Lecture 8: Recurrent Neural Networks

Pranabendu Misra
Chennai Mathematical Institute

Advanced Machine Learning 2022

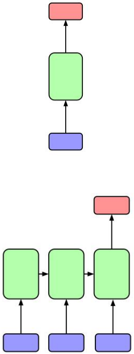
(based on slides by Madhavan Mukund)

Dealing with sequences

- Conventional neural networks map single inputs to single outputs
  - Each input/output may be a vector of values
- These are *Feed Forward Networks*.

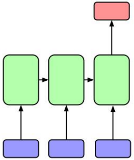
Dealing with sequences

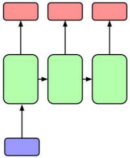
- Conventional neural networks map single inputs to single outputs
  - Each input/output may be a vector of values
- These are *Feed Forward Networks*.
- Some classification tasks require mapping a sequence of inputs to an output
  - Identifying a music of video clip



Dealing with sequences

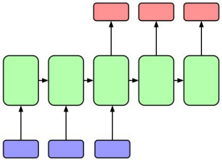
- Conventional neural networks map single inputs to single outputs
  - Each input/output may be a vector of values
- These are *Feed Forward Networks*.
- Some classification tasks require mapping a sequence of inputs to an output
  - Identifying a music of video clip
- Others require mapping a single input to a sequence of outputs
  - Generating a caption for an image





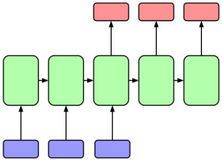
Dealing with sequences

- Mapping sequences to sequences
  - Language translation — read an entire input sentence, then generate output

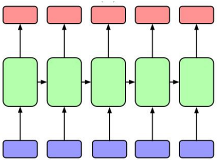


Dealing with sequences

- Mapping sequences to sequences
  - Language translation — read an entire input sentence, then generate output

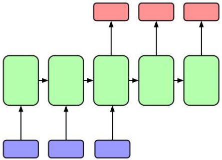


- Mapping sequences to sequences on the fly
  - Predict the next word in a sentence

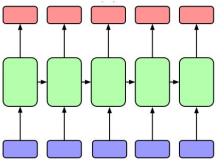


Dealing with sequences

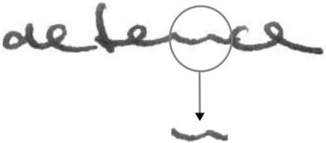
- Mapping sequences to sequences
  - Language translation — read an entire input sentence, then generate output



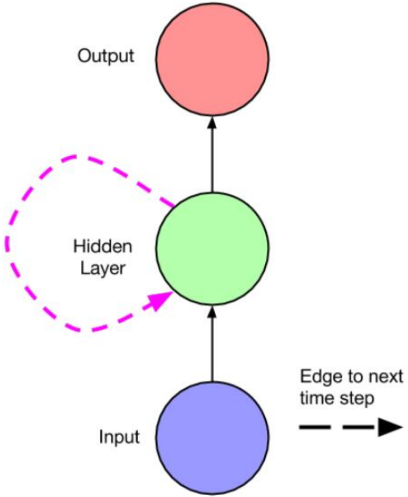
- Mapping sequences to sequences on the fly
  - Predict the next word in a sentence



- Context is important
  - The handwritten word is clearly defence
  - The n in isolation is illegible

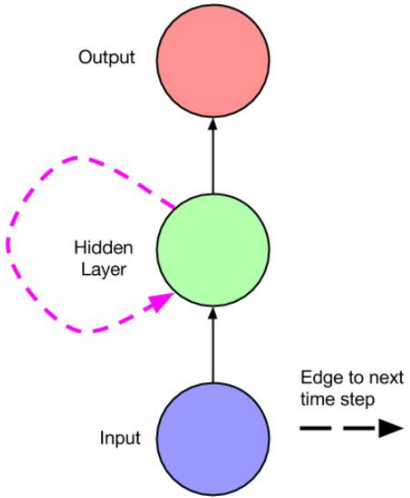


Idea: Lets also remember the past!

