Recurrent Neural Networks

Roberto Basili, Danilo Croce

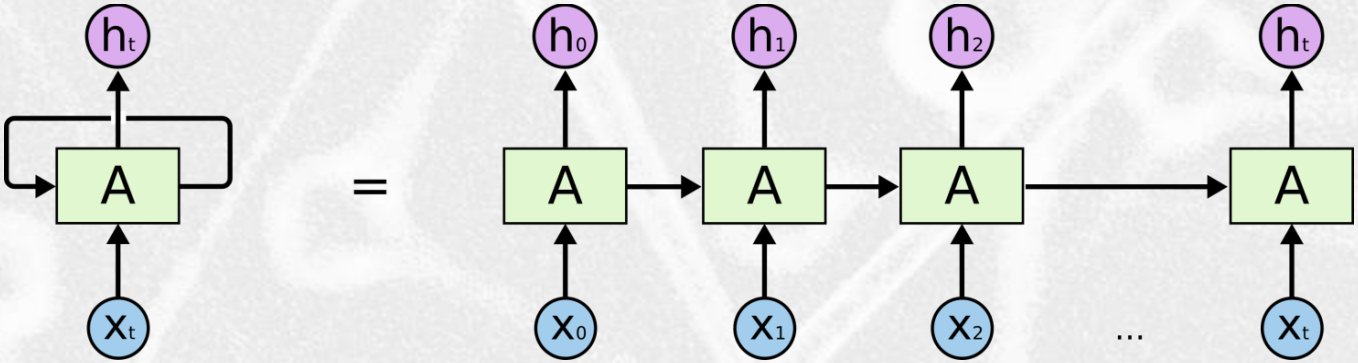
Machine Learning, Deep Learning 2023/2024

Outline

- Recurrent and Recursive Networks
- Training Recurrent Networks
- Long Short Term Memory (LSTM) networks
- LSTMS: Applications to Language Processing
- Perspectives

Recurrent Neural Networks

- Used mainly to model sequences
  - naturally applied to textual and speech problems
- A representation at time step *i* is made dependent on the representations of the preceding steps (*i-1, i-2, ...*)
  - connections between units form a directed cycle



Recurrent Neural Networks

- Commons tasks are
  - ***language models***: predict the next word in a sentence given the already seen word
  - ***speech recognition***: predict a word given the current wave form and the preceding words
  - ***machine translation***: produce a sequence in a target language given an input sequence in a source language
- The most famous and effective model of RNNs are the Long-Short Term Memory (LSTM) Networks (Sepp Hochreiter and Jürgen Schmidhuber, 1997)
  - they are meant to better deal with long-range dependencies

Neural Networks for Natural Language Processing

- Linguistic features have been highly enriched since NN language models have been introduced
  - Words, *n*-grams as well as sentences, paragraphs have been modeled through efficient and highly robust neural learners
- Representation are usually *dense embeddings*
- Making explicit Use of the contexts: *Recurrent Networks*
- Tasks have been extended beyond Classification:
  - Transducing, Ranking, Encoding, Decoding
  - Generation is a form of transduction and can be adapted to conversation

Recurrent Neural Networks

- For example, consider the classifcal form of a dynamical system

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}; \boldsymbol{\theta}),$$

- Its corresponding unfolded computational graph is as follows

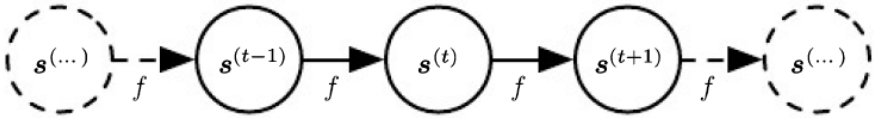
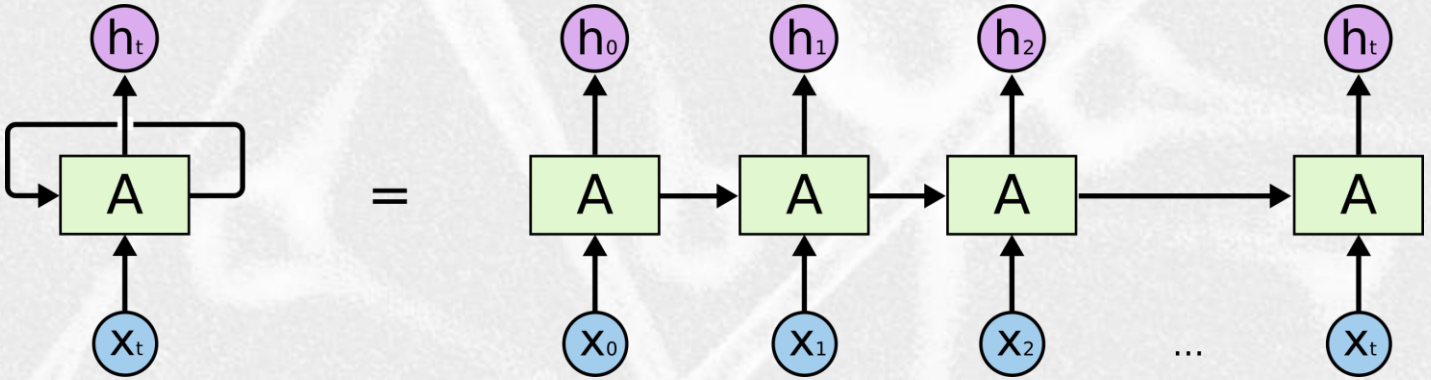


Figure 10.1: The classical dynamical system described by equation 10.1, illustrated as an unfolded computational graph. Each node represents the state at some time t, and the function f maps the state at t to the state at $t+1$. The same parameters (the same value of $\boldsymbol{\theta}$ used to parametrize f) are used for all time steps.



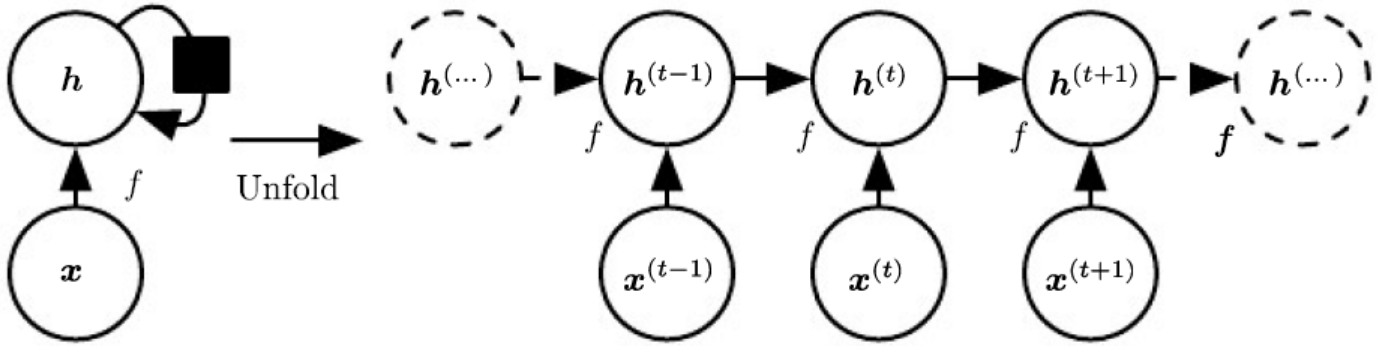


Figure 10.2: A recurrent network with no outputs. This recurrent network just processes information from the input $\boldsymbol{x}$ by incorporating it into the state $\boldsymbol{h}$ that is passed forward through time. *(Left)* Circuit diagram. The black square indicates a delay of a single time step. *(Right)* The same network seen as an unfolded computational graph, where each node is now associated with one particular time instance.

Many recurrent neural networks use equation 10.5 or a similar equation to define the values of their hidden units. To indicate that the state is the hidden units of the network, we now rewrite equation 10.4 using the variable $\boldsymbol{h}$ to represent the state,

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}), \tag{10.5}$$

illustrated in figure 10.2; typical RNNs will add extra architectural features such as output layers that read information out of the state $\boldsymbol{h}$ to make predictions.

We can represent the unfolded recurrence after t steps with a function $g^{(t)}$:

$$\boldsymbol{h}^{(t)} = g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}) \tag{10.6}$$

$$= f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}). \tag{10.7}$$

The function $g^{(t)}$ takes the whole past sequence $(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)})$ as input and produces the current state, but the unfolded recurrent structure allows us to factorize $g^{(t)}$ into repeated application of a function f.

Using a RNN

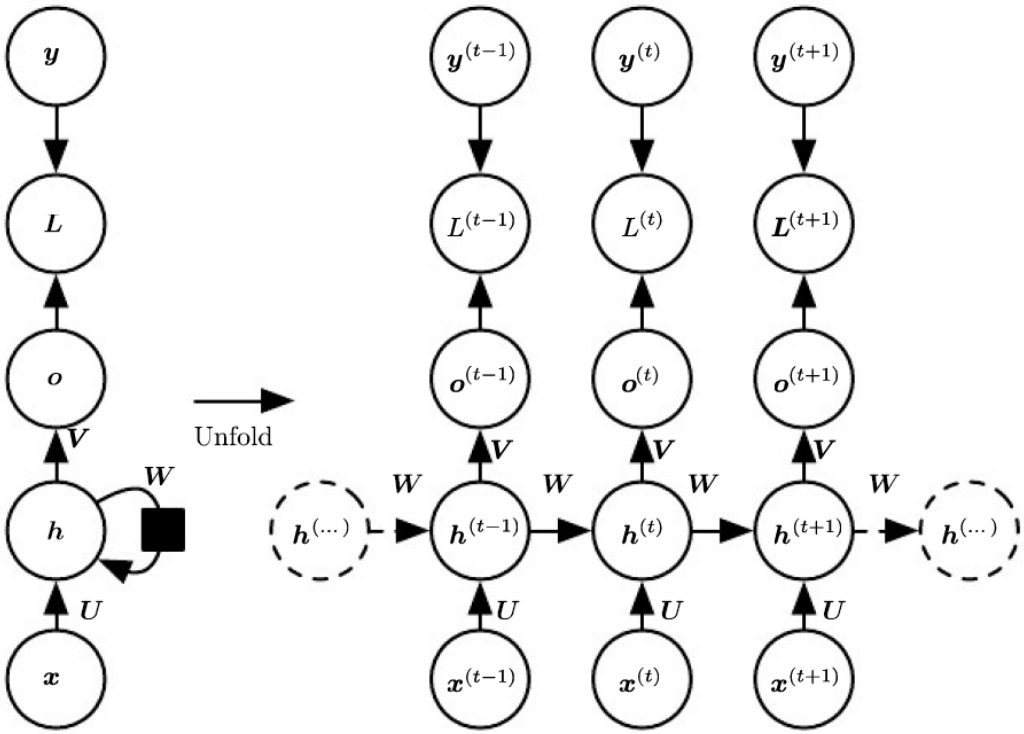
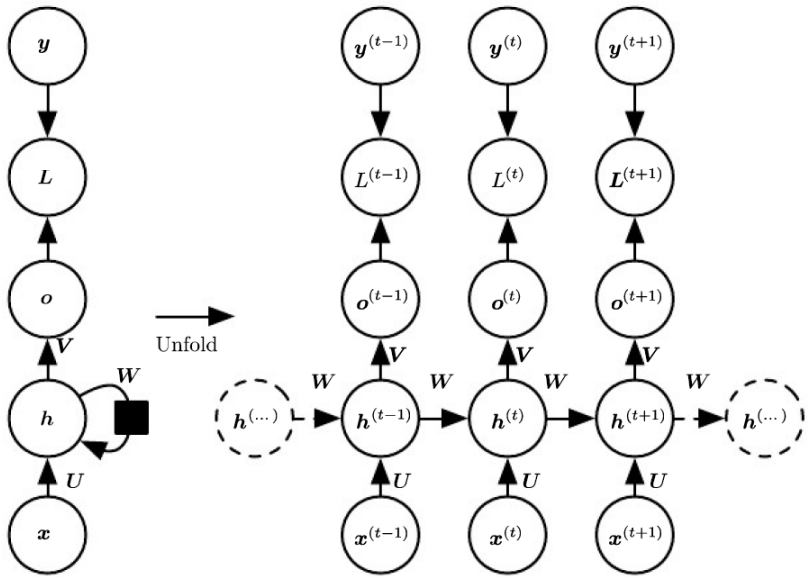


