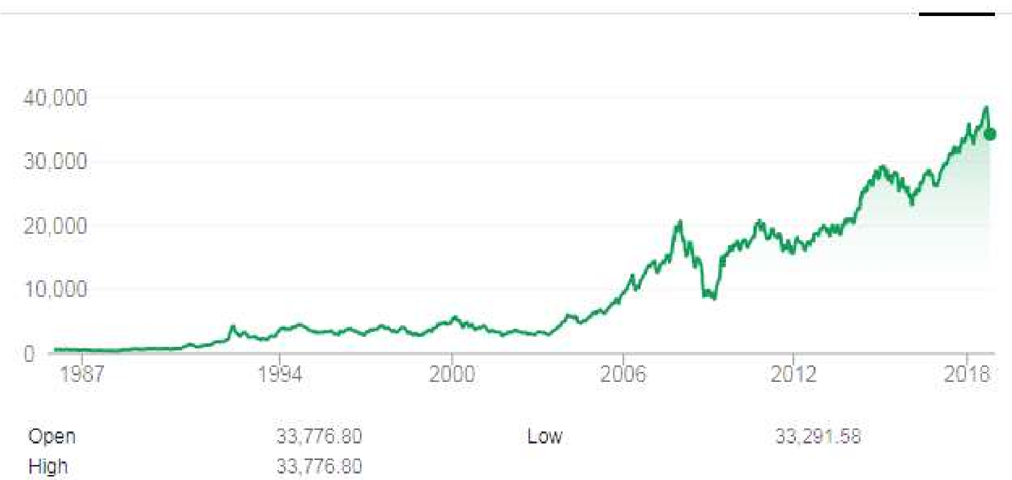
Time-Series Models

Recurrent Neural Network

Examples

BSE SENSEX



Year	Population(in Million)
1921	251
1931	279
1941	319
1951	361
1961	439
1971	548
1981	685

Time Series

- Time series is a sequence of observations often ordered in time.
- Popular Problem: Given a sequence, predict future samples.
- Applications:
  - Meteorology,
  - Finance,
  - Marketing etc.

- We want a machine learning model to understand sequences, not samples.
- Assume we have a sequence of measurements, and we want to take N sequential measurements and predict the next one.

Notation and Problem

- Notation: x[0], x[1], x[2], …, x[N].
- x[t], Where t is the time or index in the sequence.
- Assumption: Measurement at time t depends on three previous ones.
  - i.e., t-1, t-2 and t-3
- Why 3? We can have a different number.

Raw Data	
Time	Sample
1	X_1
2	X_2
3	X_3
4	X_4
5	X_5
6	X_6
7	X_7

Rearranged Data			
Feature-1	Feature-2	Feature-3	Y_i
X_1	X_2	X_3	X_4
X_2	X_3	X_4	X_5
X_3	X_4	X_5	X_6
X_4	X_5	X_6	X_7

Feature Vector	
Feature	Y_i
V_1	X_4
V_2	X_5
V_3	X_6
V_4	X_7

A Simple Model

- $X[t] = w_1 X[t-1] + w_2 X[t-2] + w_3 X[t-3] + n$ $\boxed{n \text{ is noise}}$

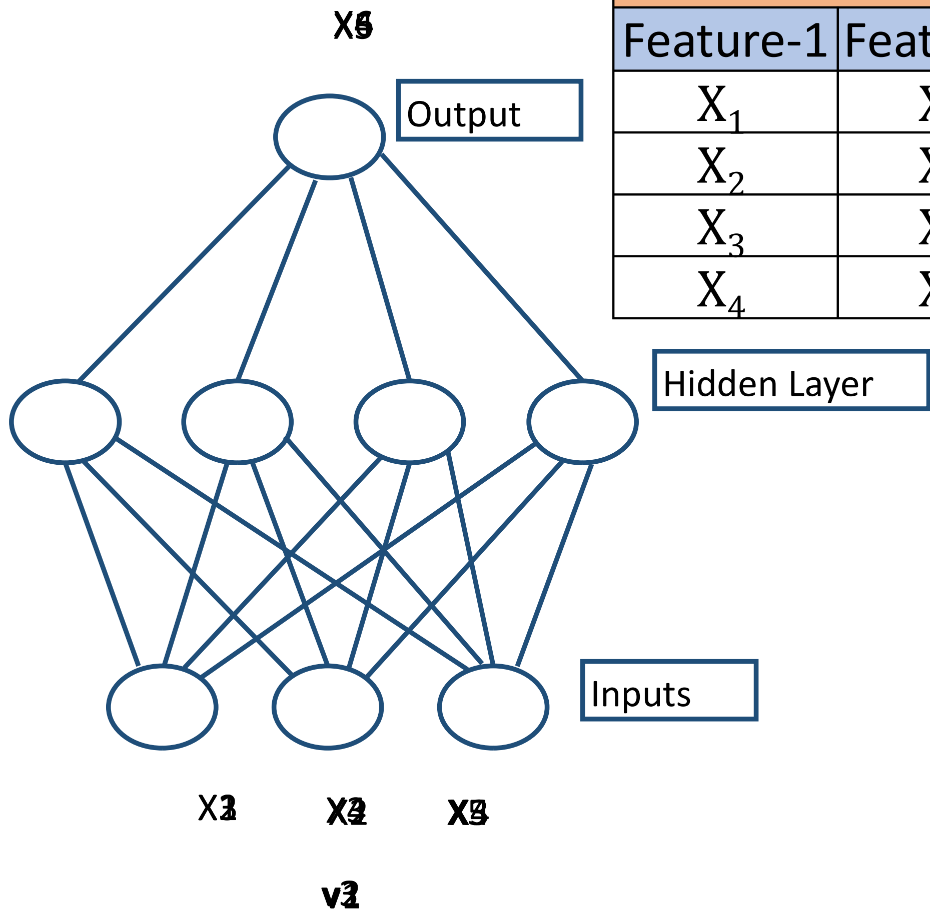
- Given the sequence $X[0], X[1], \ldots \ldots X[N]$, we find the coefficients w_1, w_2, w_3 such that the prediction error is minimal.

- $X_t = f(W, X_{t-1}, X_{t-2}, X_{t-3}) + n$

$$\Rightarrow \min_W \sum_{t=3}^{N} (X_t - f(W, X_{t-1}, X_{t-2}, X_{t-3}))^2$$

- Mean Absolute Deviation: $\sum_{i=1}^{N} \frac{|X_i - \hat{X}_i|}{N}$
- Mean Absolute Percent Error: $\frac{100}{N} \sum_{i=1}^{N} \frac{|X_i - \hat{X}_i|}{\hat{X}_i}$
- Mean Square Error: $\sum_{i=1}^{N} \frac{|X_i - \hat{X}_i|^2}{N}$
- Root Mean Square Error: $\sqrt{\sum_{i=1}^{N} \frac{|X_i - \hat{X}_i|^2}{N}}$

Neural Networks for Time Series Forecasting



Rearranged Data			
Feature-1	Feature-2	Feature-3	Y_i
X_1	X_2	X_3	X_4
X_2	X_3	X_4	X_5
X_3	X_4	X_5	X_6
X_4	X_5	X_6	X_7

Classical Models (AR and MA)

- Auto Regressive (**AR**) Model assumes: $X_t = \alpha X_{t-1} + \epsilon_t$, ($\epsilon_t$ is random uncorrelated)
  - Predict the next term in a sequence from a fixed number of previous terms.

- **AR**: A model of order p is $X_t = \sum_{i=1}^{p} \alpha_i X_{t-i} + \epsilon_t$

- Moving Average (**MA**) model assumes: $X_t = \epsilon_t + \beta\, \epsilon_{t-1}$

- **MA**: A model of order q is $X_t = \epsilon_t + \sum_{j=1}^{q} \beta_j \epsilon_{t-j}$

Classical Models (ARMA & ARIMA)

- **ARMA (p,q):** $X_t = \sum_{i=1}^{p} \alpha_i X_{t-i} + \sum_{j=0}^{q} \beta_j \epsilon_{t-j}$, **with** $\beta_0 = 1$
  - ARMA is combined from the AR and MA models to model stationary nonseasonal time series data.

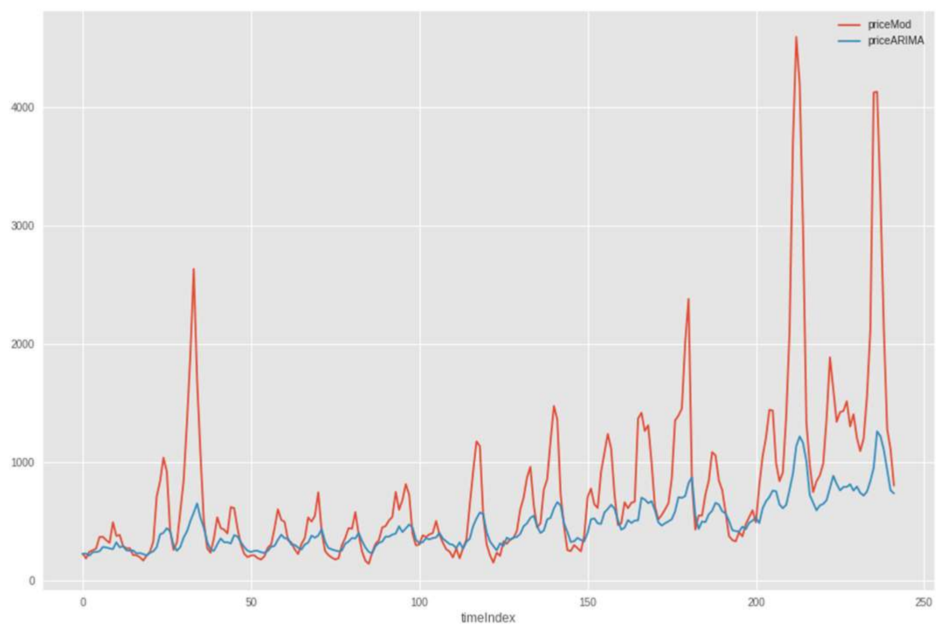
- **ARIMA (p, d, q):**
  - ARIMA is quite similar to ARMA model, with the **I** standing for Integrated, i.e. differencing.
  - A process is ARIMA (p, q, d) if $\nabla^d X$ is ARMA (p,q), where $\nabla X_t = X_t - X_{t-1}$ and $\nabla^2 X_t = \nabla(\nabla X_t)$
  - ARIMA is a combination of a number of differences already applied on the model to make it stationary, the number of previous lags along with residuals errors in order to forecast future values.

