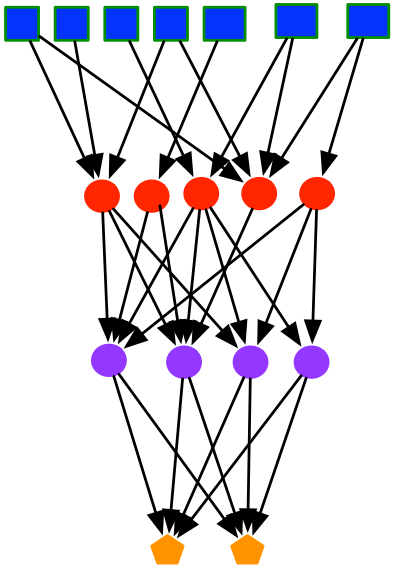
Recurrent Neural Networks (RNNs)

Keith L. Downing

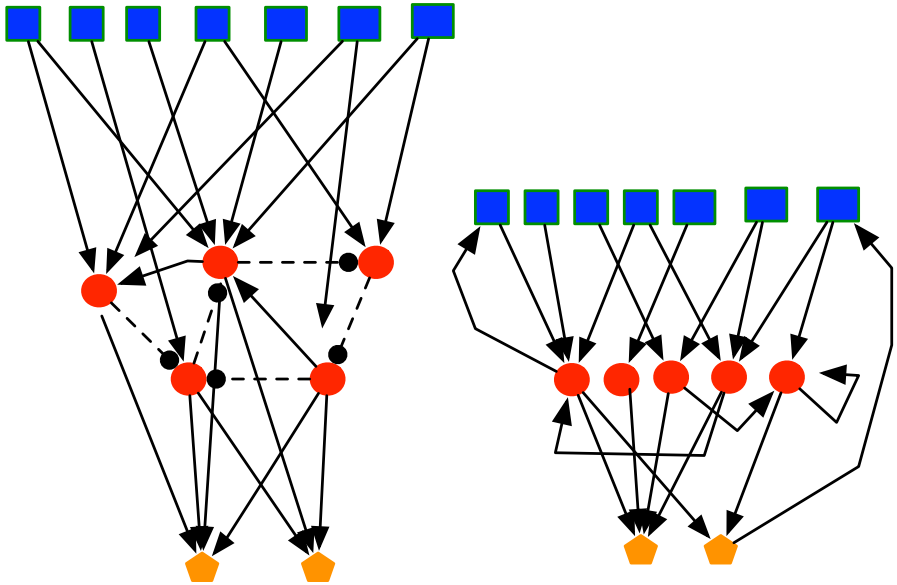
The Norwegian University of Science and Technology (NTNU)
Trondheim, Norway
keithd@idi.ntnu.no

