UNIVERSITY
OF OSLO

IN4310/3310 Recurrent Neural Networks

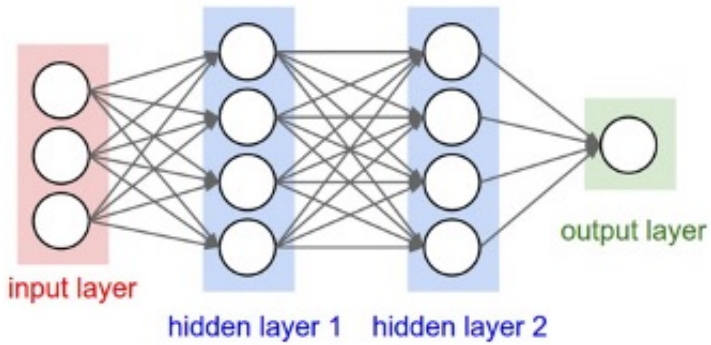
dhananjt@ifi.uio.no

Outline

Outline

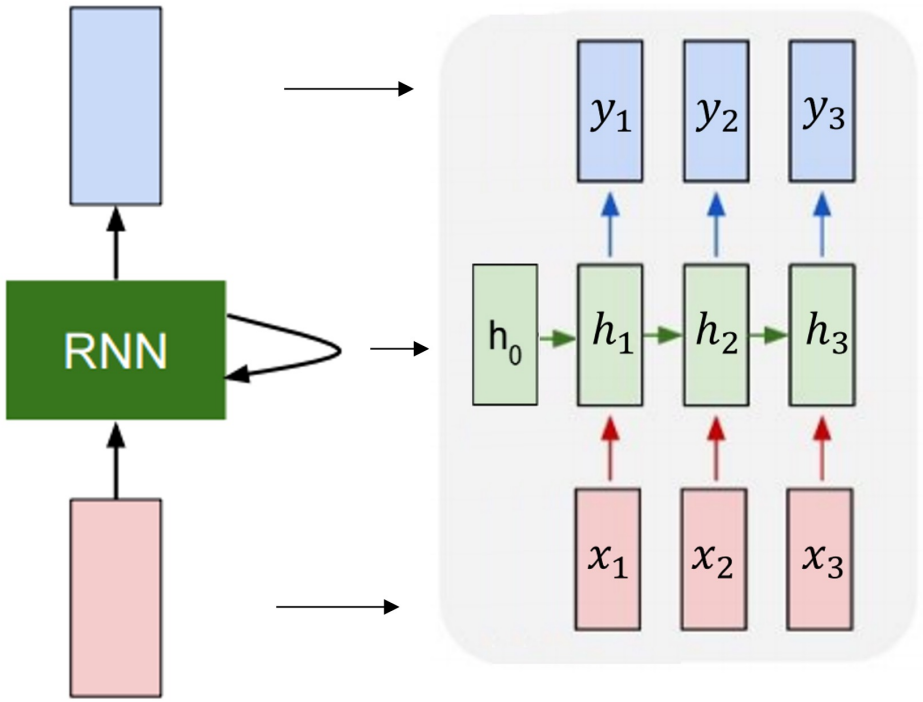
Overview

- **What have we learnt so far?**
  - Fully connected neural networks
  - Convolutional neural networks (CNNs)
- **What are their limitations?**
  - Processing data with unknown length
- **Typical applications of recurrent networks (many types of sequence data):**
  - Speech recognition
  - Music generation
  - Sentiment analysis (e.g., +ve vs –ve reviews)
  - Text analysis and translation



Recurrent Neural Network (RNN)