Figure 10.3: The computational graph to compute the training loss of a recurrent network that maps an input sequence of $\boldsymbol{x}$ values to a corresponding sequence of output $\boldsymbol{o}$ values. A loss L measures how far each $\boldsymbol{o}$ is from the corresponding training target $\boldsymbol{y}$. When using softmax outputs, we assume $\boldsymbol{o}$ is the unnormalized log probabilities. The loss L internally computes $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{o})$ and compares this to the target $\boldsymbol{y}$. The RNN has input to hidden connections parametrized by a weight matrix $\boldsymbol{U}$, hidden-to-hidden recurrent connections parametrized by a weight matrix $\boldsymbol{W}$, and hidden-to-output connections parametrized by a weight matrix $\boldsymbol{V}$. Equation 10.8 defines forward propagation in this model. *(Left)* The RNN and its loss drawn with recurrent connections. *(Right)* The same seen as a time-unfolded computational graph, where each node is now associated with one particular time instance.

Using a RNN (2)



Forward propagation begins with a specification of the initial state $\boldsymbol{h}^{(0)}$. Then, for each time step from $t = 1$ to $t = \tau$, we apply the following update equations:

$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}, \quad (10.8)$$

$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{a}^{(t)}), \quad (10.9)$$

$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)}, \quad (10.10)$$

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}(\boldsymbol{o}^{(t)}), \quad (10.11)$$

where the parameters are the bias vectors $\boldsymbol{b}$ and $\boldsymbol{c}$ along with the weight matrices $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{W}$, respectively, for input-to-hidden, hidden-to-output and hidden-to-hidden connections.

Simple RNN

11.1 Simple RNN

The simplest RNN formulation, known as an Elman Network or Simple-RNN (S-RNN), was proposed by Elman (1990) and explored for use in language modeling by Mikolov (2012). The S-RNN takes the following form:

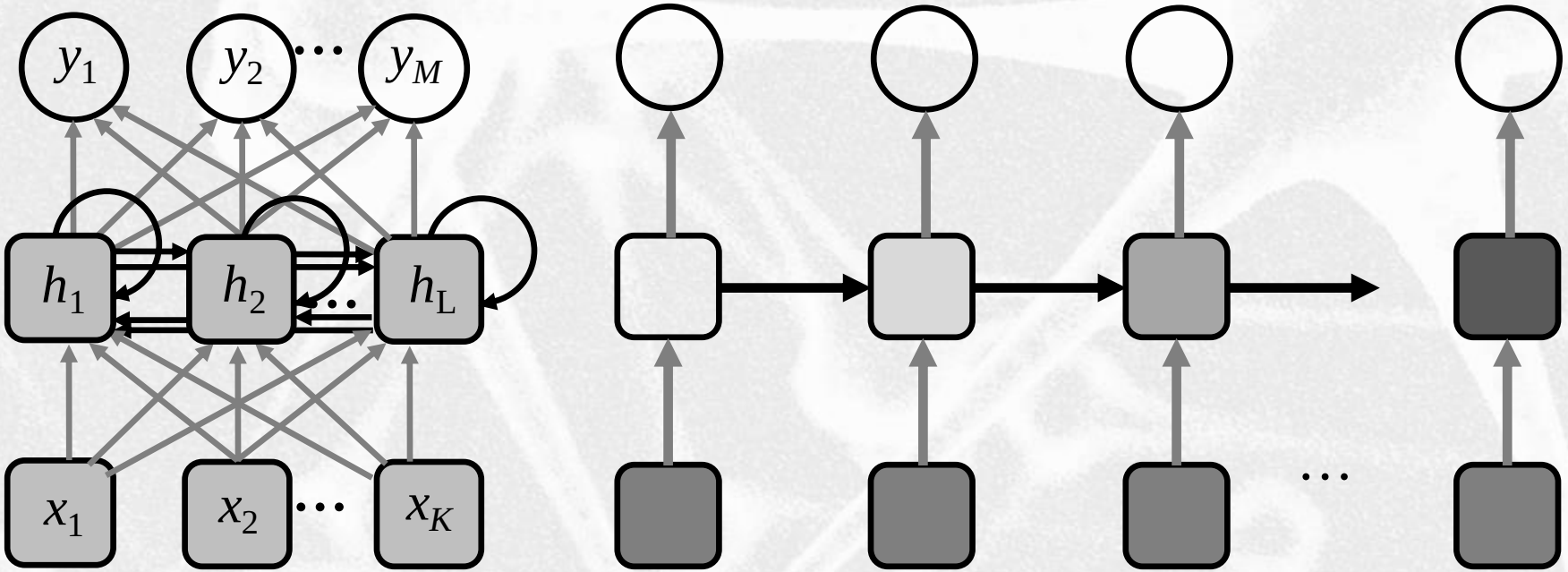
$$\begin{aligned} \mathbf{s_i} &= R_{\text{SRNN}}(\mathbf{s_{i-1}}, \mathbf{x_i}) = g(\mathbf{x_i}\mathbf{W^x} + \mathbf{s_{i-1}}\mathbf{W^s} + \mathbf{b}) \\ \mathbf{y_i} &= O_{\text{SRNN}}(\mathbf{s_i}) = \mathbf{s_i} \end{aligned} \tag{38}$$

$$\mathbf{s_i}, \mathbf{y_i} \in \mathbb{R}^{d_s}, \ \mathbf{x_i} \in \mathbb{R}^{d_x}, \ \mathbf{W^x} \in \mathbb{R}^{d_x \times d_s}, \ \mathbf{W^s} \in \mathbb{R}^{d_s \times d_s}, \ \mathbf{b} \in \mathbb{R}^{d_s}$$

That is, the state at position i is a linear combination of the input at position i and the previous state, passed through a non-linear activation (commonly tanh or ReLU). The output at position i is the same as the hidden state in that position.[71]

Recurrent neural networks (RNNs)

- An RNN can be unwrapped and implemented using the same weights and biases at each step to link units over time as shown below
- The resulting unwrapped RNN is similar to a hidden Markov model, but keep in mind that the hidden units in RNNs are not stochastic



Recurrent neural networks (RNNs)

- Recurrent neural networks apply linear matrix operations to the current observation and the hidden units from the previous time step, and the resulting linear terms serve as arguments of activation functions act():

$$\mathbf{h}_t = \mathrm{g}(\mathbf{W}_h \mathbf{x}_t + \mathbf{U}_h \mathbf{h}_{t-1} + \mathbf{b}_h)$$

$$\mathbf{o}_t = \mathrm{f}(\mathbf{W}_o \mathbf{h}_t + \mathbf{b}_o)$$

- The same matrix $\mathbf{U}_h$ is used at each time step
- The hidden units in the previous step $\mathbf{h}_{t-1}$ influence the computation of $\mathbf{h}_t$ where the current observation $\mathbf{x}_t$ contributes to a $\mathbf{W}_h\mathbf{x}_t$ term that is combined with $\mathbf{U}_h\mathbf{h}_{t-1}$ and bias $\mathbf{b}_h$ terms
- Both $\mathbf{W}_h$ and $\mathbf{b}_h$ are typically replicated over time
- The output layer is modeled by a classical neural network activation function applied to a linear transformation of the hidden units, the operation is replicated at each step.