November 7, 2017

Recurrent Networks for Sequence Learning



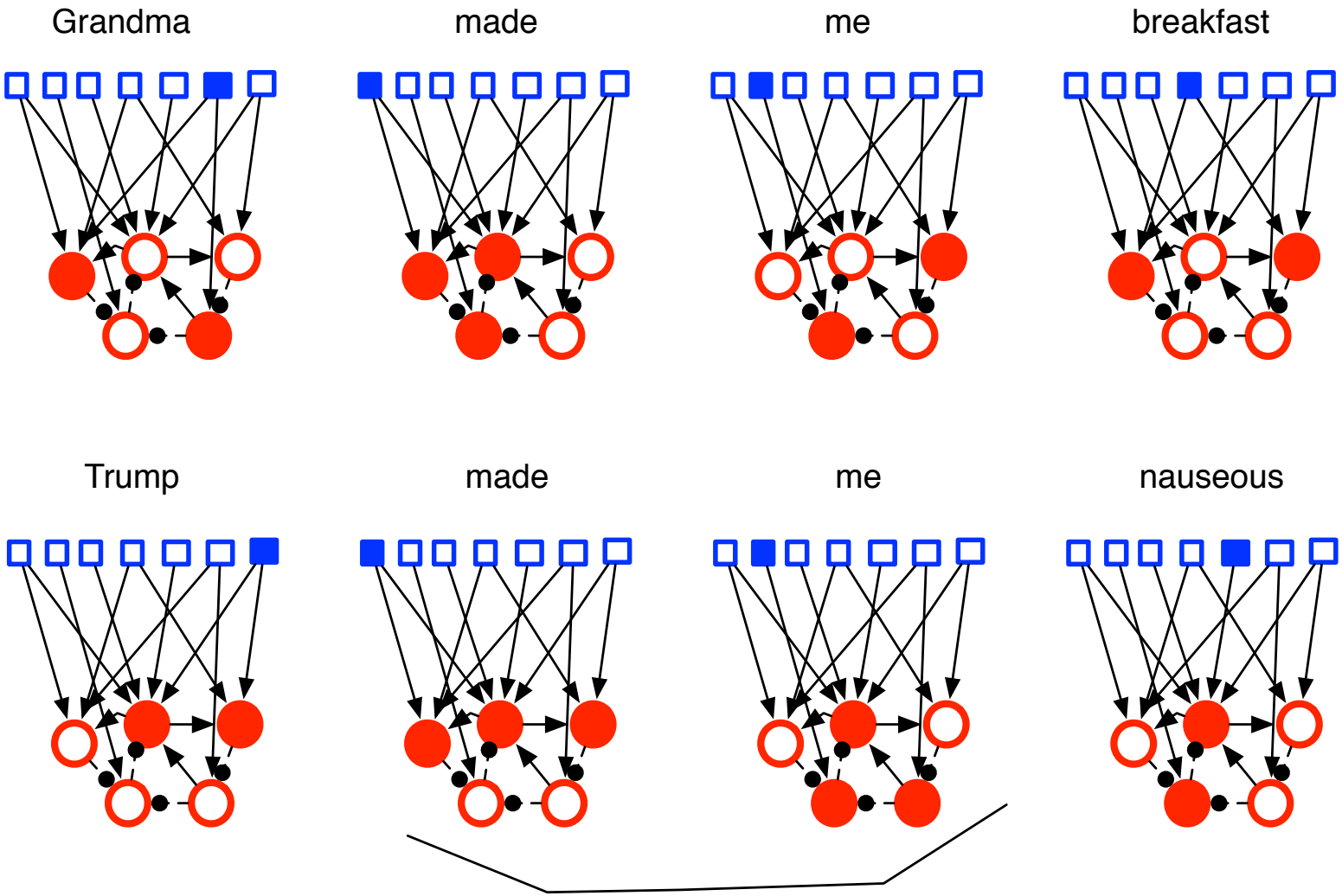
Strictly Feed-Forward



Recurrent

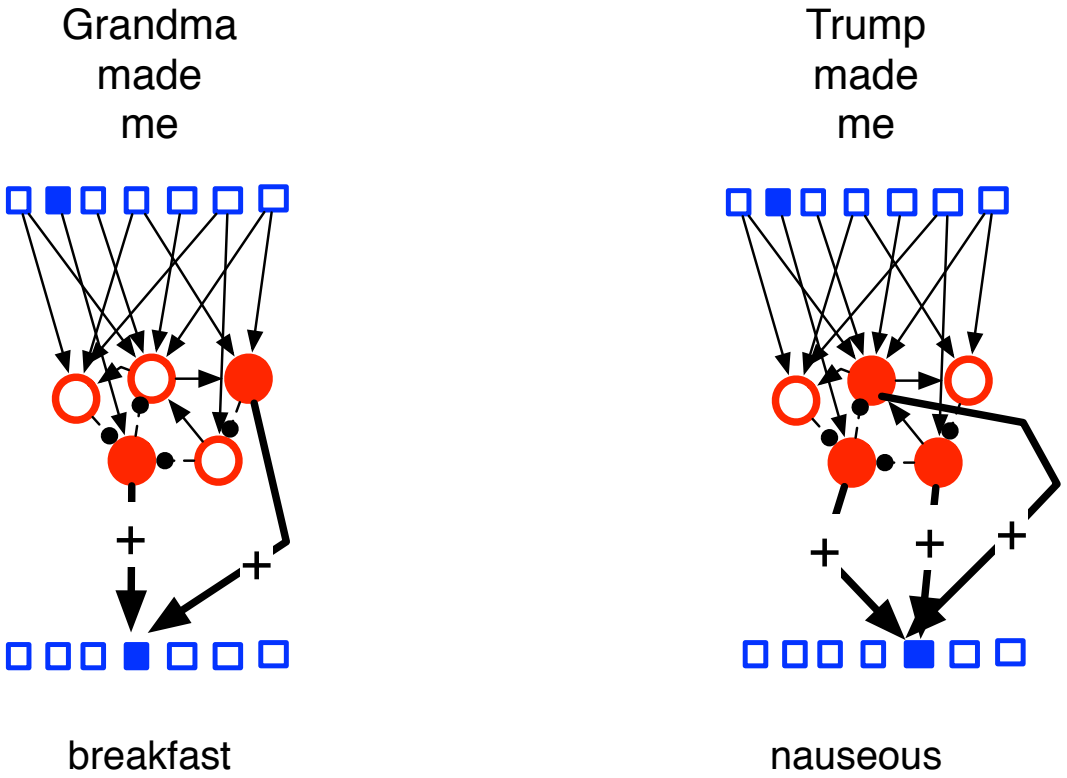
Feedback connections create directed cycles, which produce dynamic memory: current neural activation states reflect past activation states.

History-Dependent Activation Patterns



Same two inputs produce different internal patterns due to different prior contexts: Grandma -vs- Trump

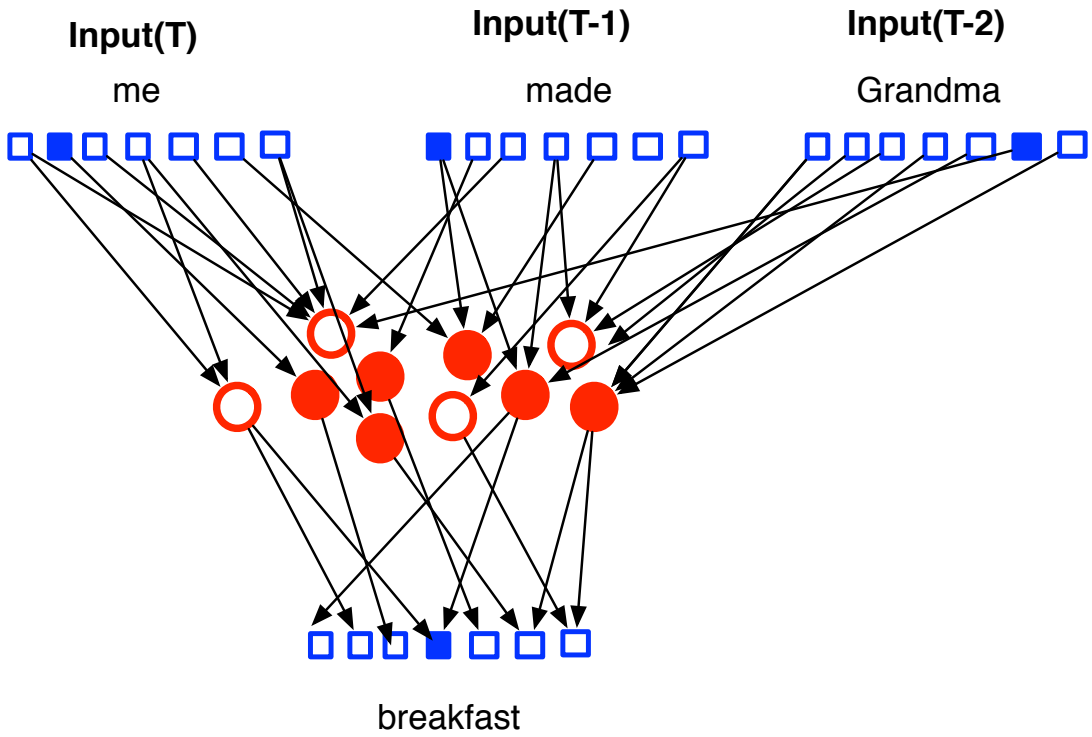
Sequence Completion = Prediction



By modifying this mapping from internal activation states to outputs (via weight modification, i.e. learning), the net can predict the next word in a sentence.

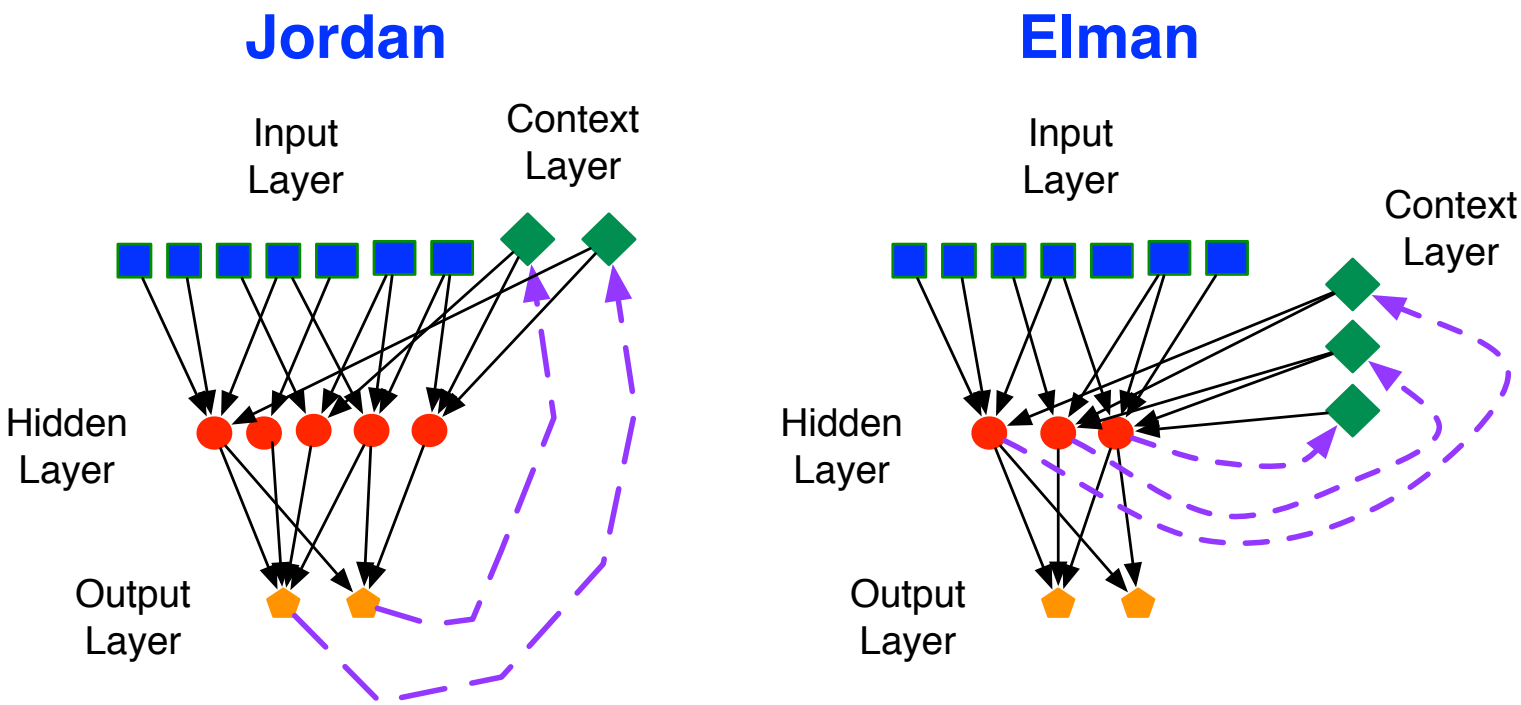
Time-Delay Nets

Include history in the input, so no recurrence needed; similar to NetTalk's use of context.



