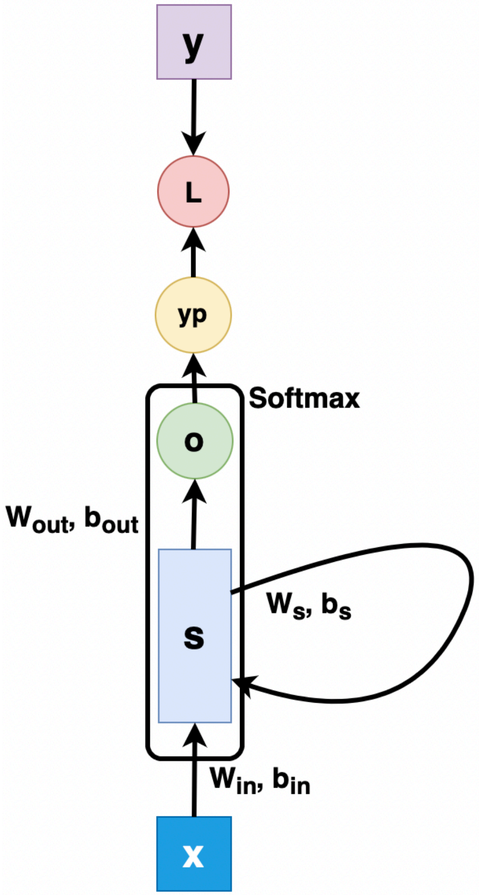
Recurrent Neural Networks

Perumadura De Silva

Recurrent Neural Networks

RNN architecture

- RNN Cell:
  - RNN Cell is designed to work with sequential inputs
  - The cell of an RNN takes an input and produces an output and a feedback of state
  - The output y_t of the cell depends on current state s_t and input x_t
  - Cell state s_t depends on all the previous states s_{t-k} (s_{t-1}, s_{t-2} … so on) and current input x_t
  - The inputs to the cells X is a sequence
  - Consider an RNN as an ANN that is copied along a time dimension
  - Each predecessor of the ANN chain passes its neurone state to a successor ANN



RNN cell

Recurrent Neural Networks

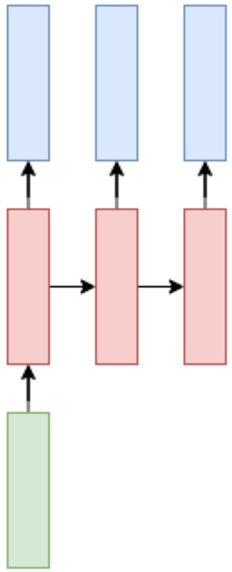
RNN architecture

- RNN Cell:
  - Each cell has 3 parameter (weight) matrices for inputs (W_{in}), outputs (W_{out}) and hidden states (W_s)
  - Cell equations:
    - Cell output: $y_t = f_{out}(W_{out} s_t)$
    - Cell state: $s_t = f_s(W_{in} x_t + W_s s_{t-1})$

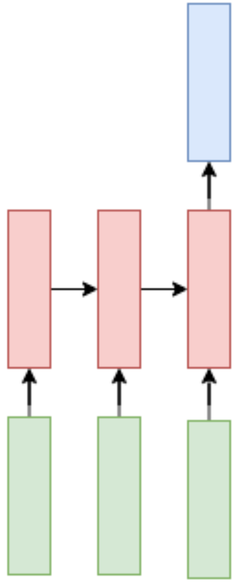
Recurrent Neural Networks

RNN architecture

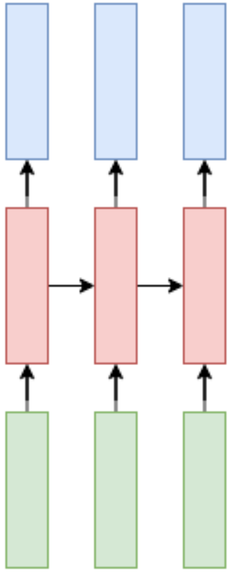
- Types of RNN layers



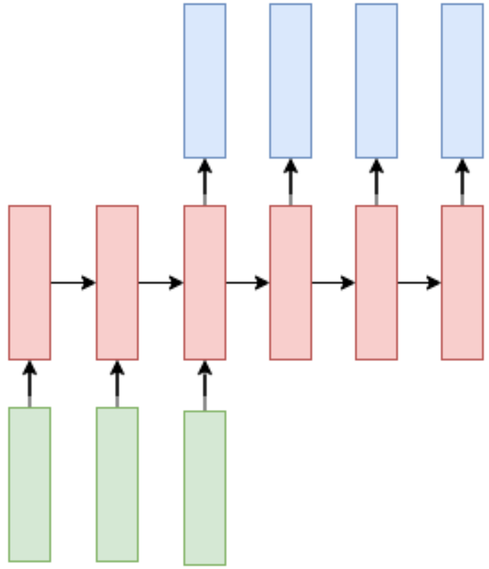
One to many mapping
E.g Image captioning



Many to one mapping
E.g Sentiment classification



Many to many equal size mapping
E.g Named Entity Recognition



Many to many different size mapping
E.g Machine translation

Recurrent Neural Networks
Training a RNN

- Training algorithm: Back Propagation Through Time (BPTT)
  - When training we do not train each recurrent layer individually with back propagation
  - Rather we calculate the error at each time step (or at the last step) and propagate it through time
  - Note: The recurrent layer weights W_{in}, W_{out}, W_s are shared across the time steps

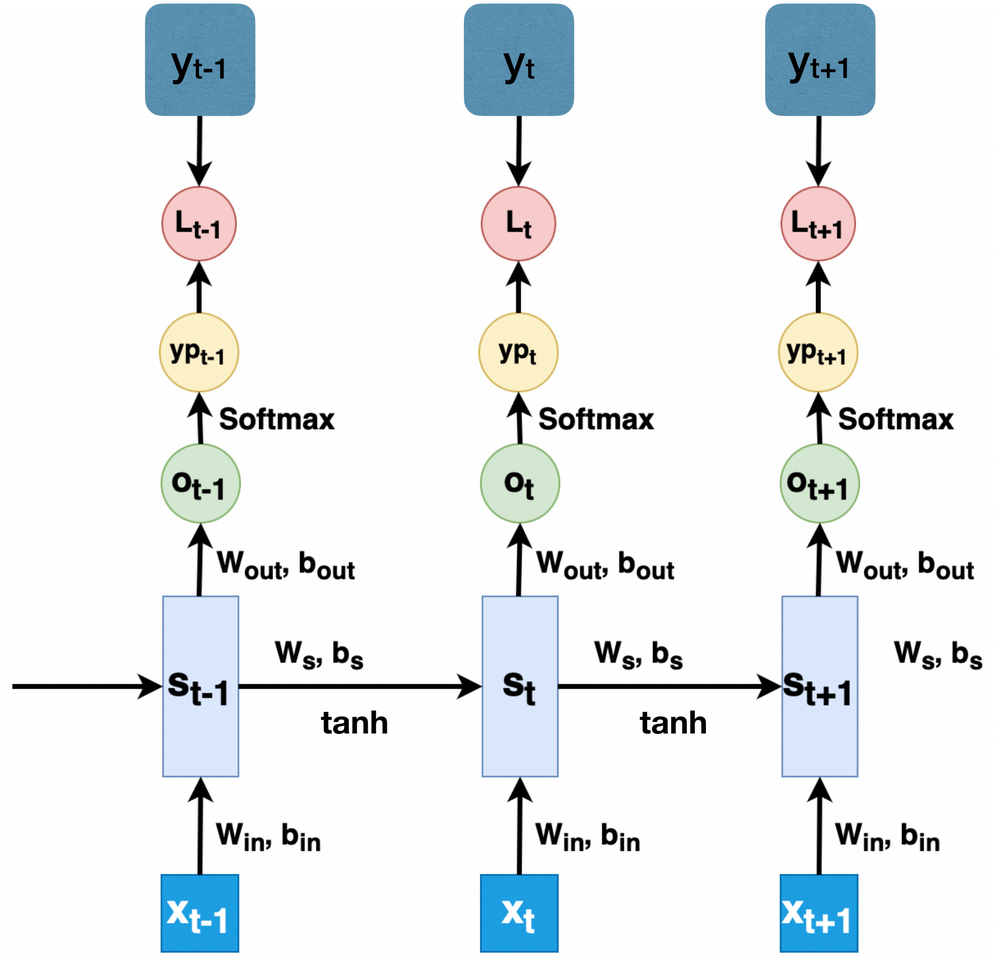
Recurrent Neural Networks

RNN Dimensions

- d_{in}: input dimension size
- d_{out}: output dimension size
- d_s: hidden dimension size
- x: inputs with size $1 \times d_{in}$
- s: hidden state with size $1 \times d_s$
- $\hat{y}$: outputs with size $1 \times d_{out}$
- T: sequence length
- t: time step in the sequence
- W_{in}: input weights of $d_{in} \times d_s$ size
- b_{in}: input bias of $1 \times d_s$
- W_s: state weights of $d_s \times d_s$ size
- b_s: state bias of $1 \times d_s$ size
- W_{out}: output weights of $d_s \times d_{out}$ size
- b_{out}: output weights of $1 \times d_{out}$ size