Many Comparisons

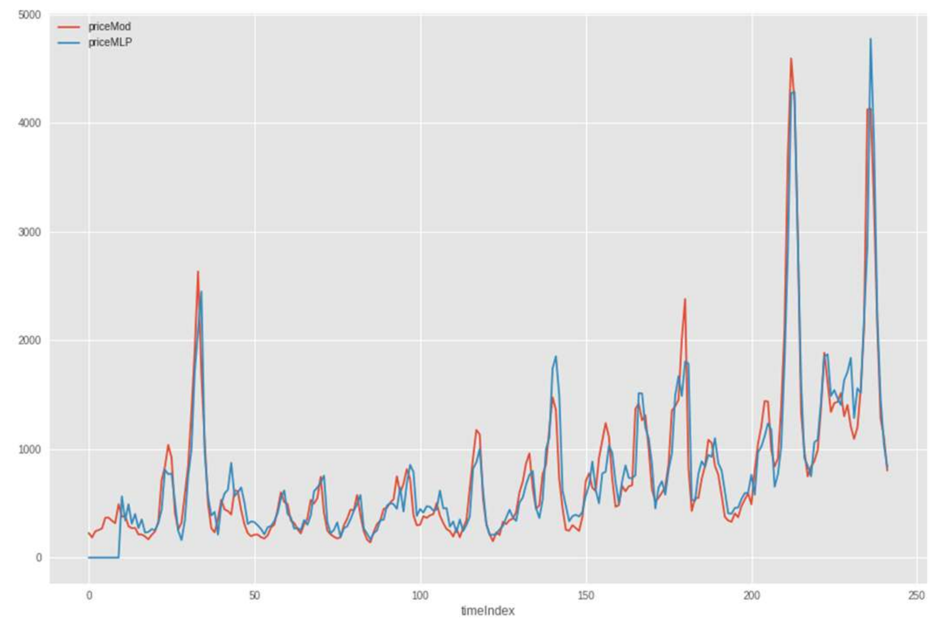
- MLP vs ARMA/ARIMA:
  - “Forecasting with artificial neural networks: The state of the art ” – 1998
    - Shows that ANNs are at par or better.
  - “Time series forecasting using a hybrid ARIMA and neural network model” G.P. Zhang (2003)
    - Shows how to get advantages of “both” worlds
- We now know more NN than what we did in 1998 or 2003!!

Prediction using ARIMA and MLP

ARIMA



MLP



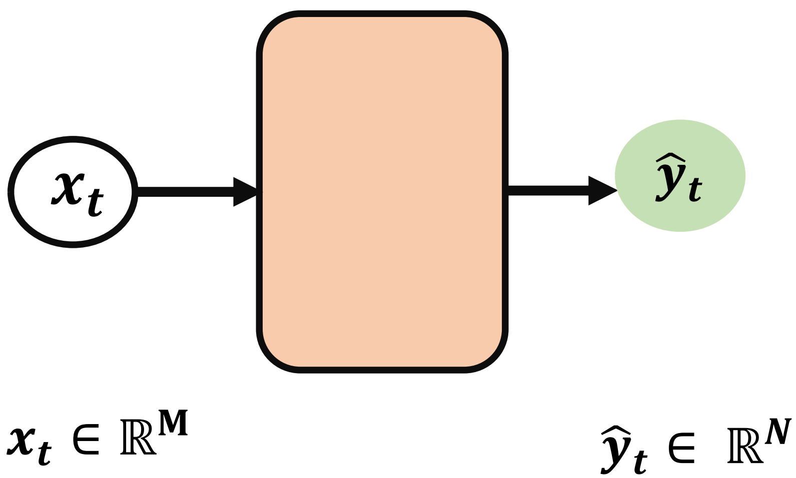
CNNs or MLPs shortcomings

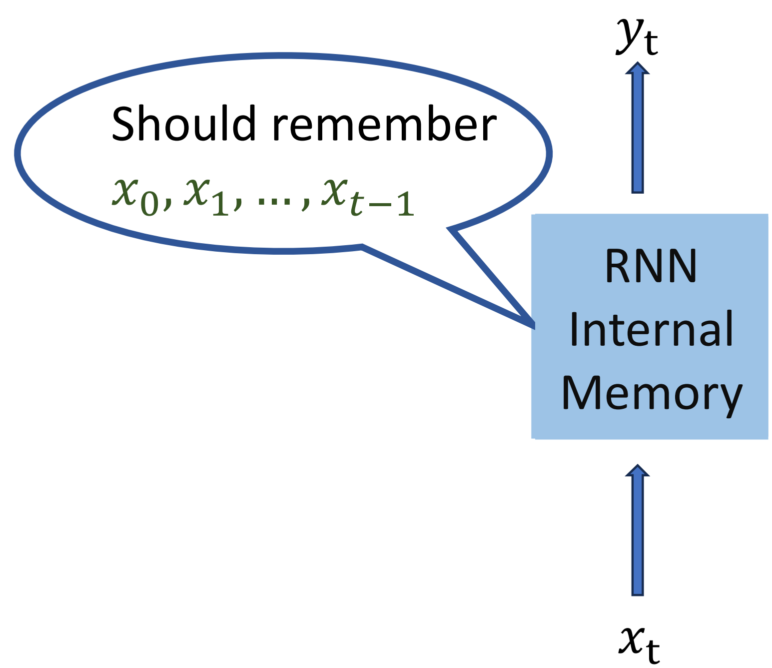
- MLPs/CNNs require fixed input and output size.
- MLPs/CNNs can’t classify inputs in multiple places.
- A fully connected network will not distinguish the order and therefore will be missing some information.
- Predicting the next term in a sequence blurs the distinction between supervised and unsupervised learning.
    - Uses method designed for supervised learning, but it doesn’t require a separate teaching signal.
    - The network needs to have a memory.

Memory

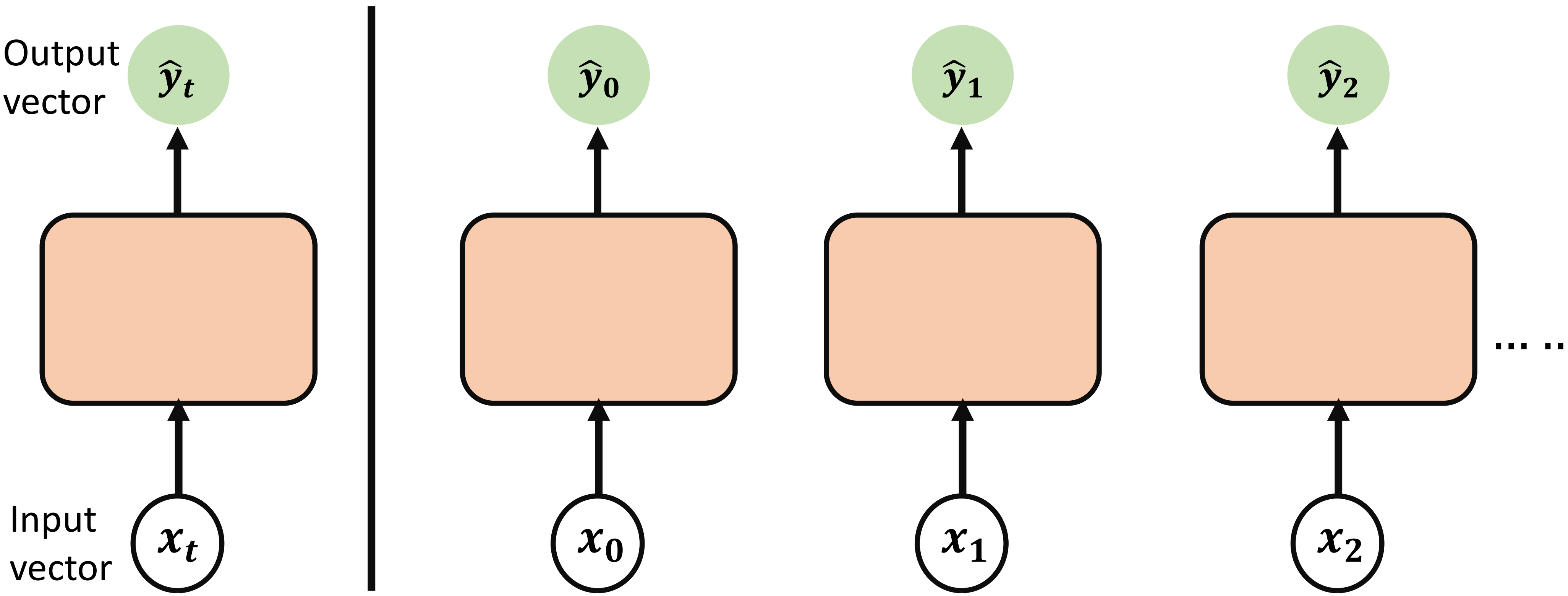
- Somehow the computational unit should remember what it has seen before