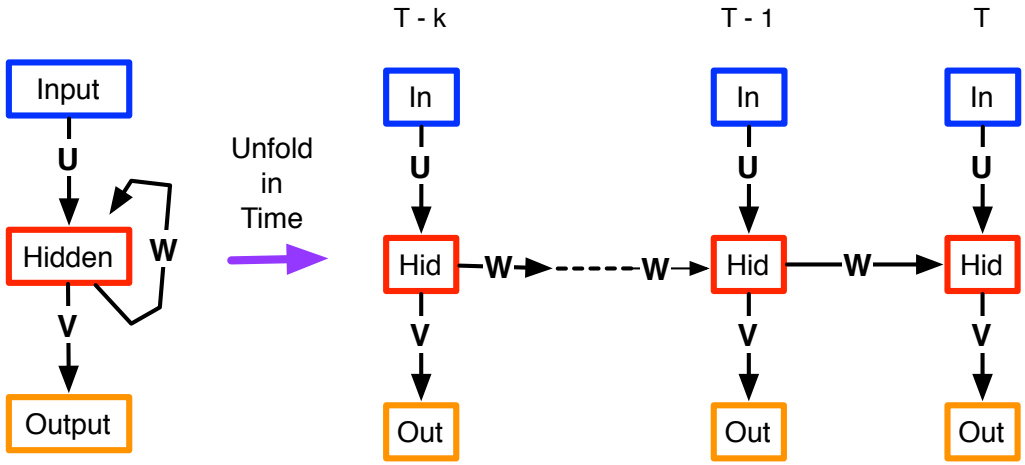
- But this requires n-grams as input, and there are many of them!
- Over-sensitivity to word position.
- Trump made me nauseous and Trump made me very nauseous would be handled very differently, because they yield different n-grams.

Elman -vs- Jordan Nets

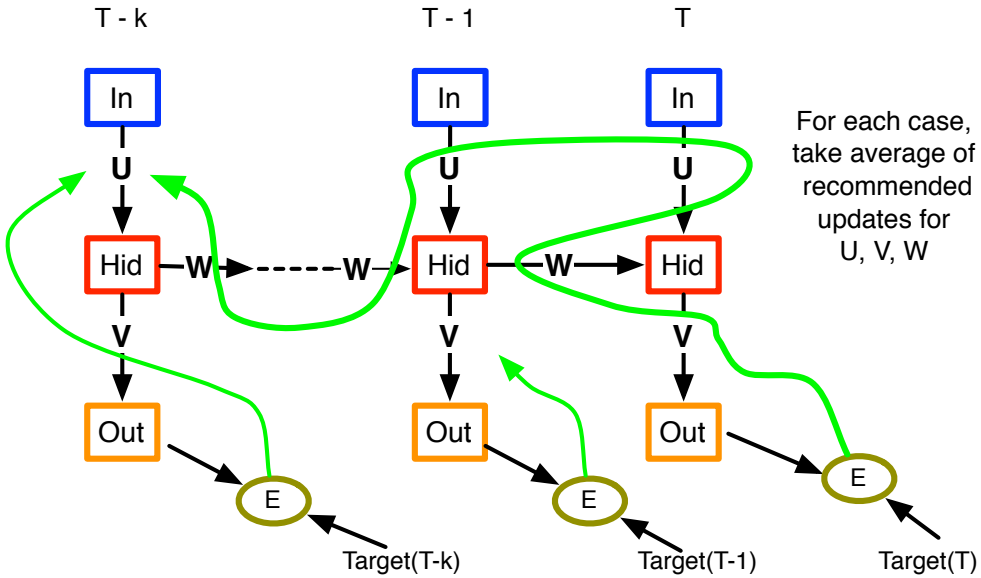


The Elman net should be more versatile, since its internal context is free to assume many activation patterns (i.e., representations), whereas the context of the Jordan net is restricted to that of the target patterns.

Backpropagation Through Time (BPTT)