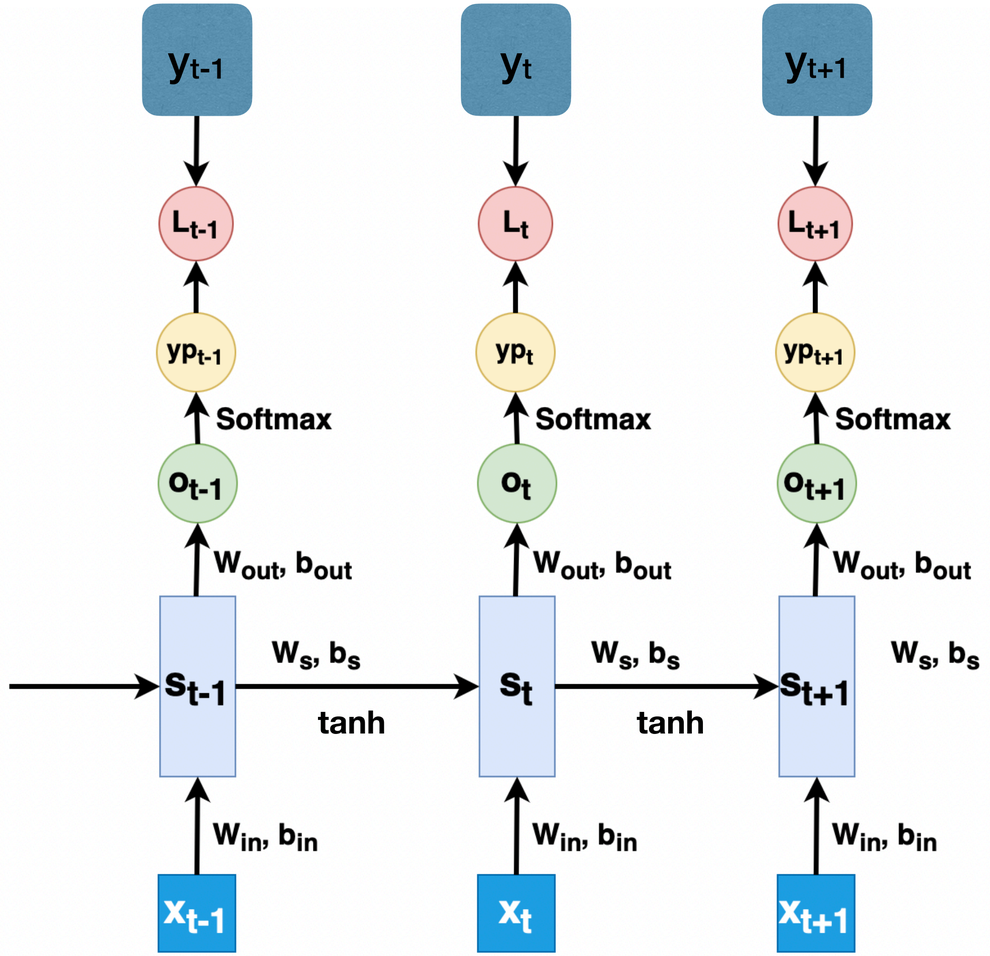
Recurrent Neural Networks
Training a RNN

- Take the scenario in the figure
  - Input to output layer: $o_t = W_{out} s_t + b_{out}$
  - Cell output: $y_t = f_{out}(o_t)$
  - Cell state: $s_t = f_s(W_{in} x_t + b_{in} + W_s s_{t-1} + b_s) = f_s([x_t; s_{t-1}]W + b_{in} + b_s)$
  - Error: $L_t = \frac{1}{2}(\hat{y}_t - y_t)^2$
- Take the intermediate state s_t
  - The error gradients w.r.t to the output weights:
    - $\frac{\partial L_t}{\partial W_{out}} = \frac{\partial E_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial o_t} \frac{\partial o_t}{\partial W_{out}}$
  - Likewise total error gradient w.r.t W_{out} is given by:
    - $\frac{\partial L}{\partial W_{out}} = \sum_{i=1}^{T} \frac{\partial L_i}{\partial W_{out}}$



Recurrent Neural Networks
Training a RNN

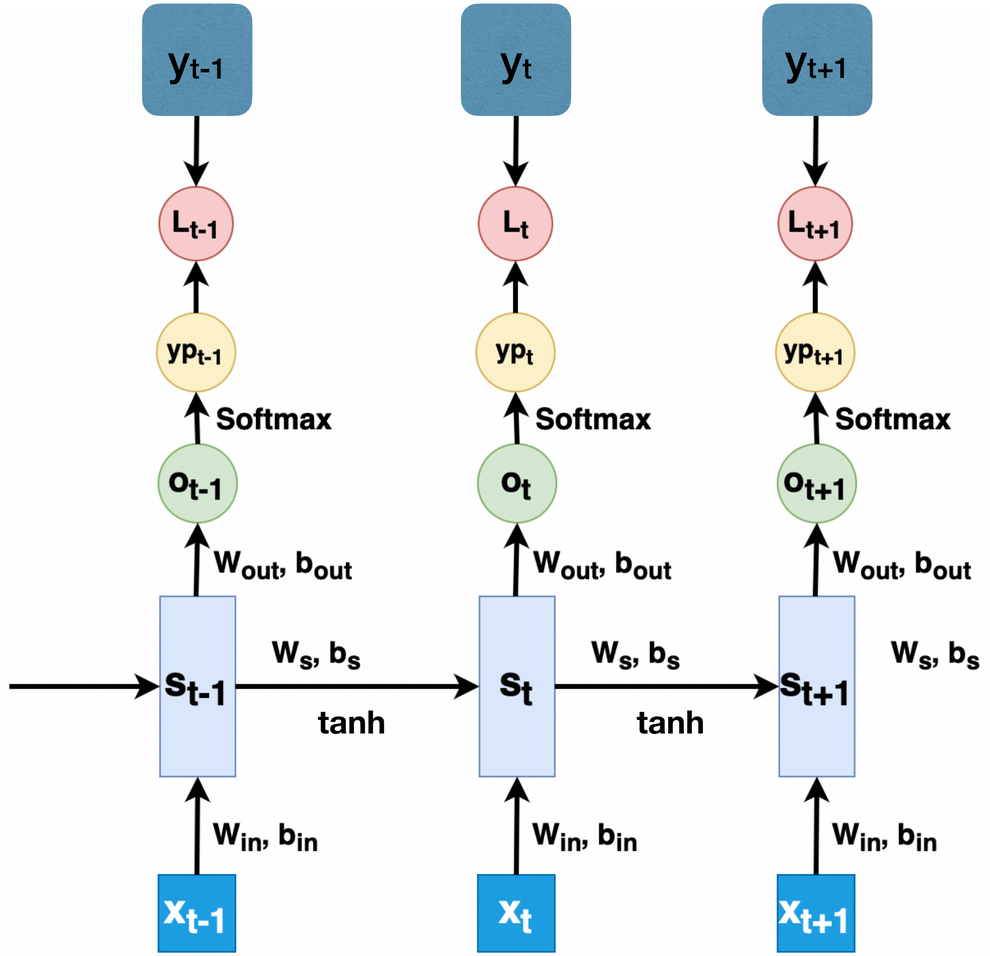
- BPTT continues:
  - Take the intermediate state s_{t+1}
    - The error gradients w.r.t to the hidden weights:
    - $\frac{\partial L_{t+1}}{\partial W_s} = \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial s_{t+1}} \frac{\partial s_{t+1}}{\partial W_s}$; Explicit gradient
    - However, $\frac{\partial s_{t+1}}{\partial W_s}$ has an implicit dependency on previous time steps $\frac{\partial s_t}{\partial W_s}$
    - $\frac{\partial L_{t+1}}{\partial W_s} = \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial s_{t+1}} \frac{\partial s_{t+1}}{\partial s_t} \frac{\partial s_t}{\partial W_s}$



