Deep learning

Recurrent neural networks[1]

Hamid Beigy

Sharif University of Technology

December 4, 2021

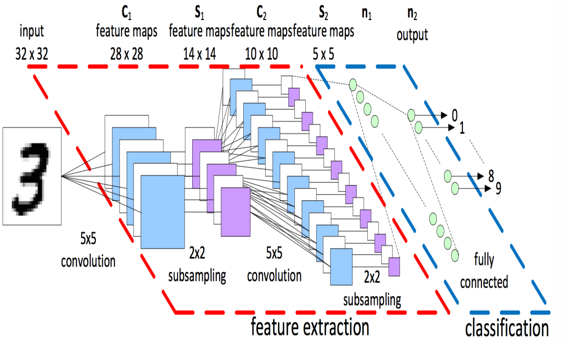
[1] Some slides are taken from B. Raj's slides

Table of contents

Introduction

Introduction

1. In previous sessions, we considered deep learning models with the following characteristics.
    - Input Layer: (maybe vectorized), quantitative representation
    - Hidden Layer(s): Apply transformations with nonlinearity
    - Output Layer: Result for classification, regression, translation, segmentation, etc.
2. Models used for supervised learning



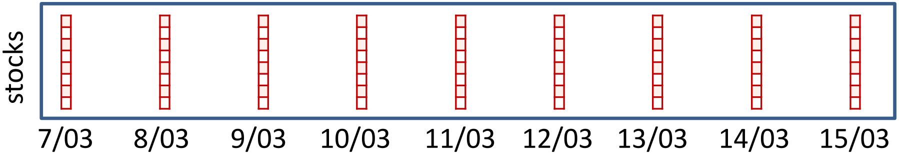
Sequence learning

1. Sequence learning is the study of machine learning algorithms designed for sequential data.
2. Language model ($p(w_t|w_{t-1}, w_{t-2}, \ldots, w_{t-n})$) is one of the most interesting topics that use sequence labeling.
3. For example, consider machine translation task.
   - We have a sentence in the source language.
   - We must translate the above sentence to the destination language.
4. How do we use feed-forward networks for solving the above machine translation problem?
5. Consider other solutions such as autoregressive models, linear dynamical systems, and hidden Markov models as an **exercise**.

Sequence learning (application)

1. Consider the stock market[2].
2. We must consider the series of stock values in the past several days to decide if it is wise to invest today (Ideally consider all of history).

To invest or not to invest?

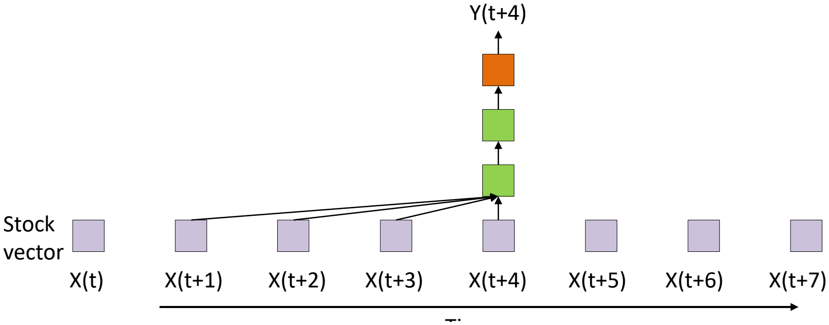


3. Inputs are vectors.
4. Output may be scalar (**Should I invest**) or vector (**should I invest in X**).

[2] From Bhiksha Raj slides

Sequence learning (application)

1. We need a network that accepts previous days and decides.



Credit: Bhiksha Raj

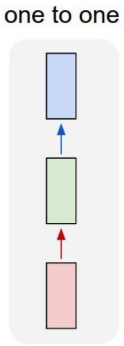
Sequence learning (Applications)

1. Although all problems in sequence learning can be converted into one with fixed-length inputs and outputs, they may involve a variable **time horizon**[3].
2. Sequence classification
   - sentiment analysis,
   - activity/action recognition,
   - DNA sequence classification,
   - action selection
3. Sequence synthesis:
   - text synthesis,
   - music synthesis,
   - motion synthesis.
4. Sequence-to-sequence translation:
   - speech recognition,
   - text translation,
   - part-of-speech tagging.

[3] From Francois Fleuret slides

Processing sequences

1. Processing sequences[4].



Vanilla Neural Networks

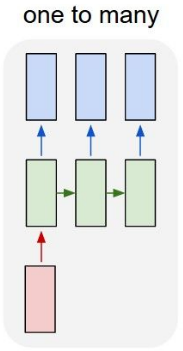
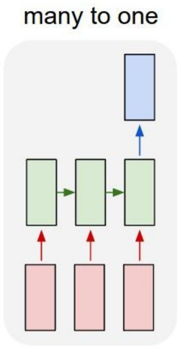
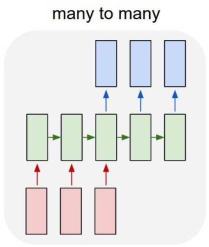


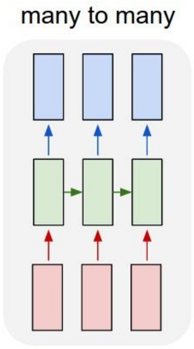
Image Captioning (image → seq of words)



Sentiment Classification (seq of words → sentiment)



Machine Translation (seq of words → seq of words)



Video classification on frame level

[4] From Fei-Fei Li et al. slides

Can we process non-sequential data sequentially?

1. We have a non-sequence data and we are processing it sequentially.
2. Is it possible?
3. Please read papers (Ba, Mnih, and Kavukcuoglu 2015)[5] and (Gregor et al. 2015)[6].

[5] Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu (2015). "Multiple Object Recognition with Visual Attention". In: *International Conference on Learning Representations*.

[6] Karol Gregor et al. (2015). "DRAW: A Recurrent Neural Network For Image Generation". In: *International Conference on Machine Learning*, pp. 1462–1471.

Why MLP is not appropriate for sequence learning?

1. MLPs only accept an input of fixed dimensionality and map it to an output of fixed dimensionality
2. In a traditional MLPs, we assume that all inputs (and outputs) are independent of each other. Why this is problematic?
3. Need to re-learn the rules from scratch each time
4. Need to resuse knowledge about the previous events to help in classifying the current

Recurrent neural networks

Introduction

1. A recurrent model maintains a recurrent state updated at each time step[7].
2. Consider input $x \in S\left(\mathbb{R}^D\right)$, and an initial state $h_0 \in \mathbb{R}^Q$, the recurrent model would compute the following sequence of recurrent states iteratively.

$$\boxed{\begin{gathered} h_t = \phi(x_t, h_{t-1}) \\ \phi : \mathbb{R}^D \times \mathbb{R}^Q \mapsto \mathbb{R}^Q \end{gathered}}$$

3. A prediction can be computed at any time step from the recurrent state

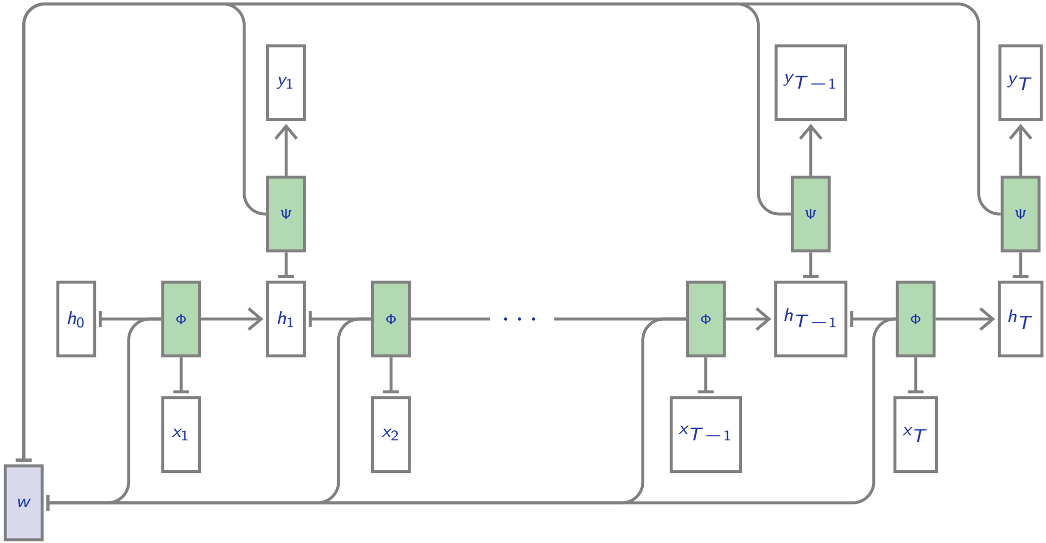
$$\boxed{\begin{gathered} y_t = \psi(h_t) \\ \psi : \mathbb{R}^Q \mapsto \mathbb{R}^C \end{gathered}}$$

[7] From Francois Fleuret slides

Recurrent neural networks

1. Recurrent neural networks are networks for handling sequential data such as a pair of sentences with different lengths and two speech signals.
2. Are weights dependent to the time instant?
3. RNN shares parameters across different parts of the model.
4. Why do we share weights?
5. When there is no parameter sharing, it would not be possible to share statistical strength and generalize to lengths of sequences not seen during training.