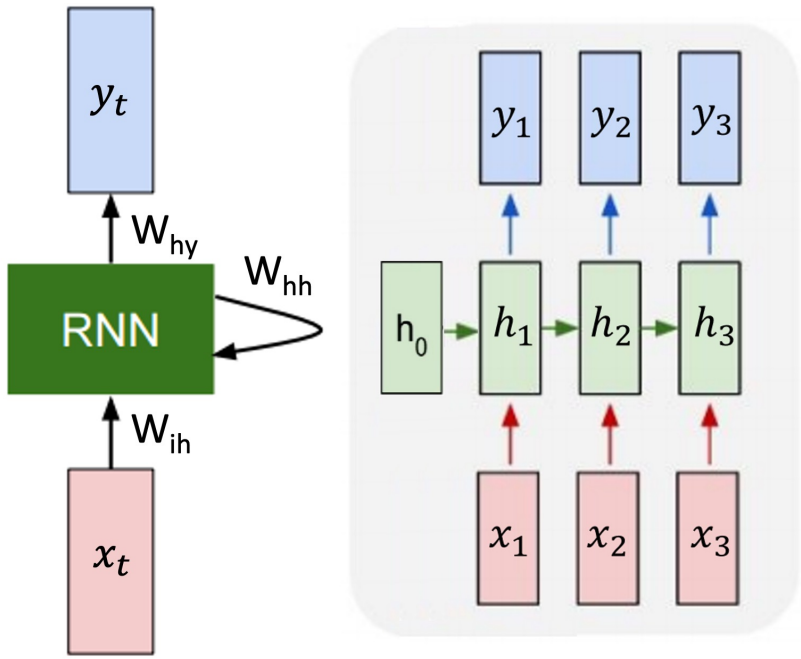
- Given a time series of input data $X = \{X_1, \ldots, X_N\}$ (text, images, time series)
- X_t is typically a vector
- Compute hidden state H_t
- Generate output Y_t



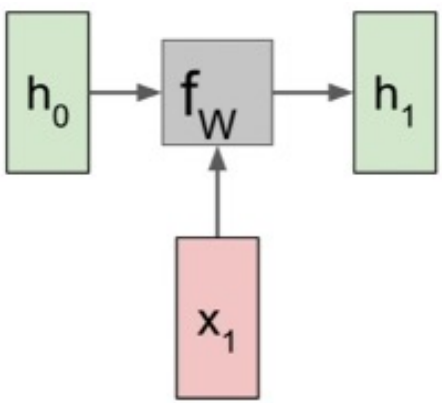
Outline

(Vanilla) Recurrent Neural Network

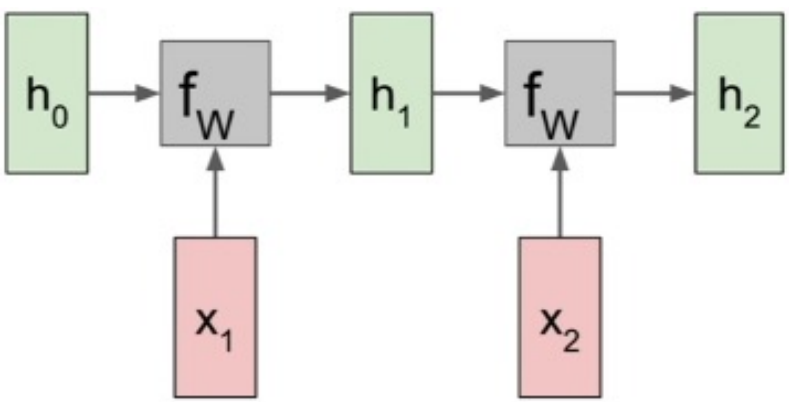
- Input vector: x_t
- Weight matrices: W_{ih}, W_{hh}, W_{hy}
- Vanilla RNN update:
  - Hidden state vector:
    $H_t = \tanh(W_{ih}x_t + W_{hh}H_{t-1} + b_h)$
  - Output: $y_t = g(W_{hy}H_t + b_y)$ where g is an activation function
- **Remark:** We often concatenate W_{hh} and W_{ih} into W, and multiply with the concatenation of $H_{t-1}$and x_t.