Recurrent Neural Networks

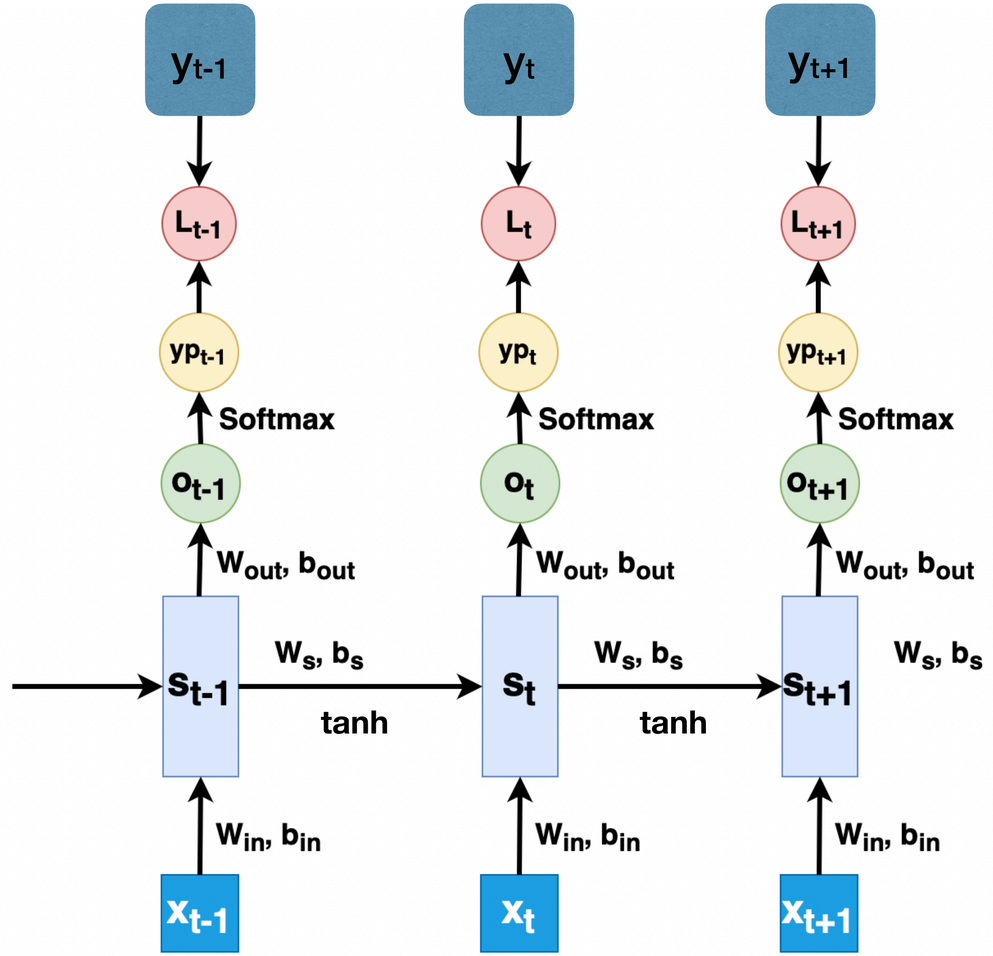
Training a RNN

- BPTT continues:
  - Take the intermediate state s_{t+1}
    - $\frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial s_{t+1}} \frac{\partial s_{t+1}}{\partial W_s} - >$
    - $\frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial s_{t+1}} \frac{\partial s_{t+1}}{\partial W_s} + \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial s_{t+1}} \frac{\partial s_{t+1}}{\partial s_t} \frac{\partial s_t}{\partial W_s} + \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial s_{t+1}} \frac{\partial s_{t+1}}{\partial s_t} \frac{\partial s_t}{\partial s_{t-1}} \frac{\partial s_{t-1}}{\partial W_s}$
    - $\frac{\partial L_{t+1}}{\partial W_s} = \sum_{k=1}^{t+1} \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial s_{t+1}} \frac{\partial s_{t+1}}{\partial s_k} \frac{\partial s_k}{\partial W_s}$
    - $\frac{\partial L_3}{\partial W_s} = \frac{\partial L_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial W_s} + \frac{\partial L_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial W_s} + \frac{\partial L_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial W_s}$



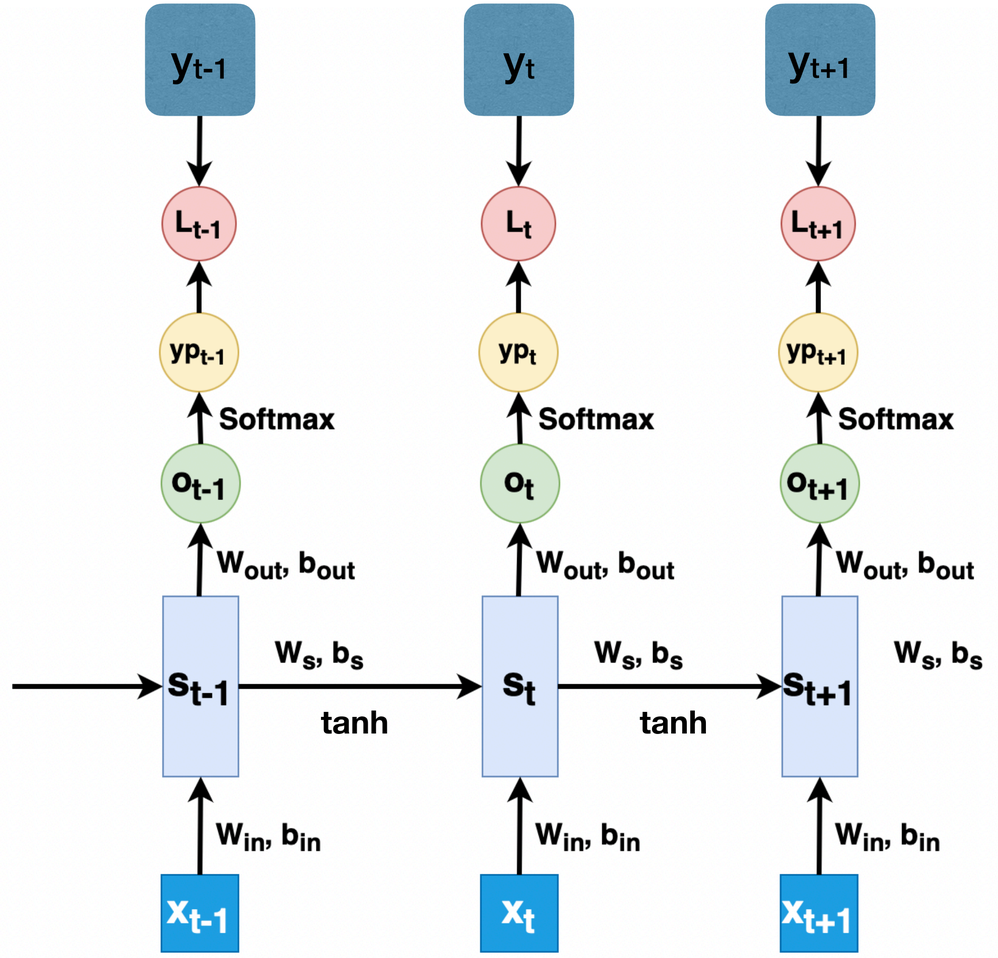
Recurrent Neural Networks
Training a RNN