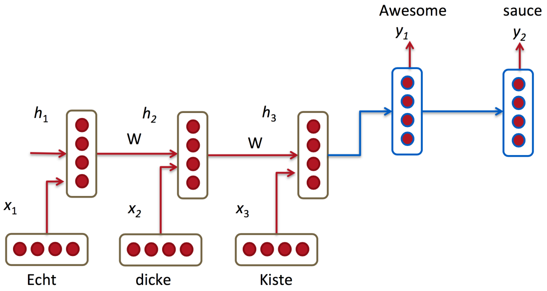
Recurrent neural network architecture[8]



[8] From Francois Fleuret slides

Recurrent neural network (Machine translation)

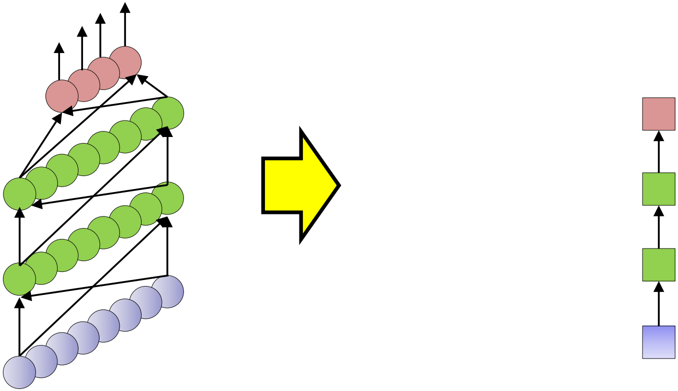
1. Machine Translation is similar to language modeling in that our input is a sequence of words in the source language.
2. We want to output a sequence of words in our target language.
3. A key difference is that the output only starts after we have seen the complete input, because the first word of our translated sentences may require information captured from the complete input sequence.



Credit: Denny Britz

Recurrent neural network (shortcut)

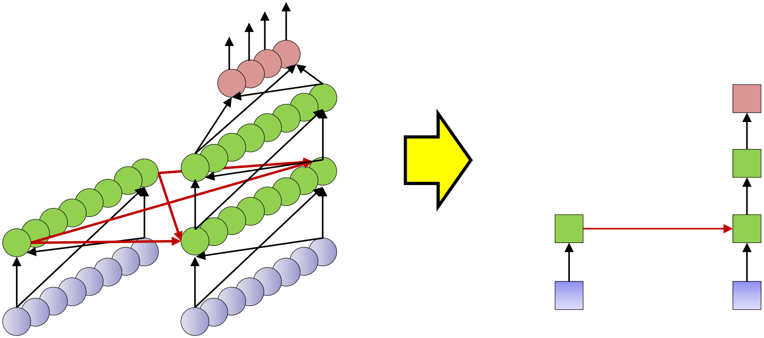
1. We often use the following shortcut to represent the recurrent neural network.



Credit: Bhiksha Raj

Recurrent neural network (shortcut)

1. We often use the following shortcut to represent the recurrent neural network.



Credit: Bhiksha Raj

Recurrent neural network (example application)[9]

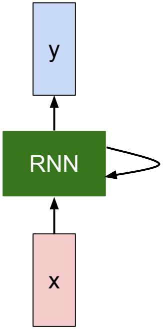
1. We consider the following simple binary sequence classification problem:
   - Label 1: the sequence is the concatenation of two identical halves,
   - label 0: otherwise

x	y
$(1, 2, 3, 4, 5, 6)$	0
$(3, 9, 9, 3)$	0
$(7, 4, 5, 7, 5, 4)$	0
$(7, 7)$	1
$(1, 2, 3, 1, 2, 3)$	1
$(5, 1, 1, 2, 5, 1, 1, 2)$	1

[9] From Francois Fleuret slides

Recurrent neural networks

1. Usually, we want to predict a vector at some steps[10].



- We can process input x by applying the following recurrence equation.

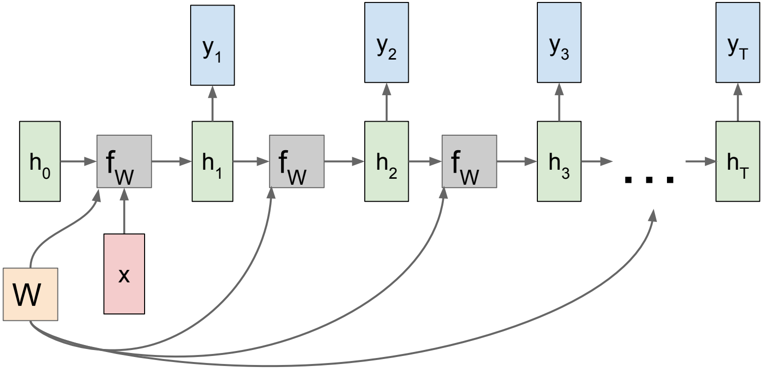
$$\boxed{h_t = f_w\left(x_t, h_{t-1}\right)}$$

- Assume that the activation function is tanh.

$$\boxed{\begin{aligned} h_t &= \tanh\left(Ux_t, Wh_{t-1}\right) \\ y_t &= Vh_t \end{aligned}}$$

[10] From Fei-Fei Li et al. slides

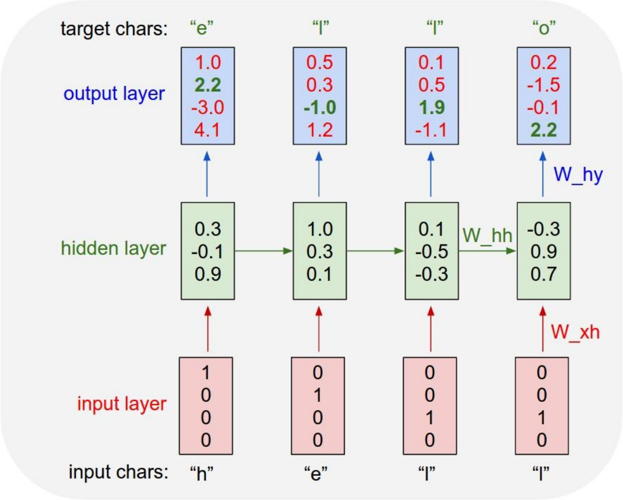
1. The computational graph is [11].



[11] From Fei-Fei Li et al. slides

Recurrent neural networks (character level language model)

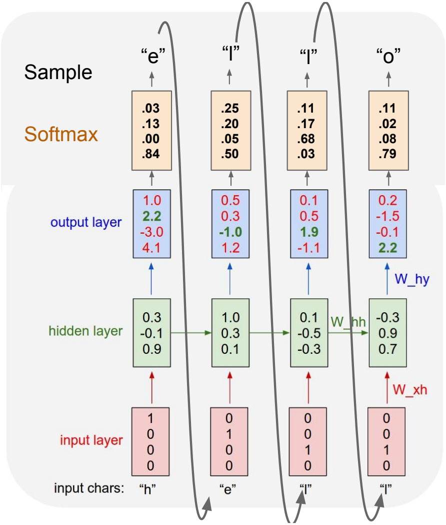
1. Assume that the vocabulary is [h,e,l,o] [12].
2. Example training sequence: hello
3. At the output layer, we use softmax.



[12] From Fei-Fei Li et al. slides

Recurrent neural networks (character level language model)

1. Assume that the vocabulary is [h,e,l,o] [13].
2. Example training sequence: hello
3. At the output layer, we use softmax.



[13] From Fei-Fei Li et al. slides

Training recurrent neural networks

Backpropagation through time

1. We have a collection of labeled samples.

$$S = \{(x_i, y_i), (x_2, y_2), \ldots, (x_m, y_m)\}$$

   where

   - $x_i = (x_{i,0}, x_{i,1}, \ldots, x_{i,T})$ is the input sequence and
   - $y_i = (y_{i,0}, y_{i,1}, \ldots, y_{i,T})$ is the output sequence.

2. The goal is to find weights of the network that minimizes the error between $\hat{y}_i = (\hat{y}_{i,0}, \hat{y}_{i,1}, \ldots, \hat{y}_{i,T})$ and $y_i = (y_{i,0}, y_{i,1}, \ldots, y_{i,T})$
3. In forward phase, the input is given to the network and the output is calculated.
4. In backward phase, the gradient of cost function with respect to the weights are calculated and the weights are updated.

Forward phase

1. The input is given to a four-layers network and the output will be calculated.
2. Consider hidden-layers denoted by $h^{(1)}$ and $h^{(2)}$.
3. The output of the first hidden layer equals to

$$\boxed{h_i^{(1)}(t) = \sigma_1\left(\sum_j u_{ij}^{(1)} x_j(t) + \sum_j w_{ij}^{(1)} h_j^{(1)}(t-1) + b_i^{(1)}\right)}$$

4. The output of the second hidden layer equals to

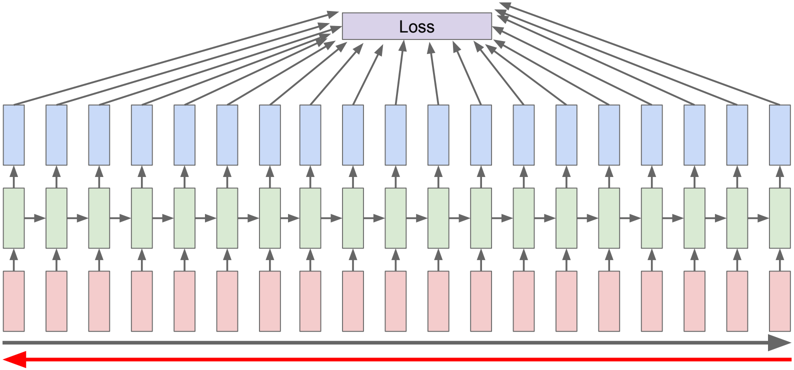
$$\boxed{h_i^{(2)}(t) = \sigma_2\left(\sum_j u_{ij}^{(2)} h_j^{(1)}(t) + \sum_j w_{ij}^{(2)} h_j^{(2)}(t-1) + b_i^{(2)}\right)}$$

5. The output equals to

$$\boxed{\hat{y}_i(t) = \sigma_3\left(\sum_j v_{ij} h_j^{(2)}(t) + c_i\right)}$$

Backpropagation through time

1. Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient[14].



