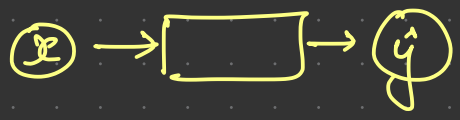
Recurrent Neural Networks

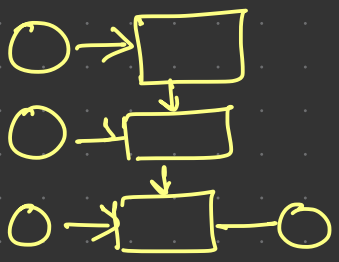
→ Sequential modelling and data

→ Sequencing model applications:

1) Binary Classification: (One to one)

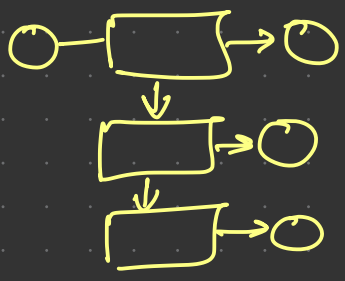


2) Sentiment classification (Many to one)

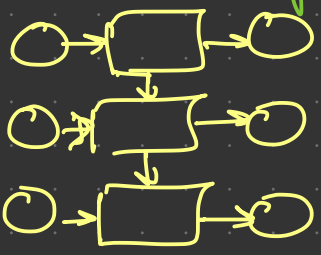


(One to many)

3) Image captioning



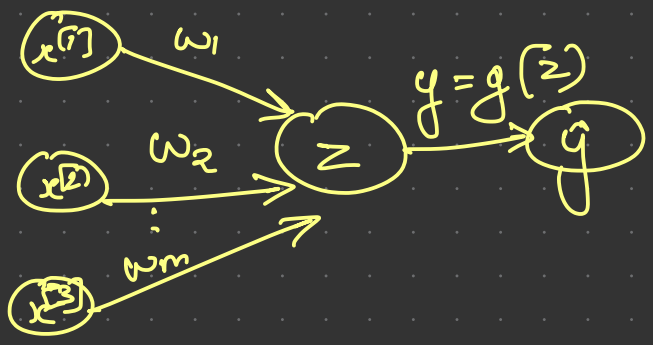
4) Machine translation (Many to many)



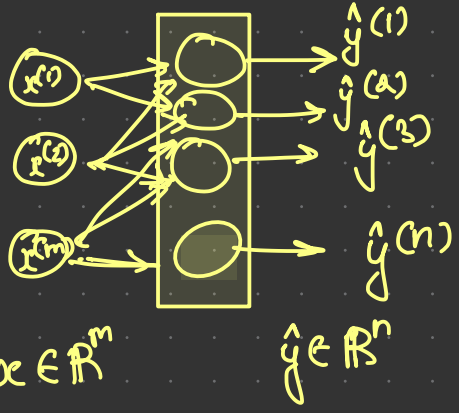
Sequences to another sequence

Neurons with recurrence:

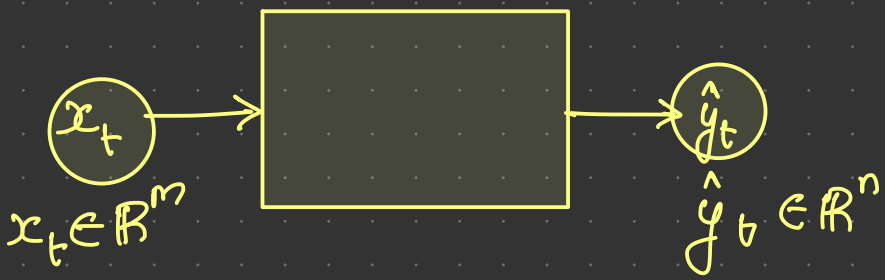
In perceptron,



Forward propagation

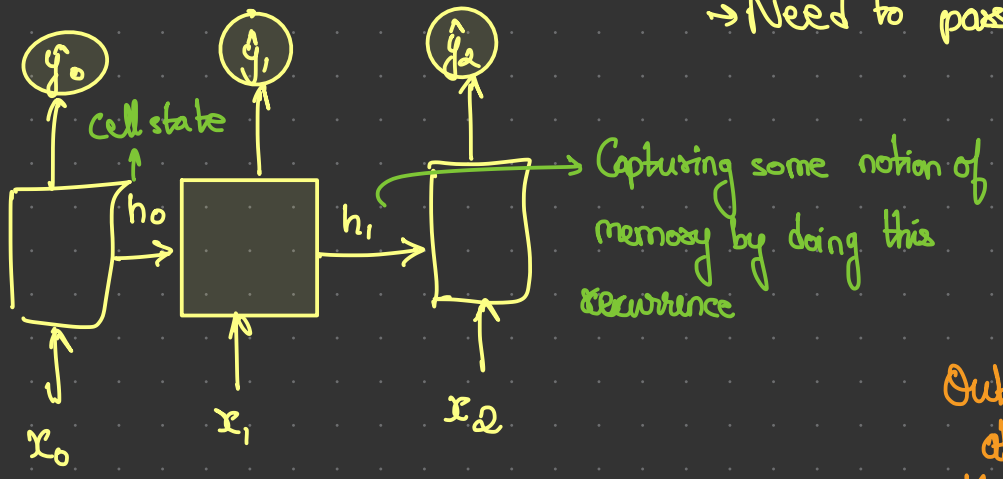


No notion of time sequences in above



If we have multiple timesteps:

→ Need to pass current info to future

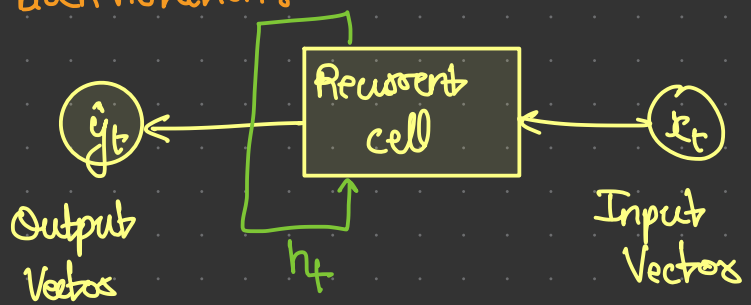


$$\hat{y}_t = f(x_t, h_{t-1})$$

Output at time t

Past Memory at prev. timestep

General block notation:



RNNs have internal state h_t, is updated at each time step

Recurrence relation:

$$h_t = f_w(x_t, h_{t-1})$$

h_t: cell state; f_w: function with weights w; x_t: input; h_{t-1}: old state

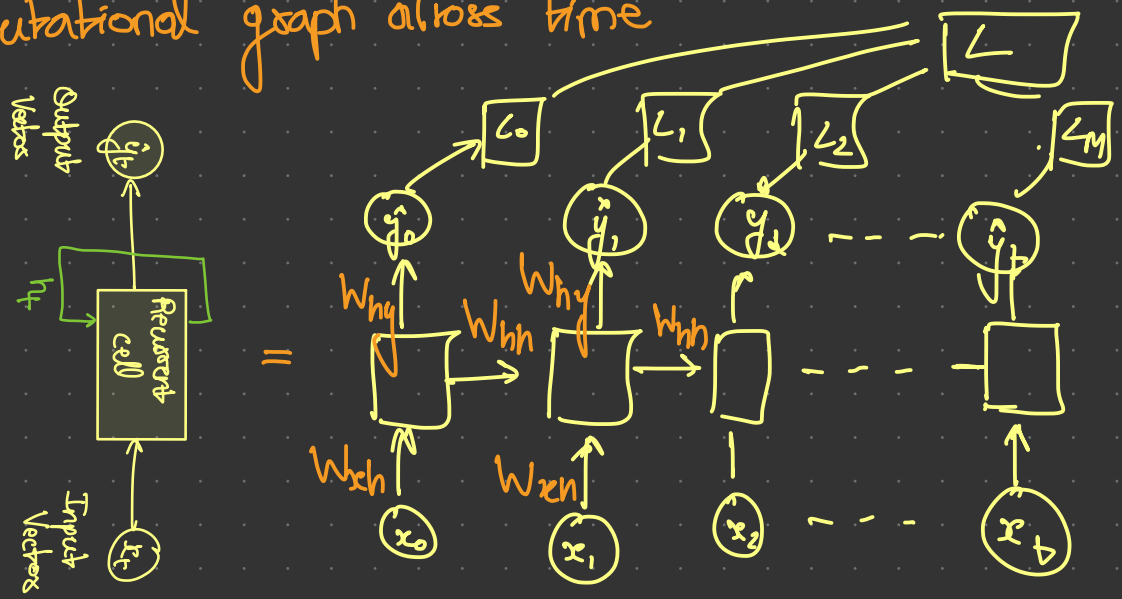
NOTE: Same params, functions are used every time step

$$h_t = \tanh\left(W_{hh}^T h_{t-1} + W_{xh}^T x_t\right) \Rightarrow \hat{y}_t = \hat{W}_{hy} h_t$$

x_t

respective weight

Computational graph across time



Design Criteria for sequence modelling in general:

1) Handle variable-length sequences
   → Ability to handle variable length of sentences etc
2) Track long-term dependencies
   To have a notion of memory
3) Maintain information about order:
   How prev. affects next
4) Share parameters across sequencies
   RNNs have all these 4

Solved examples:

Given | Predict

Represent language as NN:

"" → "" ""

Issues:
1) Usually work with mathematical operations ie. needed require it

Instead use, embedding:

↳ Transform indexes into vector of fixed size.

I. Vocabulary [Generate all possible words]

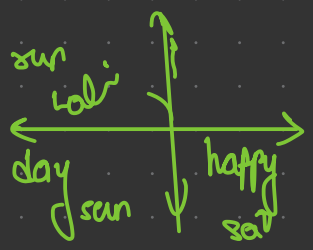
II. Indexing [□ → 1, □ → 2, ⋮, □ → N]

III. Embedding:

1) One shot embedding

i^{th} index is mapped corresponding.

2) Learned embedding



NOTE: Feedforwards loops can't handle variable lengths

1) RNNs have variable length

2), 3) Cell state dependent on prev.

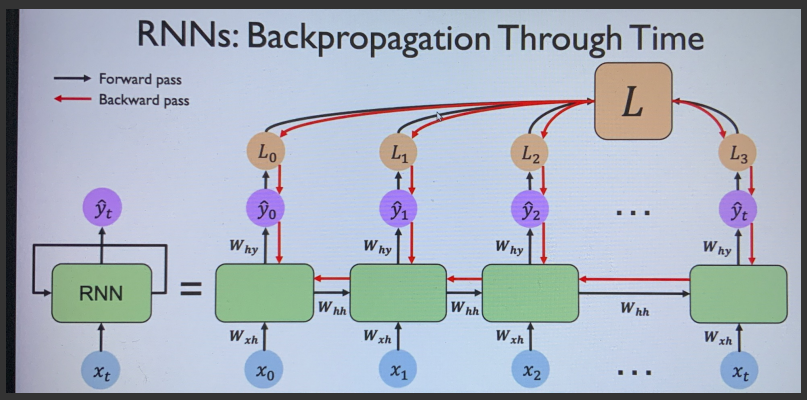
BACK PROPAGATION THROUGH TIME [BPTT]