- BPTT continues:
  - Take the intermediate state s_{t+1}
    - There is an inner chain in the equation:
    - $\frac{\partial L_{t+1}}{\partial W_s} = \sum_{k=1}^{t+1} \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial s_{t+1}} \frac{\partial s_{t+1}}{\partial s_k} \frac{\partial s_k}{\partial W_s}$;
    - Inner chain: $\frac{\partial s_{t+1}}{\partial s_t} \frac{\partial s_t}{\partial s_{t-1}} \dots \frac{\partial s_{k+1}}{\partial s_k} = \Pi_{j=k}^{t} \frac{\partial s_{j+1}}{\partial s_j}$
    - $\frac{\partial L_{t+1}}{\partial W_s} = \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial s_{t+1}} \frac{\partial s_{s+1}}{\partial W_s} + \sum_{k=1}^{t} \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial s_{t+1}} (\Pi_{j=k}^{t} \frac{\partial s_{j+1}}{\partial s_j}) \frac{\partial s_k}{\partial W_s}$
    - Likewise total error gradient w.r.t W_s is given by: $\frac{\partial L}{\partial W_s} = \sum_{i=1}^{T} \frac{\partial L_i}{\partial W_s}$



Recurrent Neural Networks
Training a RNN

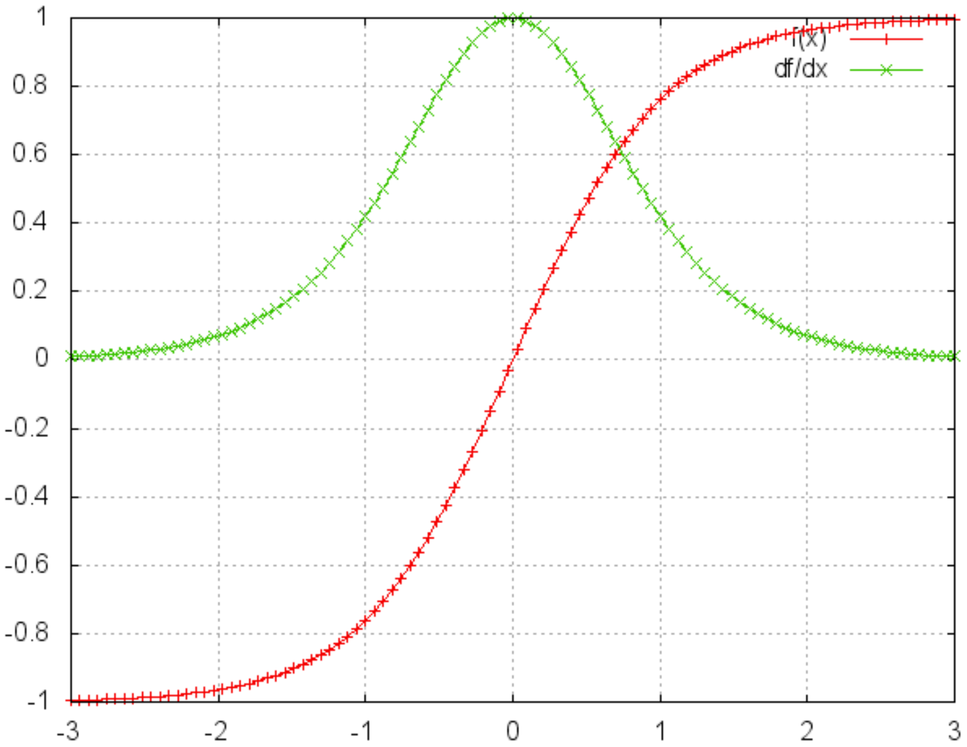
- BPTT continues:
  - Take the intermediate state s_3
    - The error gradients w.r.t to the input weights:
      - $\frac{\partial L_3}{\partial W_{in}} = \frac{\partial L_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial W_{in}}$
      - Similar to the case of hidden weights, here also $\frac{\partial s_3}{\partial W_{in}}$ has implicit dependencies.
      - $\frac{\partial L_3}{\partial W_{in}} = \frac{\partial L_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial W_{in}} + \frac{\partial L_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial W_{in}} + \frac{\partial L_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial W_{in}}$
      - $\frac{\partial L_{t+1}}{\partial W_{in}} = \sum_{k=1}^{t+1}\frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}}\frac{\partial \hat{y}_{t+1}}{\partial s_{t+1}}\frac{\partial s_{t+1}}{s_k}\frac{\partial s_k}{\partial W_{in}}$
    - Likewise total error gradient w.r.t W_{in} is given by:
      $\frac{\partial L}{\partial W_{in}} = \sum_{i=1}^{T}\frac{\partial L_i}{\partial W_{in}}$



Recurrent Neural Networks

Issues with vanilla RNN

- Vanilla RNN models have issues with learning long term dependencies
  - Long term dependencies: dependencies at time steps further from first time step (In very long sequences)
  - The vanishing gradients occur when the activations are sigmoid or tanh
  - Take: $\frac{\partial s_{t+1}}{\partial s_t} = diag(f_s^{'}(W_{in}x_{t+1} + b_{in} + W_s s_t + b_s))W_s$
  - When inputs are large, gradients get significantly small; $\Pi_{j=k}^{t}\frac{\partial s_{j+1}}{\partial s_j}$ leads to a vanishing gradient
  - When W_s is large, W_s will overpower $f_s^{'}(.)$ in $\Pi_{j=k}^{t}\frac{\partial s_{j+1}}{\partial s_j}$; leads to exploding gradient



tanh(x) function against $\frac{df}{dx}$

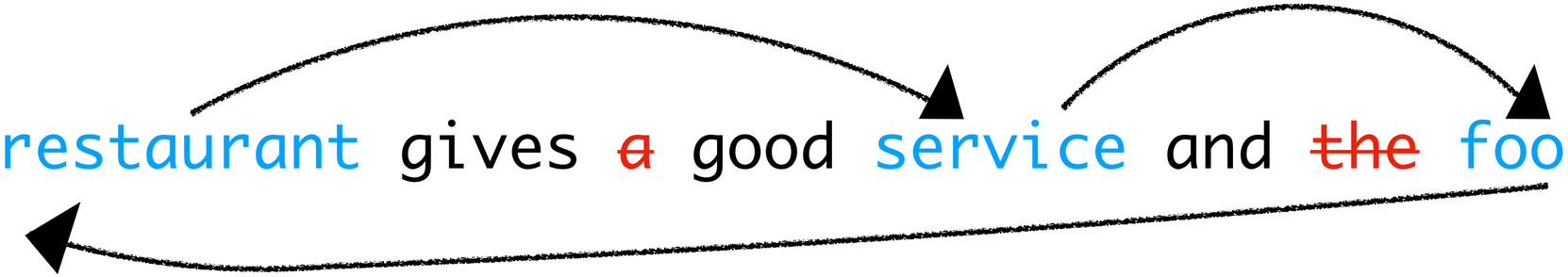
Recurrent Neural Networks
Issues with vanilla RNN

- Vanishing gradient is a difficult problem than explosive gradients
  - When gradients explode; loss will become NaN
  - However vanishing gradient is hard to observe through loss
- Solutions:
  - Exploding gradient: Clip the gradients
    - Min-Max clipping: Threshold gradients to be in a range
    - Norm clipping: given a threshold norm and type of norm ($p = 1,2,...\infty$)
      - $$\frac{\frac{\partial E}{\partial W_s} * Threshold_{norm}}{norm(\frac{\partial E}{\partial W_s})}$$
  - Vanishing gradient:
    - As one solution, a function such as ReLU can be used
    - Another better solution is to use an Long Short Term Memory (LSTM) cell instead