[14] From Fei-Fei Li et al. slides

Backpropagation through time

1. We must find

$$\boxed{\begin{array}{c}\nabla_V L(\theta)\\ \nabla_W L(\theta)\\ \nabla_U L(\theta)\\ \nabla_b L(\theta)\\ \nabla_c L(\theta)\end{array}}$$

2. Then we threat the network as usual multi-layer network and apply the backpropagation on the unrolled network.

Backpropagation through time

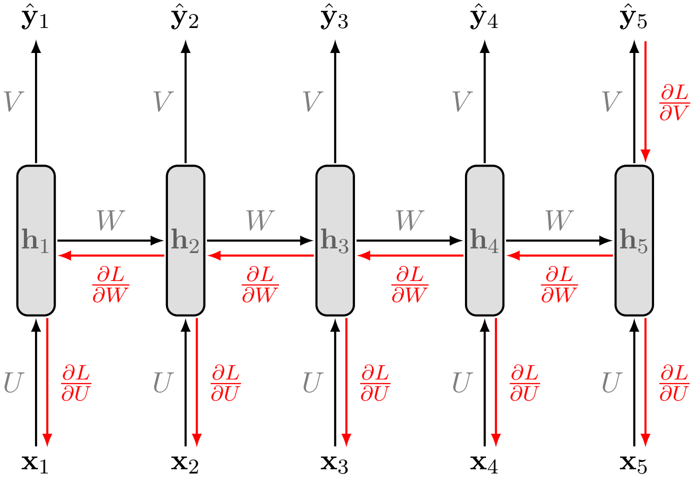
1. Let consider a network with one hidden layer.
2. We use softmax activation function for output layer
3. Suppose that $L(t)$ be the loss at time t.
4. If $L(t)$ is the negative log-likelihood of $y(t)$ given $x(1), x(2), \ldots, x(t)$, then

$$L\left(\{x(1), x(2), \ldots, x(T)\}, \{y(1), y(2), \ldots, y(T)\}\right) = \sum_t L(t)$$

$$L(t) = -\log p_{model}\left(y(t) | x(1), x(2), \ldots, x(t)\right)$$

Backpropagation through time

1. We backpropagate the gradient in the following way (L is loss function).


$\hat{\mathbf{y}}_1$
$\hat{\mathbf{y}}_2$
$\hat{\mathbf{y}}_3$
$\hat{\mathbf{y}}_4$
$\hat{\mathbf{y}}_5$
V
V
V
V
V
$\frac{\partial L}{\partial V}$
$\mathbf{h}_1$
W
$\frac{\partial L}{\partial W}$
$\mathbf{h}_2$
W
$\frac{\partial L}{\partial W}$
$\mathbf{h}_3$
W
$\frac{\partial L}{\partial W}$
$\mathbf{h}_4$
W
$\frac{\partial L}{\partial W}$
$\mathbf{h}_5$
U
$\frac{\partial L}{\partial U}$
U
$\frac{\partial L}{\partial U}$
U
$\frac{\partial L}{\partial U}$
U
$\frac{\partial L}{\partial U}$
U
$\frac{\partial L}{\partial U}$
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_3$
$\mathbf{x}_4$
$\mathbf{x}_5$

Credit Trivedi & Kondor

Backpropagation through time

1. In forward phase, hidden layer computes

$$h_t = \tanh\left(W^\top h_{t-1} + U^\top x_t + b\right)$$

2. In forward phase, output layer computes

$$\begin{aligned} o_t &= V^\top h_t + c \\ \hat{y}_t &= softmax\left(o_t\right) \end{aligned}$$

Backpropagation through time

1. Calculating the gradient

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

$$\left(\nabla_{o(t)} L\right)_i = \frac{\partial L}{\partial o_i(t)} = \frac{\partial L}{\partial L(t)} \frac{\partial L(t)}{\partial o_i(t)}$$

2. By using softmax in output layer, we have

$$\hat{y}_i(t) = \frac{e^{o_i(t)}}{\sum_k e^{o_k(t)}}$$

$$\frac{\partial \hat{y}_i(t)}{\partial o_j(t)} = \frac{\partial \frac{e^{o_i(t)}}{\sum_k e^{o_k(t)}}}{\partial o_j(t)}$$

$$= \begin{cases} \hat{y}_i(t)(1 - \hat{y}_j(t)) & i = j \\ \\ -\hat{y}_j(t)\hat{y}_i(t) & i \neq j \end{cases}$$

Backpropagation through time

1. We compute gradient of loss function with respect to $o_i(t)$

$$\begin{aligned}
L(t) &= -\sum_k y_k(t) log(\hat{y}_k(t)) \\
\frac{\partial L(t)}{\partial o_i(t)} &= -\sum_k y_k(t) \frac{\partial log(\hat{y}_k(t))}{\partial o_i(t)} \\
&= -\sum_k y_k(t) \frac{\partial log(\hat{y}_k(t))}{\partial \hat{y}_k(t)} \times \frac{\partial \hat{y}_k(t)}{\partial o_i(t)} \\
&= -\sum_k y_k(t) \frac{1}{\hat{y}_k(t)} \times \frac{\partial \hat{y}_k(t)}{\partial o_i(t)}
\end{aligned}$$

Backpropagation through time

1. We compute gradient of loss function with respect to $o_i(t)$

$$
\begin{aligned}
\frac{\partial L(t)}{\partial o_i(t)} &= -\sum_k y_k(t)\frac{1}{\hat{y}_k(t)} \times \frac{\partial \hat{y}_k(t)}{\partial o_i(t)} \\
&= -y_i(t)(1-\hat{y}_i(t)) - \sum_{k\neq i} y_k(t)\frac{1}{\hat{y}_k(t)}(-\hat{y}_k(t).\hat{y}_i(t)) \\
&= -y_i(t)(1-\hat{y}_i(t)) + \sum_{k\neq i} y_k(t).\hat{y}_i(t) \\
&= -y_i(t) + y_i(t)\hat{y}_i(t) + \sum_{k\neq i} y_k(t).\hat{y}_i(t) \\
&= \hat{y}_i(t)\left(y_i(t) + \sum_{k\neq i} y_k(t)\right) - y_i(t)
\end{aligned}
$$

Backpropagation through time

1. We had

$$\boxed{\frac{\partial L(t)}{\partial o_i(t)} = \hat{y}_i(t)\left(y_i(t) + \sum_{k\neq i} y_k(t)\right) - y_i(t)}$$

2. Since $y(t)$ is a one hot encoded vector for the labels, so $\sum_k y_k(t) = 1$ and $y_i(t) + \sum_{k\neq 1} y_k(t) = 1$. So we have

$$\boxed{\frac{\partial L(t)}{\partial o_i(t)} = \hat{y}_i(t) - y_i(t)}$$

3. This is a very simple and elegant expression.

Backpropagation through time

1. At the final time step T, $h(T)$ has only as $o(T)$ as a descendant

$$\boxed{\nabla_{h(T)} L = V^{\top} \nabla_{o(T)} L}$$

2. We can then iterate backward in time to back-propagate gradients through time, from $t = T - 1$ down to $t = 1$.

$$\boxed{\begin{aligned} \nabla_{h(t)} L &= \left(\frac{\partial h(t+1)}{\partial h(t)} \right)^{\top} \left(\nabla_{h(t+1)} L \right) \\ &\quad + \left(\frac{\partial o(t)}{\partial h(t)} \right)^{\top} \left(\nabla_{o(t)} L \right) \\ &= W^{\top} \operatorname{diag}\left(\left(1 - (h(t+1))^2 \right) \right) \left(\nabla_{h(t+1)} L \right) \\ &\quad + V^{\top} \left(\nabla_{o(t)} L \right) \end{aligned}}$$

Backpropagation through time

1. The gradient on the remaining parameters is given by

$$\nabla_c L = \sum_t \left(\frac{\partial o(t)}{\partial c(t)}\right)^\top \nabla_{o(t)} L = \sum_t \nabla_{o(t)} L$$

$$\nabla_b L = \sum_t \left(\frac{\partial h(t)}{\partial b(t)}\right)^\top \nabla_{h(t)} L = \sum_t \operatorname{diag}\left(\left(1-(h(t))^2\right)\right)\left(\nabla_{h(t)}\right) L$$

$$\nabla_V L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i(t)}\right)^\top \nabla_{V(t)} o(t) = \sum_t \left(\nabla_{o(t)} L\right) h(t)^\top$$

$$\nabla_W L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i(t)}\right)^\top \nabla_{W(t)} h_i(t) = \sum_t \operatorname{diag}\left(\left(1-(h(t))^2\right)\right)\left(\nabla_{h(t-1)} L\right)(h(t-1)$$

$$\nabla_U L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i(t)}\right)^\top \nabla_{U(t)} h_i(t) = \sum_t \operatorname{diag}\left(\left(1-(h(t))^2\right)\right)\left(\nabla_{h(t)} L\right)(x(t))^\top$$

Backpropagation through time

1. Finally, bearing in mind that the weights to and from each unit in the hidden layer are the same at every time-step, we sum over the whole sequence to get the derivatives with respect to each of the network weights.