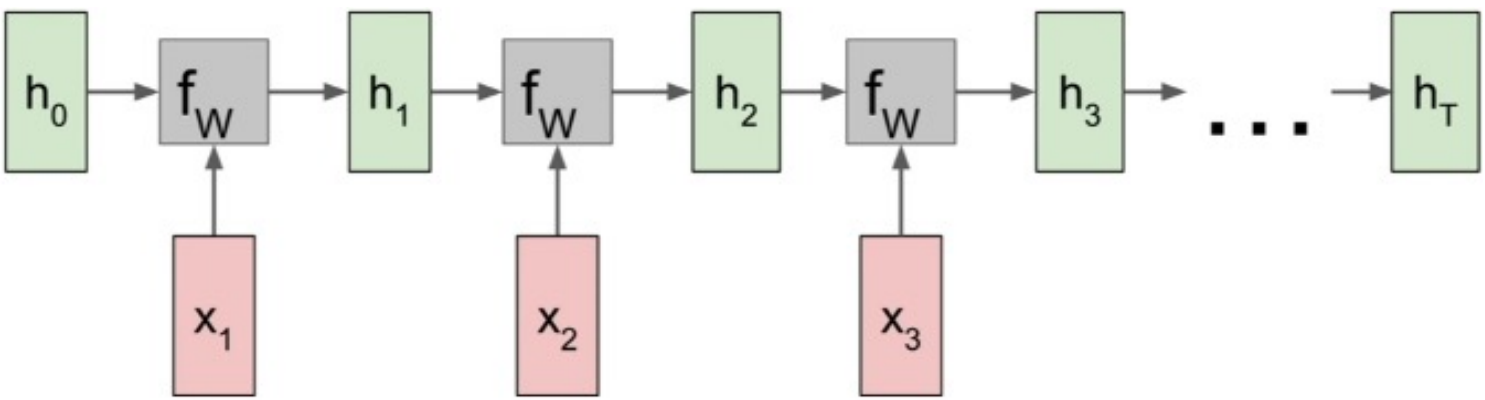
RNN: Computational Graph



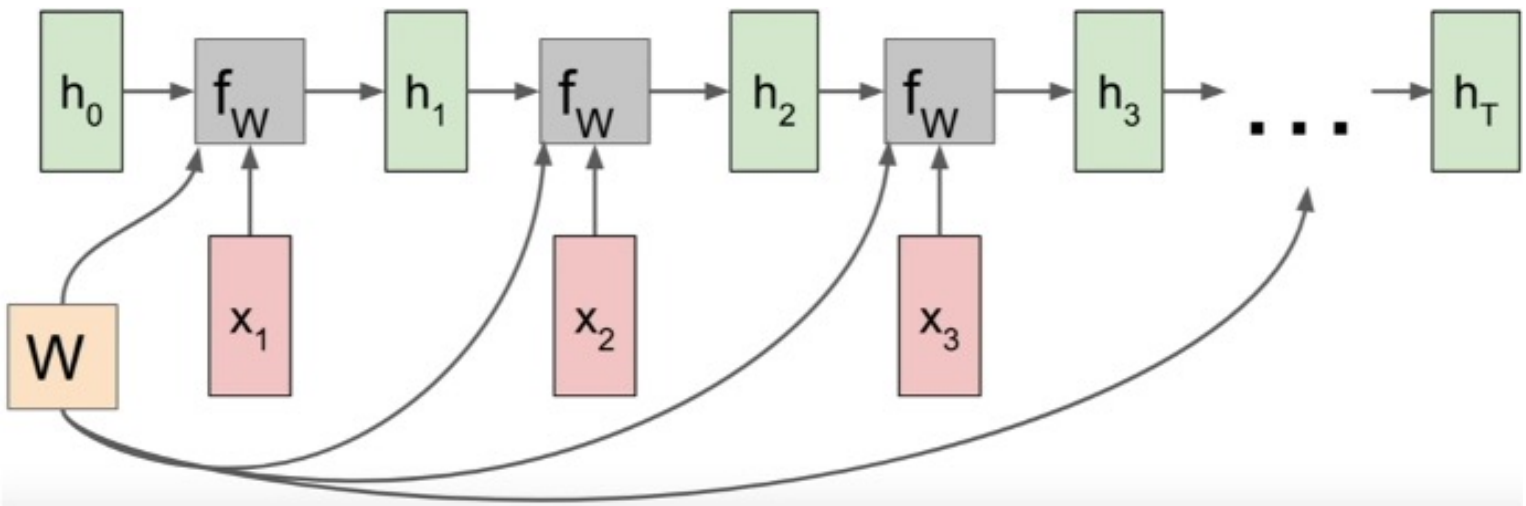