- We'll call the information the unit's state





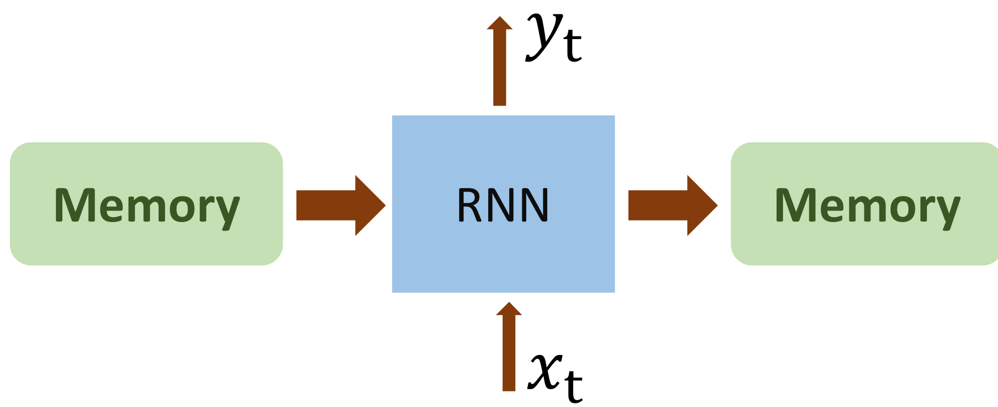
Handling Individual Time Steps

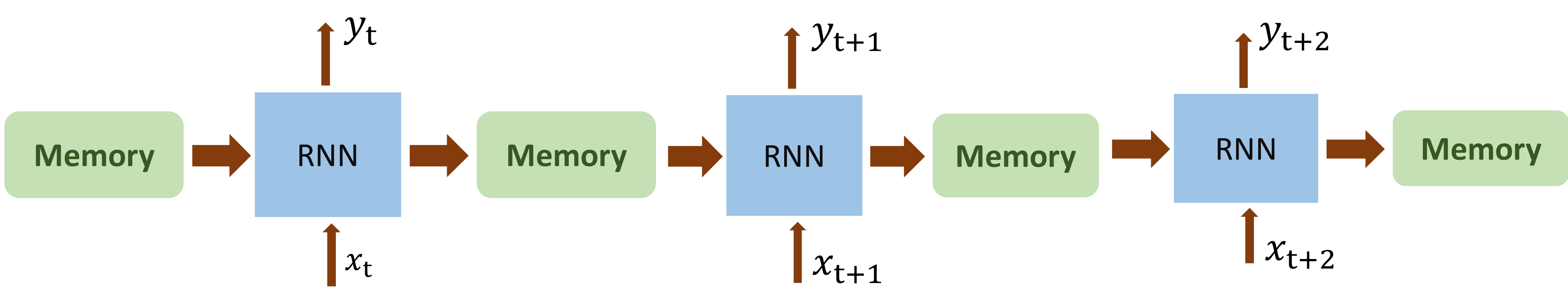


$$\hat{y}_t = f(x_t)$$

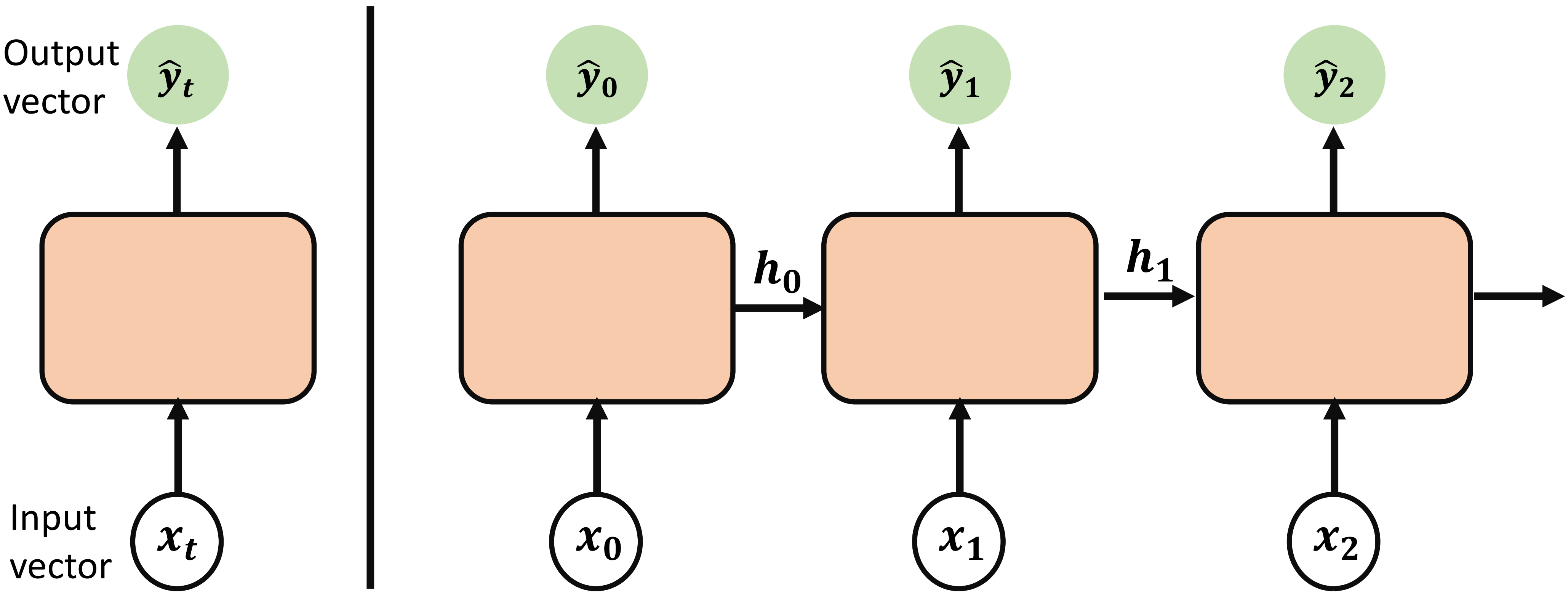
Recurrent Neural Networks

- The memory or state can be written to a file but in RNNs, we keep it inside the recurrent unit.
- In an array or in a vector!





Handling Individual Time Steps

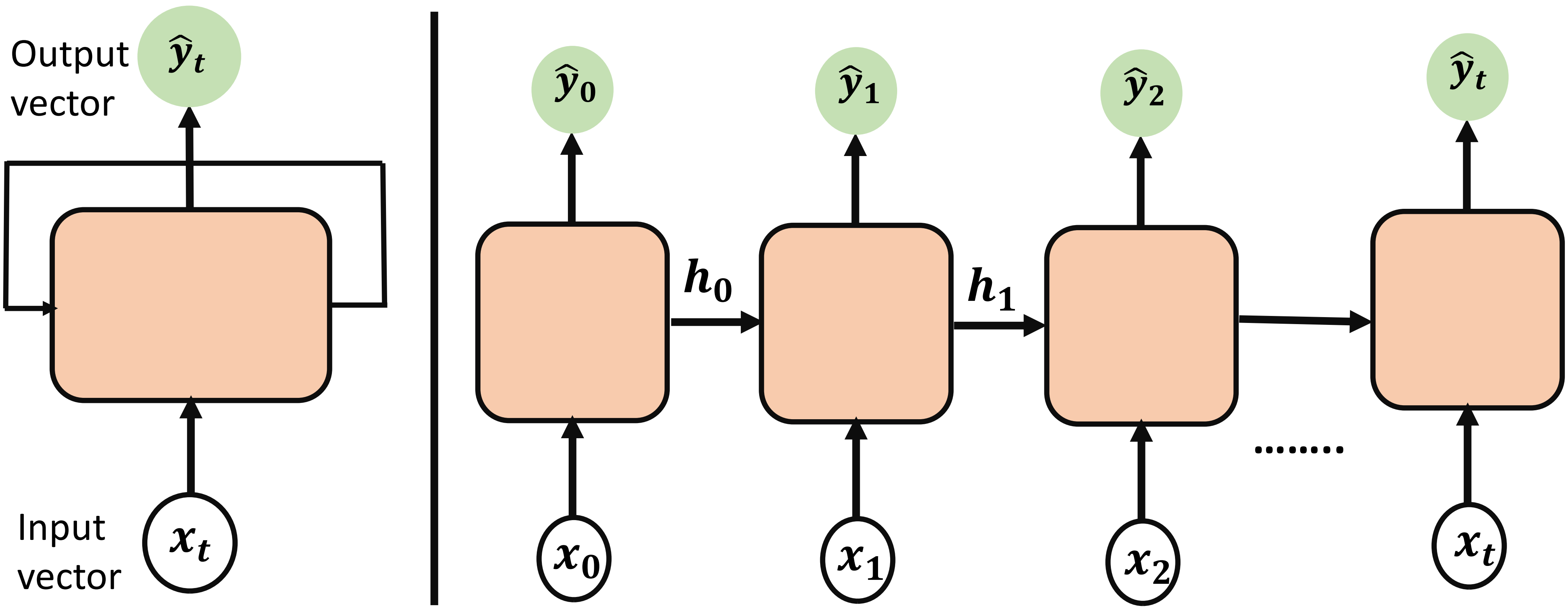


$$\hat{y}_t = f(x_t, h_{t-1})$$

Unrolled RNNs

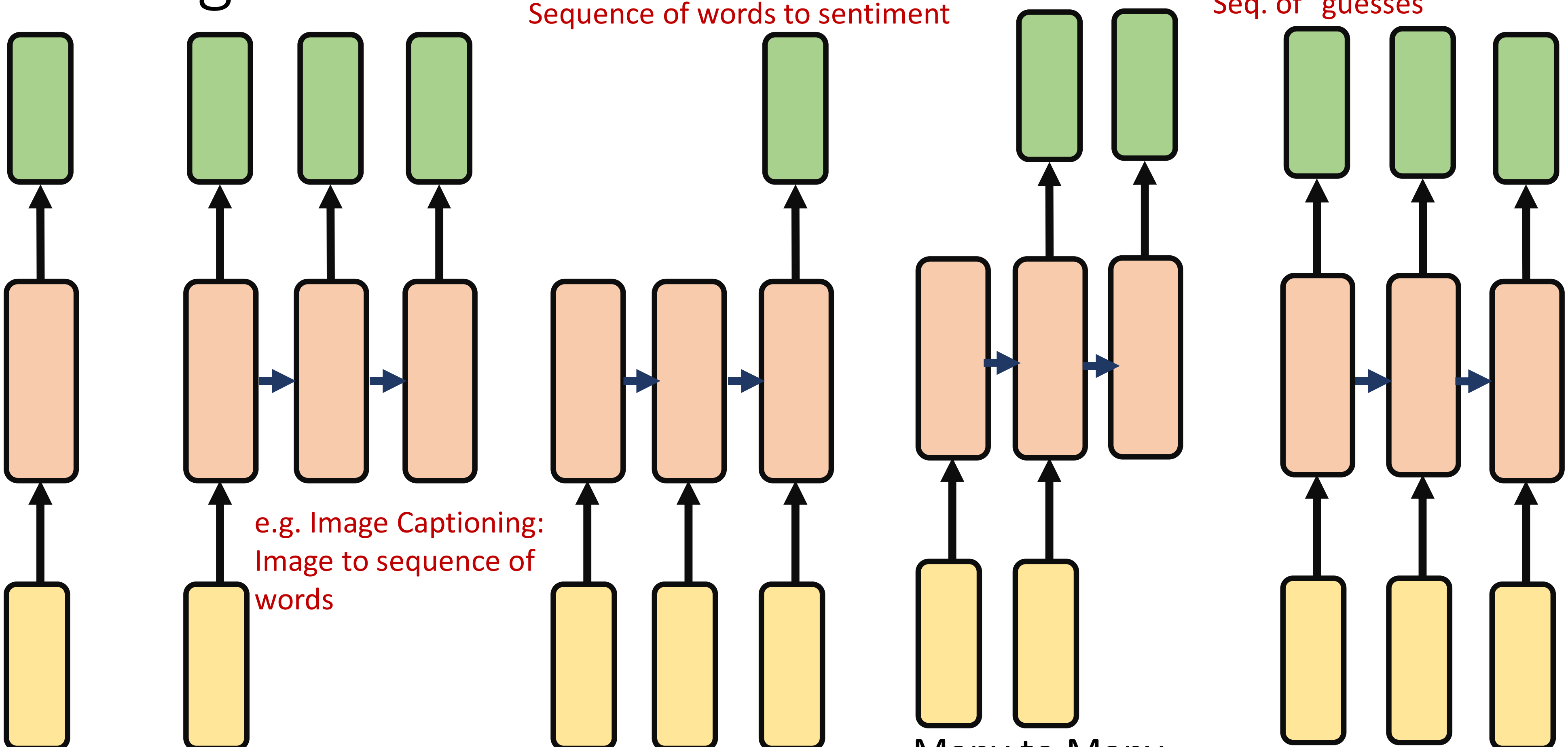
Key Idea: RNNs have an "internal state" that is updated as a sequence is processed