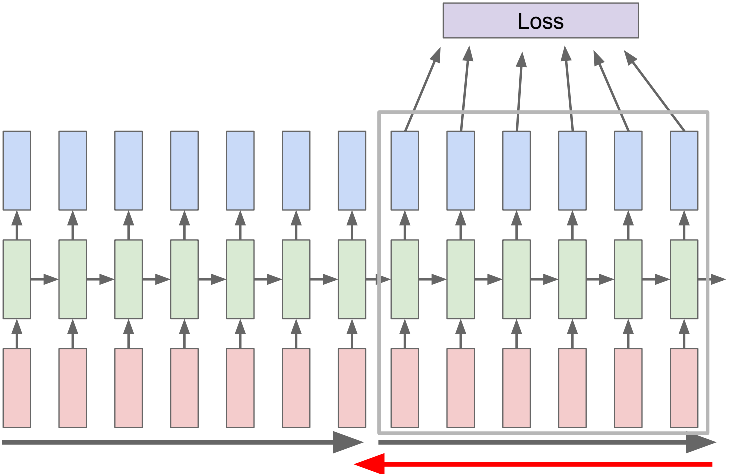
$$\Delta U = -\eta \sum_{t=1}^{T} \nabla_U L(t) \qquad \Delta b = -\eta \sum_{t=1}^{T} \nabla_b L(t)$$

$$\Delta W = -\eta \sum_{t=1}^{T} \nabla_W L(t)$$

$$\Delta V = -\eta \sum_{t=1}^{T} \nabla_V L(t) \qquad \Delta c = -\eta \sum_{t=1}^{T} \nabla_c L(t)$$

Truncated Backpropagation through time

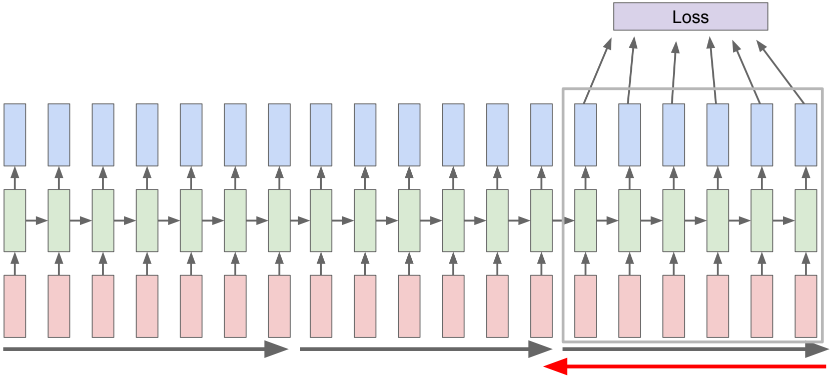
1. Run forward and backward through chunks of the sequence instead of whole sequence[15].



[15] From Fei-Fei Li et al. slides

Truncated Backpropagation through time

1. Run forward and backward through chunks of the sequence instead of whole sequence[16].

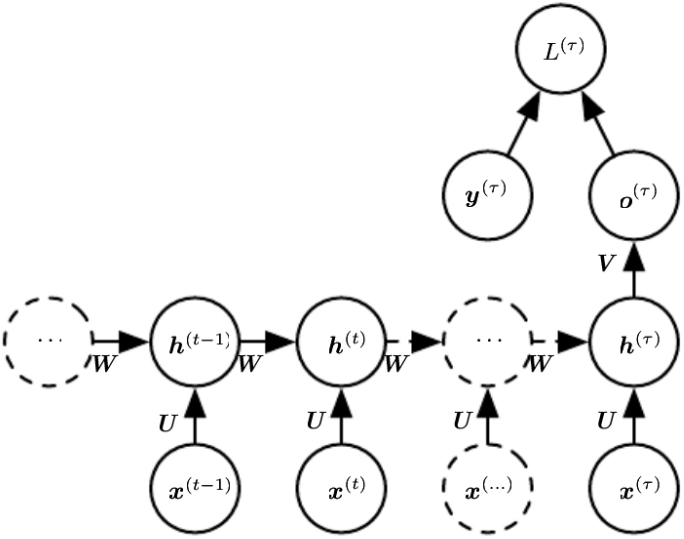


[16] From Fei-Fei Li et al. slides

Design patterns of RNN

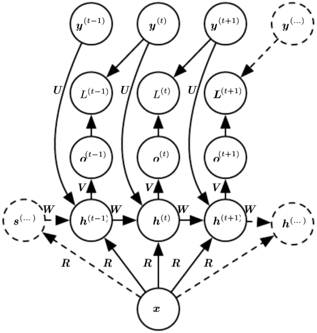
Design patterns of RNN (summarization)

1. Producing a single output and have recurrent connections between hidden units.
2. This is useful for summarizing a sequence such as sentiment analysis



Design patterns of RNN (fixed vector as input)

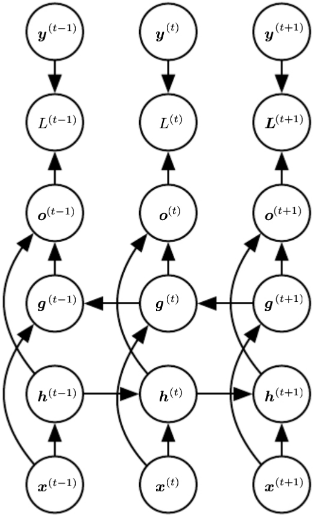
1. Sometimes we are interested in only taking a single, fixed sized vector x as input, which generates the y sequence
2. Most common ways to provide an extra input at each time step



3. Other solutions? (Please consider them)
4. Application: Image caption generation

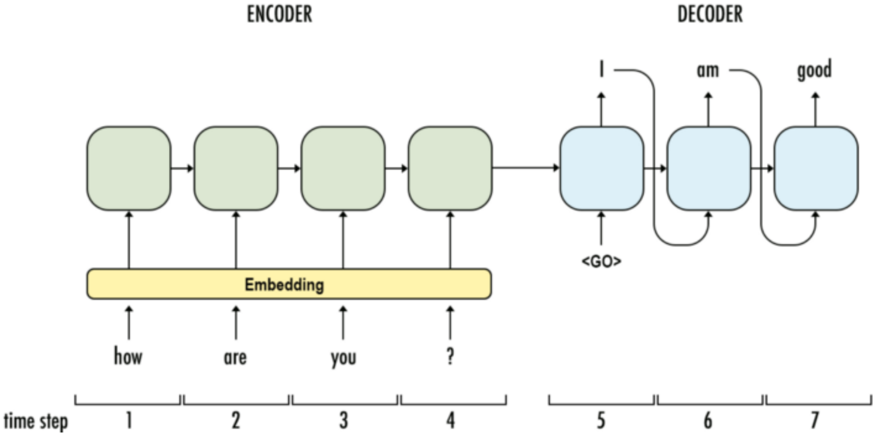
Bidirectional RNNs

1. We considered RNNs in the context of a sequence $x(t)(t = 1, \dots, T)$
2. In many applications, $y(t)$ may depend on the whole input sequence.
3. Bidirectional RNNs were introduced to address this need.

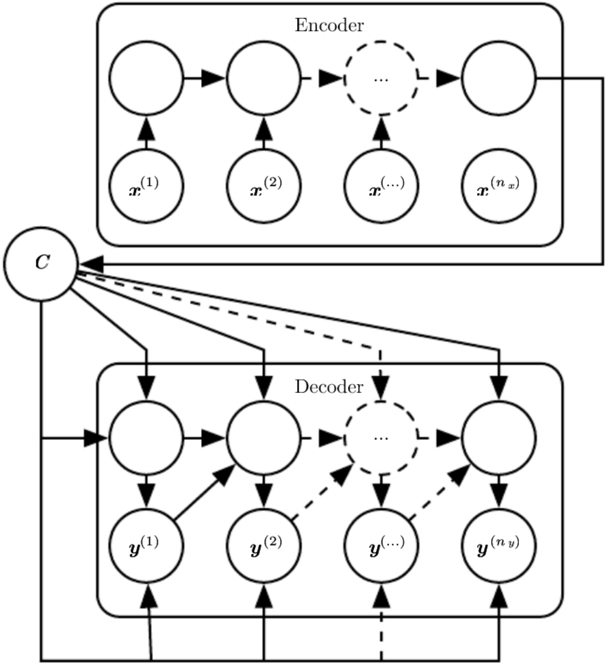


Encoder-Decoder

1. How do we map input sequences to output sequences that are not necessarily of the same length?
2. The input to RNN is called context, we want to find a representation of the context C.
3. C may be a vector or a sequence that summarizes $x = \{x(1), \dots, x(n_x)\}$



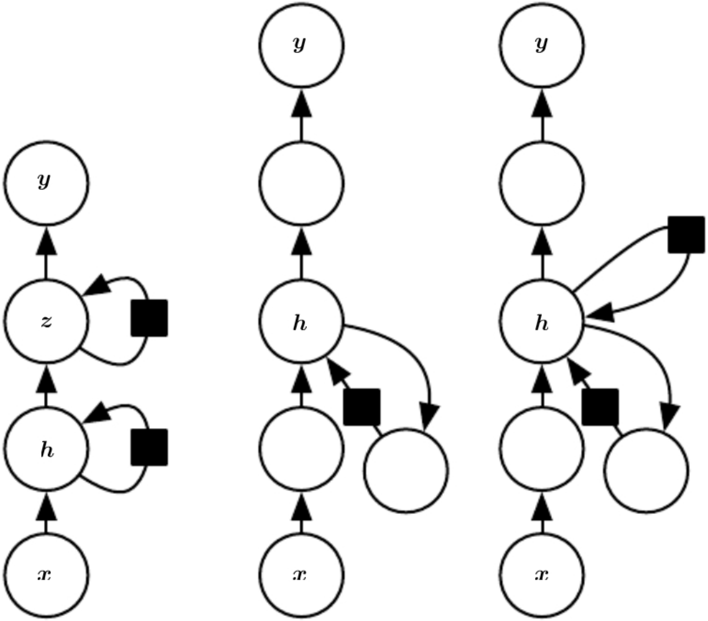
Encoder-Decoder



Deep recurrent networks

1. The computations in RNNs can be decomposed into three blocks of parameters and transformations
    - Input to hidden state
    - Previous hidden state to the next
    - Hidden state to the output
2. Each of these transforms were learned affine transformations followed by a fixed nonlinearity.
3. Introducing depth in each of these operations is advantageous.
4. The intuition on why depth should be more useful is quite similar to that in deep feed-forward networks
5. Optimization can be made much harder, but can be mitigated by tricks such as introducing skip connections

Deep recurrent networks



(a) (b) (c)

Long-term dependencies

Long-term dependencies

1. RNNs involve the composition of a function multiple times, once per time step.
2. This function composition resembles matrix multiplication.
3. Consider the recurrence relationship $h(t+1) = W^\top h(t)$
4. This is very simple RNN without a nonlinear activation and x.
5. This recurrence equation can be written as: $h(t) = (W^t)^\top h(0)$.
6. If W has an eigendecomposition of form $W = Q \Lambda Q^\top$ with orthogonal Q.
7. The recurrence becomes $h(t) = (W^t)^\top h(0) = Q^\top \Lambda^t Q h(0)$.
   Q is matrix composed of eigenvectors W.
   Λ is a diagonal matrix with eigenvalues placed on the diagonals.