In feedforward algorithms

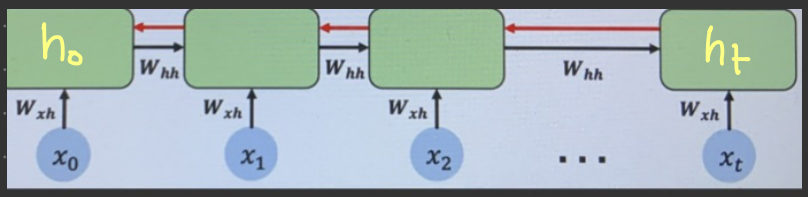
1) Take gradient of loss w.r.t. each parameter
2) Shift parameters to minimise loss

In RNNs,

1) Forward pass through time



Standard RNN Gradient Flows:



Involves many factors to get w.r.t. to h_0

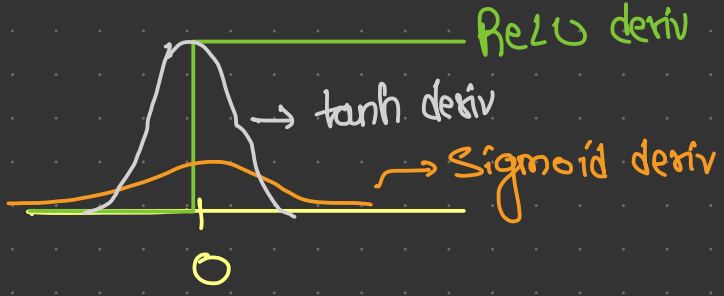
If values are too large:
[exploding gradients]
Gradient clipped to scale big gradients

If values are too small:
[vanishing gradients]
1. Activation fn
2. Weights init
3. Network Architecture

Why is vanishing a problem?
→ No learning [can't propagate error]
↳ End up biasing in short term, won't work in long term
→ Won't learn well when discussing long sentences etc
↳ long term dependence

(1) Activation Functions:

ReLU prevents f' from shrinking the gradients when $x > 0$



(2) Parameters init

$I_n = \begin{bmatrix} 1 & & 0 \\ & 1 & \\ 0 & & \ddots & 1 \end{bmatrix}$

Initialise biases to 0
Initialise weights to I
} Helps prevent weights from shrinking to zero

③ Gated Cells

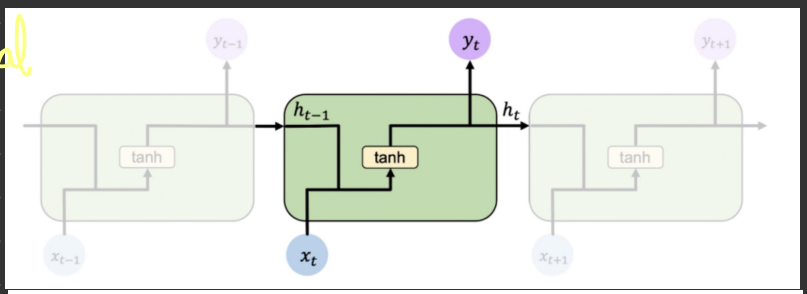
Using more complex recurrent unit with gates to control what information is passed through

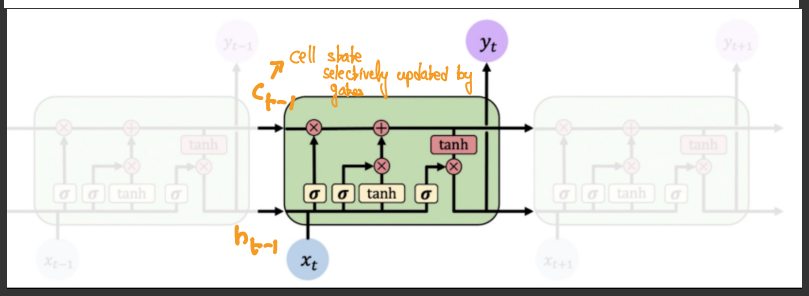
Gated cell
LSTM, GRU

↳ LSTMs: Long short term memory networks rely on a gate cell to track information through many time steps