Recurrent Neural Networks

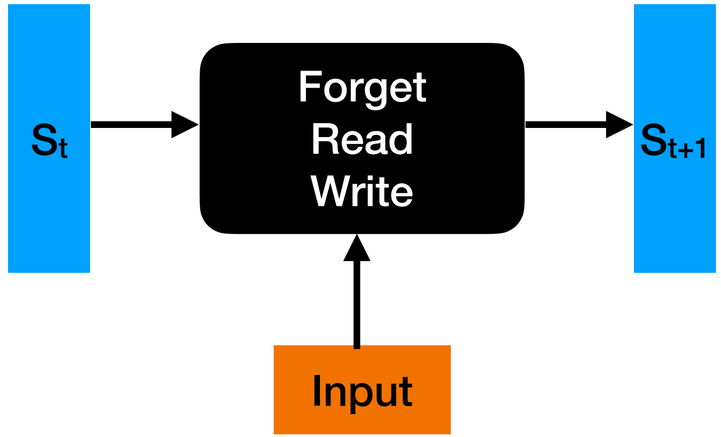
Problem setup

- Vanilla RNN try to carry all the information given at each time step
- Idea:
  - Selectively remove/forget, read and write information at each time step to regulate the information flow
  - E.g: ~~The~~ restaurant gives ~~a~~ good service and ~~the~~ food ~~is~~ delicious



Recurrent Neural Networks
Problem setup

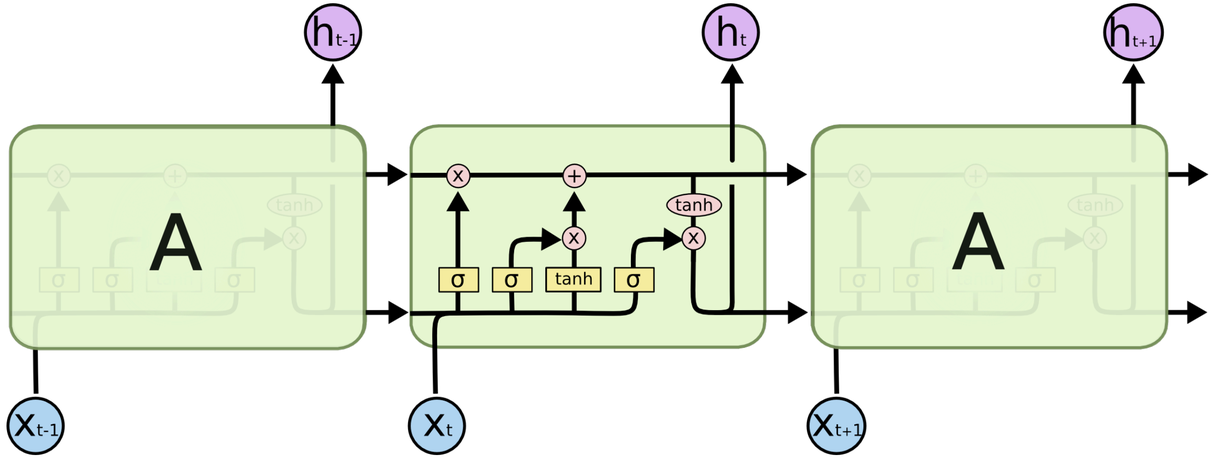
- Forget, read and write mechanisms:
  - Simply an MLP with sigmoid activation (0…1) that acts as a gate
  - learns which parameters to remove (0) and keep (1) from an input
  - $Output = gate \circ Input$; 0s at removed elements
- Feature extraction:
  - A tanh activation extracts features; which later selectively forwarded using the gates



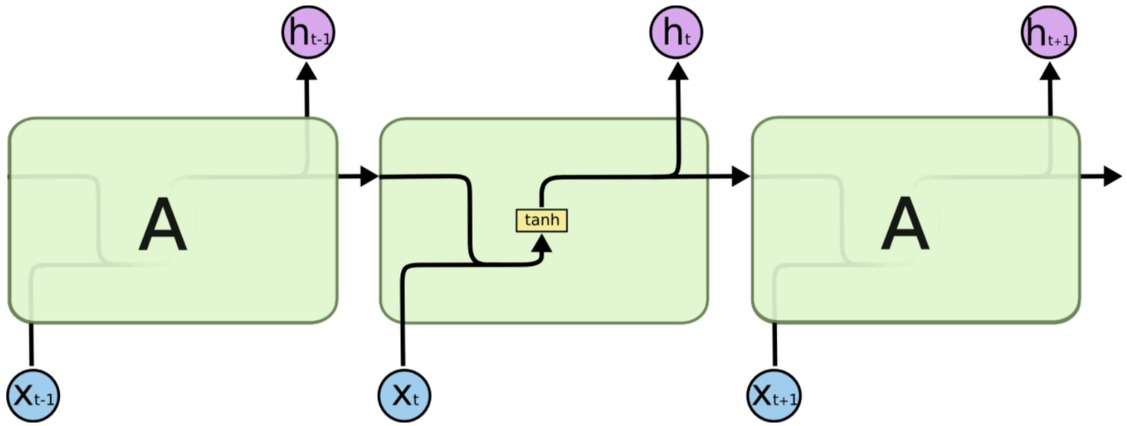
Recurrent Neural Networks

Long Short Term Memory (LSTM) networks and its variants

- LSTM Cell Compared to the Vanilla RNN Cell
  - LSTM Cell has 4 neural networks with sigmoid and tanh activations
  - Consider LSTM Cell as a system of pipes, gates (Sigmoid) and filters/feature extractors (Tanh).
  - Filters and gates operate in between cell state pipe C, hidden state pipe h and cell input pipe x



Ref.2: LSTM Cell

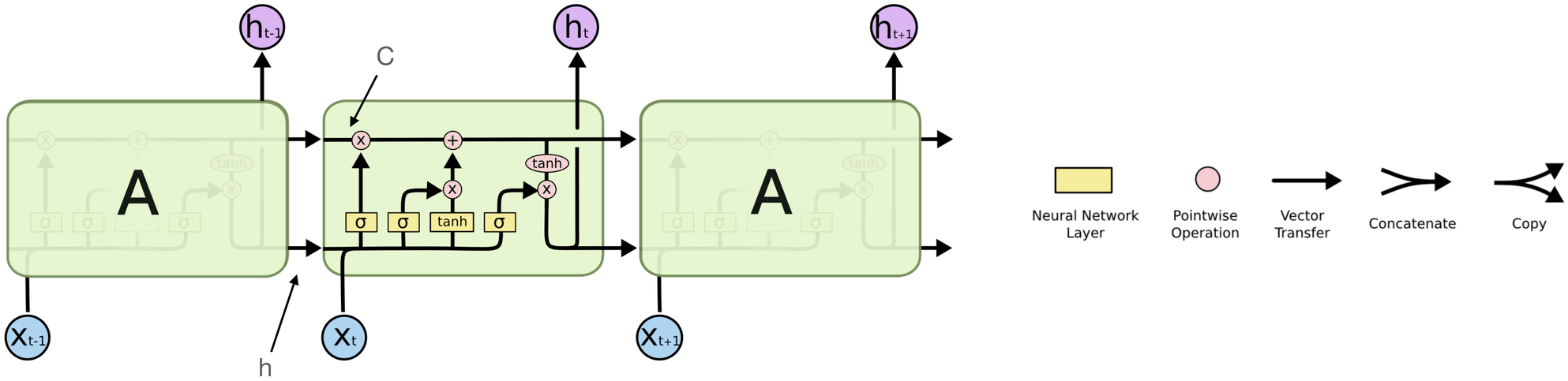


Ref.2: Vanilla RNN Cell

Recurrent Neural Networks

Long Short Term Memory (LSTM) networks and its variants

- The cell state acts as a long term memory that carry the information from all the previous time steps
- Hidden state acts as a working memory:
  - Predictions at each time step (application specific)
  - Regulate the next cell state

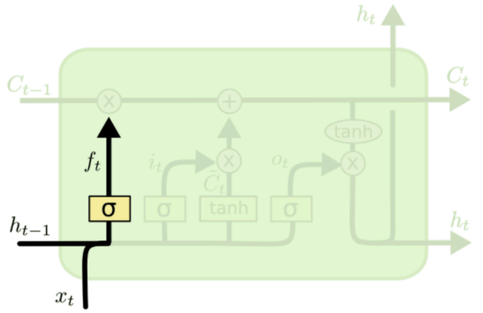


Ref.2: LSTM Cell

Recurrent Neural Networks

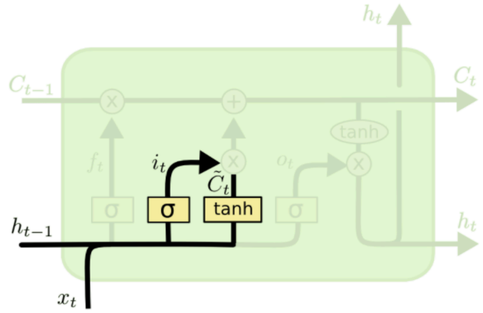
Long Short Term Memory (LSTM) networks and its variants