- Temporal dependencies
- Variable Sequence Length



$$\hat{y}_t = f(x_t, h_{t-1})$$

Modelling a RNN



One to One

One to Many

e.g. Image Captioning: Image to sequence of words

Many to One

e.g. Sentiment Classification: Sequence of words to sentiment

Many to Many

e.g. Machine translation: Seq. of words to Seq. of words

Many to Many

e.g. On-line freehand sketch recognition: Seq. of strokes to Seq. of "guesses"

http://cs231n.stanford.edu/slides/winter1516_lecture10.pdf

RNN hidden state update

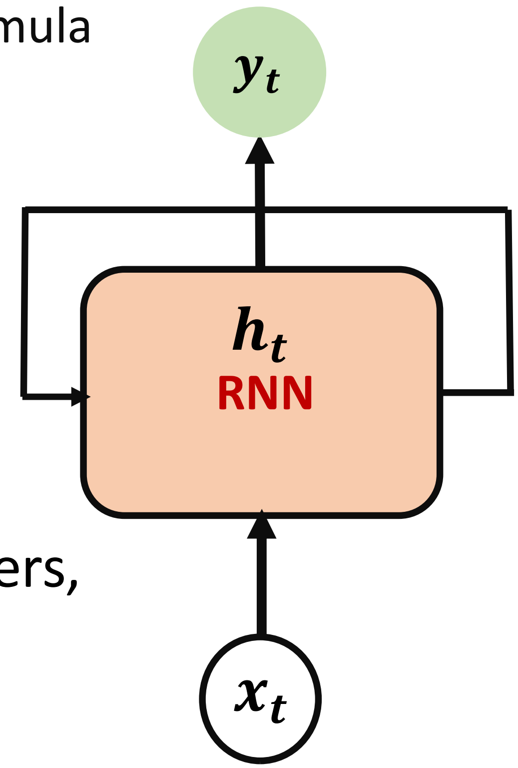
- We can process a sequence of vectors $\boldsymbol{x}$ by applying a recurrence formula at every time step: $\boldsymbol{h_t} = \boldsymbol{f_W(h_{t-1}, x_t)}$

$\mathrm{h_t}$: new state, $\mathrm{h_{t-1}}$: old state, $\mathrm{x_t}$: input vector at the time step

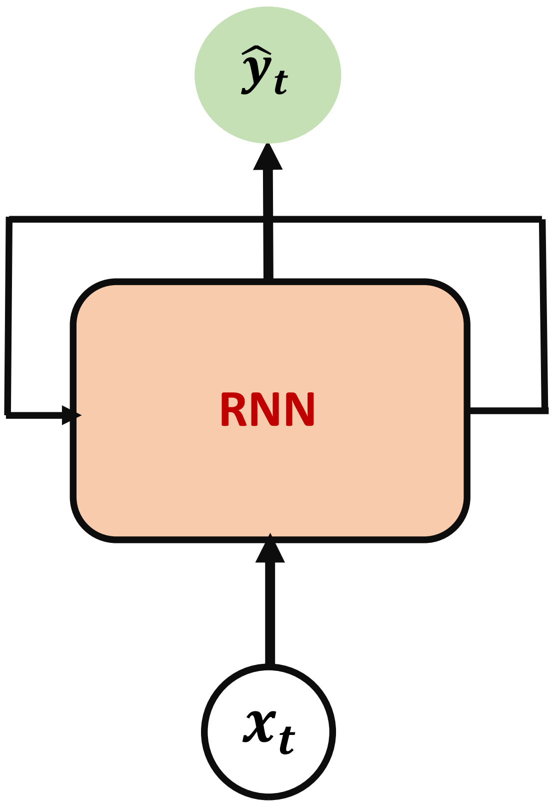
$\mathrm{f_W}$: some function with parameters W

Note: The same function and the same set of parameters are used at every time step.

- The output y_t is represented by another function of parameters, W_{hy}, where $\boldsymbol{y_t = f_{W_{hy}}(h_t)}$



RNN State Update and Output



Output Vector

$$\hat{y}_t = W_{hy}^T h_t$$

Update Hidden State

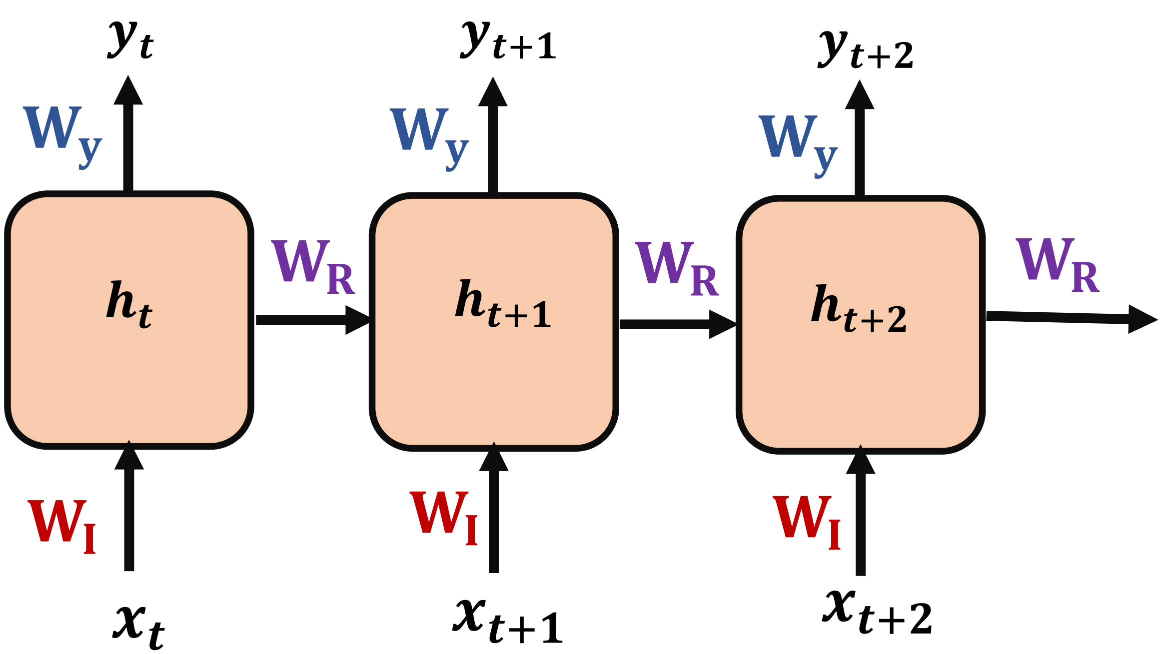
$$\mathrm{h_t = tanh(W_{hh}^T h_{t-1} + W_{xh}^T x_t)}$$

$$\mathrm{h_t = \ f_W(h_{t-1}, x_t)}$$

Input Vector

$$x_t$$

Recurrent Neural Network



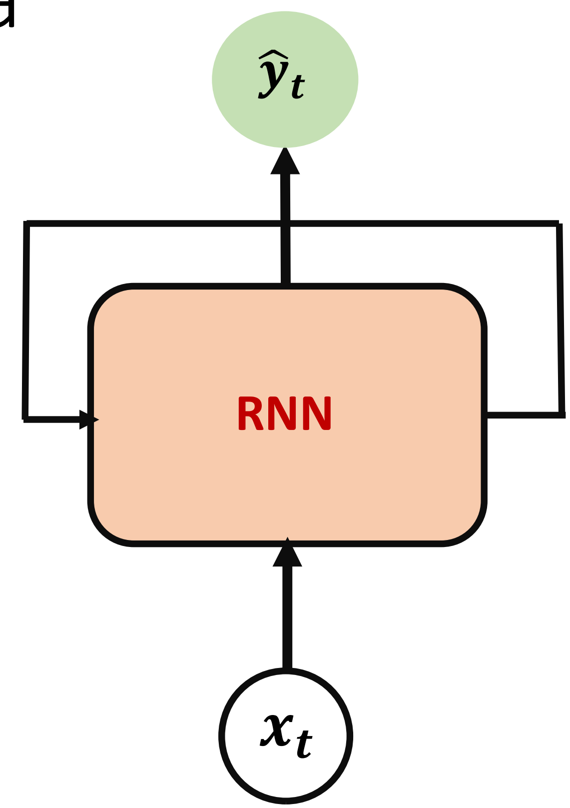
- 3 sets of parameters - $\mathrm{W_I}$, $\mathrm{W_y}$, $\mathrm{W_R}$ shared for each time-step.
- Reuse the same weight matrix at every time-step.