BPPTT

- For training a recurrent network, a solution is to unfold the recurrent structure and expand it as a feedforward neural network with a certain number of time steps: then apply traditional backpropagation onto this unfolded neural network.
- This solution is known as **Backpropagation through Time** (BPTT), independently invented by several researchers including (Robinson and Fallside, 1987; Werbos, 1988; Mozer, 1989)

The loss, exploding and vanishing gradients

- The loss for a particular sequence in the training data can be computed either at each time step or just once, at the end of the sequence.
- In either case, predictions will be made after many processing steps and this brings us to an important problem.
- The gradient for feedforward networks decomposes the gradient of parameters at layer *l* into a term that involves the product of matrix multiplications of the form $\partial J^{(l)}\mathbf{W}^{\mathrm{T}(l+1)}$ (remind the backpropagation in feedforward network)
- A recurrent network uses the same matrix at each time step, and over many steps the gradient can very easily either diminish to zero or explode to infinity—just as the magnitude of any number other than one taken to a large power either approaches zero or increases indefinitely

BPTT: the algorithm

1. Present a sequence of ***k1*** timesteps of input and output pairs to the network.
2. Unroll the network then calculate and accumulate errors across ***k2*** timesteps.
3. Roll-up the network and update weights.
4. Repeat

- The TBPTT algorithm requires the consideration of two parameters:
  - **k1**: The *number of forward-pass timesteps between updates*.
    - this influences how slow or fast training will be, given how often weight updates are performed.
  - **k2**: The *number of timesteps to which to apply BPTT*.
    - it should be large enough to capture the temporal structure in the problem for the network to learn.
    - Too large a value results in vanishing gradients

BPTT

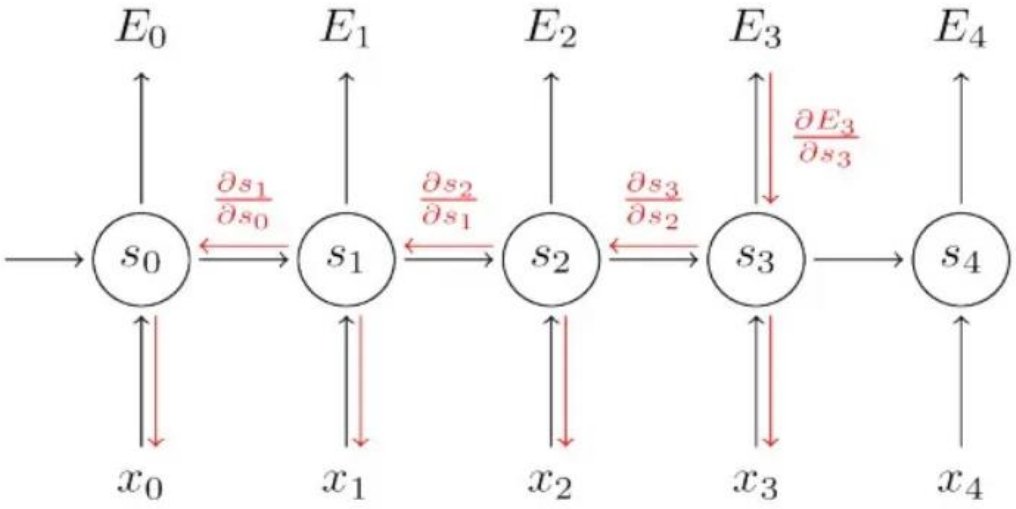
$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$$

$$\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial W}$$

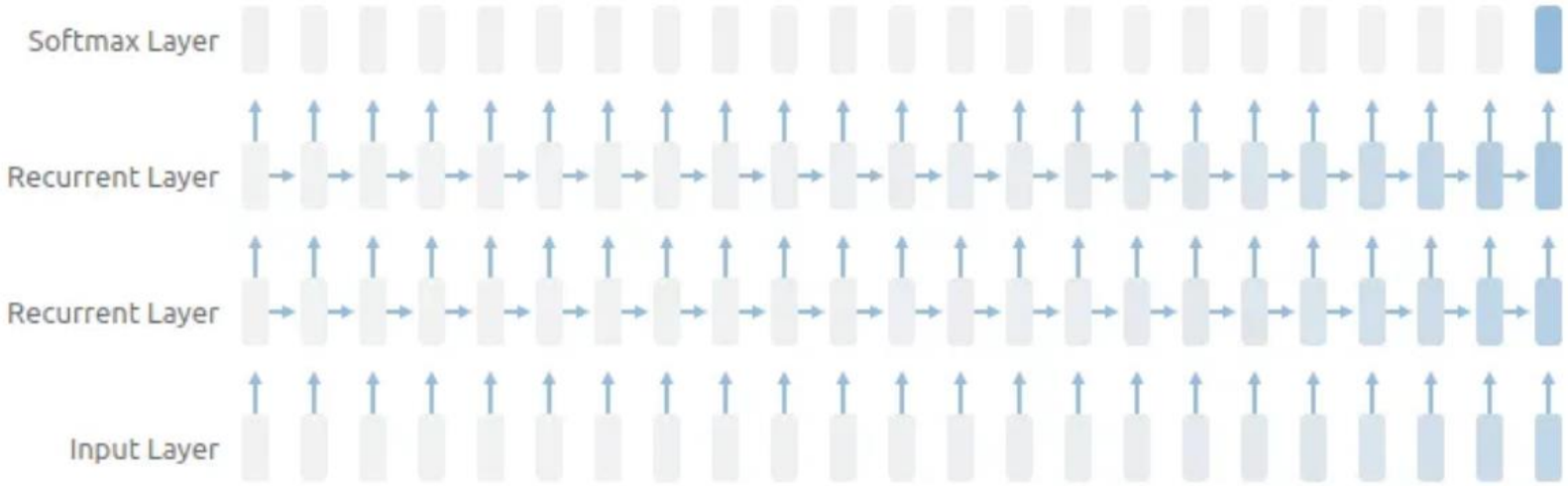
But $s_3 = \tanh(Ux_t + Ws_2)$

S_3 depends on s_2, which depends on W and s_1, and so on.

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial s_k}\frac{\partial s_k}{\partial W}$$



Vanishing Gradients



Vanishing Gradient: where the contribution from the earlier steps becomes insignificant in the gradient for the vanilla RNN unit.

Dealing with exploding gradients

- The use of L1 or L2 regularization can mitigate the problem of exploding gradients by encouraging weights to be small.
- Another strategy is to simply detect if the norm of the gradient exceeds some threshold, and if so, scale it down.
- This is sometimes called ***gradient (norm) clipping*** where for a gradient vector *g* and threshold *T*,

$$\text{if } \|\mathbf{g}\| \geq T \text{ then } \mathbf{g} \leftarrow \frac{T}{\|\mathbf{g}\|}\mathbf{g}$$

  - where T is a hyperparameter, which can be set to the average norm over several previous updates where clipping was not used.

Teacher Forcing

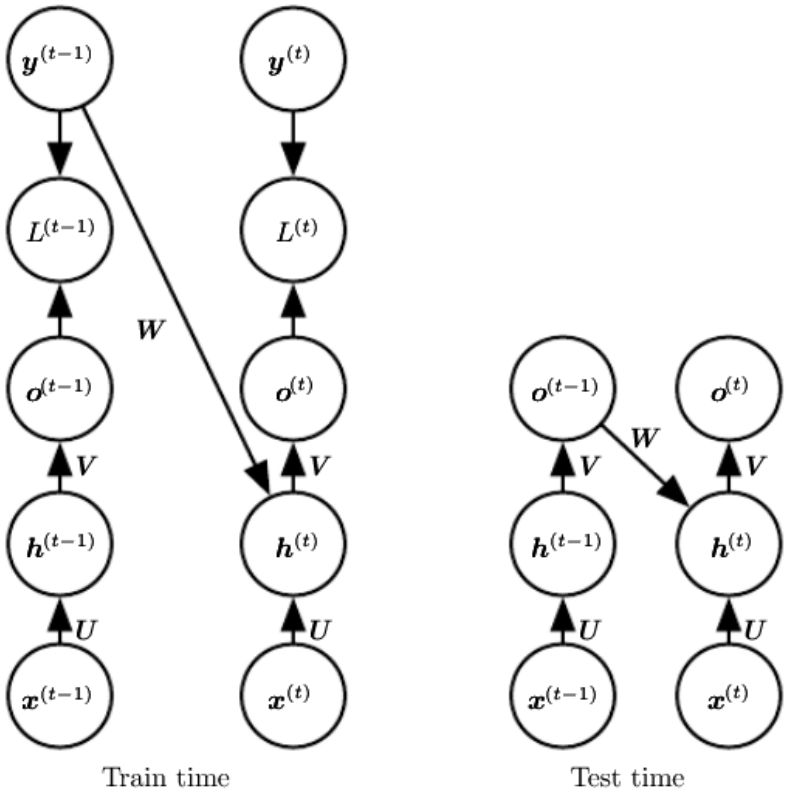
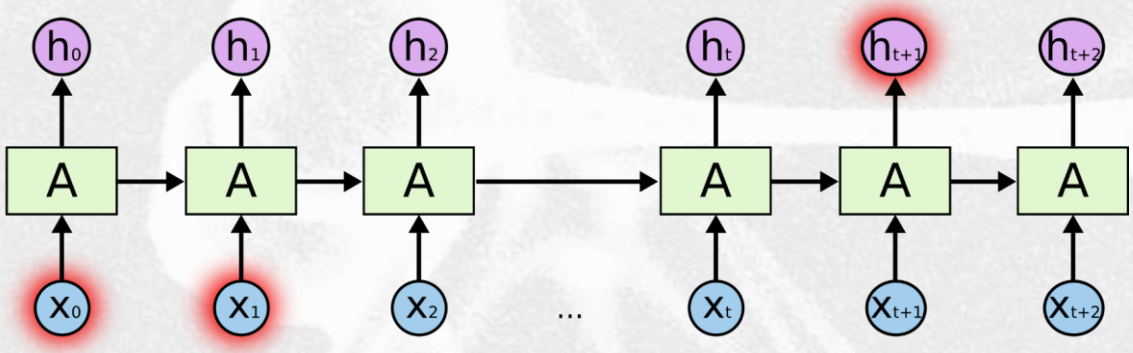
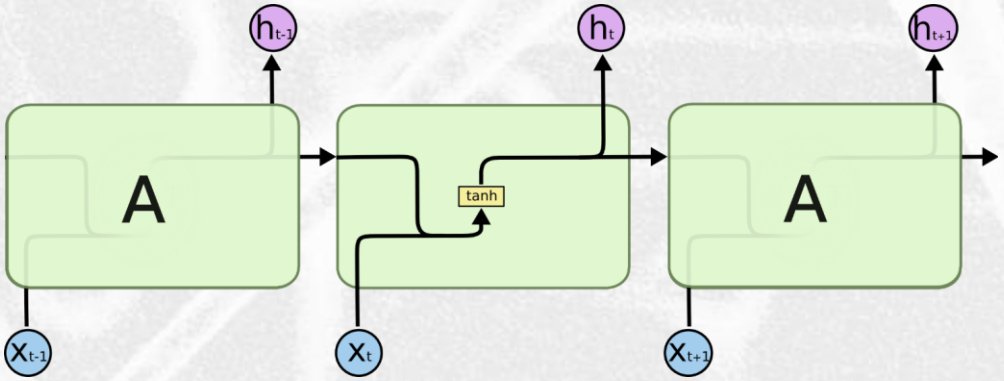


Figure 10.6: Illustration of teacher forcing. Teacher forcing is a training technique that is applicable to RNNs that have connections from their output to their hidden states at the next time step. *(Left)* At train time, we feed the *correct output* $\boldsymbol{y}^{(t)}$ drawn from the train set as input to $\boldsymbol{h}^{(t+1)}$. *(Right)* When the model is deployed, the true output is generally not known. In this case, we approximate the correct output $\boldsymbol{y}^{(t)}$ with the model's output $\boldsymbol{o}^{(t)}$, and feed the output back into the model.

