) \]

- Thus errors at \( t \) take into account equally \( \delta \) values from the entire history of computation

- Truncated/Epochwise BPTT cuts off the gradient:
  \[ \frac{\partial E(t)}{\partial w_{ij}} = \sum_{\tau = t - h}^{t} \delta_i(\tau) x_j(\tau - 1) \]

- If data is naturally organized in epochs, this is not a problem

- If data is not organized in epochs, this is not a problem either, provided \( h \) is “big enough”
RTRL

- RTRL=Real Time Recurrent Learning
- Instead of unfolding network backward in time, one can propagate error forward in time
- Compute directly $p_{ij}^k(t) = \frac{\partial y_k(t)}{\partial w_{ij}}$
  
  $p_{ij}^k(t + 1) = f'_k(s_k(t)) \sum_{l \in U} w_{kl} p_{ij}^l(t) + \delta_{ik} x_j(t)$

- $\Delta w_{ij} = -\alpha \frac{\partial E(t)}{\partial w_{ij}} = \alpha \sum_{k \in U} e_k(t) p_{ij}^k(t)$
RTRL vs BPTT

- RTRL saves from executing backward dynamics
- Temporal credit assignment solved during forward pass
- RTRL is painfully slow: for $n$ units and $n^2$ weights, $O(n^2)$ space complexity and $O(n^4)(!)$ time complexity
- Because BPTT is faster, in general RTRL is only of theoretical interest
Training Paradigms

• Epochwise training: reset system at fixed stopping points
• Continual training: never reset system
• Epochwise training provides barrier for credit assignment
• Unnatural for many tasks
• Distinct from whether weights are batchwise or iteratively updated
Teacher Forcing

- Continually-trained networks can move far from desired trajectory and never return.
- Especially true if network enters region where activations become saturated (gradients go to 0).
- Solution; replace during training actual output $y_k(t)$ with teacher signal $d_k(t)$.
- Necessary for certain problems in BPTT/RTRL networks (e.g. generating square waves).
Puzzle: How Much is Enough?

- Recall that for Truncated BPTT, the true gradient is not being computed
- How many steps do we need to “get close enough”? 
Puzzle: How Much is Enough?

- Recall that for Truncated BPTT, the true gradient is not being computed.
- How many steps do we need to “get close enough”?
- Answer: certainly not more than 50;
- Probably not more than 10(!)
Credit Assignment is Difficult

- In BPTT, error gets “diluted” with every subsequent layer (credit assignment problem):
  \[ \epsilon_k(t) = e_k(t) \]
  \[ \delta_k(\tau) = f'_k(s_k(\tau))\epsilon_k(\tau) \]
  \[ \epsilon_k(\tau - 1) = \sum_{l \in U} w_{lk} \delta_l(\tau) \]

- The logistic sigmoid \(\frac{1.0}{1.0 + (exp(-x))}\) has maximum derivative of 0.25.

- When \(|\omega| < 4.0\), error is always < 1.0
Vanishing Gradients

• Bengio, Simard & Frasconi (1994)
• Remembering a bit requires creation of attractor basin in state space
• Two cases: either system overly sensitive to noise or error gradient vanishes exponentially
• General problem (HMMs suffer something similar)
Solutions and Alternatives

- Non-gradient learning algorithms (including global search)
- Expectation Maximization (EM) training (e.g. Bengio IO/HMM)
- Hybrid architectures to aid in preserving error signals
LSTM

- Hybrid recurrent neural network
- Make hidden units linear (derivative is then 1.0)
- Linear units unstable
- Place units in a “memory block” protected by multiplicative gates
Inside an LSTM Memory Block

- Gates are standard sigmoidal units
- Input gate $y_{in}$ protects linear unit $s_c$ from spurious inputs
- Forget gate $y_{\phi}$ allows $s_c$ to empty own contents
- Output gate $y_{out}$ allows block to take itself offline and ignore error
LSTM Learning

• For linear unit $s_c$, a truncated RTRL approach is used (tables of partial derivatives)
• Everywhere else, standard backpropagation is used
• Rationale: due to vanishing gradient problem, errors would decay anyway
Properties of LSTM

• Very good at finding hierarchical structure
• Can induce nonlinear oscillation (for counting and timing)
• But error flow among blocks truncated
• Difficult to train: weights into gates are sensitive
Formal Grammars

- LSTM solves Embedded Reber Grammar faster, more reliably and with smaller hidden layer than RTRL/BPTT
- LSTM solves CSL $A^n B^n C^n$ better than RTRL/BPTT networks
- Trained on examples with $n < 10$ LSTM generalized to $n > 1000$
- BPTT-trained networks generalize to $n \leq 18$ in best case (Bodén & Wiles, 2001)