- Forget gate decides which information to keep (1) and throw (0) from the previous cell state
- Input gate decides which new information to use to update the current cell state
- Input filter predicts a temporary cell state and add to the previous cell state to produce current cell state
- Similar to input gate, output gate regulates current cell state to produce new hidden state



Ref.2: Forget gate

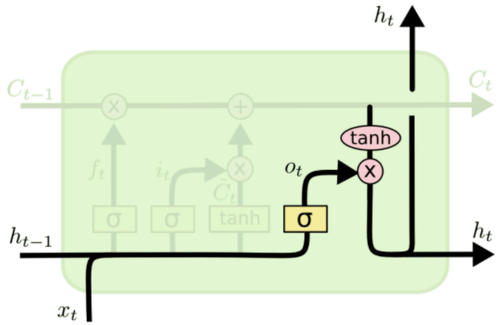
$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$



Ref.2: Input gate

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

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



Ref.2: Output gate

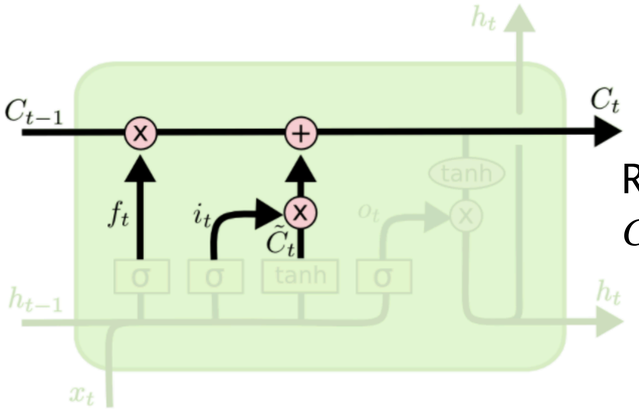
$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

$$h_t = o_t \circ tanh(C_t)$$

Recurrent Neural Networks

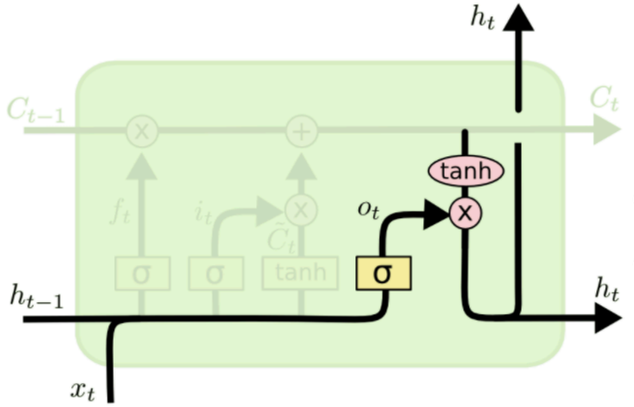
Long Short Term Memory (LSTM) networks and its variants

- Finally the new cell state is computed by:
  - Removing the information using the forget gate f_t
  - Adding the chosen information using input gate $i_t \approx 1 - f_t$
- The hidden state is updated by regulating the cell state



Ref.2: Cell state update

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



Ref.2: Output gate

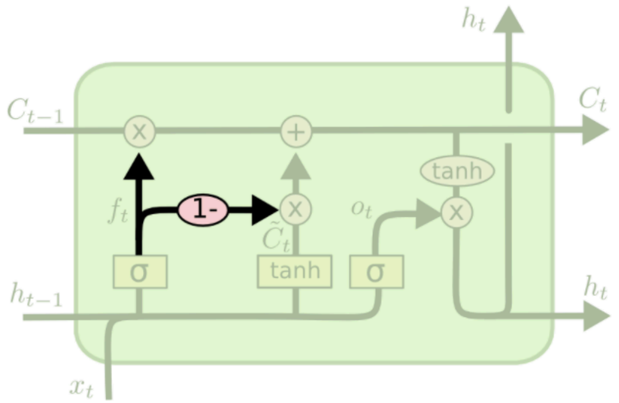
$o_t = \sigma(W_o[s_{t-1}, x_t] + b_o)$

$h_t = o_t \circ tanh(C_t)$

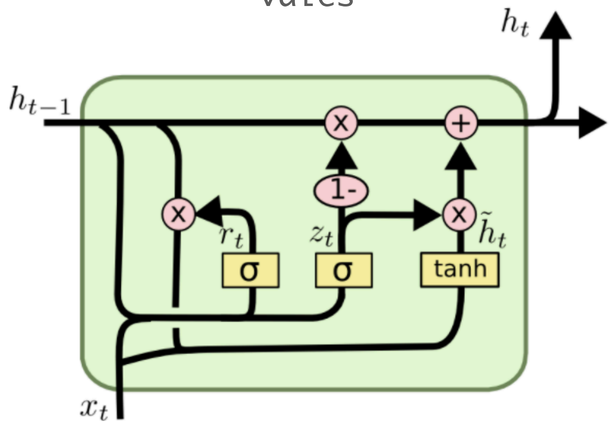
Recurrent Neural Networks

Long Short Term Memory (LSTM) networks and its variants

- Variants if LSTM Cells:
  - Cell with combined inputs and forget valves
    - $i_t = 1 - f_t$
  - Gated Recurrent Unit (GRU)
    - This is a simplified version of LSTM that combines input and forget gates: $i_t = 1 - f_t$
    - as well as hidden and cell state pipes: h_t
    - Reset gate learns to reset (completely) the previous hidden state
      - E.g: ~~The service was good,~~ but the food was awful.
    - Update gate decides what new information to use for cell state update
    - Inverse on update gate is the forget gate



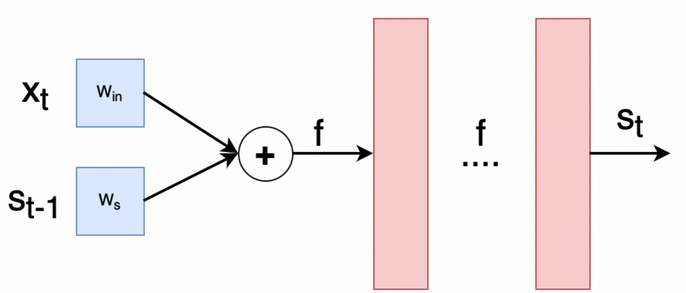
Ref.2: Combined input and foget vales



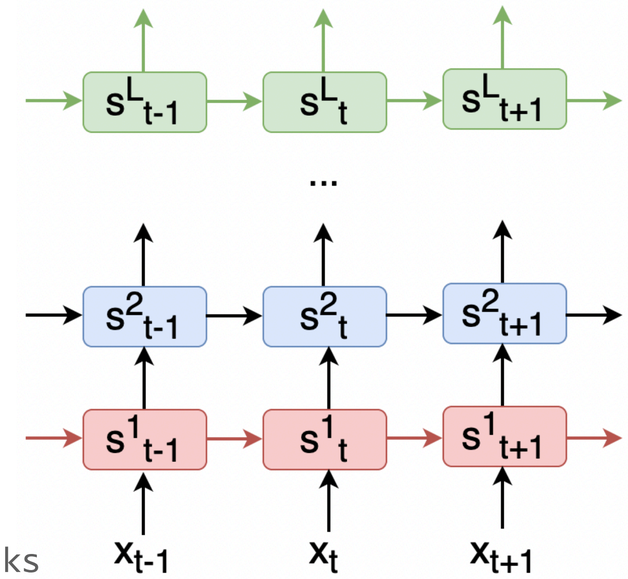
Ref.2: Gated Recurrent Unit

Recurrent Neural Networks
Configurations of RNNs/LSTMs

- Deep RNN/LSTM networks
  - In deep learning, stacking layers allows hierarchical feature extraction
  - Same concept is applied to RNNs in two different ways:
    - Increase the number of layers in each input, state and output layers
    - Stack multiple RNNs connected by hidden state