https://www.youtube.com/watch?v=Ukgii7Yd_cU

Sequence Modelling: Design Criteria

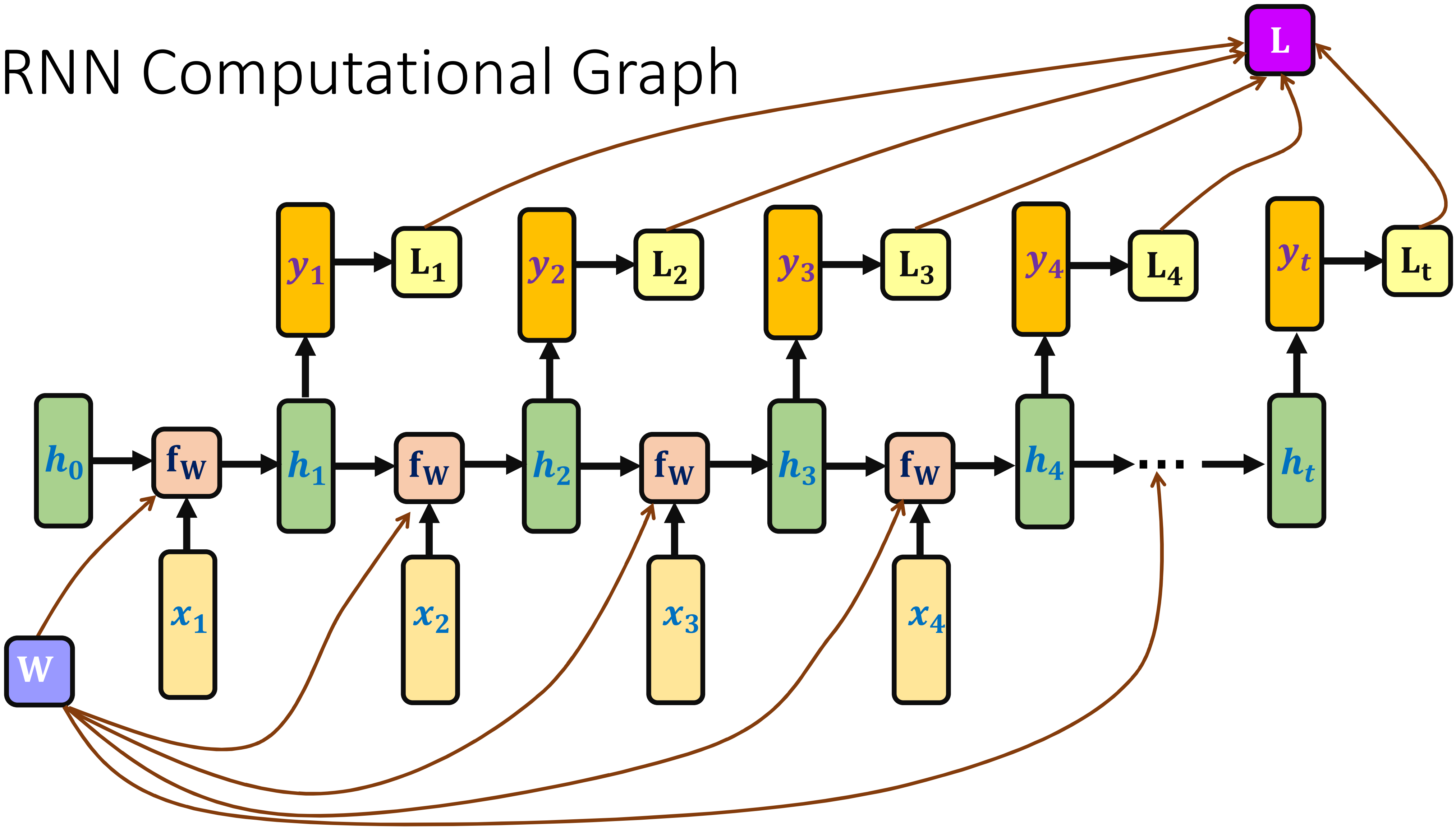
To model sequences, we need to:

1. Handle variable-length sequences
2. Track long-term dependencies
3. Maintain information about order
4. Share parameters across the sequence



Recurrent Neural Networks meets the Sequence Modelling Design Criteria

RNN Computational Graph



RNN Gradient Flow

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

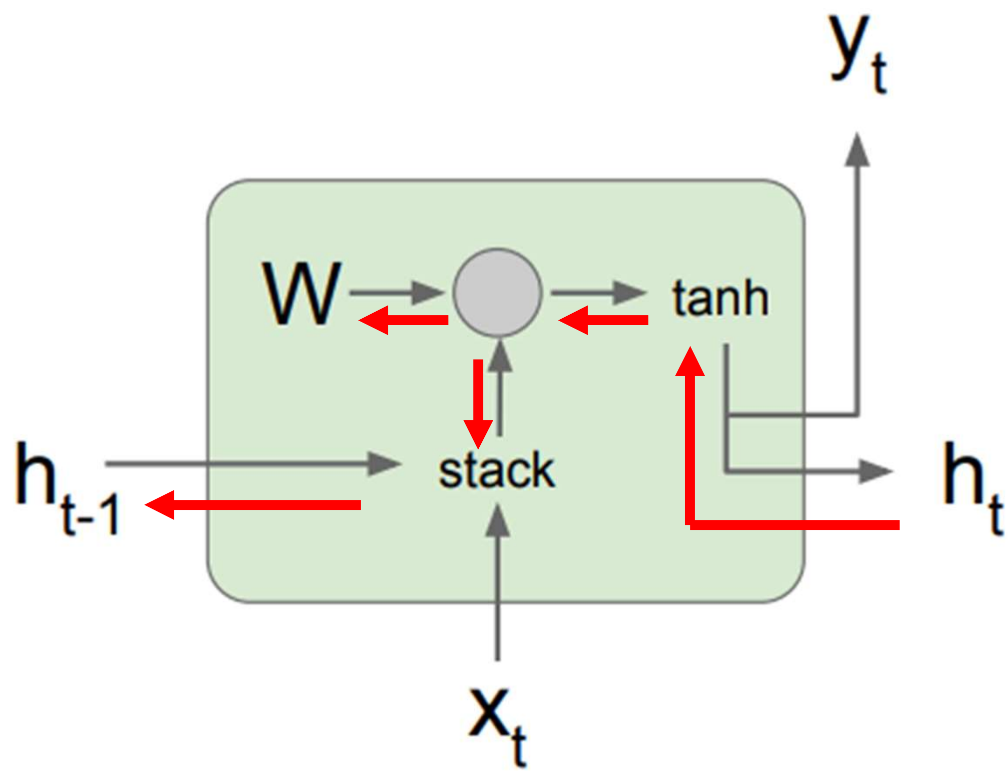
$$= \tanh\left(\begin{pmatrix} W_{hh} & W_{xh} \end{pmatrix}\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

$$= \tanh\left(W\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

$$\frac{\partial h_t}{\partial h_{t-1}} = \tanh'(W_{hh}h_{t-1} + W_{xh}x_t)W_{hh}$$

Backpropagation in time: $\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$

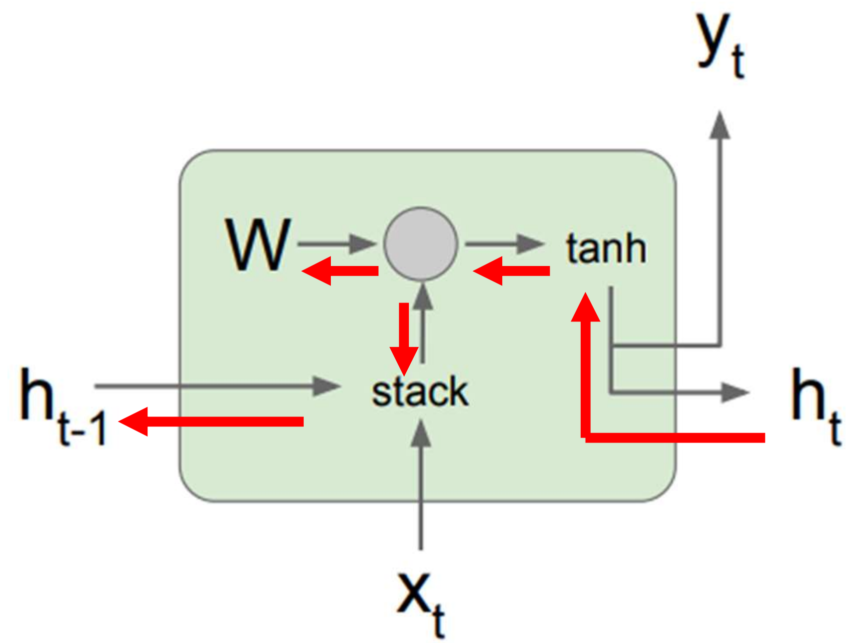
$$\frac{\partial L_t}{\partial W} = \frac{\partial L_T}{\partial h_T}\frac{\partial h_t}{\partial h_{t-1}} \dots \frac{\partial h_1}{\partial W} = \frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T} \frac{\partial h_t}{\partial h_{t-1}}\right)\frac{\partial h_1}{\partial W}$$



RNN Gradient Flow

$$h_t = \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

$$\frac{\partial h_t}{\partial h_{t-1}} = \tanh'(W_{hh}h_{t-1} + W_{xh}x_t)W_{hh}$$



Backpropagation in time:

$$\frac{\partial L_t}{\partial W} = \frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T} \frac{\partial h_t}{\partial h_{t-1}}\right)\frac{\partial h_1}{\partial W} \quad = \frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T} \boxed{\tanh'(W_{hh}h_{t-1} + W_{xh}x_t)W_{hh}}\right)\frac{\partial h_1}{\partial W}$$

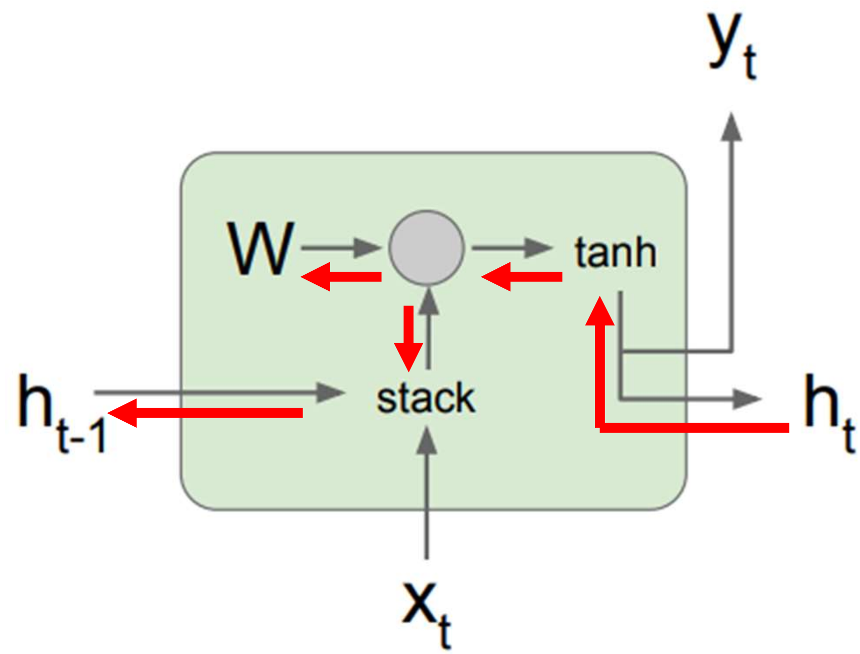
Value almost always less than one,
vanishing gradient problem

RNN Gradient Flow

- What if we assumed no non-linearity?