Long-term dependencies

1. The recurrence becomes $h(t) = (W^t)^\top h(0) = Q^\top \Lambda^t Q h(0)$.

   Q is matrix composed of eigenvectors W.

   Λ is a diagonal matrix with eigenvalues placed on the diagonals.

2. Eigenvalues are raised to t: Quickly decay to zero or explode. Consider

Vanishing gradients

$$\Lambda = \begin{bmatrix} -0.6180 & 0 \\ 0 & 1.6180 \end{bmatrix} \longrightarrow \Lambda^{10} = \begin{bmatrix} \boxed{0.0081} & 0 \\ 0 & \boxed{122.9919} \end{bmatrix}$$

Exploding gradients

3. **Problem: Gradients propagated over many stages tend to vanish (most of the time) or explode (relatively rarely)**

Why do gradients vanish or explode?

1. The expression for $h(t)$ was $h(t) = \tanh\left(Wh(t-1) + Ux(t)\right)$.
2. The partial derivative of loss wrt to hidden states equals to

$$\begin{aligned}\frac{\partial L}{\partial h(t)} &= \frac{\partial L}{\partial h(T)} \frac{\partial h(T)}{\partial h(t)} \\ &= \frac{\partial L}{\partial h(T)} \prod_{k=t}^{T-1} \frac{\partial h(k+1)}{\partial h(k)} \\ &= \frac{\partial L}{\partial h(T)} \prod_{k=t}^{T-1} D_{k+1} W_k^\top\end{aligned}$$

where

$$D_{k+1} = diag\left(1 - \tanh^2\left(Wh(t-1) + Ux(t)\right)\right)$$

Why do gradients vanish or explode?

1. For any matrices A, B , we have $\|AB\| \leq \|A\|\|B\|$

$$\left\|\frac{\partial L}{\partial h(t)}\right\| = \left\|\frac{\partial L}{\partial h(T)} \prod_{k=t}^{T-1} D_{k+1} W_k^\top\right\| \leq \left\|\frac{\partial L}{\partial h(T)}\right\| \prod_{k=t}^{T-1} \left\|D_{k+1} W_k^\top\right\|$$

2. Since $\|A\|$ equals to the largest singular value of A ($\sigma_{max}(A)$), we have

$$\left\|\frac{\partial L}{\partial h(t)}\right\| = \left\|\frac{\partial L}{\partial h(T)} \prod_{k=t}^{T-1} D_{k+1} W_k^\top\right\| \leq \left\|\frac{\partial L}{\partial h(T)}\right\| \prod_{k=t}^{T-1} \sigma_{max}(D_{k+1}) \sigma_{max}(W_k)$$

3. Hence the gradient norm can shrink to zero or grow up exponentially fast depending on the σ_{max}.

Echo state networks

1. Set the recurrent weights such that they do a good job of capturing past history and learn only the output weights
2. Methods: Echo State Machines and Liquid State Machines
3. The general methodology is called reservoir computing
4. How to choose the recurrent weights?
5. In Echo State Machines, choose recurrent weights such that the hidden-to-hidden transition Jacobian has eigenvalues close to 1

Other solutions

1. Adding skip connection through time : Adding direct connections from variables in the distant past to the variables in the present.
2. Leaky units: Having units with self-connections.
3. Removing connections: Removing length-one connections and replacing them with longer connections.

Leaky units

1. Recall that

$$\boxed{h_t = \sigma\left(W^\top h_{t-1} + U^\top x_t\right)}$$

2. Now, consider the following equation

$$\boxed{h_{t,i} = \left(1 - \frac{1}{\gamma_i}\right) h_{t-1,i} + \frac{1}{\gamma_i}\sigma\left(W^\top h_{t-1} + U^\top x_t\right)}$$

   where $1 \leq \gamma_i \leq \infty$

3. Now, considering the following cases

   When $\gamma_i = 1$, The resulting network becomes the ordinary RNN.

   When $\gamma_i = \infty$, The resulting network becomes by no changing the hidden states.

   When $\gamma_i > 1$, The gradient propagates more easily.

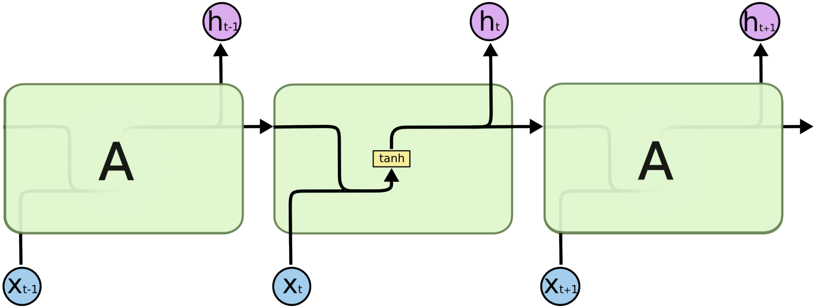
   When $\gamma_i \gg 1$, The hidden states change slowly and the network incorporates inputs and past hidden states.

Gated units

1. RNNs can accumulate but it might be useful to forget.
2. Creating paths through time where derivatives can flow.
3. Learn when to forget!
4. Gates allow learning how to read, write and forget.
5. We consider two gated units:
   - Long Short-Term Memory (LSTM)
   - Gated Recurrent Unit (GRU)

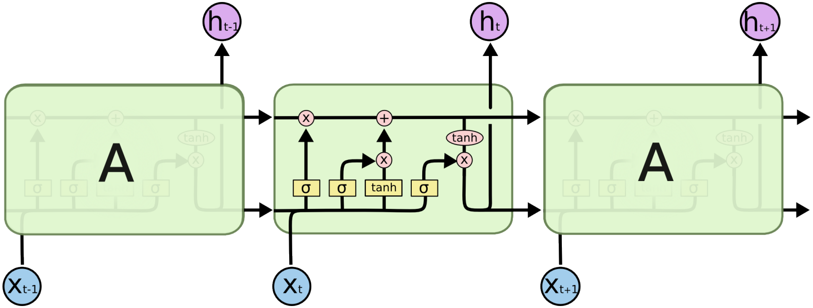
Long short term memory

1. LSTMs are explicitly designed to avoid the long-term dependency problem.
2. All recurrent neural networks have the form of a chain of repeating modules of neural network.
3. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.



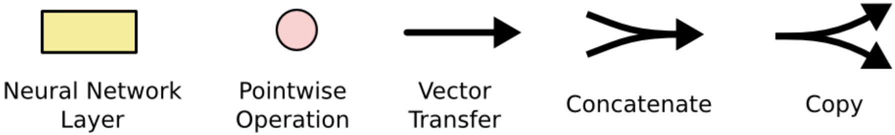
Long short term memory

1. LSTMs also have this chain like structure, but the repeating module has a different structure.
2. Instead of having a single neural network layer, there are four, interacting in a very special way.



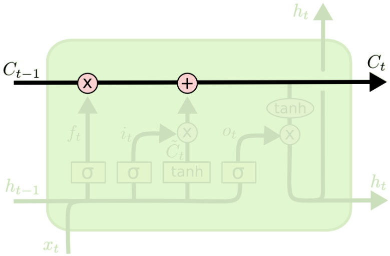
Long short term memory

1. Let us to define the following notations



2. In the above figure, each line carries an entire vector, from the output of one node to the inputs of others.
   - The pink circles represent point-wise operations, like vector addition.
   - The yellow boxes are learned neural network layers.
   - Lines merging denote concatenation
   - Line forking denote its content being copied and the copies going to different locations.

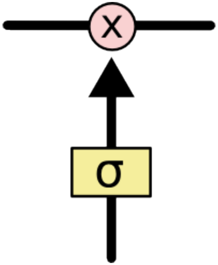
1. The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.
2. The cell state is kind of like a conveyor belt.
3. It runs straight down the entire chain, with only some minor linear interactions.
4. It's very easy for information to just flow along it unchanged.



5. The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called **gates**

Long short term memory

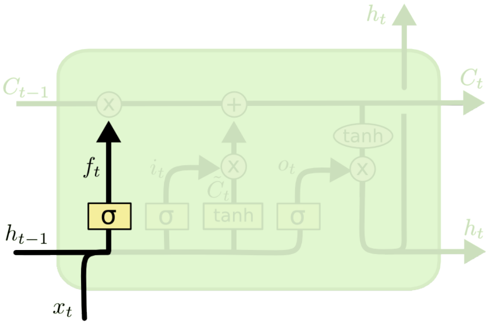
1. Gates are a way to optionally let information through.
2. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.



3. The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means let nothing through, while a value of one means let everything through!
4. An LSTM has three of these gates, to protect and control the cell state.

Long short term memory

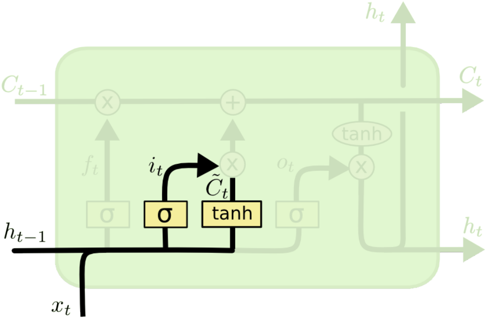
1. This decision is made by a sigmoid layer called the forget gate layer.
2. It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}.