Backpropagating Error

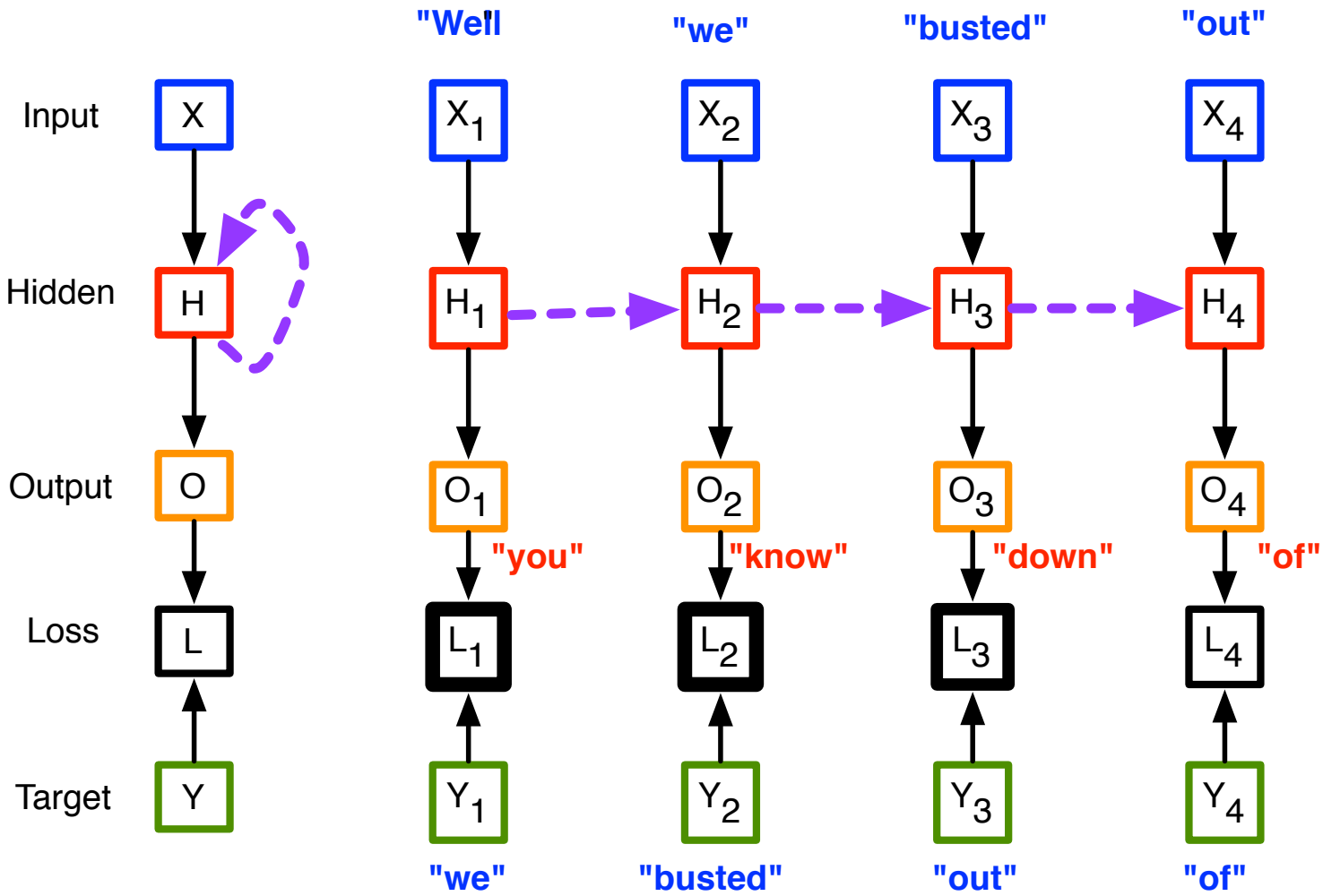


Applications of Recurrent NNs

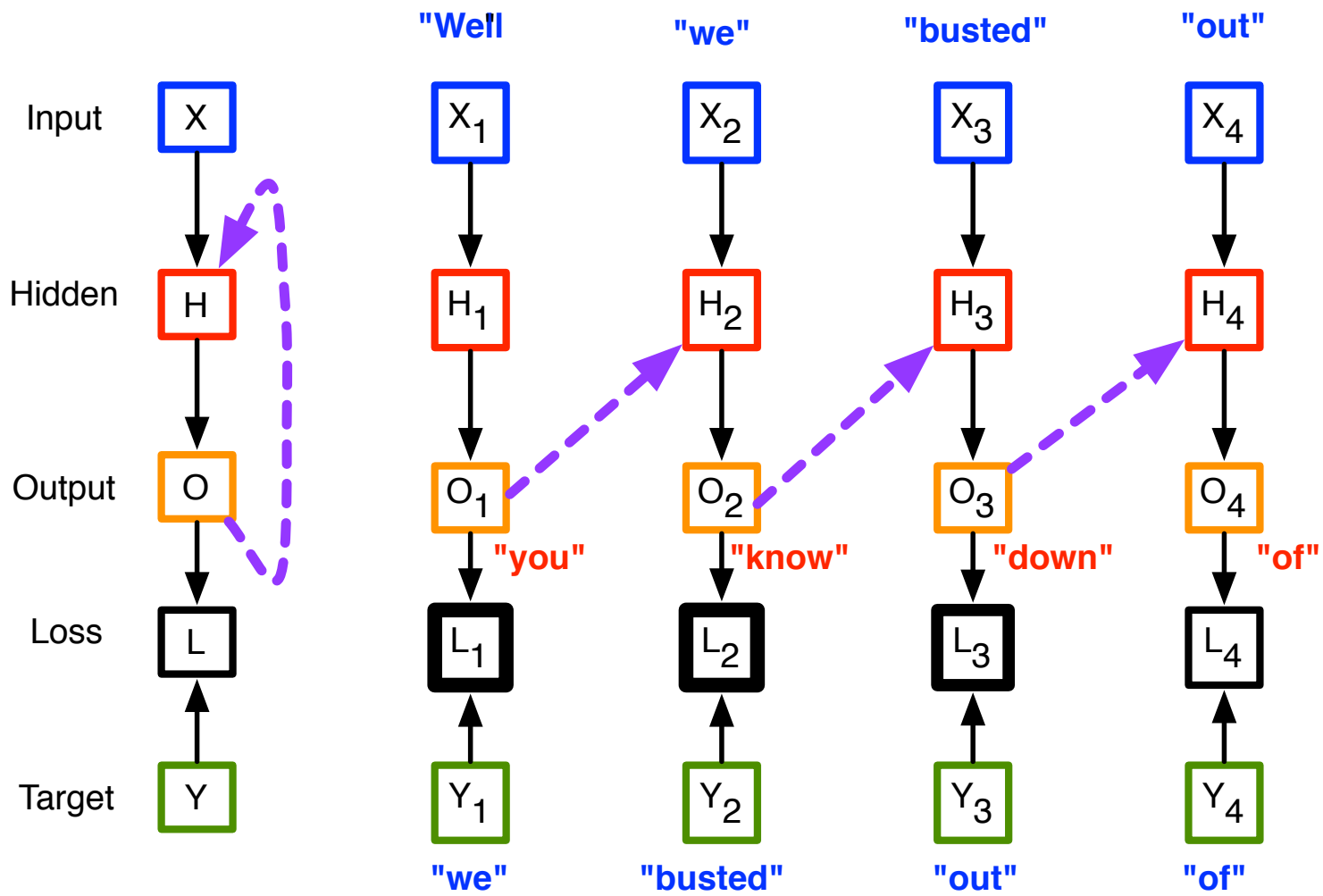
1. Language Modeling - Given a sequence of words, **predict** the next one.
2. Text Generation - Given a few words, produce a meaningful sequence.
3. Machine Translation - Given a sequence of words in one language, produce a semantically equivalent sequence in another language.
4. Speech Recognition - Given a sequence of acoustic signals, produce a sequence of phonemes.
5. Image Captioning - Given an image, produce a short text description. This typically requires RNNs + Convolution Nets.

- Most of the top RNN systems for natural language (NL) applications now use Long Short-Term Memory (LSTM; Hochreiter + Schmidhuber (1997))
- RNNs used for any NL task yield useful internal vector representations (a.k.a. embeddings) for NL expressions, wherein **semantically** similar expressions have similar vectors.