$$h_t = W_{hh} h_{t-1} + W_{xh} x_t$$

$$\frac{\partial h_t}{\partial h_{t-1}} = W_{hh}$$



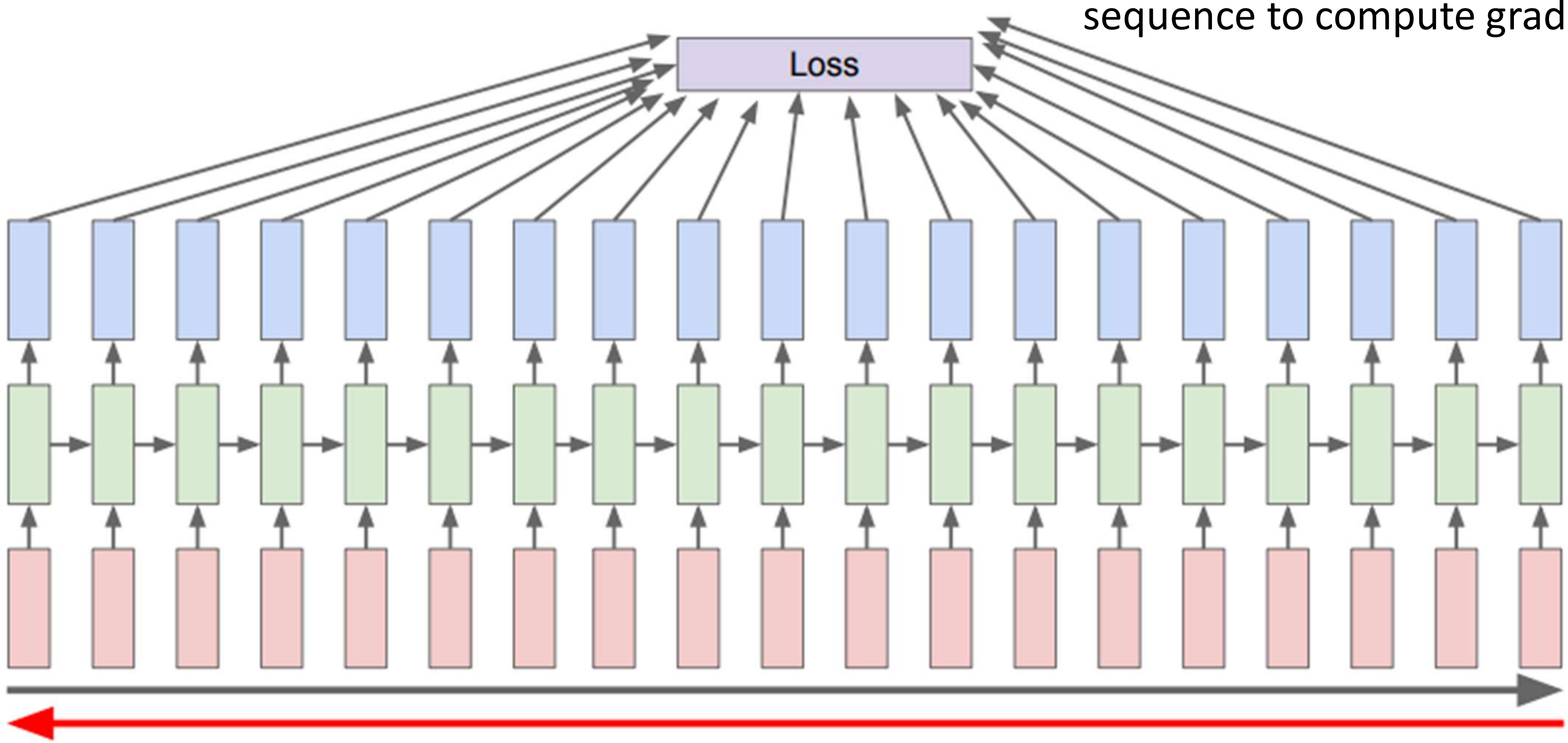
Backpropagation in time: $\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$

$$\frac{\partial L_t}{\partial W} = \frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T} \frac{\partial h_t}{\partial h_{t-1}}\right)\frac{\partial h_1}{\partial W} = \frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T} W_{hh}\right)\frac{\partial h_1}{\partial W} = \frac{\partial L_T}{\partial h_T} W_{hh}^{T-1} \frac{\partial h_1}{\partial W}$$

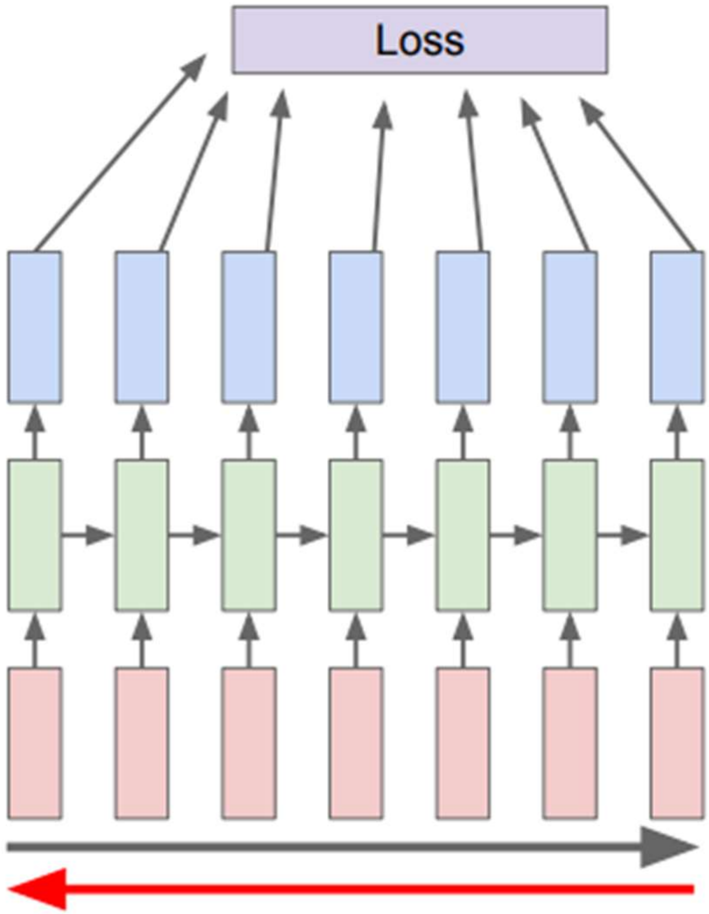
- Largest singular value > 1: Exploding Gradient ⟹ Go with gradient clipping, scale gradient if it's norm is too big
- Largest singular value < 1: Vanishing Gradient ⟹ Change RNN architecture