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

Long short term memory

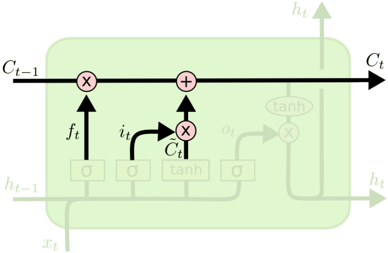
1. The next step is to decide what new information we're going to store in the cell state.
2. This has the two following parts that could be added to the state.
   - A sigmoid layer called the input gate layer decides which values we'll update.
   - A tanh layer creates a vector of new candidate values, $\tilde{C}_t$.
3. Then we'll combine these two to create an update to the state.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Long short term memory

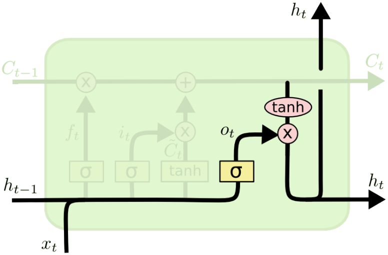
1. Now, we must update the old cell state, C_{t-1} into the new cell state C_t.
2. We multiply the old state by f_t, forgetting the things we decided to forget earlier.
3. Then we add $i_t \times \tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.



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

Long short term memory

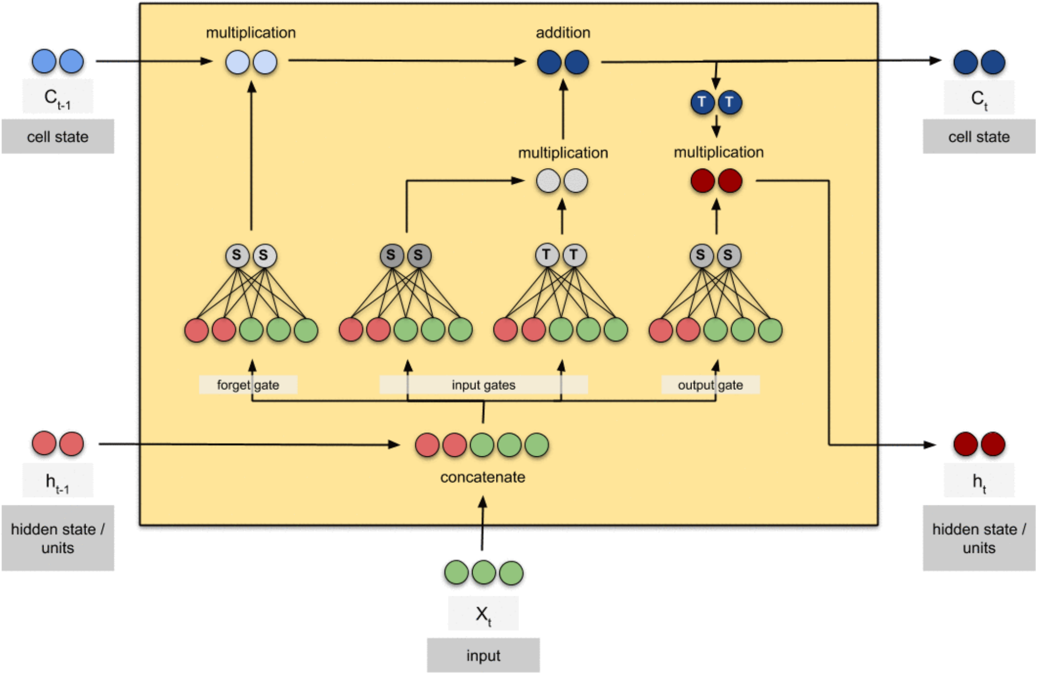
1. Finally, we need to decide what we're going to output.
2. This output will be based on our cell state, but will be a filtered version.
   - First, we run a sigmoid layer which decides what parts of the cell state we're going to output.
   - Then, we put the cell state through tanh and multiply it by the output of the sigmoid gate.
3. So that we only output the parts we decided to.



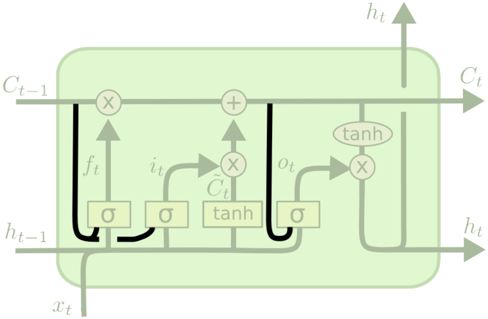
$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$

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

Long short term memory



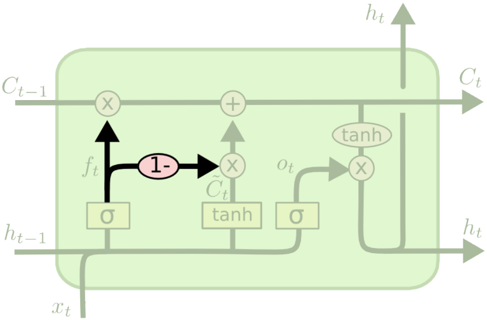
1. One popular LSTM variant is adding peephole connections. This means that we let the gate layers look at the cell state (Gers and Schmidhuber 2000).



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1},} h_{t-1}, x_t] \ + \ b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1},} h_{t-1}, x_t] \ + \ b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \ + \ b_o\right)$$

Variants on Long short term memory

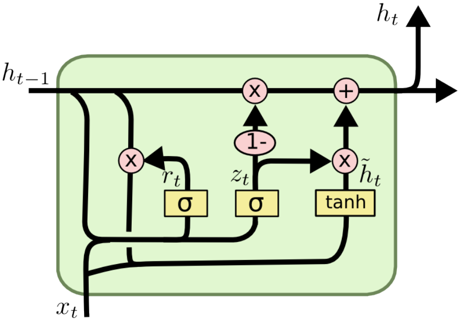
1. Another variation is to use coupled forget and input gates.
2. Instead of separately deciding what to forget and what we should add new information to, we make those decisions together.
3. We only forget when we're going to input something in its place. We only input new values to the state when we forget something older.



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

Variants on Long short term memory (GRU)

1. GRU combines the forget and input gates into a single (Cho et al. n.d.).
2. It merges the cell and hidden states, and makes some other changes.
3. The resulting model is simpler than standard LSTM models, and has been growing increasingly popular.



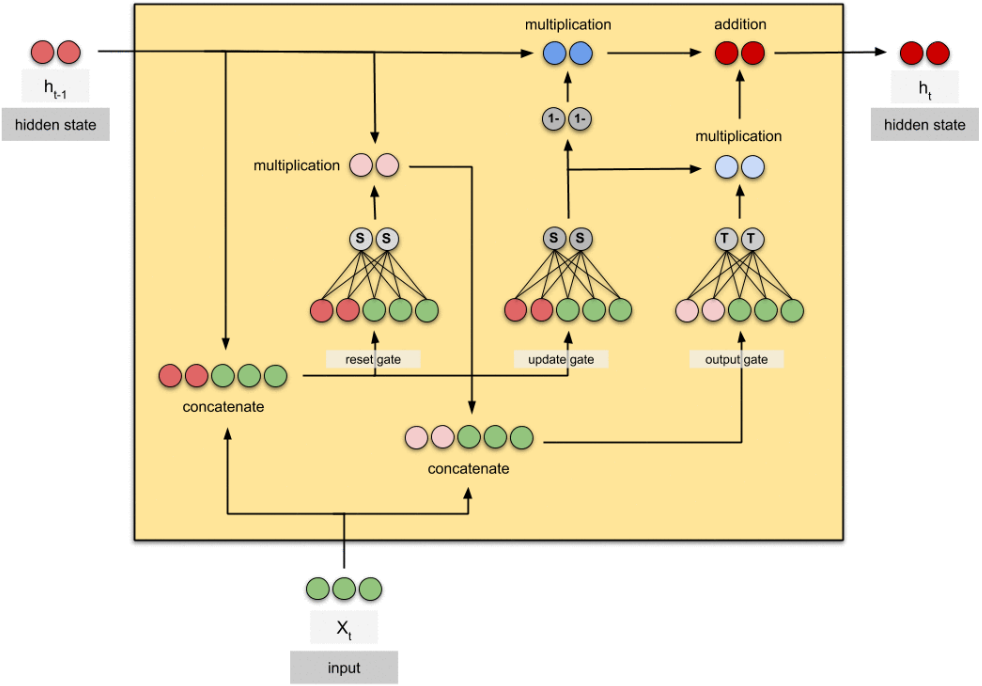
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Gated recurrent units (GRU)



Variants on Long short term memory

1. There are also other LSTM variants such as
    - Jan Koutnik, et. al. "A clockwork RNN", Proceedings of the 31st International Conference on International Conference on Machine Learning, 2014.
    - Kaisheng Yao, et. al. "Depth-Gated Recurrent Neural Networks", https://arxiv.org/pdf/1508.03790v2.pdf.

Optimization for long-term dependencies

1. Two simple solutions for gradient vanishing / exploding.
   - For vanishing gradients: Initialization + ReLus
   - Trick for exploding gradient: clipping trick

     When $\|g\| > v$ Then

     $$g \leftarrow \frac{vg}{\|g\|}$$

2. Vanishing gradient happens when the optimization gets stuck in a saddle point, the gradient becomes too small for the optimization to progress. This can also be fixed by using gradient descent with momentum or other methods.
3. Exploding gradient happens when the gradient becomes too big and you get numerical overflow. This can be easily fixed by initializing the network's weights to smaller values.

