COMS0018: Recurrent Neural Networks

Dima Damen
Dima.Damen@bristol.ac.uk

Bristol University, Department of Computer Science
Bristol BS8 1UB, UK

November 10, 2019

Introduction

- By the end of this course you will be familiar with 3 types of DNNs
    - Fully-Connected DNN
    - Convolutional DNN
    - Recurrent DNN
- The topic of today's lecture will be Recurrent Neural Networks (RNNs)

Introduction

- Similar to CNNs, RNNs are specialised in processing certain types of data
- CNNs were designed to deal with grid-like data
- RNNs are designed for processing sequential data

$$x_1, x_2, \cdots, x_t$$

Introduction

- Similar to CNNs, RNNs are specialised in processing certain types of data
- CNNs were designed to deal with grid-like data
- RNNs are designed for processing sequential data

$$x_1, x_2, \cdots, x_t$$

- Importantly, RNNs can scale to much longer sequences than would be practical for networks without sequence-based specialisation
- Most RNN architectures are designed to process sequences of variable lengths

CNN vs RNN

- What is the difference between 1-D CNN and RNN?
- 1-D CNN allows sharing parameters across time but is shallow
- In 1-D CNN, the output is a function of a small number of neighbouring members of the input

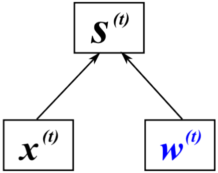
CNN vs RNN

- What is the difference between 1-D CNN and RNN?
- 1-D CNN allows sharing parameters across time but is shallow
- In 1-D CNN, the output is a function of a small number of neighbouring members of the input
- In contrary, RNNs share parameters with all previous members of the output
- This results in RNNs sharing parameters through a very deep computational graph

Recurrent Neural Networks

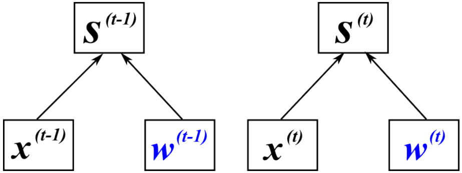
- For a certain time (t), the output function is the same as for other DNNs

$$s^{(t)} = f(x^{(t)}; w^{(t)})$$



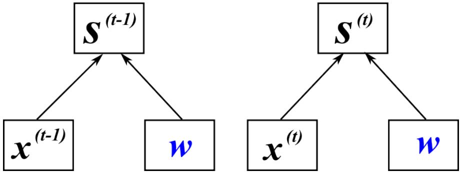
Recurrent Neural Networks

- And similarly for input at other times $t-1$



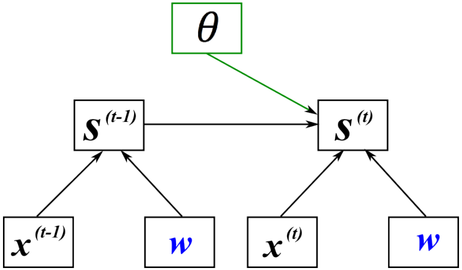
Recurrent Neural Networks

- Weights can though be shared across time



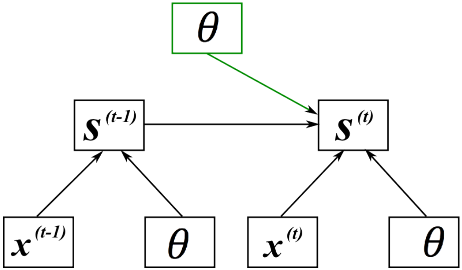
Recurrent Neural Networks

- RNNs emphasise the relationship between outputs over time
- w is called ‘input-to-hidden’ weights
- θ is called ‘hidden-to-hidden’ weights



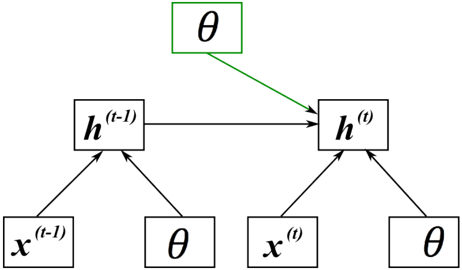
Recurrent Neural Networks

- Both w and θ are parameters for the RNN, we can thus use θ to refer to both



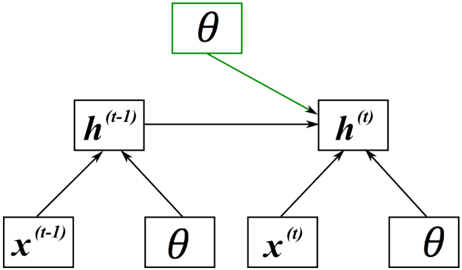
Recurrent Neural Networks

- To further emphasise that s is typically a hidden state of the system, we follow the book's notation using h