Long-term Dependencies with one single layer





Dealing with Long-term Dependencies

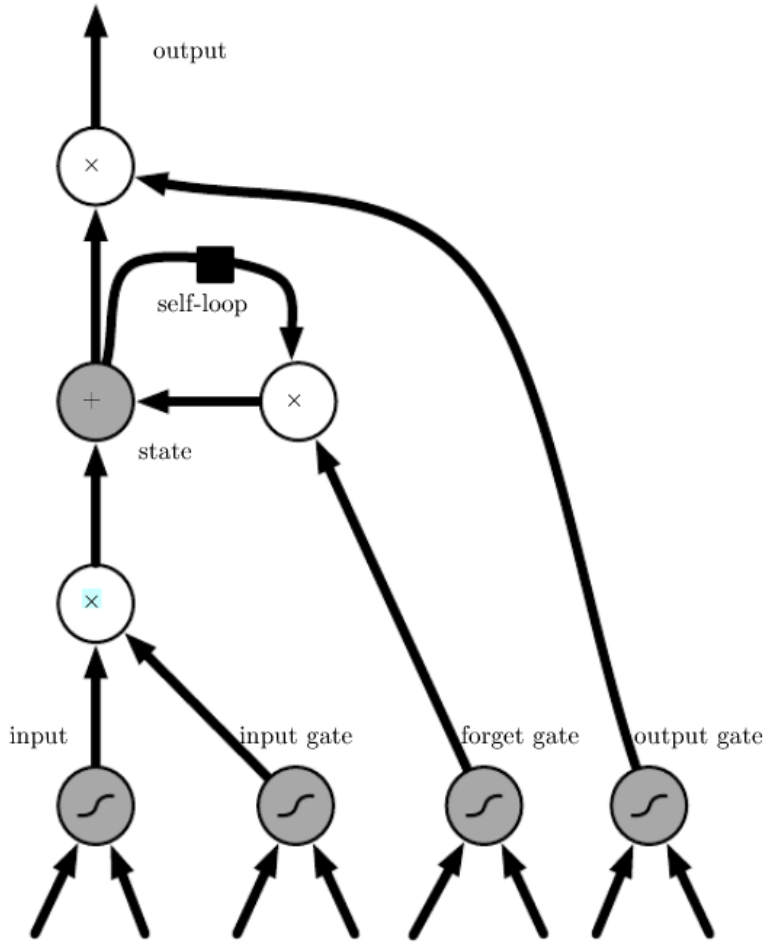
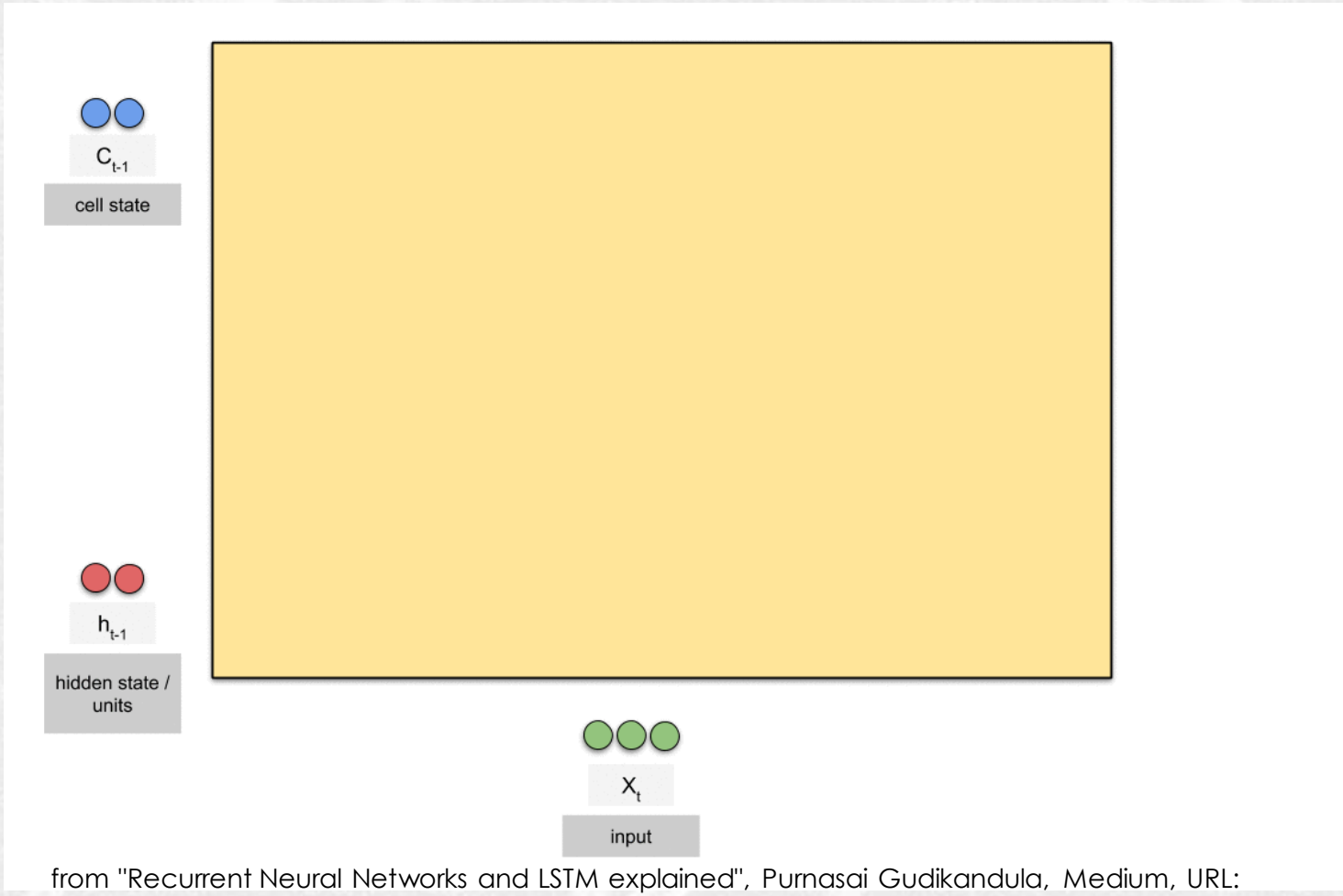


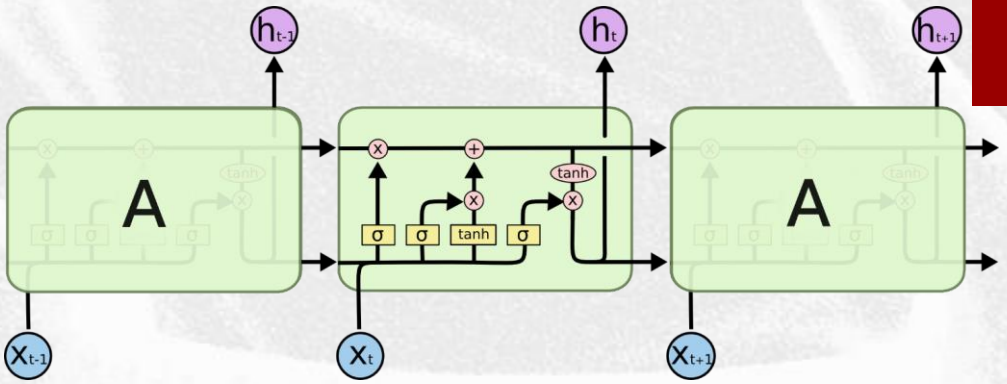
Figure 10.16: Block diagram of the LSTM recurrent network "cell." Cells are connected recurrently to each other, replacing the usual hidden units of ordinary recurrent networks. An input feature is computed with a regular artificial neuron unit. Its value can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate. The output of the cell can be shut off by the output gate. All the gating units have a sigmoid nonlinearity, while the input unit can have any squashing nonlinearity. The state unit can also be used as an extra input to the gating units. The black square indicates a delay of a single time step.

LSTMs



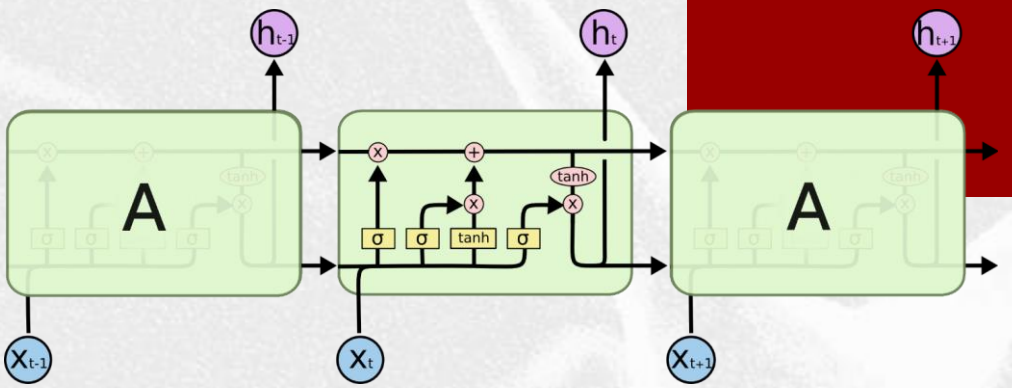
from "Recurrent Neural Networks and LSTM explained", Purnasai Gudikandula, Medium, URL: https://purnasaigudikandula.medium.com/recurrent-neural-networks-and-lstm-explained-7f51c7f6bbb9

LSTMS (Hochreiter & Schmidhuber, 1997)