Optimization for long-term dependencies

1. Gradient clipping helps to deal with exploding gradients but it does not help with vanishing gradients.
2. Another way is encourage creating paths in the unfolded recurrent architecture along which the product of gradients is near 1.
3. Solution: regularize to encourage information flow.
4. We want that $\left(\nabla_{h(t)} L\right) \frac{\partial h(t)}{\partial h(t-1)}$ to be as large as $\nabla_{h(t)} L$.
5. One regularizer is

$$\sum_t \left(\frac{\left\| \left(\nabla_{h(t)} L\right) \frac{\partial h(t)}{\partial h(t-1)} \right\|}{\left\| \nabla_{h(t)} L \right\|} - 1 \right)^2$$

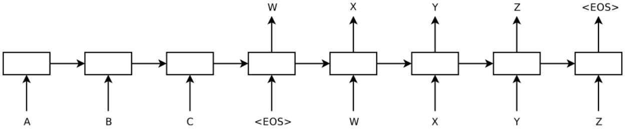
6. Computing this regularizer is difficult and its approximation is used.

Attention models

Sequence to sequence modeling

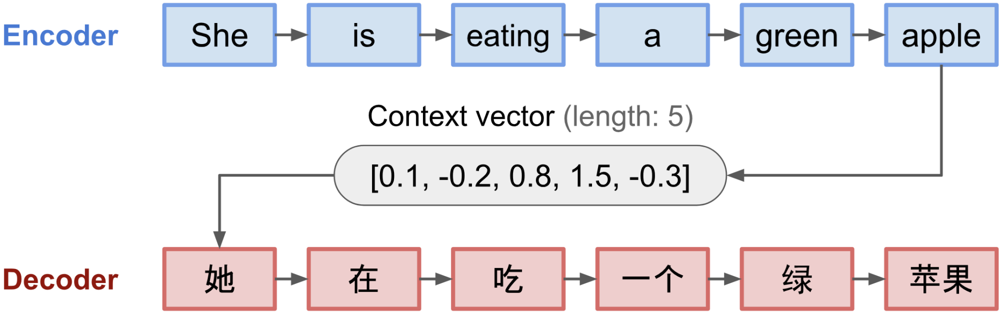
1. Sequence to sequence modeling transforms an input sequence (source) to a new one (target) and both sequences can be of arbitrary lengths.
2. Examples of transformation tasks include
   - Machine translation between multiple languages in either text or audio
   - Question-answer dialog generation
   - Parsing sentences into grammar trees.
3. The sequence to sequence model normally has an encoder-decoder architecture, composed of
   - An encoder processes the input sequence and compresses the information into a context vector of a fixed length.

     This representation is expected to be a good summary of the meaning of the whole source sequence.
   - A decoder is initialized with the context vector to emit the transformed output. The early work only used the last state of the encoder network as the decoder initial state.



Credit: Roger Grosse and Jimmy Ba

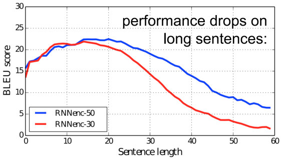
1. Both the encoder and decoder are recurrent neural networks such as LSTM or GRU units.



2. A critical disadvantage of this fixed-length context vector design is incapability of remembering long sentences.

Attention models

1. RNNs cannot remember longer sentences and sequences due to the vanishing/exploding gradient problem.
2. The performance of the encoder-decoder network degrades rapidly as the length of the input sentence increases.



3. In psychology, attention is the cognitive process of selectively concentrating on one or a few things while ignoring others.

Example (Counting the number of people in a photo)

Counting the number of heads and ignoring the rest.

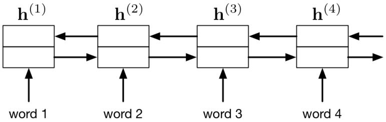
Attention models

1. The attention mechanism was born to help memorize long source sentences in neural machine translation (NMT) (Bahdanau, Cho, and Bengio 2015).
2. Instead of building a single context vector out of the encoder's last hidden state, the goal of attention is to create shortcuts between the context vector and the entire source input.
3. The weights of these shortcut connections are customizable for each output element.
4. The alignment between the source and target is learned and controlled by the context vector.
5. Essentially the context vector consumes three pieces of information:
    - Encoder hidden states
    - Decoder hidden states
    - Alignment between source and target

1. Assume that we have a source sequence x of length n and try to output a target sequence y of length m

$$x = [x_1, x_2, \ldots, x_n]$$
$$y = [y_1, y_2, \ldots, y_m]$$

2. The encoder is a bidirectional RNN with a forward hidden state $\overrightarrow{h}_i$ and a backward one $\overleftarrow{h}_i$.



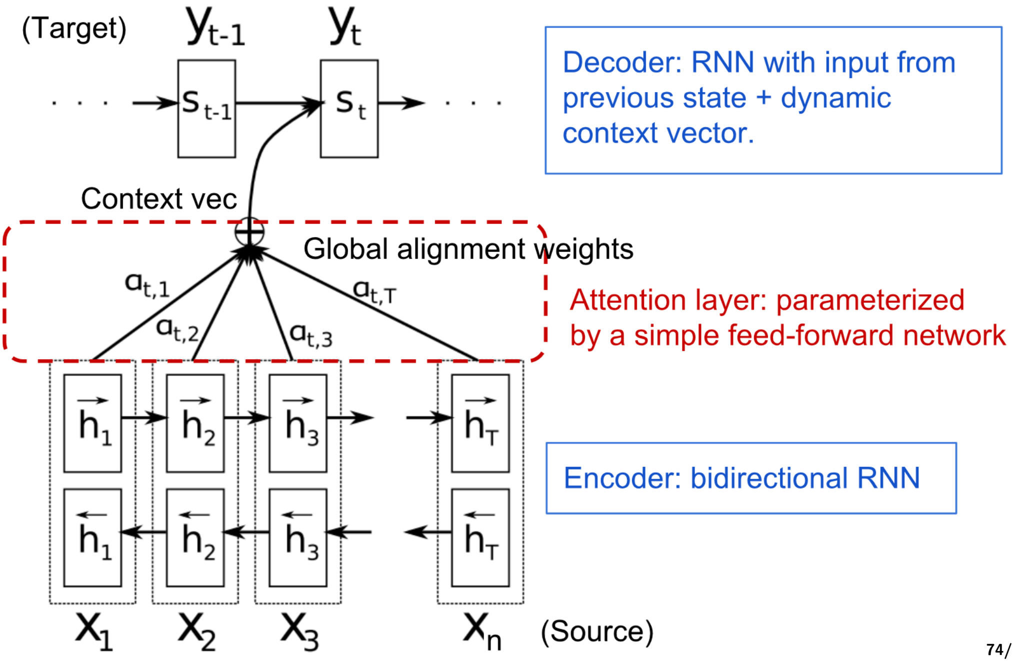
Credit: Roger Grosse and Jimmy Ba

3. A simple concatenation of two represents the encoder state.
4. The motivation is to include both the preceding and following words in the annotation of one word.

$$\boldsymbol{h}_i = \left[\overrightarrow{\boldsymbol{h}}_i^\top ; \overleftarrow{\boldsymbol{h}}_i^\top\right]^\top \quad i = 1, 2, \ldots, n$$

Attention models

1. Model of attention



Decoder: RNN with input from previous state + dynamic context vector.

Attention layer: parameterized by a simple feed-forward network

Encoder: bidirectional RNN

1. The decoder network has hidden state $\boldsymbol{s}_t = f(\boldsymbol{s}_{t-1}, y_{t-1}, \mathbf{c}_t)$ at position $t = 1, 2, \ldots, m$.
2. The context vector $\mathbf{c}_t$ is a sum of hidden states of the input sequence, weighted by alignment scores:

$$
\begin{aligned}
\mathbf{c}_t &= \sum_{i=1}^{n} \alpha_{t,i} \boldsymbol{h}_i && \text{Context vector for output } y_t \\
\alpha_{t,i} &= \text{align}(y_t, x_i) && \text{How well two words } y_t \text{ and } x_i \text{ are aligned.} \\
&= \frac{exp\left(\text{score}(\boldsymbol{s}_{t-1}, \boldsymbol{h}_i)\right)}{\sum_{j=1}^{n} \exp\left(\text{score}(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j)\right)} && \text{Softmax of predefined alignment score.}
\end{aligned}
$$

3. The alignment model assigns a score $\alpha_{t,i}$ to the pair of (y_t, x_i) based on how well they match.
4. The set of $\{\alpha_{t,i}\}$ are weights defining how much of each source hidden state should be considered for each output.

Attention models

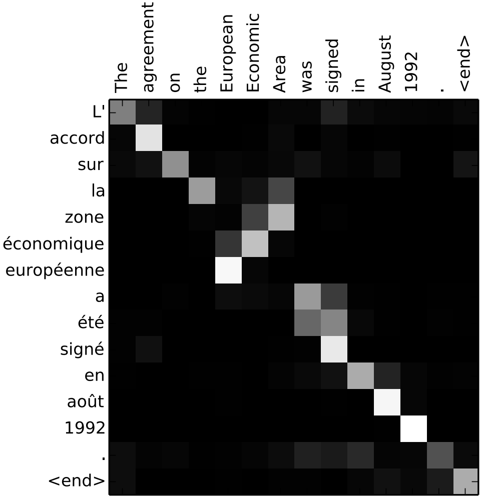
1. The alignment score α is parametrized by a feed-forward network with a single hidden layer (Bahdanau, Cho, and Bengio 2015).
2. This network is jointly trained with other parts of the model.
3. The score function is in the following form.