Recurrent Neural Networks

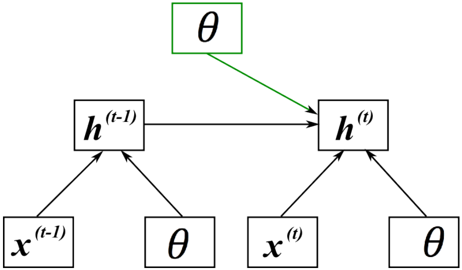
- Accordingly



Recurrent Neural Networks

- Accordingly

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



Unfolding RNNs

- The equation below is typically **unrolled** for a finite number of steps

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

- For 3 time steps

$$\begin{aligned} h^3 &= f(h^2, x^3; \theta) \\ &= f(f(h^1, x^2; \theta), x^3; \theta) \end{aligned}$$

- Note that the function f and the parameters θ are believed to be shared for all temporal steps
- Regardless of the sequence length, the learnt model f and parameters θ always have the same size - as it focuses on the transition over consecutive inputs/outputs as opposed to a variable-length past

Unfolding RNNs

- The different RNN architectures learn to use $h^{(t)}$ as a **lossy** summary of the past input up to t.

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

Unfolding RNNs

- The different RNN architectures learn to use $h^{(t)}$ as a **lossy** summary of the past input up to t.

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

- The learning is by default lossy, as it aims to map from an arbitrary length sequence $(x^{(t)}, x^{(t-1)}, \cdots, x^{1})$, to a fixed length output $h^{(t)}$

Unfolding RNNs

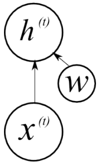
- The different RNN architectures learn to use $h^{(t)}$ as a **lossy** summary of the past input up to t.

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

- The learning is by default lossy, as it aims to map from an arbitrary length sequence $(x^{(t)}, x^{(t-1)}, \cdots, x^1)$, to a fixed length output $h^{(t)}$
- Depending on the training criteria, the learning selectively ‘keeps’ some part of the past and ‘forgets’ others

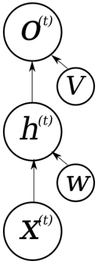
Training an RNN

- Note that h is the hidden representation of the RNN, not its output,
- So for every timestep,



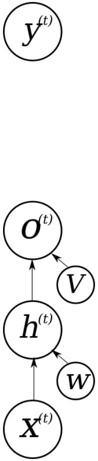
Training an RNN

- Note that h is the hidden representation of the RNN, not its output,
- So for every timestep, an output $o^{(t)}$ would be predicted
- V is 'hidden-to-output' weights



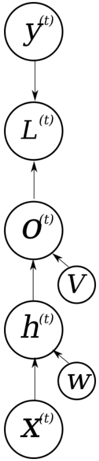
Training an RNN

- The hidden state represents the summary of the past,
- The output can then be compared to a given label $y^{(t)}$



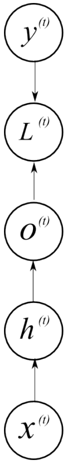
Training an RNN

- The output can then be compared to a given label $y^{(t)}$
- Using a specified loss function L that measures how far each output o is from the corresponding target label y



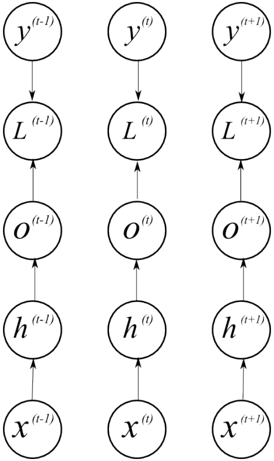
Training an RNN

- Note that we remove the parameter/weight W, V for simplicity



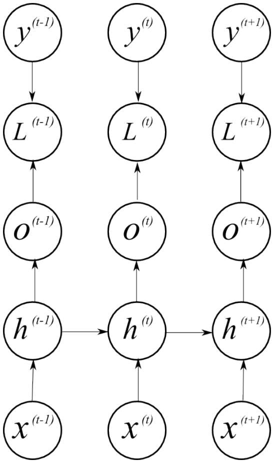
Training an RNN

- Next, we consider the connectivity across time steps
- Different variants of RNNs are available as follows,



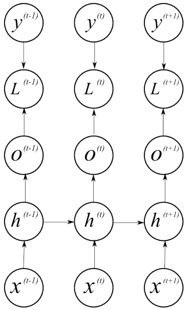
RNN Types

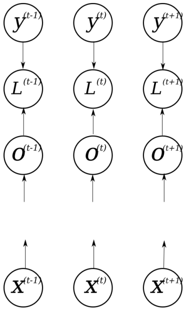
1. The first type produces an output for every time step, with recurrent connections between hidden units



RNN Types

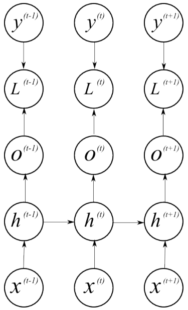
- Do we need these hidden units??