RNN: Computational Graph



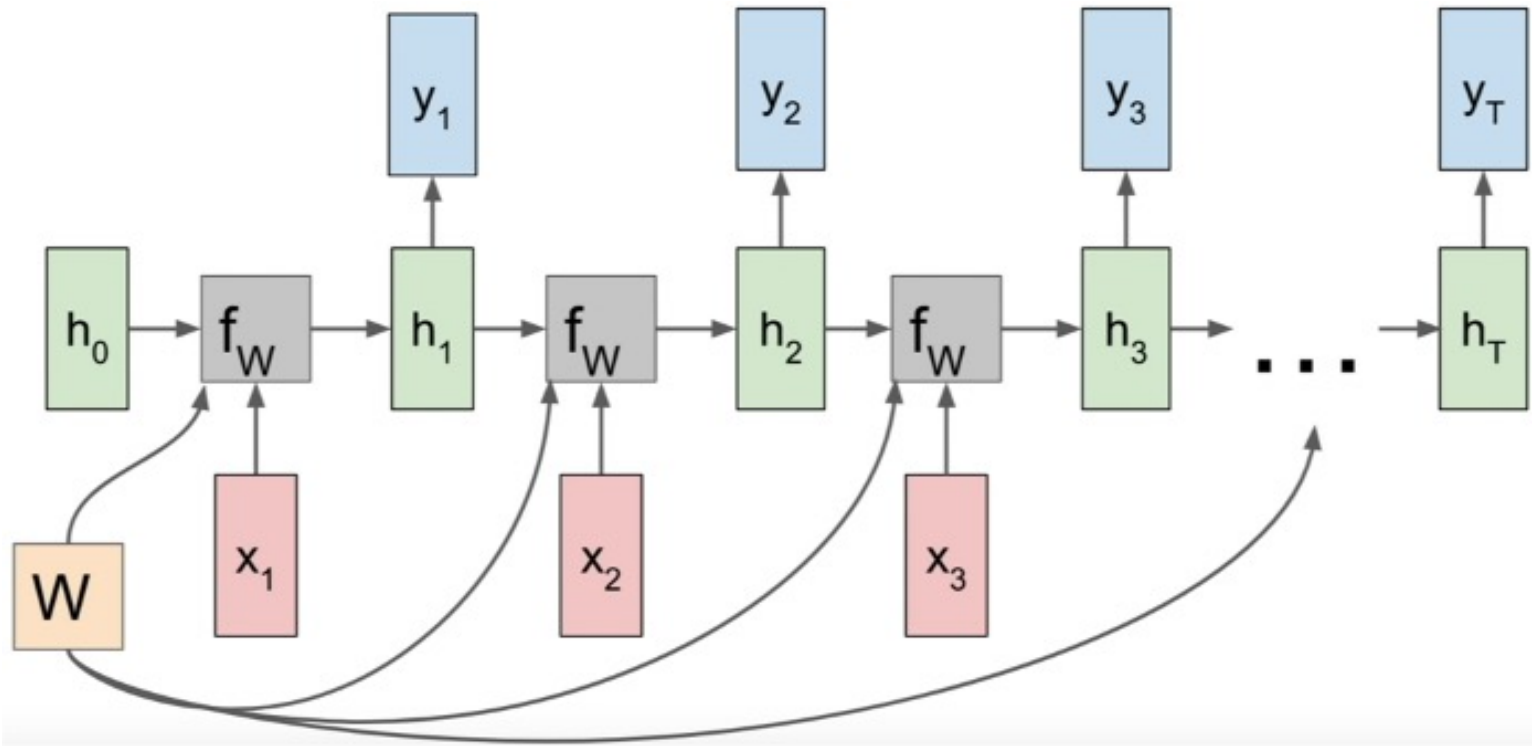