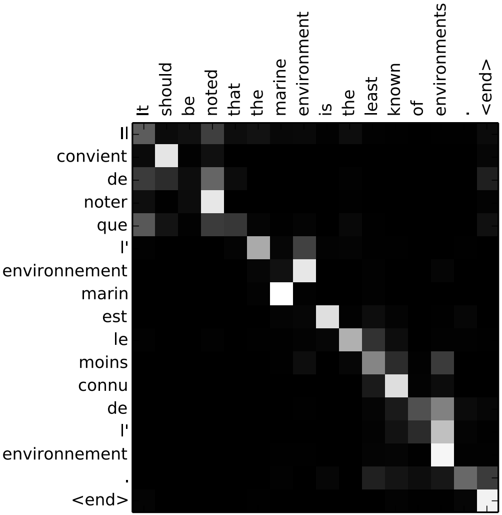
$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$$

where both $\mathbf{V}_a$ and $\mathbf{W}_a$ are weight matrices to be learned in the alignment model.

Alignment scores

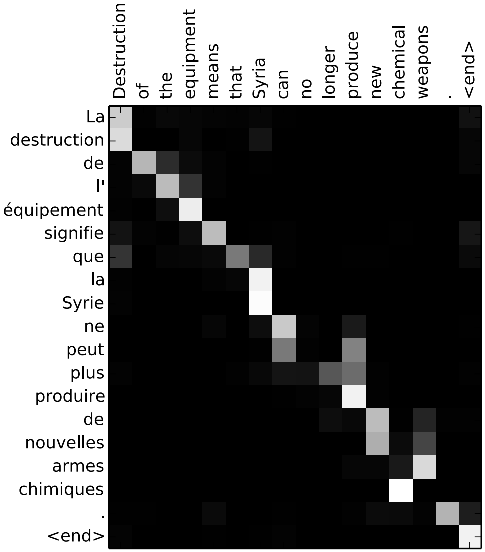
1. The matrix of alignment scores explicitly show the correlation between source and target words.

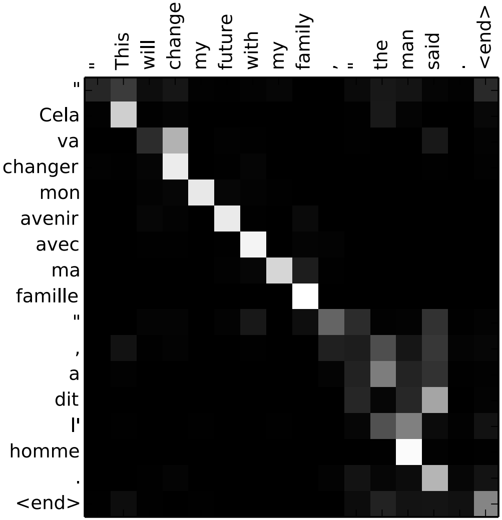




Alignment scores

1. The matrix of alignment scores explicitly show the correlation between source and target words.

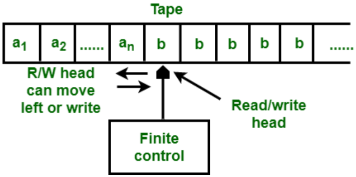




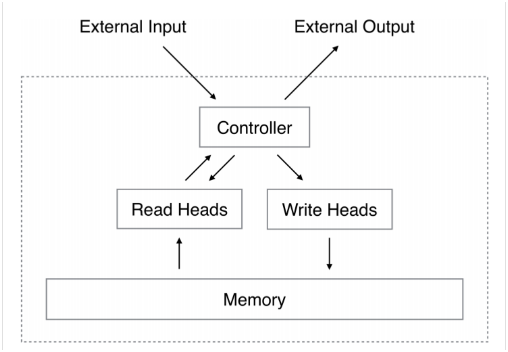
Augmented recurrent neural networks

Neural Turing machine

1. Turing machines, which are proposed by Alan Turing in (1936–7), are simple abstract computational devices intended to help investigate the extent and limitations of what can be computed.



2. Neural Turing machines combine a RNN with an external memory bank (Graves, Wayne, and Danihelka 2014).



Neural Turing machine

1. Memory is an array of vectors, for example array with M entry where each entry is a vector of size N.
2. Two types of addressing are used to address the memory.
   - Content-based addressing.
   - Location-based addressing.
3. Reading from memory

$$r_t \leftarrow \sum_{i}^{M} w_t(i)\mathcal{M}_t(i)$$

$$0 \leq w_t(i) \leq 1$$

$$\sum_{i=1}^{M} w_t(i) = 1$$

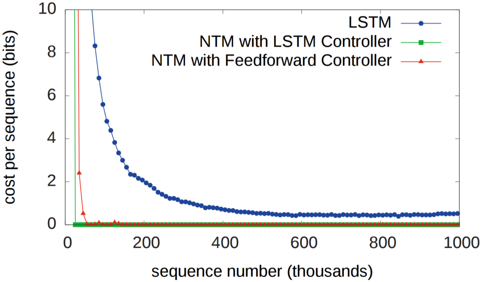
4. Writing to the memory

$$\mathcal{M}_t^{erased}(i) \leftarrow \mathcal{M}_{t-1}(i)[\mathbf{1} - w_t(i)e_t]$$

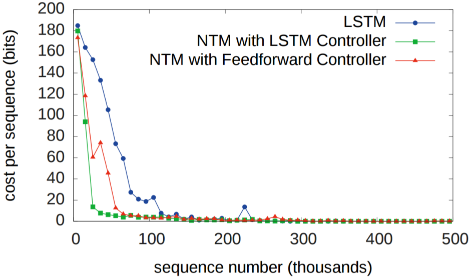
$$\mathcal{M}_t(i) \leftarrow \mathcal{M}_t^{erased}(i) + w_t(i)a_t$$

Neural Turing machine

1. Copying (store and recall a long sequence of arbitrary information).

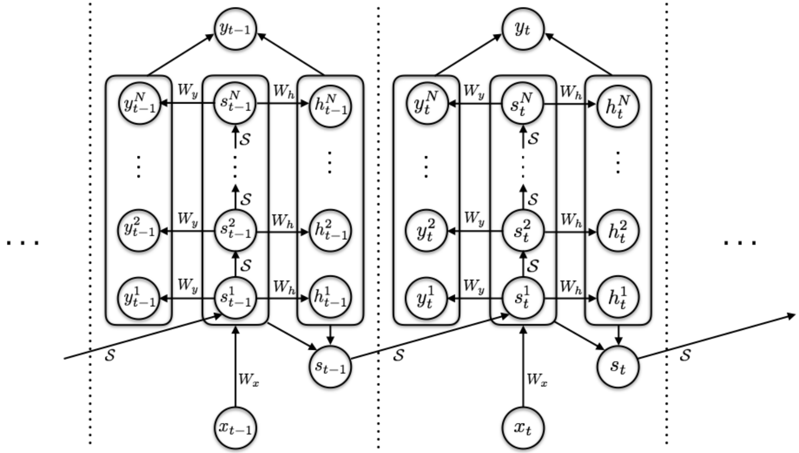


2. Repeated copying (output the copied sequence a specified number of times and then emit an end-of-sequence marker).



Adaptive Computation Time

1. Standard RNNs do the same amount of computation for each time step. .
2. Adaptive computation time (ACT) is a way for RNNs to do different amounts of computation each step (Graves 2016).
3. The big picture idea is simple: allow the RNN to do multiple steps of computation for each time step.



4. Comparison of fixed and adaptive computation times are done in (Fojo, Campos, and Giró-i-Nieto 2018).

Neural programmer[17]

1. Neural nets are excellent at many tasks, but they also struggle to do some basic things like arithmetic, which are trivial in normal approaches to computing.
2. It would be really nice to have a way to fuse neural nets with normal programming, and get the best of both worlds.
3. Neural programmer is one approach to this. It learns to create programs in order to solve a task (Neelakantan, Le, and Sutskever 2016).
4. In fact, it learns to generate such programs without needing examples of correct programs.
5. It discovers how to produce programs as a means to the end of accomplishing some task.

[17] This slide taken from (Olah and Carter 2016)

Reading

Readings

1. Chapter 10 of Deep Learning Book[18]
2. For more information about the augmented networks, please read (Olah and Carter 2016).

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. MIT Press.

References i

- Ba, Jimmy, Volodymyr Mnih, and Koray Kavukcuoglu (2015). “Multiple Object Recognition with Visual Attention”. In: *International Conference on Learning Representations*.
- Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2015). “Neural Machine Translation by Jointly Learning to Align and Translate”. In: *International Conference on Learning Representations*.
- Cho, Kyunghyun et al. (n.d.). “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *Conference on Empirical Methods in Natural Language Processing*, pp. 1724–1734.
- Fojo, Daniel, Vıctor Campos, and Xavier Giró-i-Nieto (2018). “Comparing Fixed and Adaptive Computation Time for Recurrent Neural Networks”. In: *Proceedings of the 6th International Conference on Learning Representations*.
- Gers, Felix A. and Jürgen Schmidhuber (2000). “Recurrent Nets that Time and Count”. In: *International Joint Conference on Neural Networks*, pp. 189–194.
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. MIT Press.

- Graves, Alex (2016). "Adaptive Computation Time for Recurrent Neural Networks". In: *CoRR* abs/1603.08983. URL: http://arxiv.org/abs/1603.08983.
- Graves, Alex, Greg Wayne, and Ivo Danihelka (2014). "Neural Turing Machines". In: *CoRR* abs/1410.5401. URL: http://arxiv.org/abs/1410.5401.
- Gregor, Karol et al. (2015). "DRAW: A Recurrent Neural Network For Image Generation". In: *International Conference on Machine Learning*, pp. 1462–1471.
- Neelakantan, Arvind, Quoc V. Le, and Ilya Sutskever (2016). "Neural Programmer: Inducing Latent Programs with Gradient Descent". In: *International Conference on Learning Representations*.
- Olah, Chris and Shan Carter (2016). "Attention and Augmented Recurrent Neural Networks". In: *Distill*.

Questions?