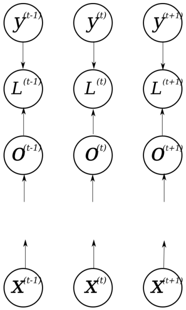


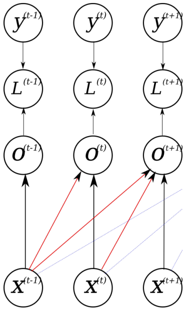


RNN Types

- Do we need these hidden units??

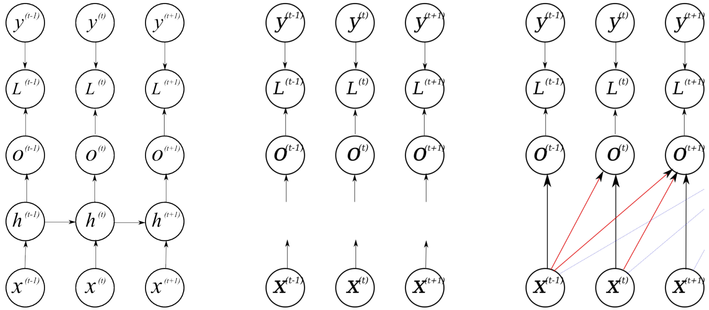






RNN Types

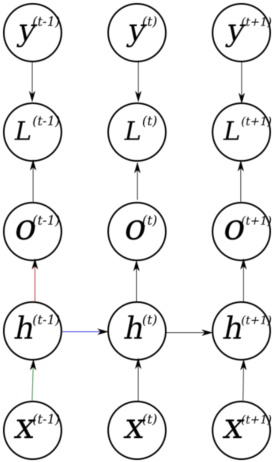
- Do we need these hidden units??



- An observation in the distant pass might influence the decision **via its effect on** h

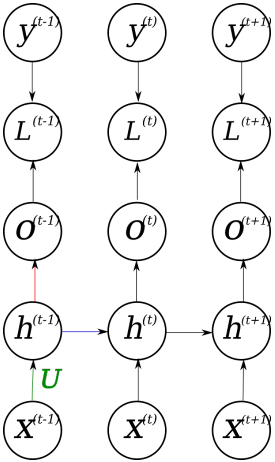
Training an RNN

- In this case, three weights matrices can be envisaged



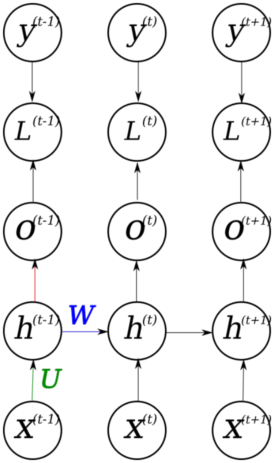
Training an RNN

- input to hidden connections are parameterised by weight U



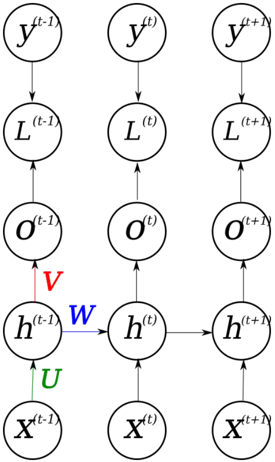
Training an RNN

- hidden-to-hidden recurrent connections parameterised by weight W



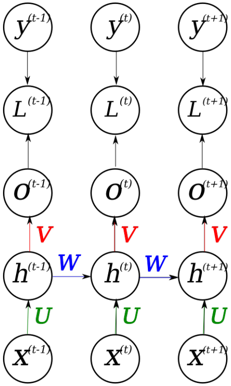
Training an RNN

- hidden-to output connections are parameterised by weight V



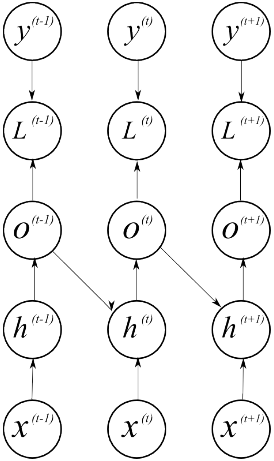
Training an RNN

- and these are shared across timesteps
- This relies on the assumption that the same parameter can be used for different time steps, i.e. the temporal dependency is **stationary**, i.e. does not depend on t



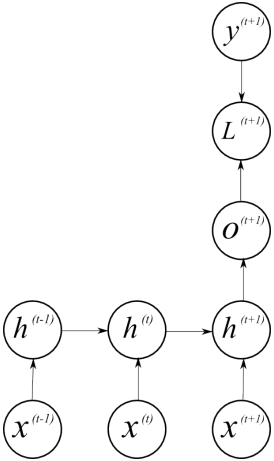
RNN Types

2. The second type produces an output for every time step, with recurrent connections from output to hidden units at the next time step