Backpropagation through time

Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient

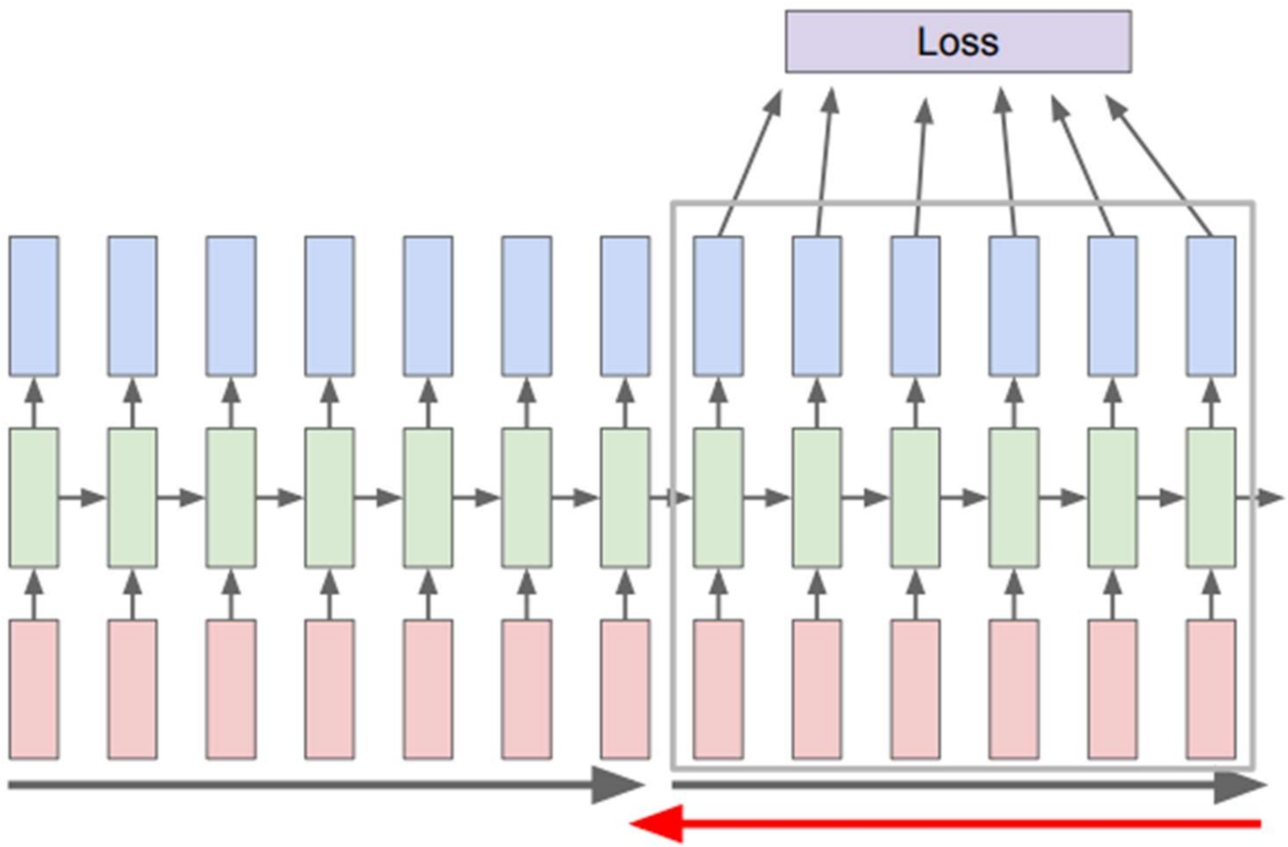


Truncated Backpropagation through time



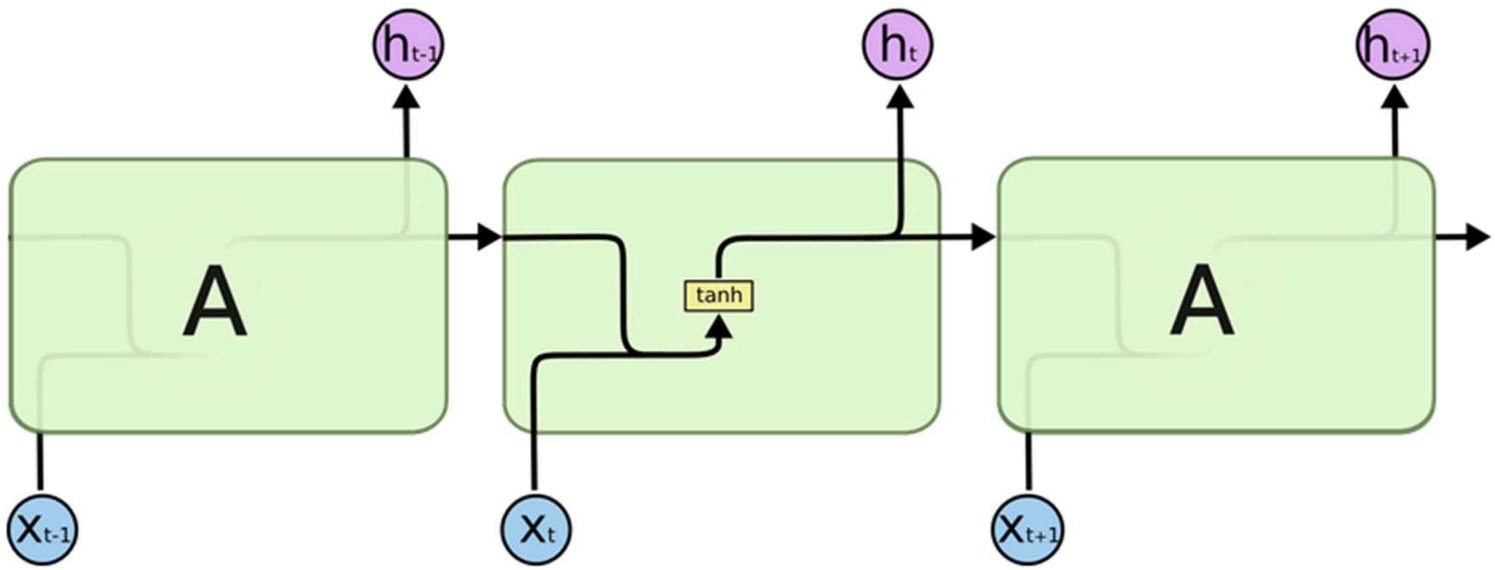
Run forward and backward through chunks of the sequence instead of whole sequence

Truncated Backpropagation through time



Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps

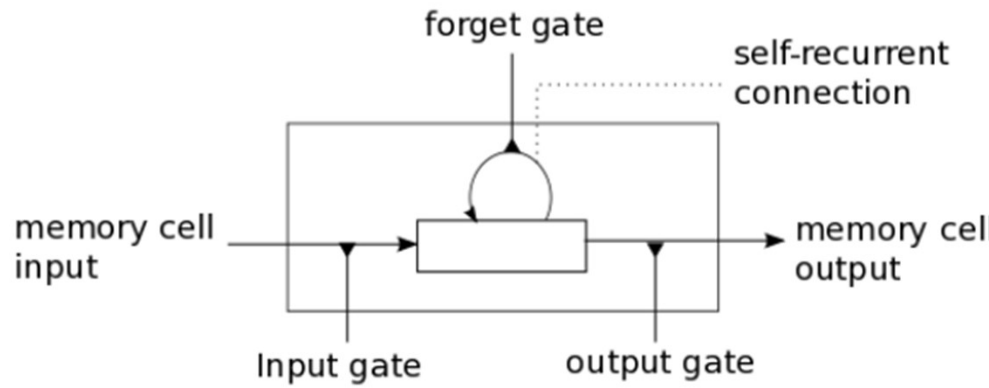
Standard RNN Architecture



The repeating module in a standard RNN contains a single layer.

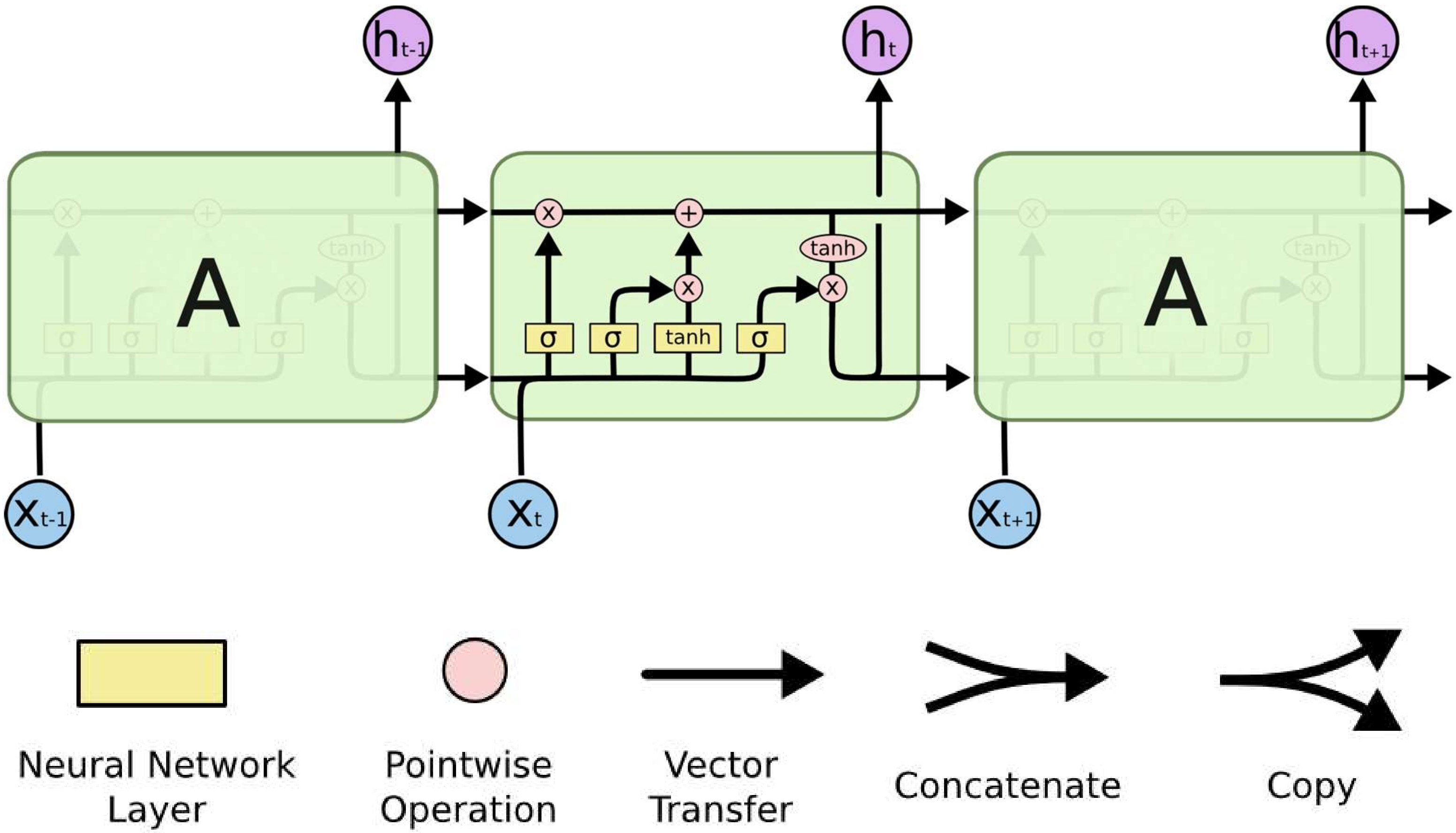
Long Short-Term Memory

- LSTM networks, add additional gating units in each memory cell.
    - Forget gate
    - Input gate
    - Output gate



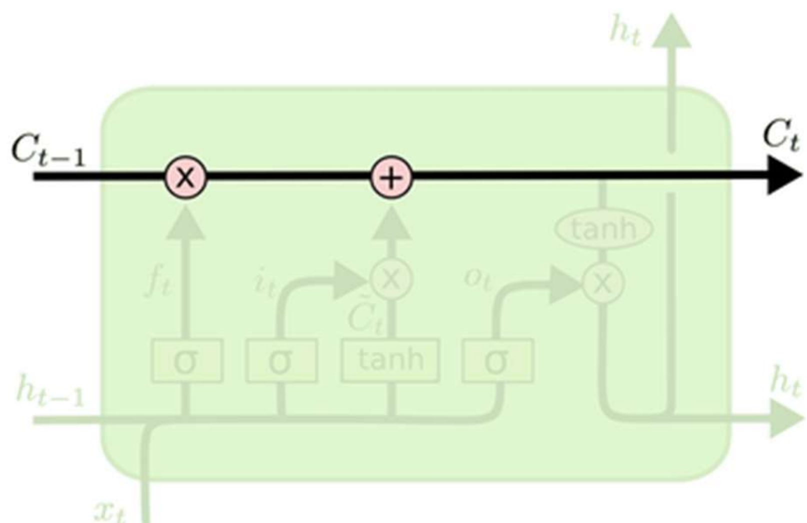
- Prevents vanishing/exploding gradient problem and allows network to retain state information over longer periods of time.

LSTM Network Architecture



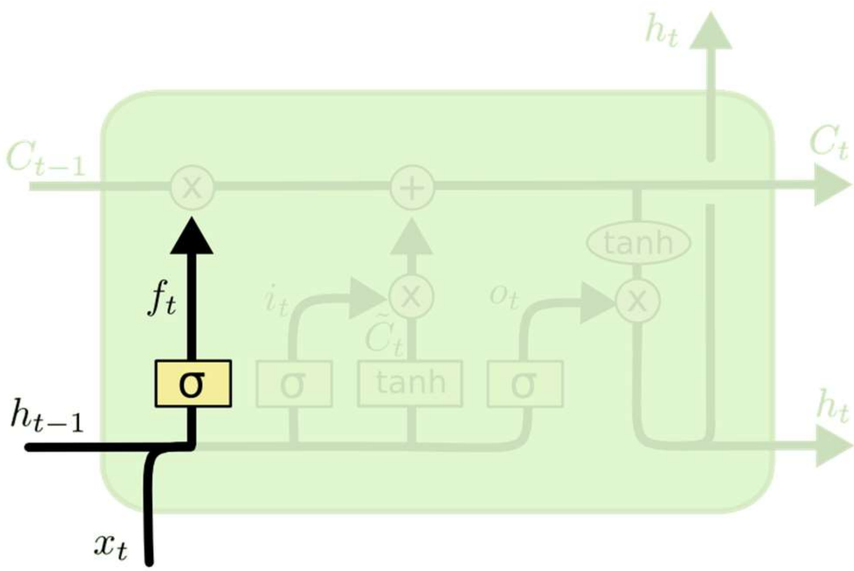
Cell State

- Maintains a vector C_t that is the same dimensionality as the hidden state, h_t
- Information can be added or deleted from this state vector via the forget and input gates.