Deep hidden layer: f: nonlinear function, s: state, x: input

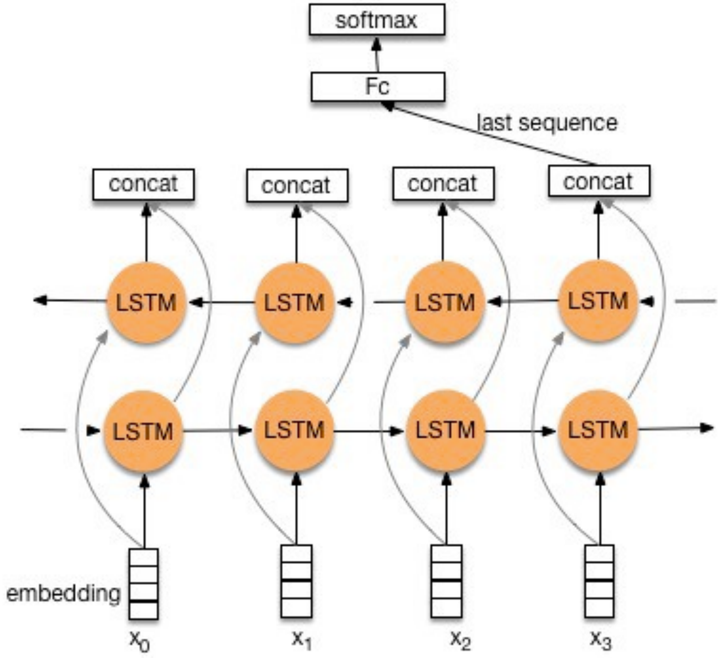


Stacked RNN with L layers. Pass the hidden states from top to bottom as inputs

How to Construct Deep Recurrent Neural Networks
https://arxiv.org/pdf/1312.6026.pdf

Recurrent Neural Networks
Configurations of RNNs/LSTMs

- Bi-Directional LSTM networks:
  - These type of networks take the input sequence in both forward and backward direction
  - Advantage: an intermediate time step has information about both future and past time steps
  - Disadvantage: A complete input sequence is required; not suitable for tasks such as speech recognition



Ref.3: Bidirectional RNN for sequence classification

Recurrent Neural Networks
Natural language processing

- RNNs are widely used for natural language processing (NLP) tasks:
  - Sentiment classification: Classify if a sentences is negative or positive: Many to One architecture
  - Neural machine translation (NMT): Translate from one language to the other: Many to Many architecture
  - Question and answering (Chatbots): Many to Many architecture
- However, neural networks work with numbers not text:
  - Given a text dataset determine the vocabulary size
  - Assign a unique value (Index) to every word (Tokenisation)
  - Option-1: One-hot encode the words in a sentence using the vocabulary index
  - Option-2: One-hot encode and train a word embedding model
    - Embedding vector is a dense, real valued vectors that is learned to represent a word

Recurrent Neural Networks
Natural language processing

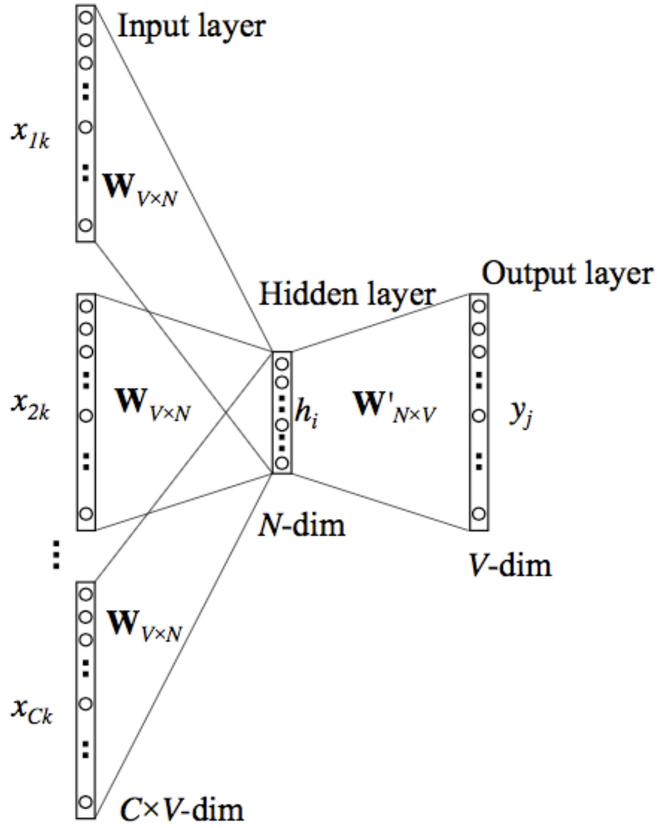
- Layer-1: Text preprocessing layer (Text tokenisation)
  - The purpose of text tokenisation is to convert a sentence into a set of tokens
  - These tokens can be based on words, characters and sub-words (N-grams)
  - E.g: Today is a very sunny day
    - Word tokens: [“Today”, “is”, “a”, ….]
    - Character tokens: [“T”, “o”, “d”,”a”, ….]
    - Sub-words (2-gram): [“To”, “od”, “da”, “ay”, …..]
  - Once the sentence is split into tokens, these tokens are associated with values
  - These values are used for word embedding
- The tokeniser can create a vocabulary form the given text dataset and this vocabulary will be used for associating a value
- In the case of word tokens; the vocabulary will contain all the unique words in the given corpus

Recurrent Neural Networks
Sentiment classification model architecture

- Layer-2: Embedding Layer:
  - Word embedding is the process of converting the words into real valued vectors
  - A neural network can be used to convert the one hot vectors to more robust representations
    - Onehot encoded inputs -> ANN -> Encoded outputs of defined size
    - This encoding ANN is trained semi-supervisely with back propagation
  - Once the words are embedded, similar words will have a small distance in the vector space (e.g: cosine similarity to measure the distance)
  - Another very popular word embedding method is Word2Vec method
  - This method uses a trainable neural network to learn representations of context or words

Recurrent Neural Networks
Sentiment classification model architecture

- Layer-2: Embedding Layer: Word2Vec CBOW
  - Take a sentence: “I go to the mall today”
  - We can make each word a onehot vector of [VOCAB_SIZE]
  - Continuous Bag of Word Method (CBOW):
    - Take a sliding window of size 3:
    - [“to”, “the”, “mall”]; “mall” is the target word
    - The context is the neighbouring words [“to”, “the”]
    - In CBOW a neural network is used to predict $P(target|context)$



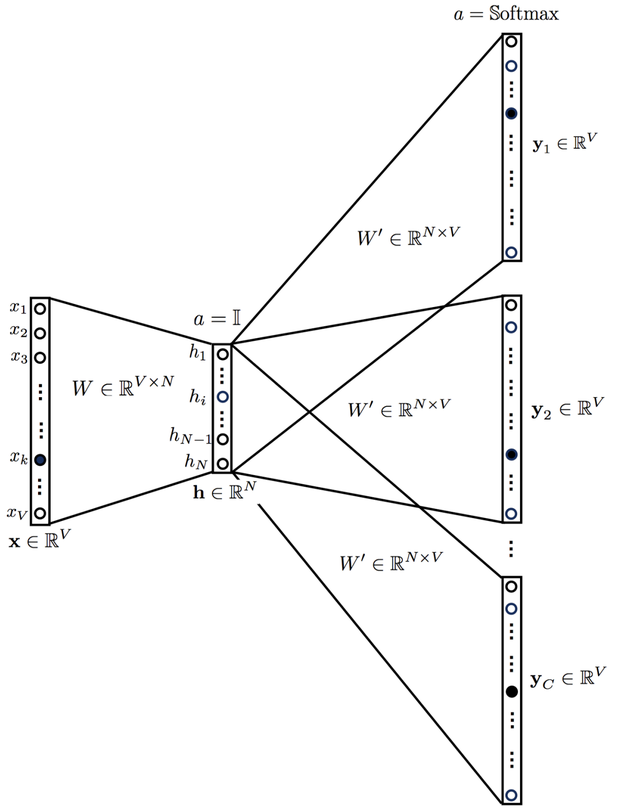
Ref.4: CBOW embedding model

Recurrent Neural Networks

Sentiment classification model architecture

- Layer-2: Embedding Layer: Word2Vec Skip-Gram
  - Take the previous sentence: “I go to the mall today”
  - Skip-Gram model:
    - Take a sliding window of size 3:
    - [“to”, “the”, “mall”]; “the” is the target word
    - The context is the neighbouring words