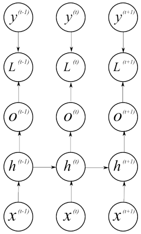
RNN Types

3. The third type reads multiple timesteps before producing a single output



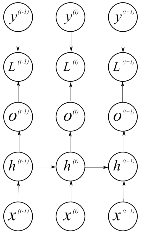
RNN Types

- The left model can be used to learn any function computable with a Turing machine

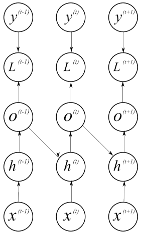


RNN Types

- The left model can be used to learn any function computable with a Turing machine
- The middle model is less powerful. The information captured by the output o is the only information it can send to the future. Unless o is very high-dimensional and rich, it will lack important information from the past.



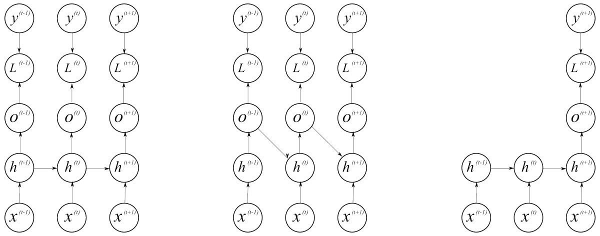
$y^{(t-1)}$ $y^{(t)}$ $y^{(t+1)}$
$L^{(t-1)}$ $L^{(t)}$ $L^{(t+1)}$
$o^{(t-1)}$ $o^{(t)}$ $o^{(t+1)}$
$h^{(t-1)}$ $h^{(t)}$ $h^{(t+1)}$
$x^{(t-1)}$ $x^{(t)}$ $x^{(t+1)}$



$y^{(t-1)}$ $y^{(t)}$ $y^{(t+1)}$
$L^{(t-1)}$ $L^{(t)}$ $L^{(t+1)}$
$o^{(t-1)}$ $o^{(t)}$ $o^{(t+1)}$
$h^{(t-1)}$ $h^{(t)}$ $h^{(t+1)}$
$x^{(t-1)}$ $x^{(t)}$ $x^{(t+1)}$

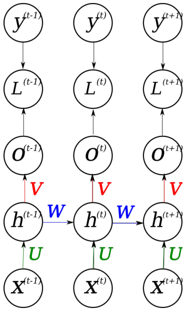
RNN Types

- The left model can be used to learn any function computable with a Turing machine
- The middle model is less powerful. The information captured by the output o is the only information it can send to the future. Unless o is very high-dimensional and rich, it will lack important information from the past.
- The right model can be used to produce summaries (e.g. classifications of full sentences)



Training RNNs

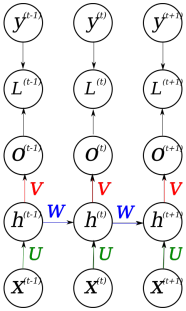
- We will next focus on the first model to understand how the forward pass can be obtained



Training RNNs

- We will next focus on the first model to understand how the forward pass can be obtained

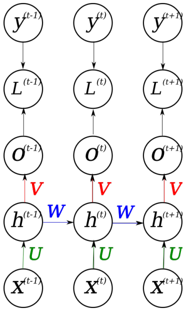
$$h^{(t)} = g(Wh^{(t-1)} + Ux^{(t)} + b)$$



Training RNNs

- We will next focus on the first model to understand how the forward pass can be obtained

$$\begin{aligned} h^{(t)} &= g(Wh^{(t-1)} + Ux^{(t)} + b) \\ o^{(t)} &= g(Vh^{(t)} + c) \end{aligned}$$

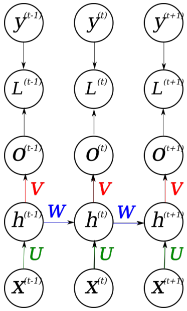


Training RNNs

- We will next focus on the first model to understand how the forward pass can be obtained

$$\begin{aligned} h^{(t)} &= g(Wh^{(t-1)} + Ux^{(t)} + b) \\ o^{(t)} &= g(Vh^{(t)} + c) \end{aligned}$$

- The total loss would them be the sum of all losses over all time steps



Training RNNs

- However, computing the gradient of this function is expensive
- It requires performing a forward propagation pass of the unrolled network, followed by backward propagation pass throughout time

Training RNNs

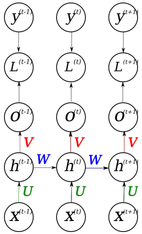
- However, computing the gradient of this function is expensive
- It requires performing a forward propagation pass of the unrolled network, followed by backward propagation pass throughout time
- The runtime is $O(\tau)$ and cannot be reduced by parallelisation because the forward pass is sequential
- All computations in the forward pass must be stored until reused during the backward pass, making the memory cost $O(\tau)$ as well