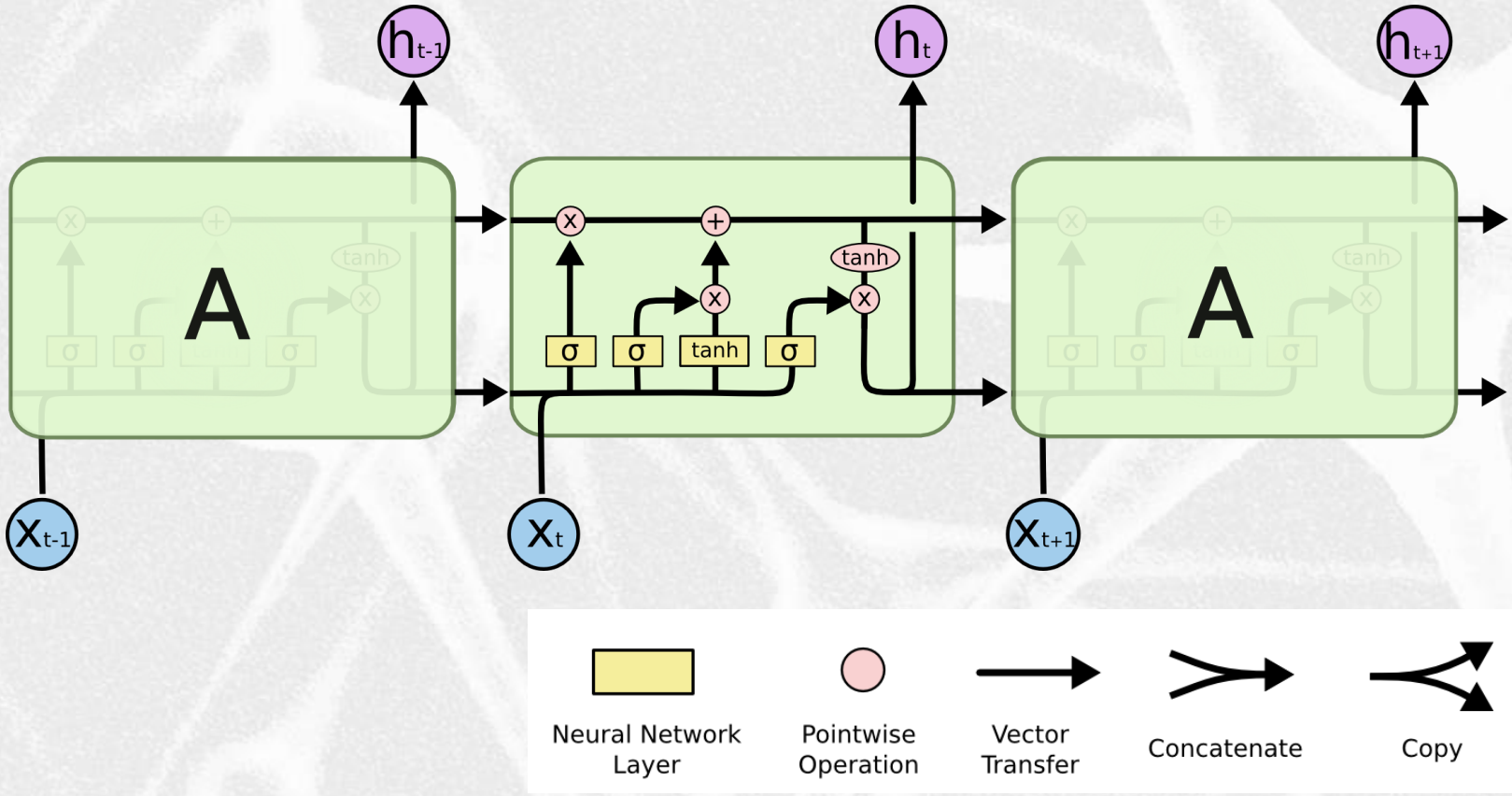
- The **Long Short-Term Memory (LSTM)** architecture (Hochreiter & Schmidhuber, 1997) was designed to solve the vanishing gradients problem.
- Main idea: to introduce as part of the state representation also specialized **memory cells** (a vector C) that can preserve gradients across time.
- Access to the memory cells is controlled by gating components, i.e. smooth mathematical functions that simulate logical gates.

LSTMS

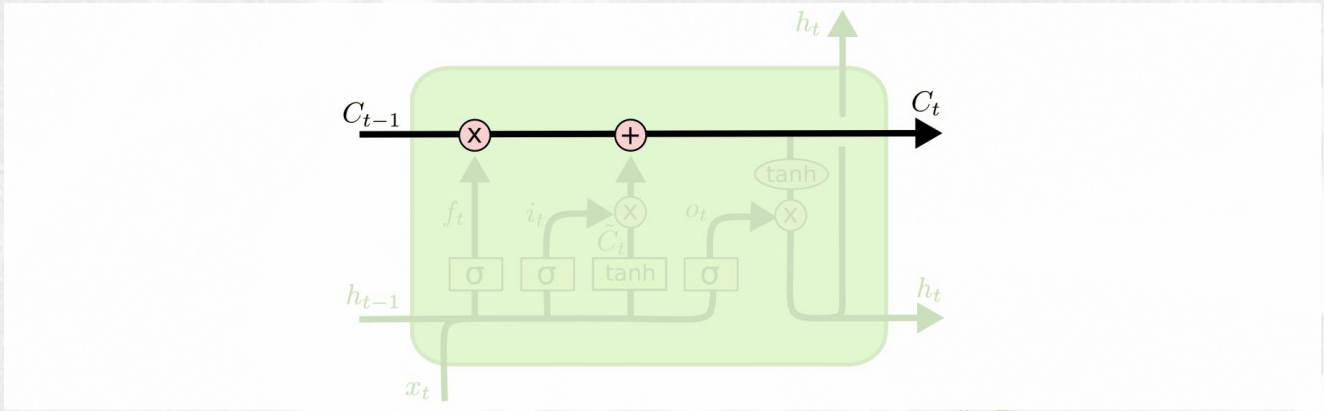


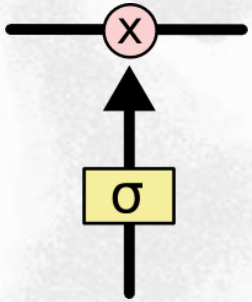
- At each input state, a gate is used to decide:
  - **how much of the new input** should be written to the memory cell,
  - **how much of the current content** of the memory cell should be **forgotten**.
- Concretely, a gate g in $[0;1]^n$ is a vector of values in the range [0; 1] that is multiplied **component-wise** with another vector C in R^n, and the result is then added to another vector.
- Indices in C corresponding to near-one values in g are allowed to pass, while those corresponding to near-zero values are blocked.

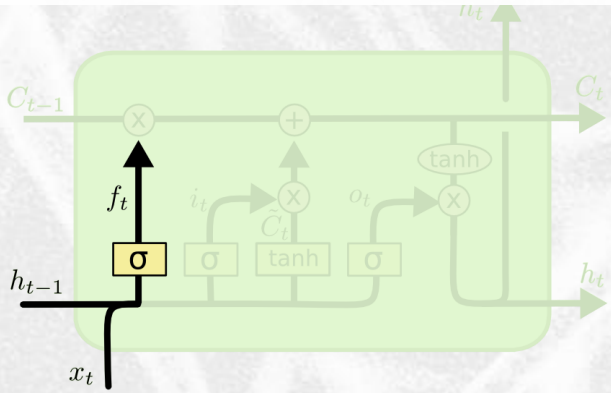
4 layer RNNS


h_{t-1}
h_t
h_{t+1}
A
A
tanh
σ
σ
tanh
σ
tanh
X_{t-1}
X_t
X_{t+1}
Neural Network Layer
Pointwise Operation
Vector Transfer
Concatenate
Copy