      [“to”, “mall”]
    - In Skip-Gram a neural network is used to predict $P(context|target)$

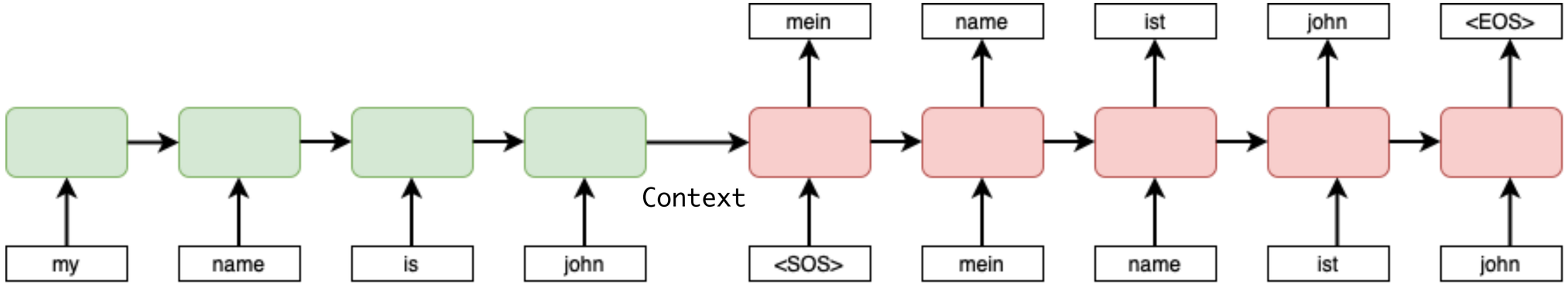


Ref.4: Skip-Gram embedding model

Recurrent Neural Networks

Neural machine translation: Training

- Translate from a source language to a target language
- Use teacher forcing for training
- Seq2Seq encoder decoder architecture:



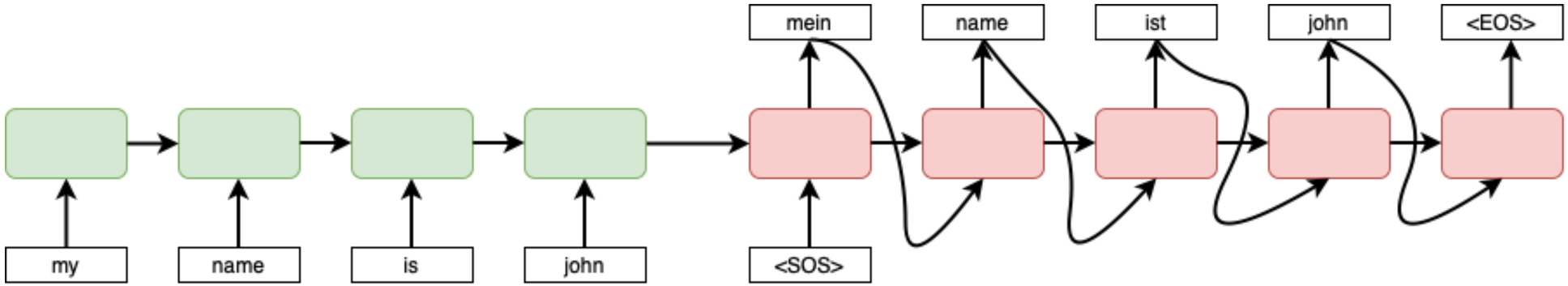
Encoder RNN encodes the source language to context vector

Decoder RNN initialised with a encoded context translates the source language to target language

Recurrent Neural Networks

Neural machine translation: Inference

- Get the encoded context of the source sentence
- Given the context as initial state:
  - Sample one output at a time an reinject the output as inputs to the next time step

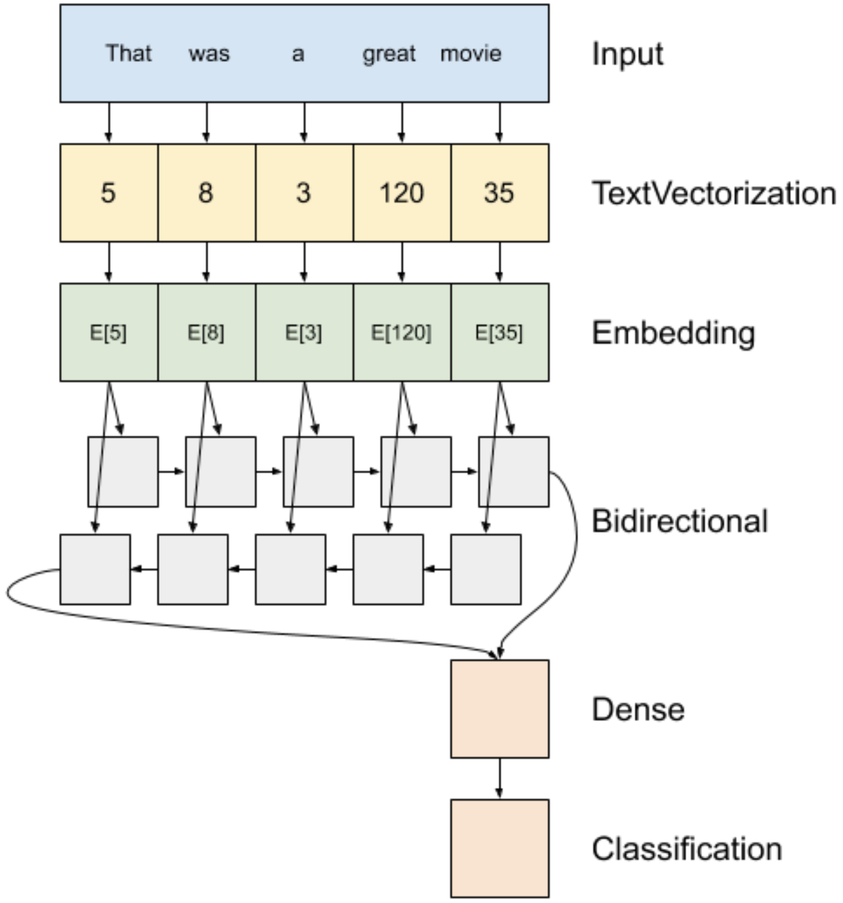


Recurrent Neural Networks
Sentiment classification

- Sentiment classification example:
  - A sentiment classification problem involves a text dataset containing text inputs and labels for each input
  - Task is to design a recurrent type neural network to predict the sentiment given a text input
- Layer-3: LSTM and MLP layers:
  - The LSTM layer extracts time dependent features from a encoded sequence of a sentence
  - It can be unidirectional or bidirectional LSTM layer
  - It is possible to stack a set of LSTM layers and construct a deep LSTM networks
  - The final state of the LSTM layer is given to a MLP (Dense) layer for sentiment prediction

Recurrent Neural Networks

Sentiment classification



Ref.5: Sentiment classification model architecture

Reference

- Ref.1: http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/
- Ref.2: https://colah.github.io/posts/2015-08-Understanding-LSTMs/
- Ref.3: https://towardsdatascience.com/understanding-bidirectional-rnn-in-pytorch-5bd25a5dd66
- Ref.4: https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa
- Ref.5: https://www.tensorflow.org/tutorials/text/text_classification_rnn
- GRU: https://d2l.ai/chapter_recurrent-modern/gru.html
- Tokenization and n-grams:
  - https://albertauyeung.github.io/2018/06/03/generating-ngrams.html
  - https://www.analyticsvidhya.com/blog/2020/05/what-is-tokenization-nlp/
- Word2Vec
  - https://medium.com/@zafaralibagh6/simple-tutorial-on-word-embedding-and-word2vec-43d477624b6d
- Keras embedding layer:
  - https://machinelearningmastery.com/use-word-embedding-layers-deep-learning-keras/
- GRU:
  - https://ieeexplore.ieee.org/document/818041