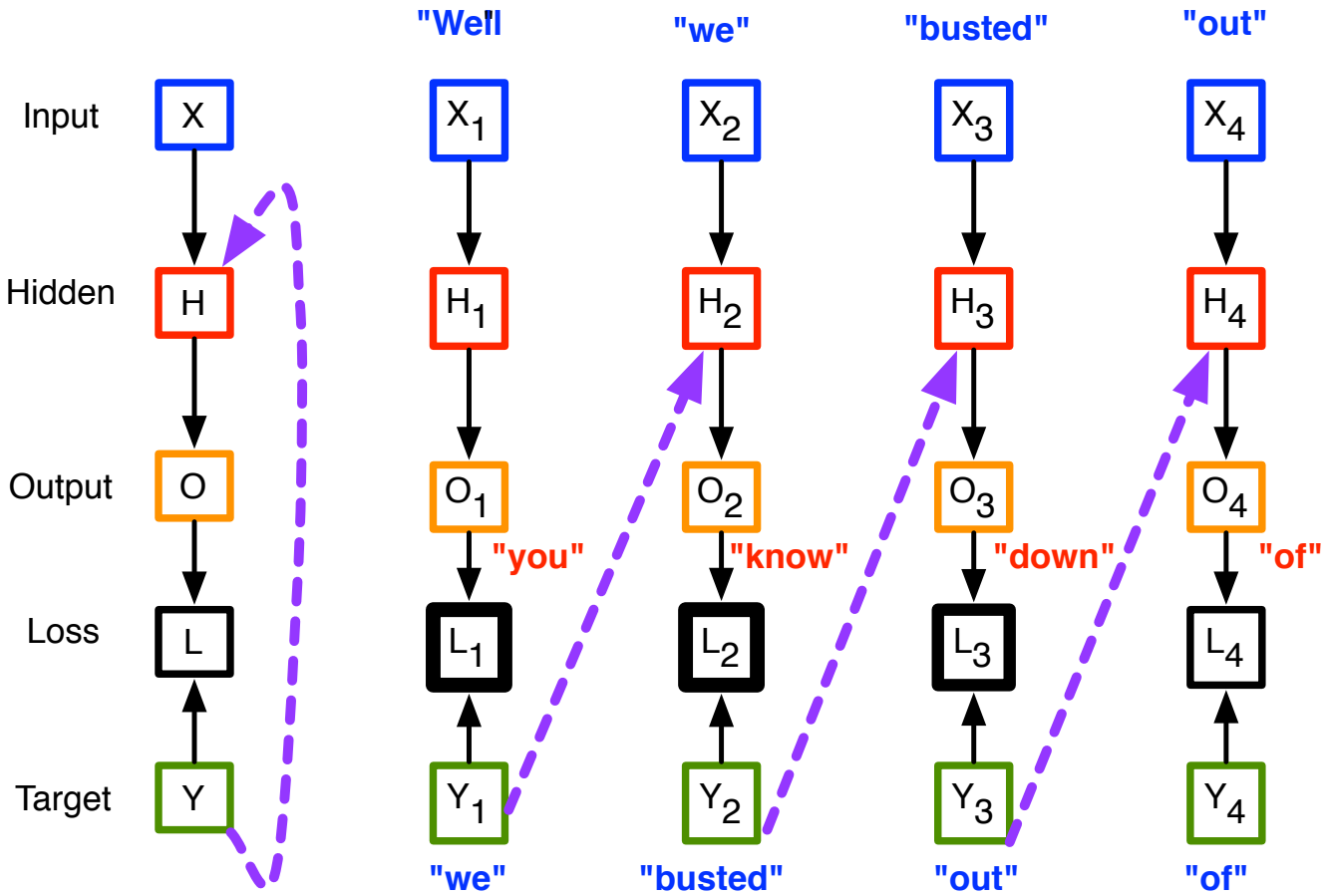
Recurrent Architectures: Elman Net



Recurrent Architectures: Jordan Net



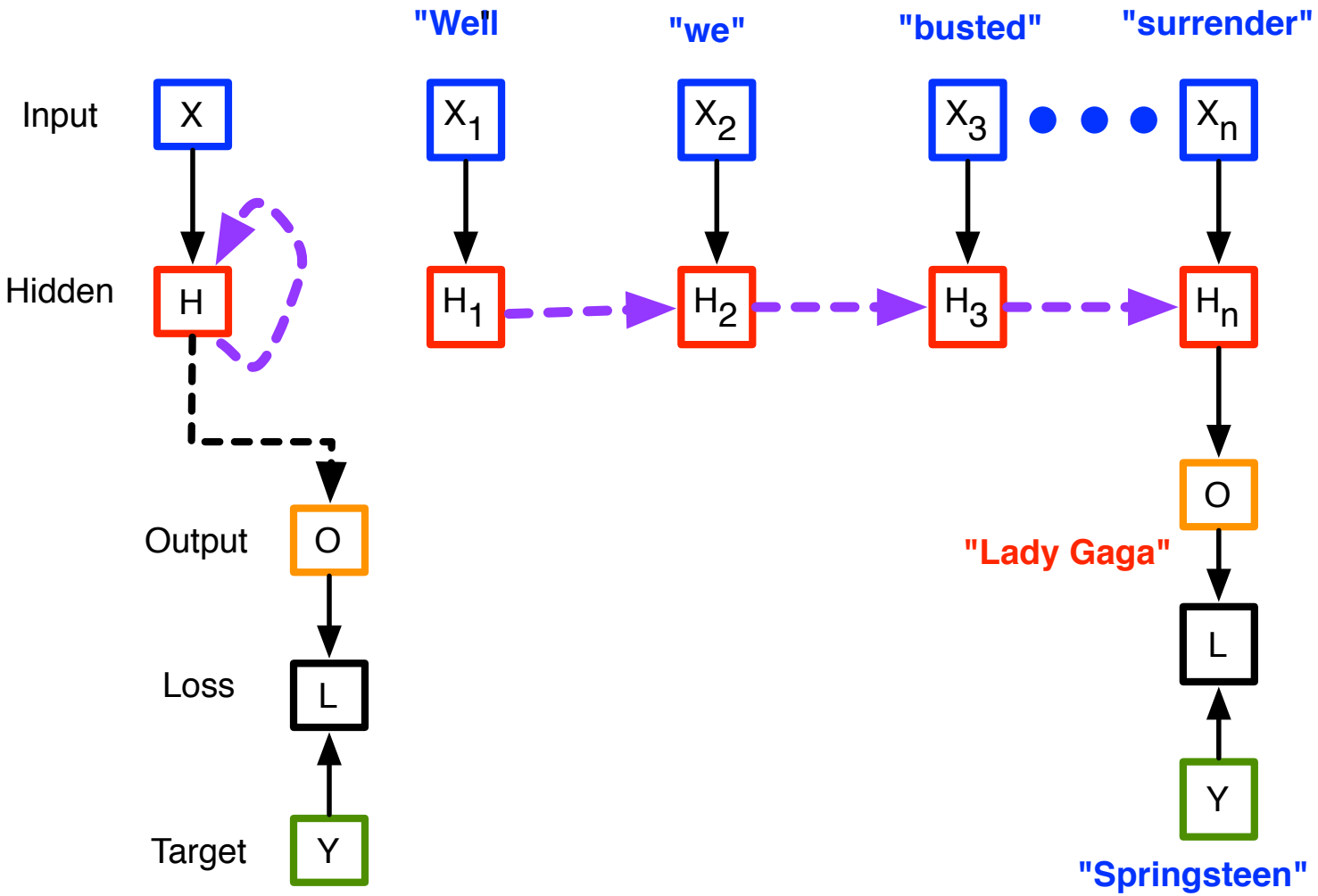
Recurrent Architectures: Teacher Forcing



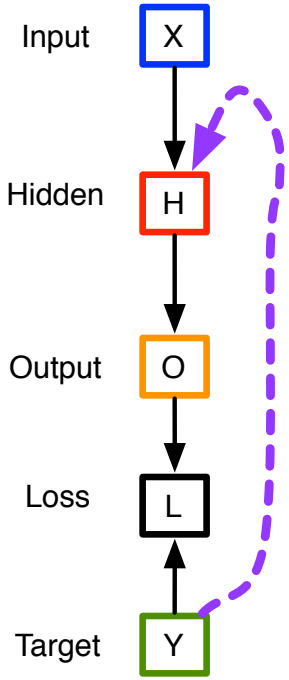
During testing, the output (not the target) feeds back to H.

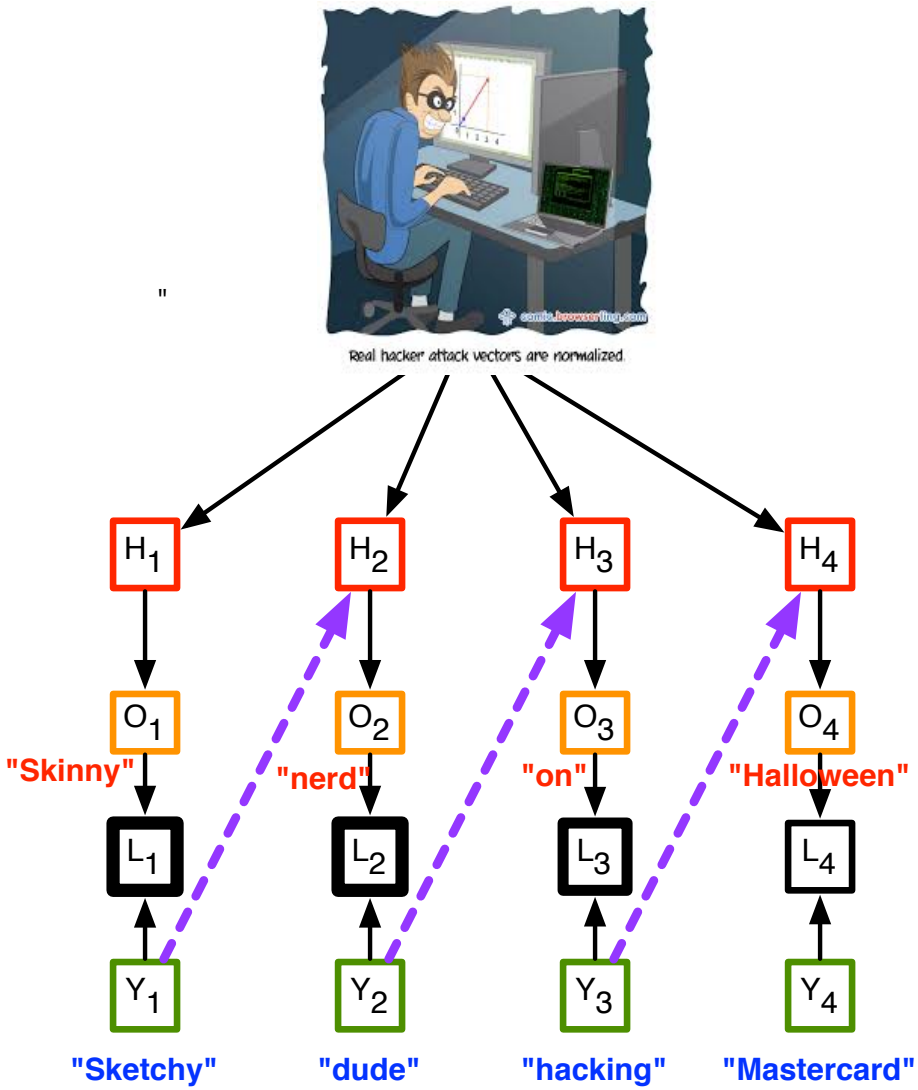
Since Y is independent of H, the gradients do not propagate backwards in time: the full power of BPTT is not needed for learning.