Training RNNs

- However, computing the gradient of this function is expensive
- It requires performing a forward propagation pass of the unrolled network, followed by backward propagation pass throughout time
- The runtime is $O(\tau)$ and cannot be reduced by parallelisation because the forward pass is sequential
- All computations in the forward pass must be stored until reused during the backward pass, making the memory cost $O(\tau)$ as well
- This back-propagation algorithm is called back-propagation through time (BPTT)

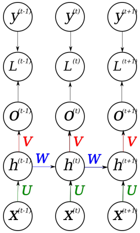
Training RNNs

- Practically, computing the gradient through an RNN is straightforward.
- The generalised back-propagation algorithm is applied to the unrolled network.



Training RNNs

- Practically, computing the gradient through an RNN is straightforward.
- The generalised back-propagation algorithm is applied to the unrolled network.
- For the network below, the parameters are: U, V, W, b, c,
- The nodes indexed by t: $x^{(t)}, o^{(t)}, L^{(t)}$ as well as the hidden node $h^{(t)}$



Training RNNs

- After the forward pass, gradient is first computed at the nodes immediately preceding the final loss

$$\frac{\partial L}{\partial L^{(t)}} = 1$$

- We can then calculate the loss using softmax and cross-entropy, given the output $o^{(t)}$

$$\frac{\partial L}{\partial o^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i,y^{(t)}}$$

- We then work our way backwards, from the end of the sequence to the start

Training RNNs

- Because the parameters are shared across many time steps, calculating the derivative might seem confusing
- Calculating the derivative $\nabla_W L$ operator, should take into account the contribution of W from all edges in the graph.
- To resolve this, we introduce copies of W at different time steps $W^{(t)}$
- We then calculate the gradient at time step t to be $\nabla_{W^{(t)}}$

Training RNNs

- For the five parameters c, b, V, W, U, the gradients are given by

$$\begin{aligned}
\nabla_c L &= \sum_t \nabla_{o^{(t)}} L \\
\nabla_b L &= \sum_t \Big(\frac{\partial h^{(t)}}{\partial b^{(t)}}\Big)^T \nabla_{h^{(t)}} L \\
\nabla_V L &= \sum_t \sum_i \Big(\frac{\partial L}{\partial o^{(t)}}\Big)^T \nabla_{V^{(t)}} o_i^{(t)} = \sum_t (\nabla_{o^{(t)}} L) h^{(t)^T} \\
\nabla_W L &= \sum_t \sum_i \Big(\frac{\partial L}{\partial h^{(t)}}\Big)^T \nabla_{W^{(t)}} h_i^{(t)} = \sum_t diag\big(1 - (h^{(t)})^2\big)(\nabla_{h^{(t)}} L) h^{(t-1)^T} \\
\nabla_U L &= \sum_t \sum_i \Big(\frac{\partial L}{\partial h^{(t)}}\Big)^T \nabla_{U^{(t)}} h_i^{(t)} = \sum_t diag\big(1 - (h^{(t)})^2\big)(\nabla_{h^{(t)}} L) x^{(t)^T}
\end{aligned}$$

Other RNN Types

- Bi-directional RNNs
- Encoder-Decoder Sequence-to-Sequence architecture
- Gated RNNs
- Long-Short Term Memory (LSTMs)

Bi-directional RNNs

- We might want the prediction $y^{(t)}$ to depend on the *whole* sequence, its past and future

Bi-directional RNNs

- We might want the prediction $y^{(t)}$ to depend on the *whole* sequence, its past and future
- This is particularly of relevance to speech recognition, machine translation or audio analysis

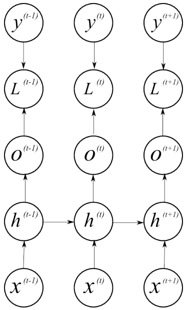
Bi-directional RNNs

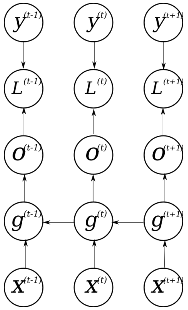
- We might want the prediction $y^{(t)}$ to depend on the *whole* sequence, its past and future
- This is particularly of relevance to speech recognition, machine translation or audio analysis
- As the name suggests, bidirectional RNNs combine an RNN that moves forward through time, beginning from the start of the sequence, with another that moves backward through time, beginning from the end of the sequence.