… The memory component and the gates



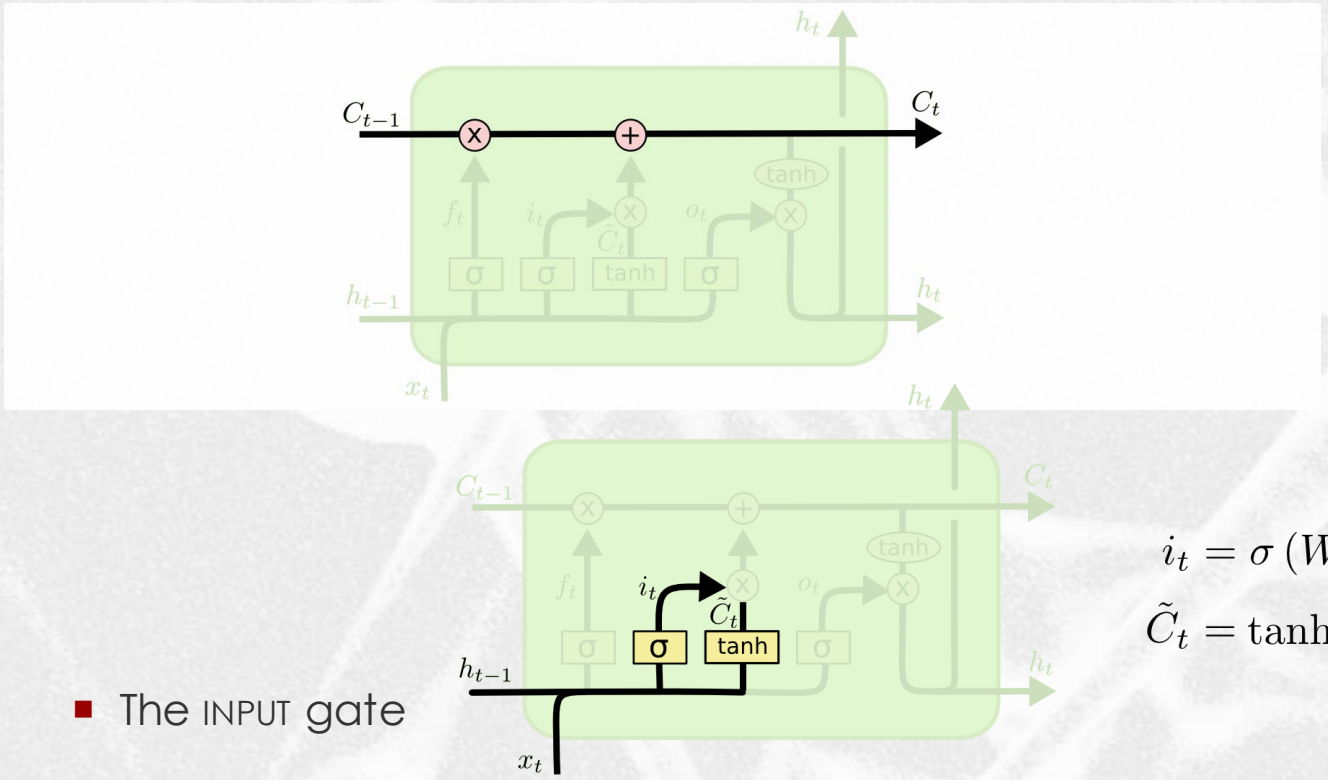


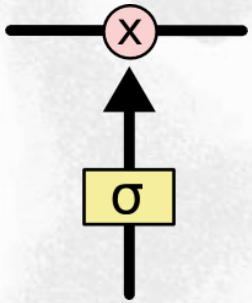


- The FORGET gate

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

… The memory component and the gates



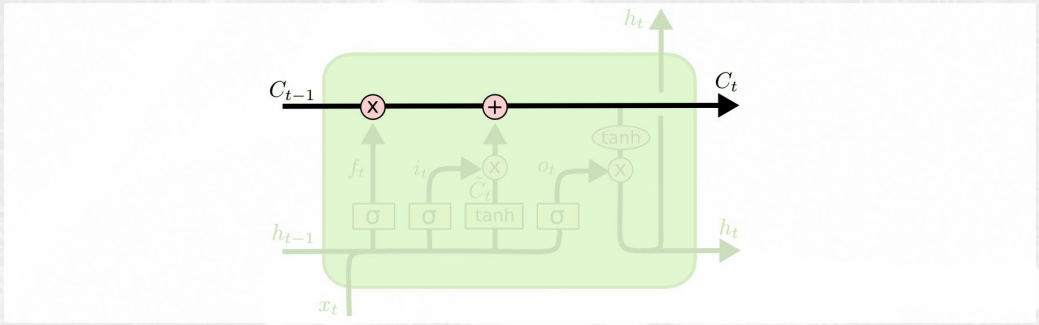


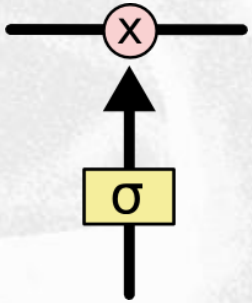
- The INPUT gate

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

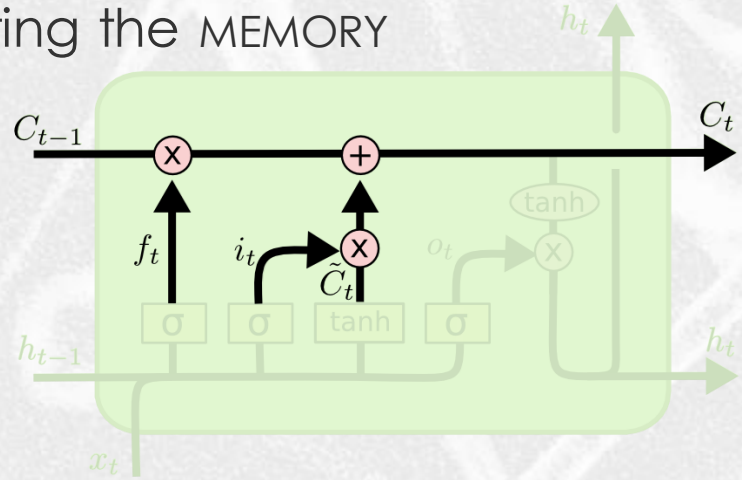
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

… The memory component and the gates



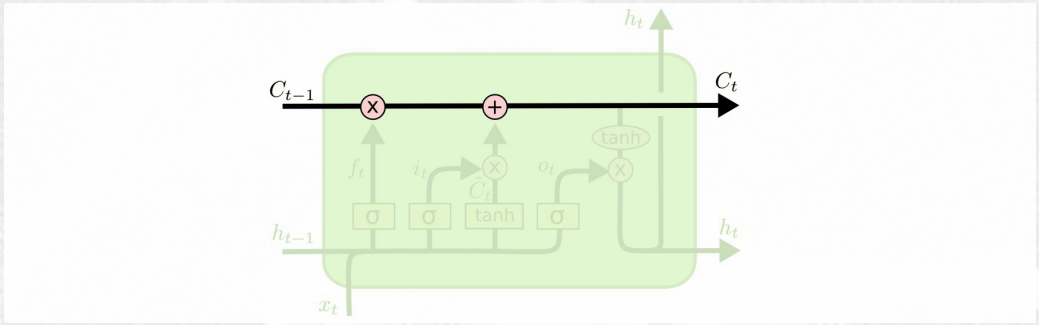


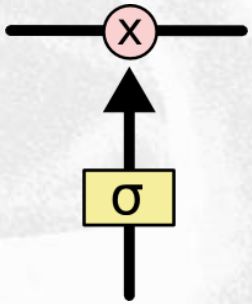
- Updating the MEMORY



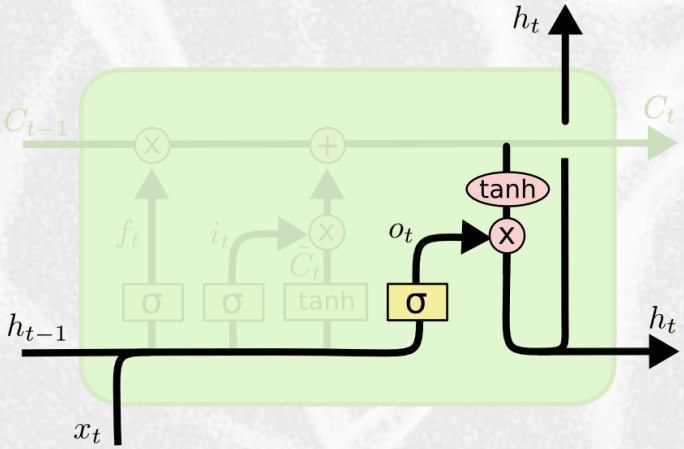
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

… The memory component and the gates





- Computing the OUTPUT



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

LSTMS

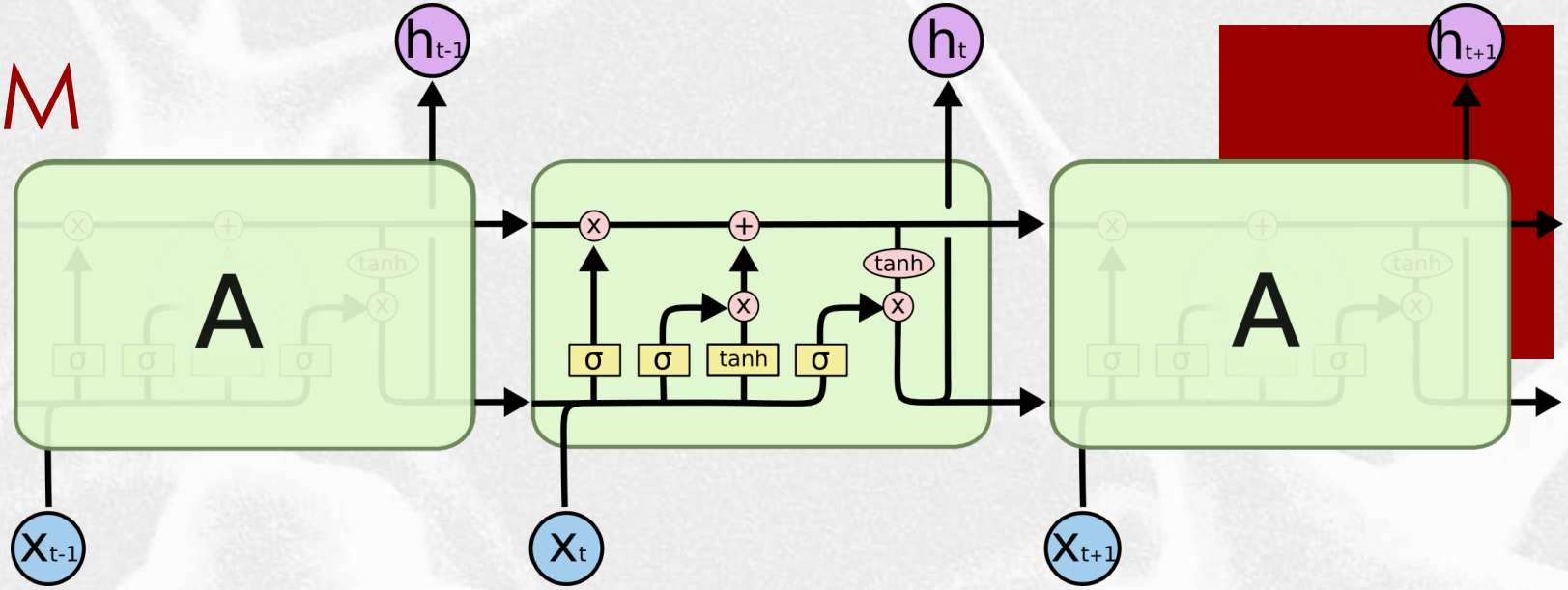
Mathematically, the LSTM architecture is defined as:[72]

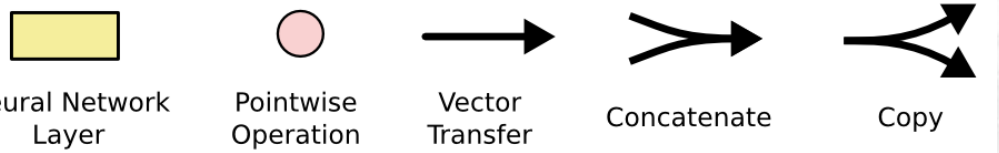
$$
\begin{aligned}
\mathbf{s_j} = R_{\text{LSTM}}(\mathbf{s_{j-1}}, \mathbf{x_j}) &= [\mathbf{c_j}; \mathbf{h_j}] \\
\mathbf{c_j} &= \mathbf{c_{j-1}} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i} \\
\mathbf{h_j} &= \tanh(\mathbf{c_j}) \odot \mathbf{o} \\
\mathbf{i} &= \sigma(\mathbf{x_j}\mathbf{W^{xi}} + \mathbf{h_{j-1}}\mathbf{W^{hi}}) \\
\mathbf{f} &= \sigma(\mathbf{x_j}\mathbf{W^{xf}} + \mathbf{h_{j-1}}\mathbf{W^{hf}}) \\
\mathbf{o} &= \sigma(\mathbf{x_j}\mathbf{W^{xo}} + \mathbf{h_{j-1}}\mathbf{W^{ho}}) \\
\mathbf{g} &= \tanh(\mathbf{x_j}\mathbf{W^{xg}} + \mathbf{h_{j-1}}\mathbf{W^{hg}})
\end{aligned}
\tag{39}
$$

$$
\mathbf{y_j} = O_{\text{LSTM}}(\mathbf{s_j}) = \mathbf{h_j}
$$

$$
\mathbf{s_j} \in \mathbb{R}^{2 \cdot d_h}, \quad \mathbf{x_i} \in \mathbb{R}^{d_x}, \quad \mathbf{c_j}, \mathbf{h_j}, \mathbf{i}, \mathbf{f}, \mathbf{o}, \mathbf{g} \in \mathbb{R}^{d_h}, \quad \mathbf{W^{xo}} \in \mathbb{R}^{d_x \times d_h}, \quad \mathbf{W^{ho}} \in \mathbb{R}^{d_h \times d_h},
$$