Recurrent Architectures: Delayed Supervision



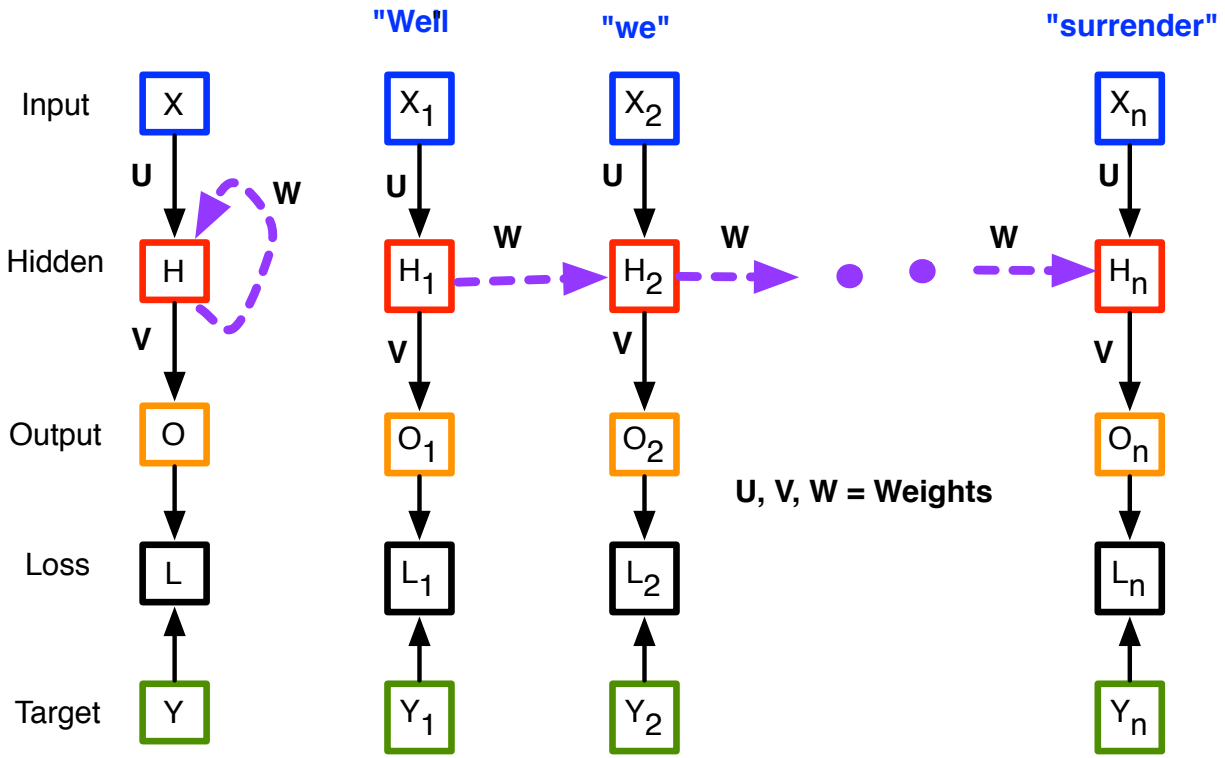
Recurrent Architectures: Captioning





Calculating RNN Gradients

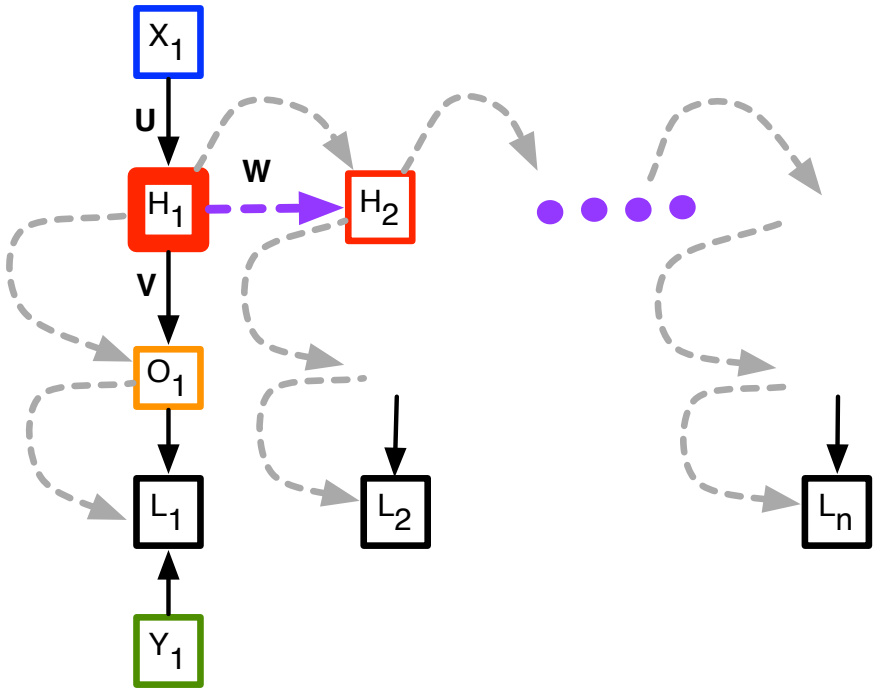
Standard calculations once the function graph is unrolled



$$\frac{\partial L}{\partial H_1} = \frac{\partial O_1}{\partial H_1} \bullet \frac{\partial L_1}{\partial O_1} + \frac{\partial H_2}{\partial H_1} \bullet \frac{\partial L}{\partial H_2}$$

Calculating RNN Gradients

The effects of H_i upon L involve all paths from H_i to L_i, L_{i+1}, $\ldots L_n$.



$$\frac{\partial L}{\partial H_1} = \frac{\partial O_1}{\partial H_1} \bullet \frac{\partial L_1}{\partial O_1} + \frac{\partial H_2}{\partial H_1} \bullet \frac{\partial L}{\partial H_2}$$

Jacobian involving the Hidden Layer

Derivatives of Loss w.r.t. Hidden Layer

Note: the hidden layer's activation levels change with each recurrent loop, so there is no single Jacobian, $\frac{\partial L}{\partial H}$. We must calculate $\frac{\partial L}{\partial H_i} \forall i$

$$\frac{\partial L}{\partial H_i} = \frac{\partial O_i}{\partial H_i} \bullet \frac{\partial L_i}{\partial O_i} + \frac{\partial H_{i+1}}{\partial H_i} \bullet \frac{\partial L}{\partial H_{i+1}}$$