Bi-directional RNNs

- We might want the prediction $y^{(t)}$ to depend on the *whole* sequence, its past and future
- This is particularly of relevance to speech recognition, machine translation or audio analysis
- As the name suggests, bidirectional RNNs combine an RNN that moves forward through time, beginning from the start of the sequence, with another that moves backward through time, beginning from the end of the sequence.
- This allows the output unit $o^{(t)}$ to compute a representation that depends on *both the past and the future*, without specifying any fixed-size window around t

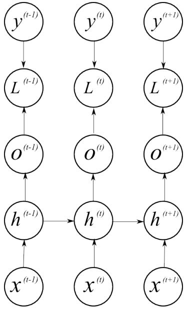
Bi-directional RNNs

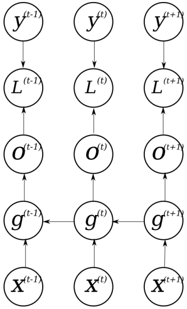


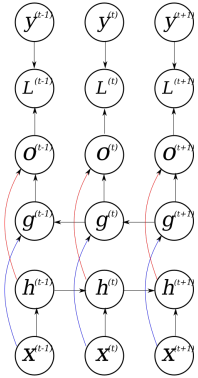


Bi-directional RNNs

- A typical bidirectional RNN

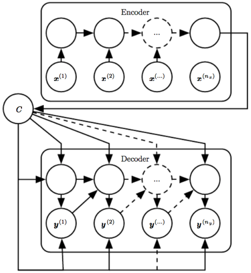






Encoder-Decoder RNN

- Mapping a variable-length sequence to another variable-length sequence
- You can refer to c as the context
- The encoder *reads* the input, emitting the context - a function of its hidden states
- The decoder *writes* the fixed-level output sequence



Goodfellow p384

Long-Term Dependencies

- All previously mentioned architectures are good at learning short-term dependencies, without specifying a fixed-length
- However, gradients propagated over longer time tend to either vanish or explode

Long-Term Dependencies

- All previously mentioned architectures are good at learning short-term dependencies, without specifying a fixed-length
- However, gradients propagated over longer time tend to either vanish or explode
- Even when attempting to resolve the problem, by selecting parameter spaces where the gradients do not vanish or explode, the problem persists
- The gradient of a long-term interaction will always have exponentially smaller magnitude than the gradient of a short-term interaction

Long-Term Dependencies

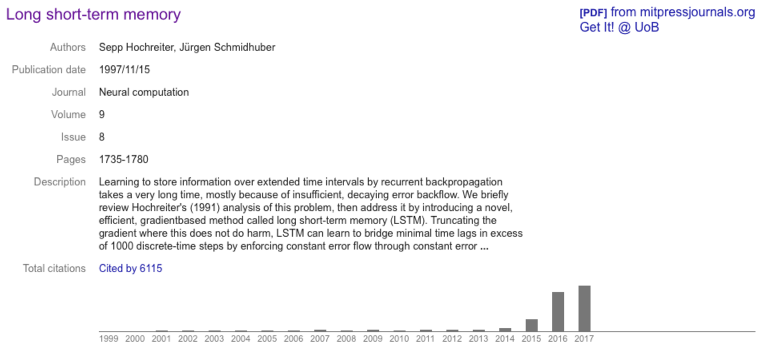
- All previously mentioned architectures are good at learning short-term dependencies, without specifying a fixed-length
- However, gradients propagated over longer time tend to either vanish or explode
- Even when attempting to resolve the problem, by selecting parameter spaces where the gradients do not vanish or explode, the problem persists
- The gradient of a long-term interaction will always have exponentially smaller magnitude than the gradient of a short-term interaction
- The most effective solution for long-term dependencies are gated RNNs

Gated RNNs

- Gated RNNs are based on creating paths through time
- This is achieved through connection weights that change at each time step
- This allows the network to *accumulate* information over a long duration - often referred to as memory
- It also allows the network to *forget* old states when needed
- Previous approaches attempted to set these accumulation and forgetting gates manually, while gated RNNs attempt to learn to decide when to do that.
- One of the most popular gated RNNs are LSTMs

LSTM

- First proposed by Hochreiter and Schmidhuber in 1997



Long short-term memory

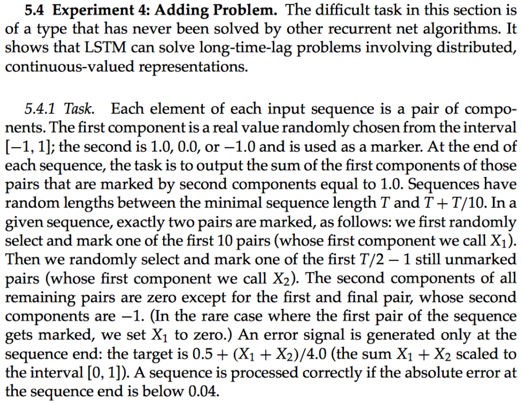
[PDF] from mitpressjournals.org
Get It! @ UoB