Forget Gate

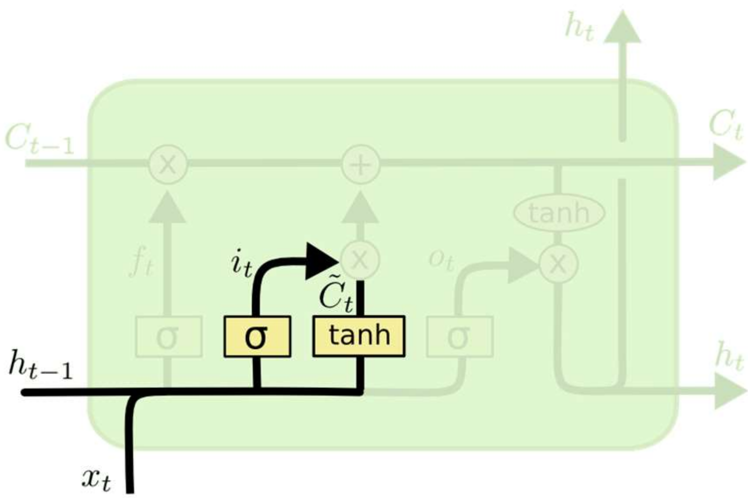
- Forget gate computes a 0-1 value using a logistic sigmoid output function from the input, x_t, and the current hidden state, h_{t-1}:
- Multiplicatively combined with cell state, "forgetting" information where the gate outputs something close to 0.



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

Input Gate

- First, determine which entries in the cell state to update by computing 0-1 sigmoid output.
- Then determine what amount to add/subtract from these entries by computing a tanh output (valued –1 to 1) function of the input and hidden state.

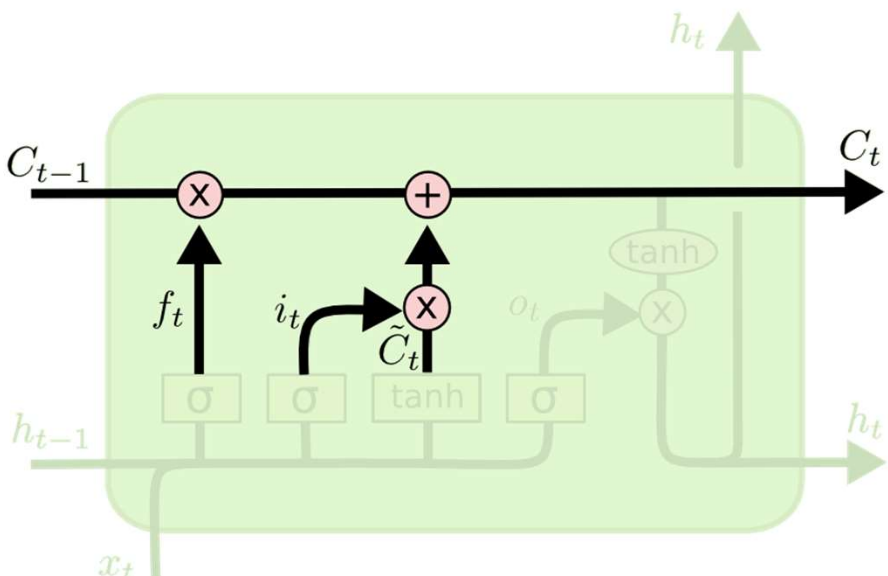


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

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

Updating the Cell State

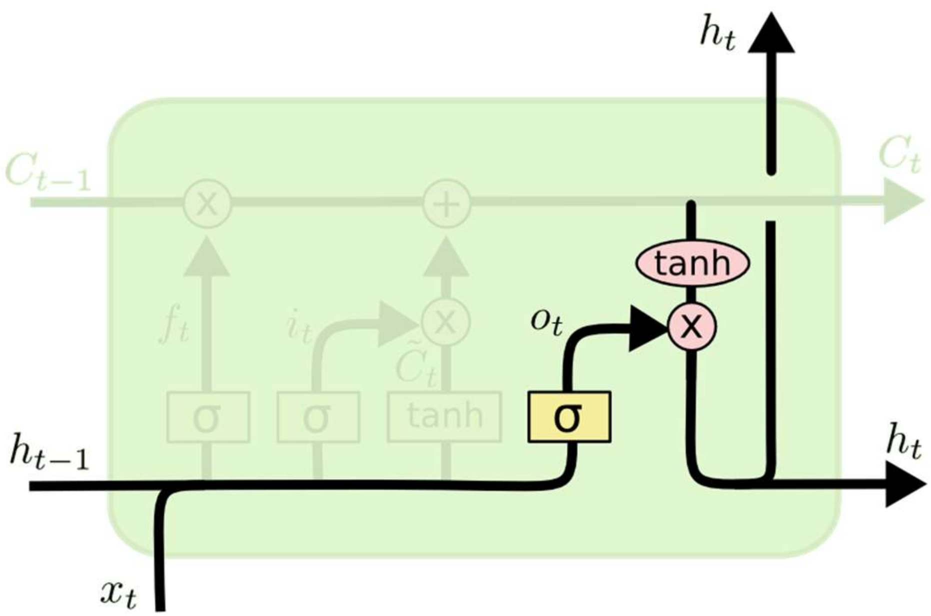
- Cell state is updated by using component-wise vector multiply to "forget" and vector addition to "input" new information.



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

Output Gate

- Hidden state is updated based on a "filtered" version of the cell state, scaled to –1 to 1 using tanh.
- Output gate computes a sigmoid function of the input and current hidden state to determine which elements of the cell state to "output".



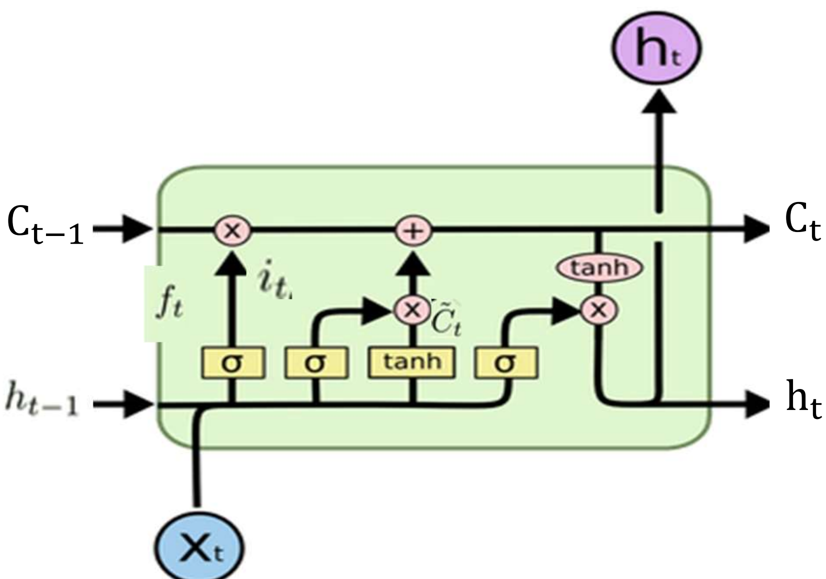
$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$

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

$$\begin{pmatrix} f_t \\ i_t \\ o_t \\ \tilde{C}_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W_g \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$C_t = f_t \odot c_{t-1} + i_t \odot \tilde{C}_t$$

$$h_t = o_t \odot \tanh C_t$$



LSTM [Hochreiter et al., 1997]

- **Forget gate** (f_t)**:** Defines how much of the previous state you want to let through.
- **Input gate** (i_t)**:** Defines how much of the newly computed state for the current input you want to let through.
- **Output gate** (o_t)**:** Defines how much of the internal state you want to expose to the external network.
- $\widetilde{C}_t$: "candidate" hidden state that is computed based on the current input and the previous hidden state.
- C_t**:** the internal memory of the unit. Intuitively it is a combination of how we want to combine previous memory and the new input.
- Given the memory C_t we finally compute the output **hidden state** h_t by multiplying the memory with the output gate.

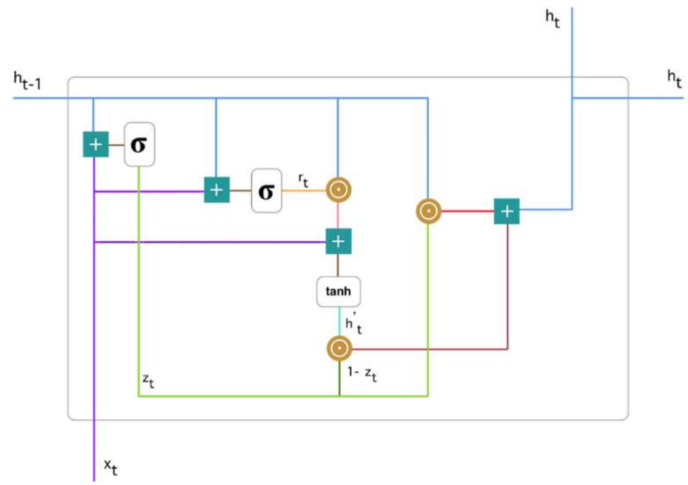
Do LSTMs solve the vanishing gradient problem?

- The LSTM architecture makes it easier for the RNN to preserve information over many timesteps.
  - e.g., if the f = 1 and the i = 0, then the information of that cell is preserved indefinitely.
  - By contrast, it's harder for vanilla RNN to learn a recurrent weight matrix W that preserves info in hidden state

- LSTM doesn't guarantee that there is no vanishing/exploding gradient, but it does provide an easier way for the model to learn long-distance dependencies.

Gated Recurrent Unit (GRU)

1. Update gate: $z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$
2. Reset gate: $r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$
3. New memory content: $h_t' = \tanh(Wx_t + r_t \odot Uh_{t-1})$
4. Final memory: $h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h_t'$



Learning phrase representations using rnn encoder-decoder for statistical machine translation, Cho et al. 2014

RNN Summary

- Can process any length input. Computation for step t can use information from many steps back.

- Vanilla RNNs are simple but don’t work very well - Common to use LSTM or GRU: their additive interactions improve gradient flow.

- Model size doesn’t increase for longer input - Same weights applied on every timestep, so there is symmetry in how inputs are processed.

- LSTMs, better at capturing long-term dependencies compared to vanilla RNNs, may still struggle with very long sequences or maintaining context over extended periods.

- Computationally Intensive, Difficult in Parallelization, Limited Interpretability.

- Architectures like Transformers with their self-attention mechanisms have addressed these.

Further Readings

- https://karpathy.github.io/2015/05/21/rnn-effectiveness/
- https://cs231n.stanford.edu/slides/2023/lecture_8.pdf
- https://cs231n.stanford.edu/slides/2020/lecture_10.pdf