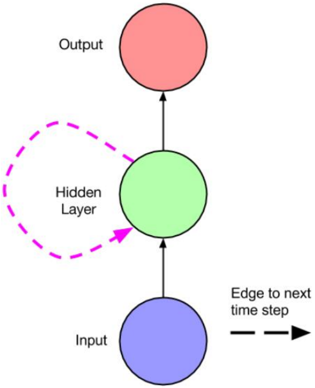


Incorporating memory

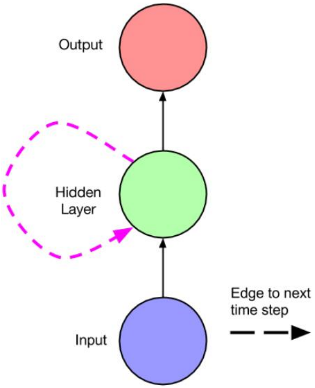
- Input sequence $x^{(1)}, x^{(2)}, \ldots, x^{(t)}$
- Output sequence $\hat{y}^{(1)}, \hat{y}^{(2)}, \ldots, \hat{y}^{(t)}$
- Allow $\hat{y}^{(t)}$ to also depend on previous inputs $x^{(1)}, x^{(2)}, \ldots, x^{(t-1)}$
- Hidden state : $h^{(t)}$
  - $h^{(t)}$ depends on current input and previous state
  - $h^{(t)} = f(W^{hx}x^{(t)} + W^{hh}h^{(t-1)} + b_h)$
- Output is a function of the current state
  - $y^{(t)} = g(W^{yh}h^{(t)} + b_y)$