Authors	Sepp Hochreiter, Jürgen Schmidhuber
Publication date	1997/11/15
Journal	Neural computation
Volume	9
Issue	8
Pages	1735-1780
Description	Learning to store information over extended time intervals by recurrent backpropagation takes a very long time, mostly because of insufficient, decaying error backflow. We briefly review Hochreiter's (1991) analysis of this problem, then address it by introducing a novel, efficient, gradientbased method called long short-term memory (LSTM). Truncating the gradient where this does not do harm, LSTM can learn to bridge minimal time lags in excess of 1000 discrete-time steps by enforcing constant error flow through constant error ...
Total citations	Cited by 6115

Google Scholars 2017

LSTM

- The Adding Problem



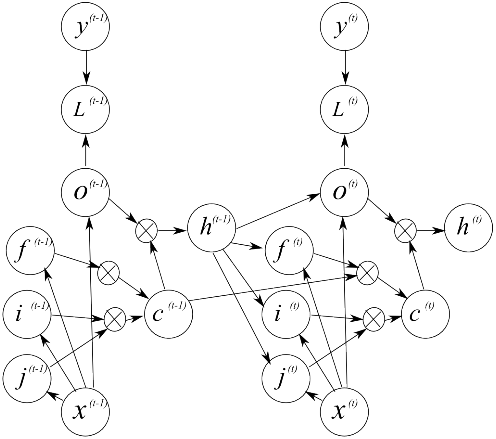
5.4 Experiment 4: Adding Problem. The difficult task in this section is of a type that has never been solved by other recurrent net algorithms. It shows that LSTM can solve long-time-lag problems involving distributed, continuous-valued representations.

5.4.1 Task. Each element of each input sequence is a pair of components. The first component is a real value randomly chosen from the interval $[-1, 1]$; the second is 1.0, 0.0, or -1.0 and is used as a marker. At the end of each sequence, the task is to output the sum of the first components of those pairs that are marked by second components equal to 1.0. Sequences have random lengths between the minimal sequence length T and $T + T/10$. In a given sequence, exactly two pairs are marked, as follows: we first randomly select and mark one of the first 10 pairs (whose first component we call X_1). Then we randomly select and mark one of the first $T/2 - 1$ still unmarked pairs (whose first component we call X_2). The second components of all remaining pairs are zero except for the first and final pair, whose second components are -1. (In the rare case where the first pair of the sequence gets marked, we set X_1 to zero.) An error signal is generated only at the sequence end: the target is $0.5 + (X_1 + X_2)/4.0$ (the sum $X_1 + X_2$ scaled to the interval $[0, 1]$). A sequence is processed correctly if the absolute error at the sequence end is below 0.04.

Hochreiter and Schmidhuber (1997). Long-Short Term Memory. Neural Computation

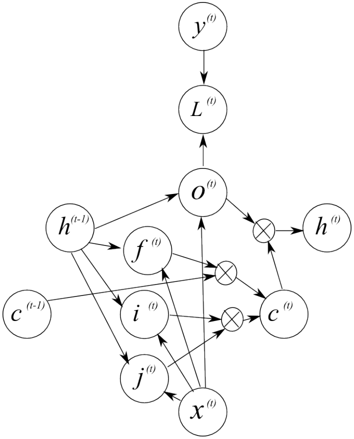
Two-slice LSTM Figure

- LSTM looks significantly more complex than an RNN until you start dissecting it

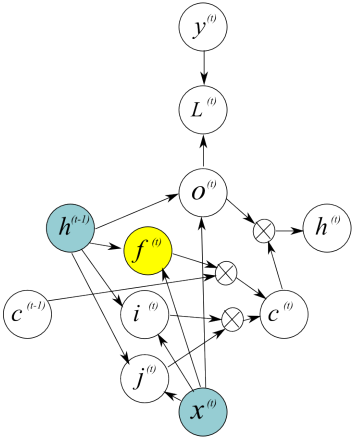


Two-slice LSTM Figure

- Let's first simplify by keeping a single timestamp and only its dependencies $h^{(t-1)}$, $c^{(t-1)}$



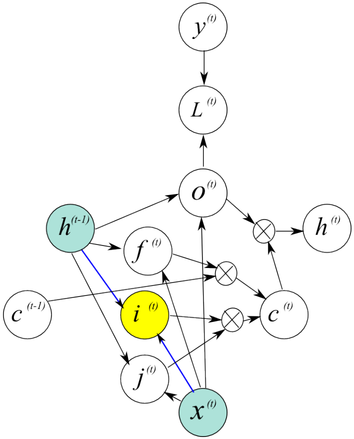
LSTM



- The ability to forget the past is controlled by what is rightly named, the *forget* gate $f^{(t)}$
- The weight of the forget gate is set to a value between 0 and 1 using a sigmoid function

$$
\begin{aligned}
f_i^{(t)} &= \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right) \\
\boldsymbol{f}^{(t)} &= \boldsymbol{b}^f + \boldsymbol{U}^f \boldsymbol{x}^{(t)} + \boldsymbol{W}^f \boldsymbol{h}^{(t-1)}
\end{aligned}
$$