LSTM Networks

- Contains computational blocks that control info flow
- → Able to track info throughout time step

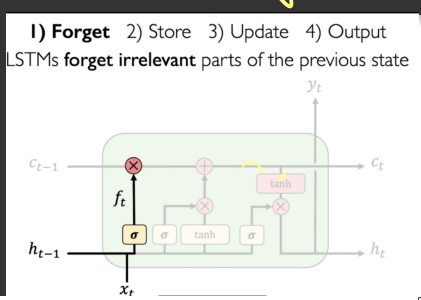


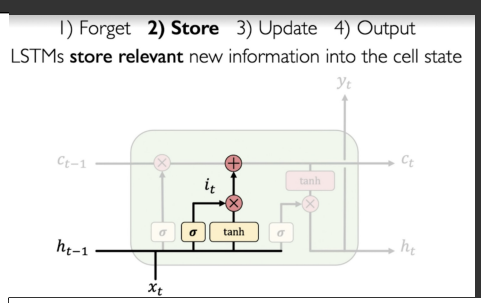


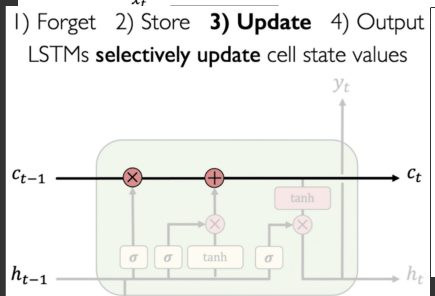
Info is added (or) subtracted through gates optionally let info through

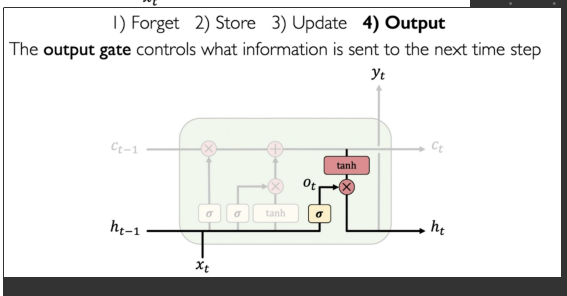
Steps: 1) forget
2) Store
3) Update
4) Output

→ Regulates info flow

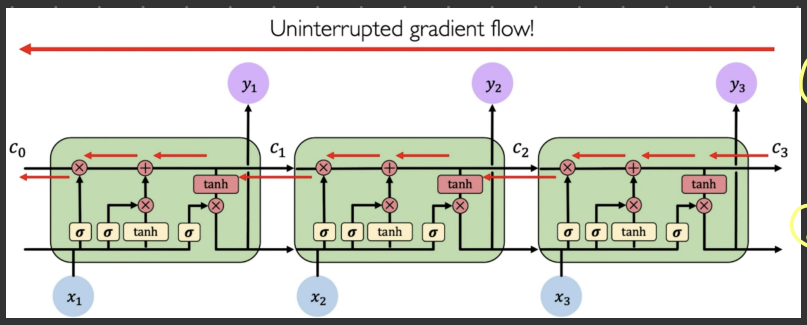








Allows uninterrupted gradient flow:



Key ideas of LSTM:

(1) Maintain separate cell state from outputted

(2) Use gates to control flow:

→ Forget: Get rid of irrelevant info
→ Store: Relevant info from current input
→ Selectively UPDATE cell state
→ OUTPUT gate returns filtered version of cell state

(3) Backprop with uninterrupted gradient flow

Practical Applications:

1) Music generation
2) Sentiment classification
3) Machine Translation
4) Weather modelling

Issues of RNNs:

1) Encoding bottleneck
2) Not efficient as its sequential [no parallelisation]
3) Not long memory [limited capacity]

Solutions:

[ATTENTION]

block is to learn from input which point and states to attend to [very efficient]

→ Provides learnable memory access
→ Only one pass needed (no back pass)