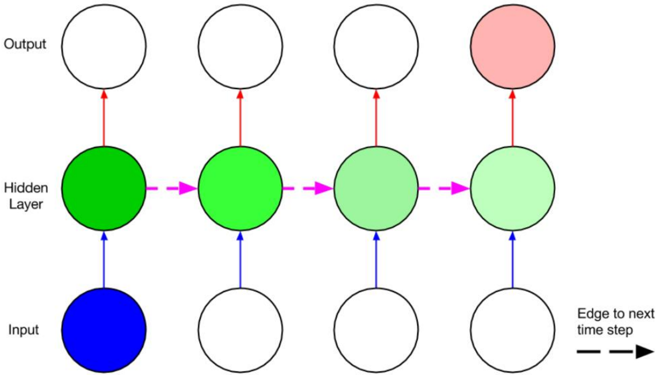


Time unrolling

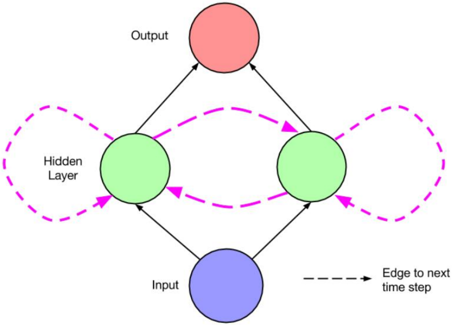


Time unrolling

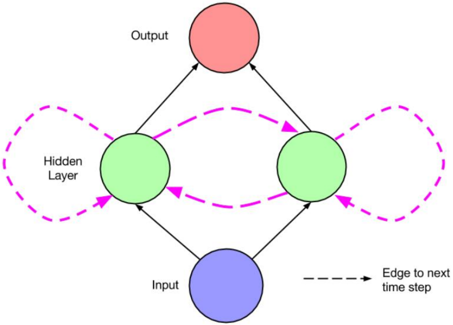


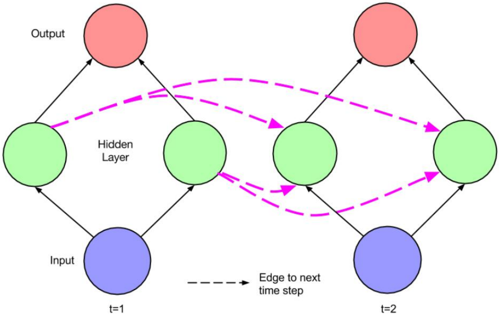


Time unrolling

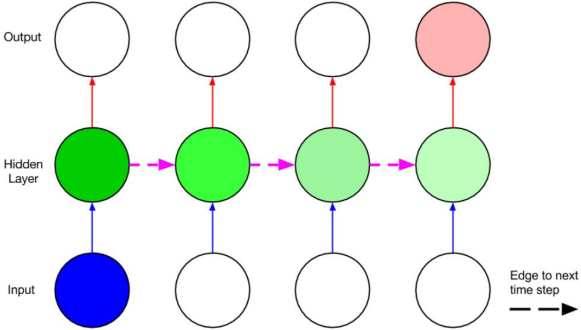


Time unrolling



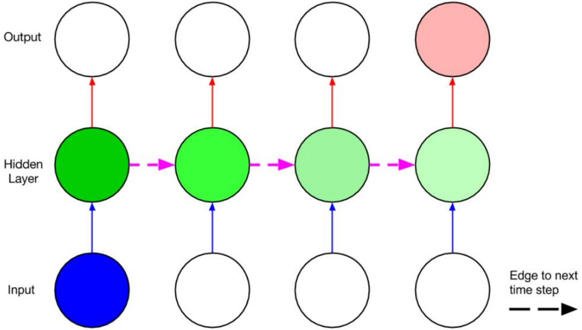


Time Unrolling and Back Propagation Through Time



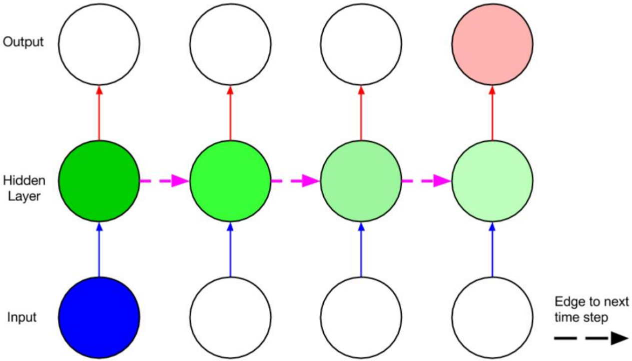
- Time Unrolling makes it a (larger) Feed-Forward Network
- but all *copies* share the parameters, so number of parameters doesn't increase.
- So we can do back-propagation to update the weights

Vanishing and Exploding Gradients gradients



- Unfortunately, we end-up with a very deep network
- Only the most recent parts of the input sequence are remembered; earlier parts are forgotten
- Back-Propagation suffers from vanishing or exploding gradients when unrolled over many time steps.

Vanishing and Exploding Gradients gradients



- Also, can't unroll infinitely far into the future.
- Truncated BPTT is what is done in practice, and also addresses these issues to an extent.
- Unfortunately, long-term context is lost.

Problem: We are unable to remember important parts of information far into the past.

Problem: We are unable to remember important parts of information far into the past.

Why this happens:

- Every piece x_i of the input sequence is treated in the same manner by the RNN.
- So important and non-important pieces both modify the hidden state, causing important pieces of the input sequences further in the past to be forgotten.

Problem: We are unable to remember important parts of information far into the past.

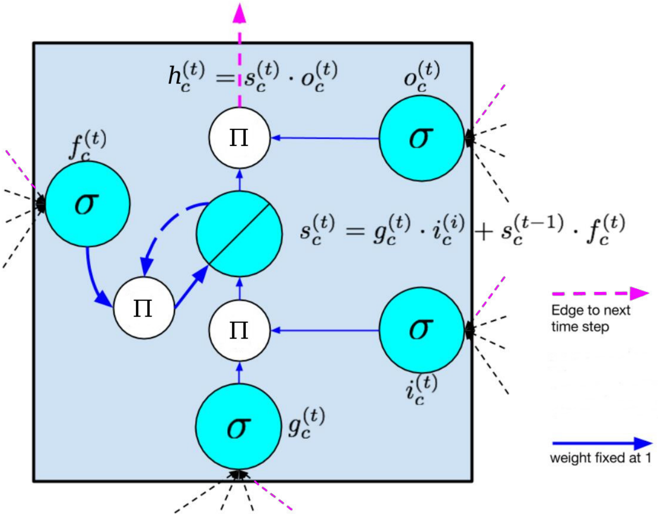
Idea: A mechanism that learns to distinguish between important and not-important parts, and remembers the important parts.

Problem: We are unable to remember important parts of information far into the past.

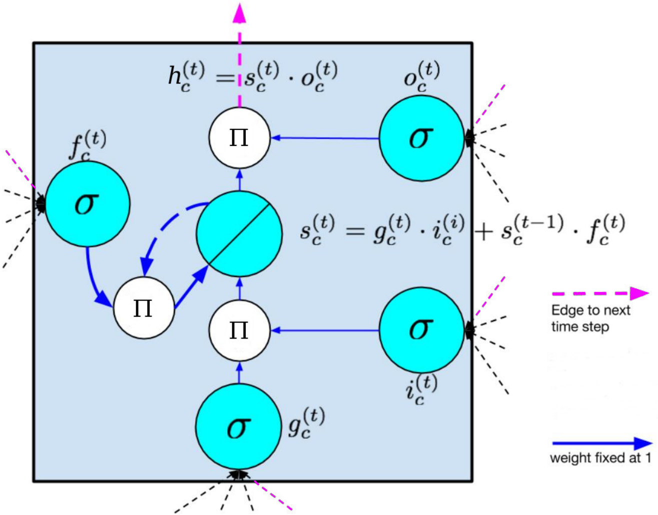
Idea: A mechanism that learns to distinguish between important and not-important parts, and remembers the important parts.