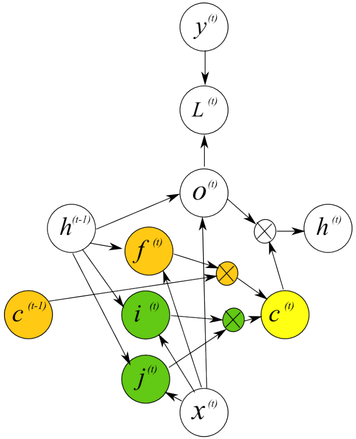
LSTM



- The **external input gate** unit $\boldsymbol{i}$ is computed in the same way as the forget gate
- Again, its weight is set to a value between 0 and 1 using a sigmoid function

$$\boldsymbol{i}^{(t)} = \boldsymbol{b}^i + U^i\boldsymbol{x}^{(t)} + W^i\boldsymbol{h}^{(t-1)}$$

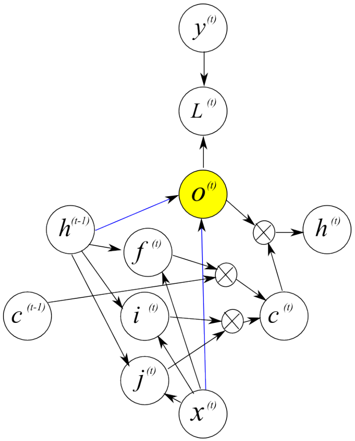
LSTM



- The **state gate** unit $\boldsymbol{c}$ is then updated as follows
- Where $\circ$ is an element-wise multiplication

$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{i} \circ \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \circ \sigma\left(\boldsymbol{b}^{j} + U^{j}\boldsymbol{x}^{(t)} + W^{j}\boldsymbol{h}^{(t-1)}\right)$$

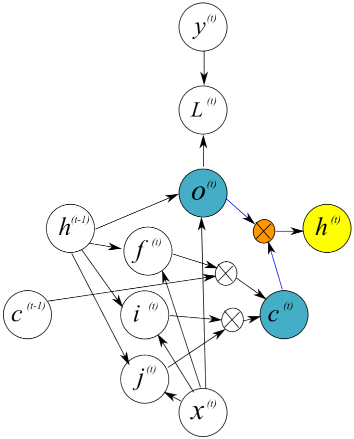
LSTM



- The **output gate** unit $\boldsymbol{o}$ also uses a sigmoid function

$$\boldsymbol{o}^{(t)} = \sigma\left(\boldsymbol{b}^{o} + U^{o}\boldsymbol{x}^{(t)} + W^{o}\boldsymbol{h}^{(t-1)}\right)$$

LSTM



- And finally

$$\boldsymbol{h}^{(t)} = \tanh(c^{(t)}) \circ o^{(t)}$$

Gated RNNs

- While LSTMs proved useful for both artificial and real data, questions were asked on whether this level of complexity is necessary
- Recently, gated RNNs (GRUs) are increasingly used where a single gating unit controls the forget factor and the update factor as follows,

Gated RNNs

- An update gate

$$\boldsymbol{u}^{(t)} = \sigma\left(\boldsymbol{b}^u + U^u \boldsymbol{x}^{(t)} + W^u \boldsymbol{h}^{(t)}\right)$$

- A reset gate

$$\boldsymbol{r}^{(t)} = \sigma\left(\boldsymbol{b}^r + U^r \boldsymbol{x}^{(t)} + W^r \boldsymbol{h}^{(t)}\right)$$

- A single update equation

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + (1 - u_i^{(t-1)}) \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t-1)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)}\right)$$

Gated RNNs

- An update gate

$$\boldsymbol{u}^{(t)} = \sigma\left(\boldsymbol{b}^u + U^u \boldsymbol{x}^{(t)} + W^u \boldsymbol{h}^{(t)}\right)$$

- A reset gate

$$\boldsymbol{r}^{(t)} = \sigma\left(\boldsymbol{b}^r + U^r \boldsymbol{x}^{(t)} + W^r \boldsymbol{h}^{(t)}\right)$$

- A single update equation

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + (1 - u_i^{(t-1)}) \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t-1)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)}\right)$$

- The reset gates chose to ignore parts of the state vector
- The update gates linearly gate any dimension, copying or completely ignoring it
- The reset gates control which parts of the state get used, introducing an additional nonlinear effect in the relationship with the past state.

Further Reading

- Deep Learning

  Ian Goodfellow, Yoshua Bengio, and Aaron Courville
  MIT Press, ISBN: 9780262035613.

  - Chapter 10 – Sequence Modeling: Recurrent and Recursive Nets