LSTM





Mathematically, the LSTM architecture is defined as:[72]

$$
\begin{aligned}
\mathbf{s_j} = R_{\text{LSTM}}(\mathbf{s_{j-1}}, \mathbf{x_j}) &= [\mathbf{c_j}; \mathbf{h_j}] \\
\mathbf{c_j} &= \mathbf{c_{j-1}} \odot \mathbf{f} + \mathbf{g} \odot \mathbf{i} \\
\mathbf{h_j} &= \tanh(\mathbf{c_j}) \odot \mathbf{o} \\
\mathbf{i} &= \sigma(\mathbf{x_j}\mathbf{W^{xi}} + \mathbf{h_{j-1}}\mathbf{W^{hi}}) \\
\mathbf{f} &= \sigma(\mathbf{x_j}\mathbf{W^{xf}} + \mathbf{h_{j-1}}\mathbf{W^{hf}}) \\
\mathbf{o} &= \sigma(\mathbf{x_j}\mathbf{W^{xo}} + \mathbf{h_{j-1}}\mathbf{W^{ho}}) \\
\mathbf{g} &= \tanh(\mathbf{x_j}\mathbf{W^{xg}} + \mathbf{h_{j-1}}\mathbf{W^{hg}})
\end{aligned}
\tag{39}
$$

$$
\mathbf{y_j} = O_{\text{LSTM}}(\mathbf{s_j}) = \mathbf{h_j}
$$

$$
\mathbf{s_j} \in \mathbb{R}^{2 \cdot d_h}, \ \mathbf{x_i} \in \mathbb{R}^{d_x}, \ \mathbf{c_j}, \mathbf{h_j}, \mathbf{i}, \mathbf{f}, \mathbf{o}, \mathbf{g} \in \mathbb{R}^{d_h}, \ \mathbf{W^{xo}} \in \mathbb{R}^{d_x \times d_h}, \ \mathbf{W^{ho}} \in \mathbb{R}^{d_h \times d_h},
$$

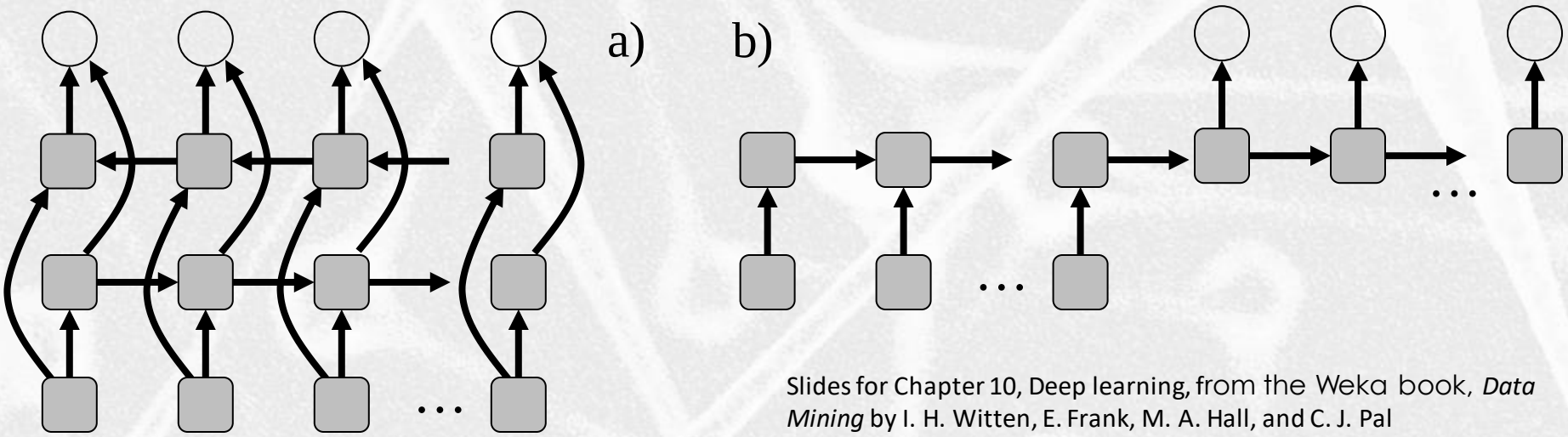
LSTMs and vanishing gradients

- The so-called "long short term memory" (LSTM) RNN architecture was specifically created to address the vanishing gradient problem.
- Uses a combination of hidden units, elementwise products and sums between units to implement gates that control "memory cells".
- Memory cells are designed to retain information without modification for long periods of time.
- They have their own input and output gates, which are controlled by learnable weights that are a function of the current observation and the hidden units at the previous time step.
- As a result, backpropagated error terms from gradient computations can be stored and propagated backwards without degradation.

Other RNN architectures

a) Recurrent networks can be made bidirectional, propagating information in both directions
   - They have been used for a wide variety of applications, including protein secondary structure prediction and handwriting recognition

b) An "encoder-decoder" network creates a fixed-length vector representation for variable-length inputs, the encoding can be used to generate a variable-length sequence as the output
   - Particularly useful for machine translation



Training different Types of RNNs

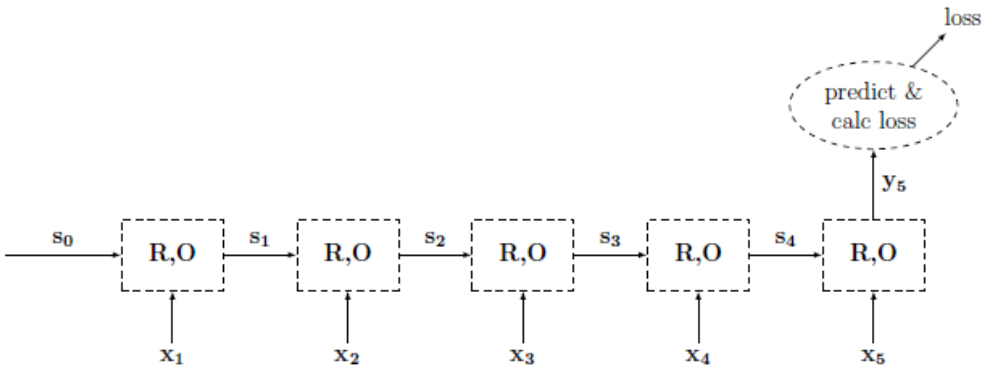


Figure 7: Acceptor RNN Training Graph.

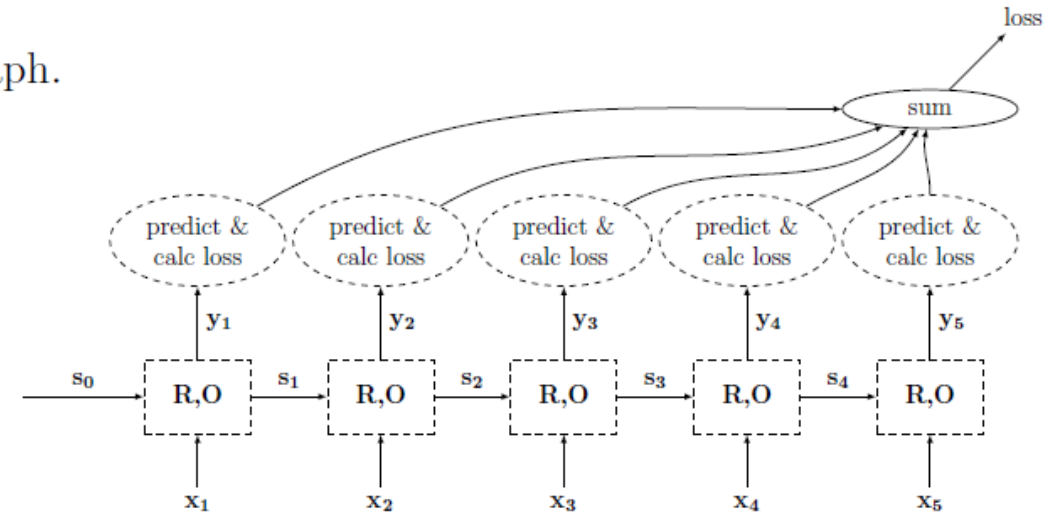


Figure 8: Transducer RNN Training Graph.