This recurrent relationship can create a long product (in blue):

$$\frac{\partial L}{\partial H_i} = \cdots + \frac{\partial H_{i+1}}{\partial H_i} \bullet \frac{\partial H_{i+2}}{\partial H_{i+1}} \bullet \cdots \bullet \frac{\partial H_n}{\partial H_{n-1}} \bullet \frac{\partial L}{\partial H_n} + \cdots$$

Each term includes the weights (W):

$$\frac{\partial H_{k+1}}{\partial H_k} = \frac{\partial s_{k+1}}{\partial H_k} \times \frac{\partial H(s_{k+1})}{\partial s_{k+1}} = W^T \times \frac{\partial H(s_{k+1})}{\partial s_{k+1}}$$

where s_{k+1} = sum of weighted inputs to H at time k+1.

$$\frac{\partial H_{k+1}}{\partial H_k} = \begin{pmatrix} \frac{\partial h_{1,k+1}}{\partial h_{1,k}} & \frac{\partial h_{1,k+1}}{\partial h_{2,k}} & \cdots & \frac{\partial h_{1,k+1}}{\partial h_{m,k}} \\ \frac{\partial h_{2,k+1}}{\partial h_{1,k}} & \frac{\partial h_{2,k+1}}{\partial h_{2,k}} & \cdots & \frac{\partial h_{2,k+1}}{\partial h_{m,k}} \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial h_{m,k+1}}{\partial h_{1,k}} & \frac{\partial h_{m,k+1}}{\partial h_{2,k}} & \cdots & \frac{\partial h_{m,k+1}}{\partial h_{m,k}} \end{pmatrix}$$

Finishing the Calculation of $\partial H_{k+1}/\partial H_k$

- Assume H uses tanh as its activation function
- $\frac{\partial tanh(x)}{\partial x} = 1 - tanh(x)^2$.
- $s_{i,k+1}$ = sum of weighted inputs to hidden node h_i on the k+1 recursion cycle.
- $h_{i,k+1}$ = output of hidden node i on cycle k+1.

$$\frac{\partial H(s_{k+1})}{\partial s_{k+1}} = \begin{vmatrix} \frac{\partial h_{1,k+1}}{\partial s_{1,k+1}} \\ \frac{\partial h_{2,k+1}}{\partial s_{2,k+1}} \\ \vdots \end{vmatrix} = \begin{vmatrix} 1 - h_{1,k+1}^2 \\ 1 - h_{2,k+1}^2 \\ \vdots \end{vmatrix}$$

- For each node in H (h_i), there is a) one row in $\frac{\partial H(s_{k+1})}{\partial s_{k+1}}$ and b) one row in W^T (for all incoming weights to h_i).
- So multiply each weight in the ith row of W^T by the ith member of the column vector to produce $\frac{\partial H_{k+1}}{\partial H_k}$.

Finishing the Calculation of $\partial H_{k+1}/\partial H_k$

$$\frac{\partial H_{k+1}}{\partial H_k} = \begin{pmatrix} (1-h_{1,k+1}^2)w_{11} & (1-h_{1,k+1}^2)w_{21} & \cdots & (1-h_{1,k+1}^2)w_{m1} \\ (1-h_{2,k+1}^2)w_{12} & (1-h_{2,k+1}^2)w_{22} & \cdots & (1-h_{2,k+1}^2)w_{m2} \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

- Since row-by-row multiplication is not a standard array operation, the column vector can be converted into a diagonal matrix (with the only non-zero values along the main diagonal) denoted $diag(1-H^2)$. Then we can use standard matrix multiplication ($\bullet$):

$$\frac{\partial H_{k+1}}{\partial H_k} = diag(1-H^2) \bullet W^T =$$

$$\begin{pmatrix} (1-h_{1,k+1}^2) & 0 & \cdots & 0 \\ 0 & (1-h_{2,k+1}^2) & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix} \bullet \begin{pmatrix} w_{11} & w_{21} & w_{31} & \cdots \\ w_{12} & w_{22} & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

Jacobians of Loss w.r.t. Weights

Notice the sums across all recurrent cycles (t).

$$\frac{\partial L}{\partial V} = \sum_{t=1}^{n} \frac{\partial O_t}{\partial V} \bullet \frac{\partial L_t}{\partial O_t}$$

- where $\frac{\partial O_t}{\partial V}$ is a 3-d array containing the values of H_t; the details of $\frac{\partial L_t}{\partial O_t}$ depend upon the actual loss function and activation function of O.

$$\frac{\partial L}{\partial U} = \sum_{t=1}^{n} \frac{\partial H_t}{\partial U} \bullet \frac{\partial L}{\partial H_t}$$

- where $\frac{\partial H_t}{\partial U}$ is a 3-d array containing the values of X_t.

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{n-1} \frac{\partial H_{t+1}}{\partial W} \bullet \frac{\partial L}{\partial H_{t+1}}$$

- where $\frac{\partial H_{t+1}}{\partial W}$ is a 3-d array containing the values of H_t.

- These gradient products include the activations of X and H. When these approach 0, the gradients vanish.
- When the sigmoid or tanh is near saturation (0 or 1 for sigmoid; -1 or 1 for tanh), the diagonal matrix in $\frac{\partial H_{k+1}}{\partial H_k}$ approaches zero; and the gradients vanish again.

Handling Uncooperative Gradients

- Recurrence creates long product gradients which often vanish (and occasionally explode).
- How can we combat this?

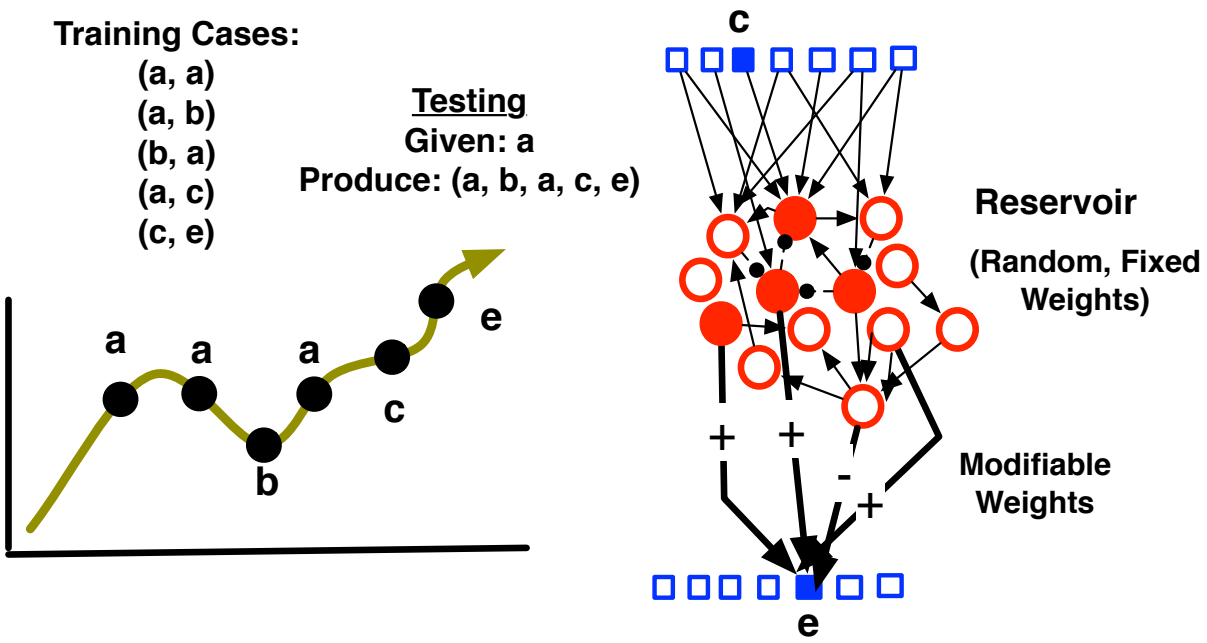
Innovative Network Architectures

- Reservoir Computing: Echo-State Machines and Liquid-State Machines
- Gated RNNs: Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU)
- Explicit Memory - Neural Turing Machine

Supplementary Techniques

- Multiple Timescales - using different degrees of leak/forgetting and skip connections.
- Gradient Clipping - setting an upper bound on gradient magnitude.
- Regularizing - adding penalty for gradient decay to the loss function.