- Don't update the hidden state all the time, but only when necessary.
  Use gates to control information flow
- A gate is typically a neuron or a simple neural network with sigmoid activation, that takes the current input $x^{(t)}$ and the previous state $h^{(t)}$ and outputs a vector with values in $[0, 1]$.
  0 means gate is closed; 1 means gate is open
- We thus arrive at *Gated RNNs.* Intuitively, gates learn to distinguish important and non-important information. They only let important information update the internal-state.

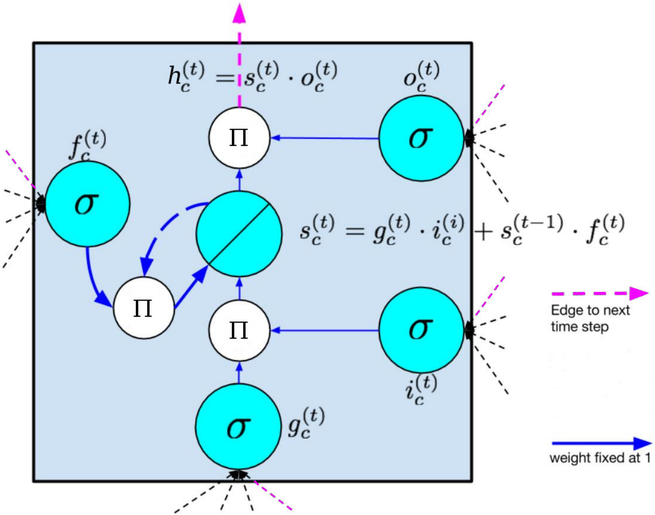
LSTM



- Long Short Term Memory (LSTM) are a popular variant of gated RNNs.
- They use multiple gates to decide how the internal state / memory $s_c^{(t)}$ is updated.
- Here, $x_c^{(t)}$ is the input at time t, which is supplied to all the gates along with the previous hidden state $h_c^{(t-1)}$
- Π represents point-wise product of two vectors.



- $f_c^{(t)}$ is the *Forget Gate*. Decides what bits of $s_c^{(t-1)}$ is retained in $s_c^{(t)}$ and what is forgotten.
- $i_c^{(t)}$ is the *Input Gate*. Decides what bits of the input $x_c^{(t)}$ are added to $s_c^{(t)}$.
- $g(t)_c$ is the *Input Node* which encodes the input $x^{(t)}$. It's output is what is actually added to $s_c^{(t)}$ instead of $x^{(t)}$.



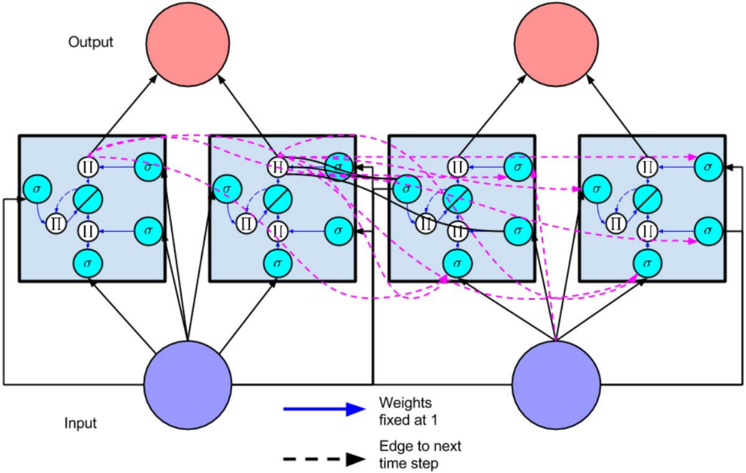
- We update the internal state as

$$s_c^{(t)} = \Pi(g_c^{(t)}, i_c^{(t)}) + \Pi(s_c^{(t-1)}, f_c^{(t)})$$

- The Output Gate $o_c^{(t)}$ decides what bits of the internal state $s^{(t)}$, is output to the hidden state $h_c(t)$

$$h_c^{(t)} = \Pi(s_c^{(t)}, o_c^{(t)})$$

LSTM unrolled in time



Bidirectional RNN

- Useful when we need context from both past and "future" e.g. translation, handwriting recognition etc.
- The whole input must be available; not online.
- Training via BPTT