Training different Types of RNNs

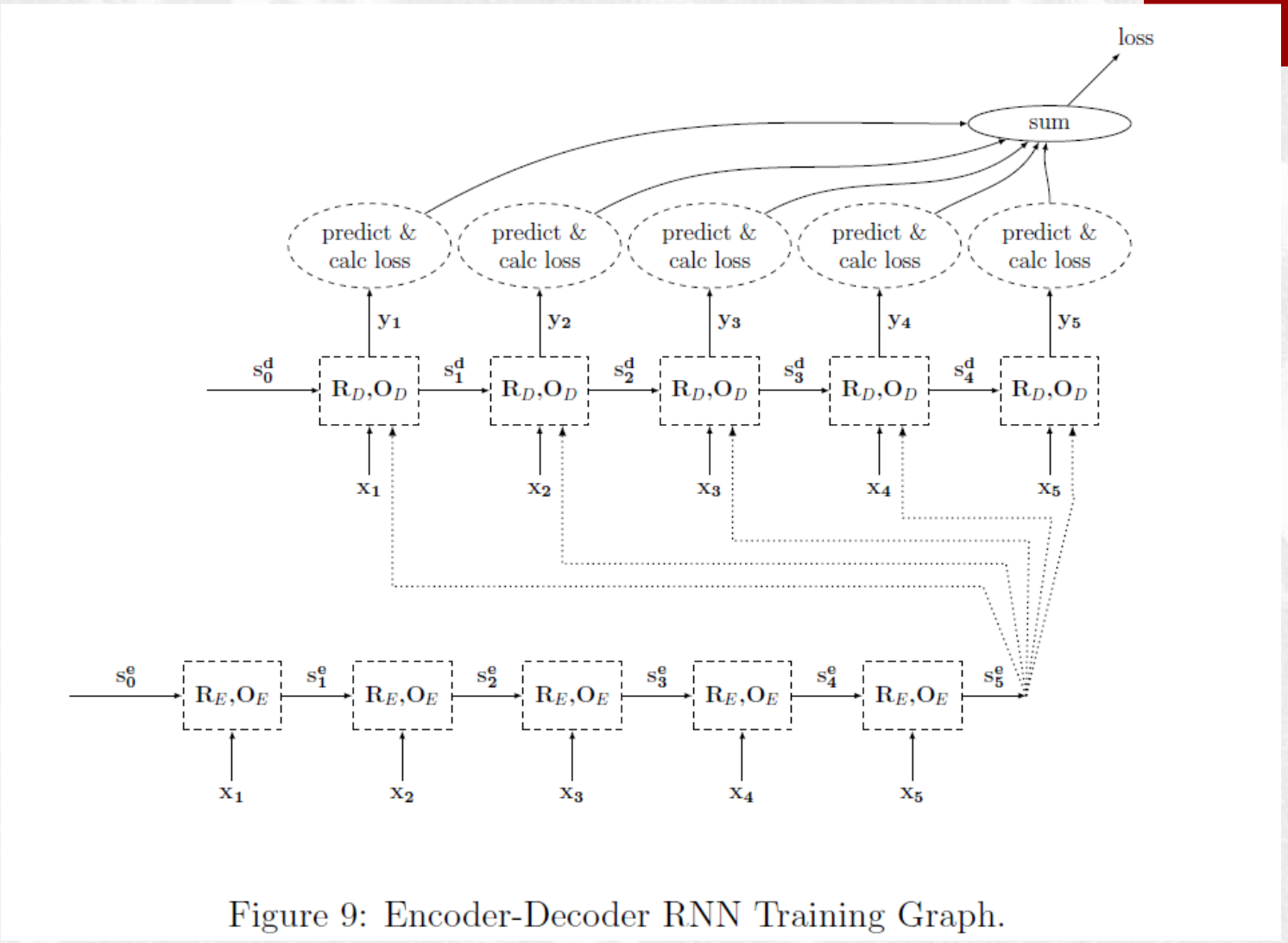


Figure 9: Encoder-Decoder RNN Training Graph.

Training different Types of RNNs

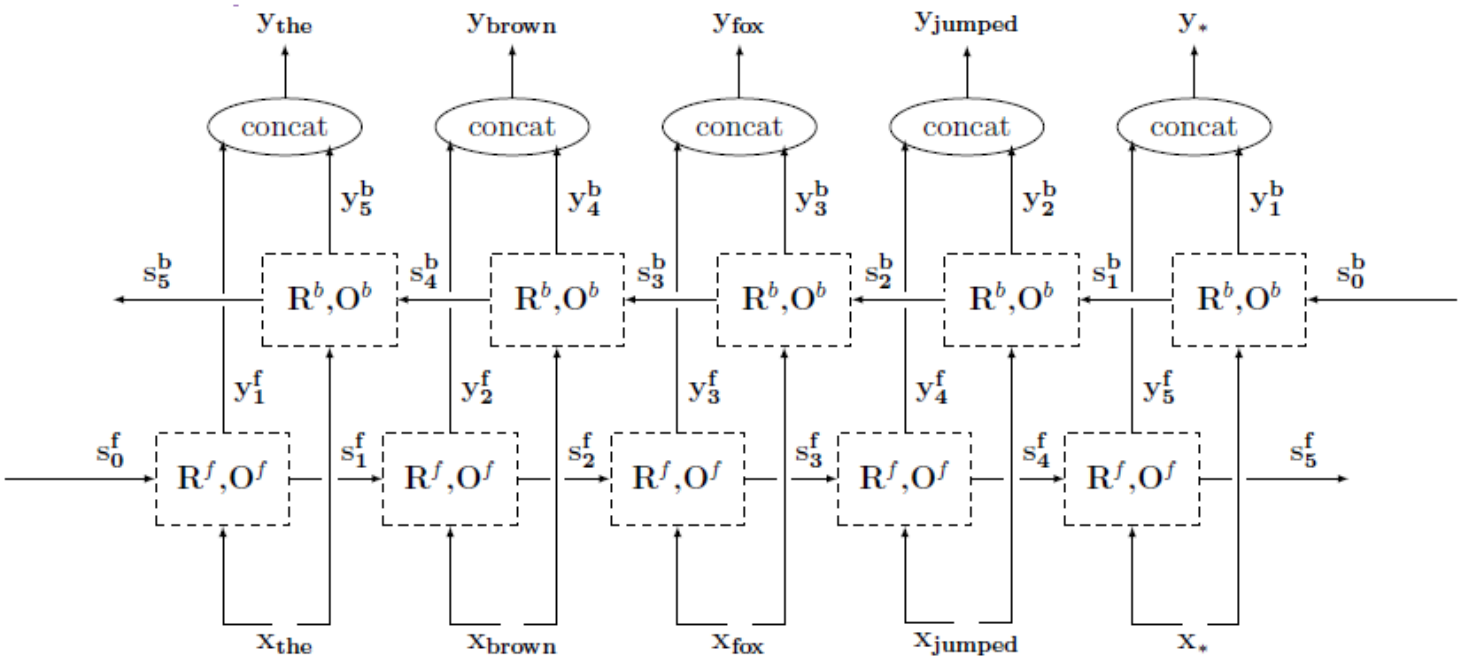
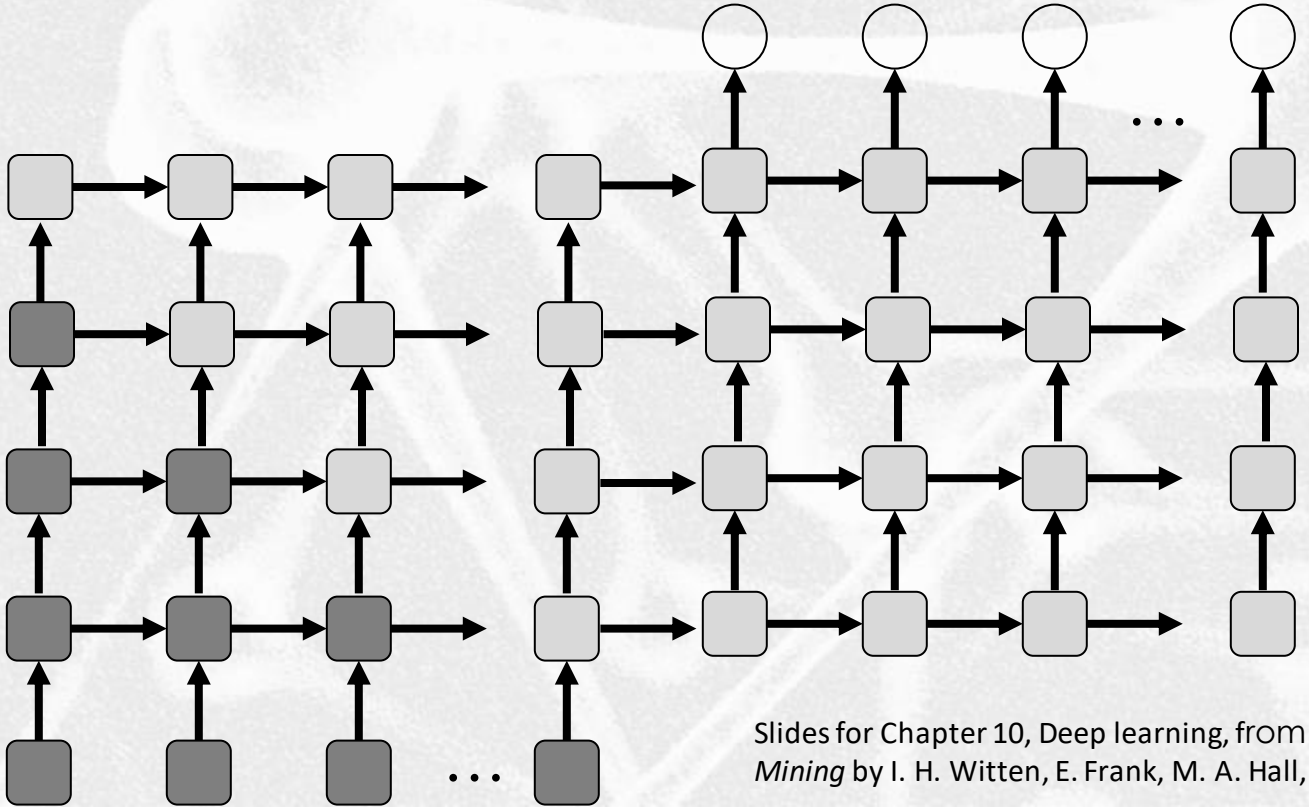


Figure 11: biRNN over the sentence "the brown fox jumped .".

Encoder-decoder deep architectures

- Given enough data, a deep encoder-decoder architecture (see below) can yield results that compete with hand-engineered translation systems.

- The connectivity structure means that partial computations in the model can flow through the graph in a wave (darker nodes in fig.)



RNNs - Bibliographic Notes & Further Readings

- Graves et al. (2009) demonstrate how recurrent neural networks are particularly effective at handwriting recognition,
- Graves et al. (2013) apply recurrent neural networks to speech.
- The form of gradient clipping presented above was proposed by Pascanu et al. (2013).
- Hochreiter and Schmidhuber (1997) is the seminal work on the "Long Short-term Memory" architecture for recurrent neural networks;
  - our explanation follows Graves and Schmidhuber (2005)'s formulation.
- Yoav Goldberg, A Primer on Neural Network Models for Natural Language Processing, Journal of Artificial Intelligence Research volume 57 pp 345-420, 2016
- Greff et al. (2015)'s paper "LSTM: A search space odyssey" explored a wide variety of variants and finds that:
  - none of them significantly outperformed the standard LSTM architecture; and
  - forget gates and the output activation function were the most critical components. Forget gates were added by Gers et al. (2000).

RNNs - Bibliographic Notes & Further Readings

- IRNNs were proposed by Le et al. (2015)
- Chung et al. (2014) proposed gated recurrent units
- Schuster and Paliwal (1997) proposed bidirectional recurrent neural networks
- Chen and Chaudhari (2004) used bi-directional networks for protein structure prediction; Graves et al. (2009) used them for handwriting recognition
- Cho et al. (2014) used encoder-decoder networks for machine translation, while Sutskever et al. (2014) proposed deep encoder-decoder networks and used them with massive quantities of data
- For further accounts of advances in deep learning and a more extensive history of the field, consult the reviews of LeCun et al. (2015), Bengio (2009), and Schmidhuber (2015)