Reservoir Computing



- Echo State Machines (ESM) and Liquid State Machines (LSM)
- MANY reservoir neurons with random and recurrent connections allow sequences of inputs to create complex internal states (a.k.a. **contexts**).
- Only tuneable synapses: reservoir to outputs $\rightarrow$ no long gradients.
- Drawback: initializing reservoir weights is tricky.

Even RNNs Forget

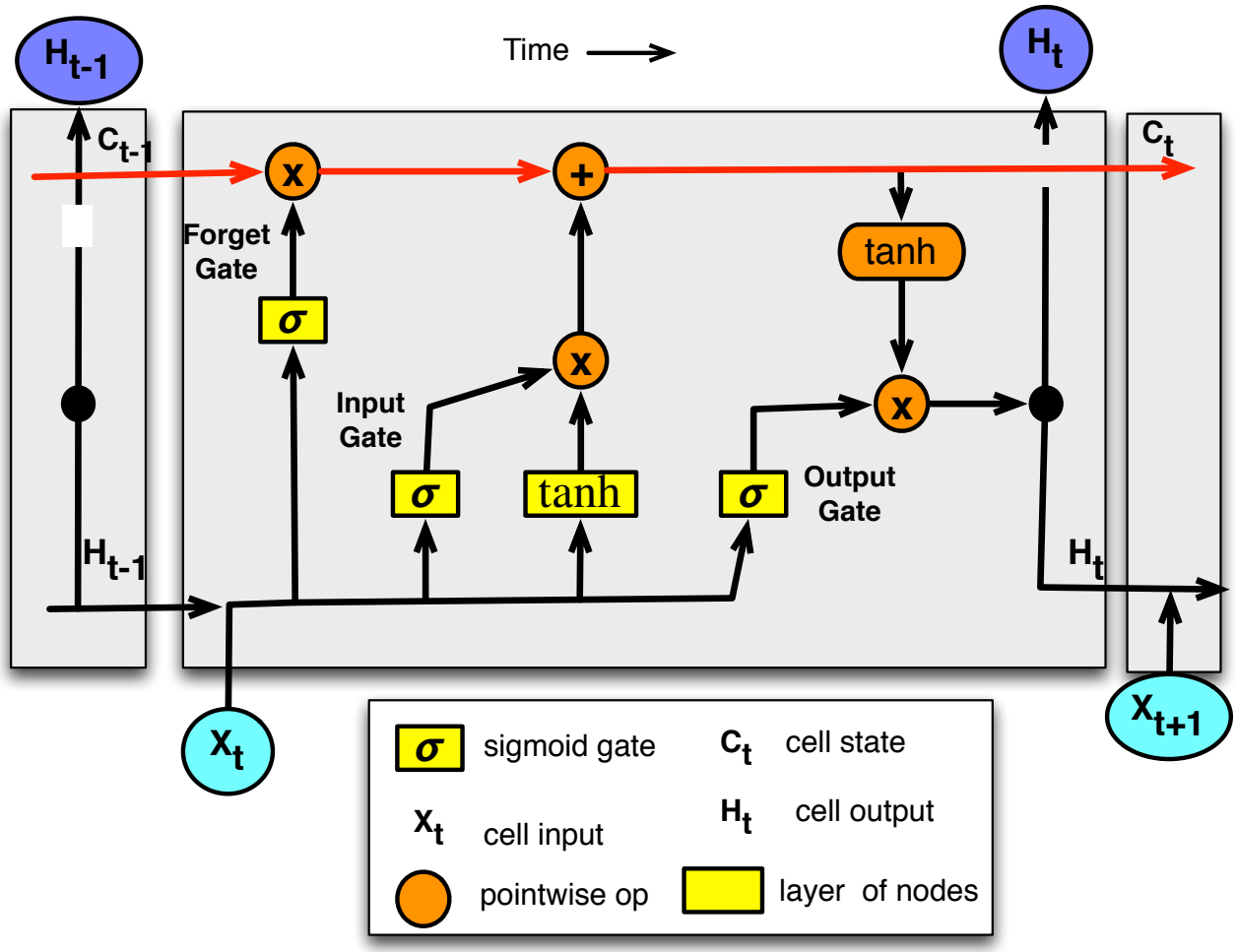
- Repeated multiplication by hidden-to-hidden weights, along with the integration of new inputs, causes memories (i.e. hidden-node states) to fade over time.
- For some tasks, e.g. reinforcement learning, this is desirable: older information should have a diminishing effect upon current decision-making.
- In other tasks, such as natural-language processing, some key pieces of *old* information need to retain influence.

  *I found my **bicycle** in the same town and on the same street, but not on the same front porch from which I had reported to the police that **it** had been stolen.*
- What was stolen?
- A clear memory of *bicycle* must be maintained throughout processing of the entire sentence (or paragraph, or story).

Long Short-Term Memory (LSTM)

- Invented by Hochreiter and Schmidhuber in 1997.
- Breakthrough in 2009: Won Int'l Conf. on Doc. Analysis and Recognition (ICDAR) competition.
- 2013: Record for speech recognition on DARPA's TIMIT dataset.
- Currently used in commercial speech-recognition systems by major players: Google, Apple, Microsoft...
- Critical basis of RNN success, which is otherwise very limited.
- Hundreds, if not thousands, of LSTM variations. One of the most popular is the GRU (Gated Recurrent Unit), which has fewer gates but still good performance. Also, Phased LSTM's handle asynchronous input.
- Easily combined with other NN components, e.g., convolution.
- Increased memory retention $\rightarrow$ hidden layers continue to produce (some) high values even when external inputs are low $\rightarrow$ gradients do not vanish.
- See Understanding LSTM Networks. Colah's Blog (August, 2015).

The LSTM Module



Long Short-Term Memory (LSTM)

- Yellow boxes are full layers of nodes that are preceded by a matrix of learnable weights and that produce a vector of values.
- Orange ovals have unweighted vectors as inputs. They perform their operation (+ or $\times$) pointwise: either individually on each item of a vector, or pairwise with corresponding elements of two vectors (e.g. the state vector and the forget vector). A.k.a. Hadamard Product.
- The cell's internal state, C (a vector), runs along the top (red line) of the diagram.
- C is selectively modified, as controlled by the forget and input gates,
- The yellow tanh produces *suggested* contributions to C_{t-1}.
- The modified state (C_t) is preserved (along the red line) and passed to the next time step.
- C_t also passes through the (orange) tanh, whose result is also gated to produce the cell's output value, H_t.
- H_t along with the next input X_{t+1} comprise the complete set of weighted inputs to the 3 gates and (yellow) tanh at time t+1.

Supplementary Techniques for Gradient Management

These are minor adjustments, applicable to many NN architectures.

Gradient Clipping

- Oversized gradients $\rightarrow$ Large jumps in search space $\rightarrow$ risky in rugged error landscapes.
- Reduce problem by restricting gradients to a pre-defined range.

Gradient-Sensitive Regularization

- Develop a metric for amount of gradient decay across recurrent steps.
- Include this value as a penalty in the loss function.

Facilitate Multiple Time Scales

- Skip connections over time - implement delays by linking output of a layer at time t **directly** to input at time t+d.
- Variable leak rates - neurons have diverse leak parameters. More leak $\rightarrow$ more forgetting $\rightarrow$ shorter timescale of information integration.