RNN: Computational Graph



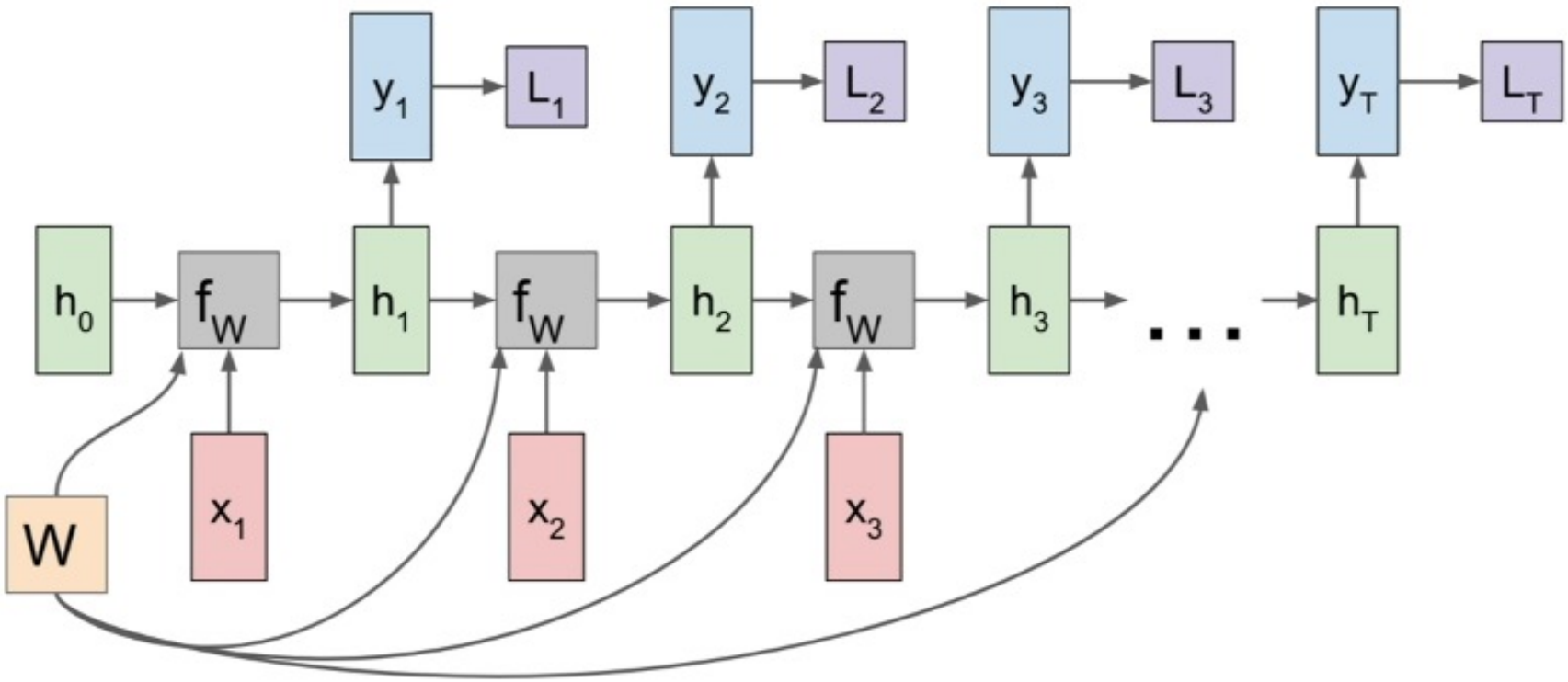
RNN: Computational Graph



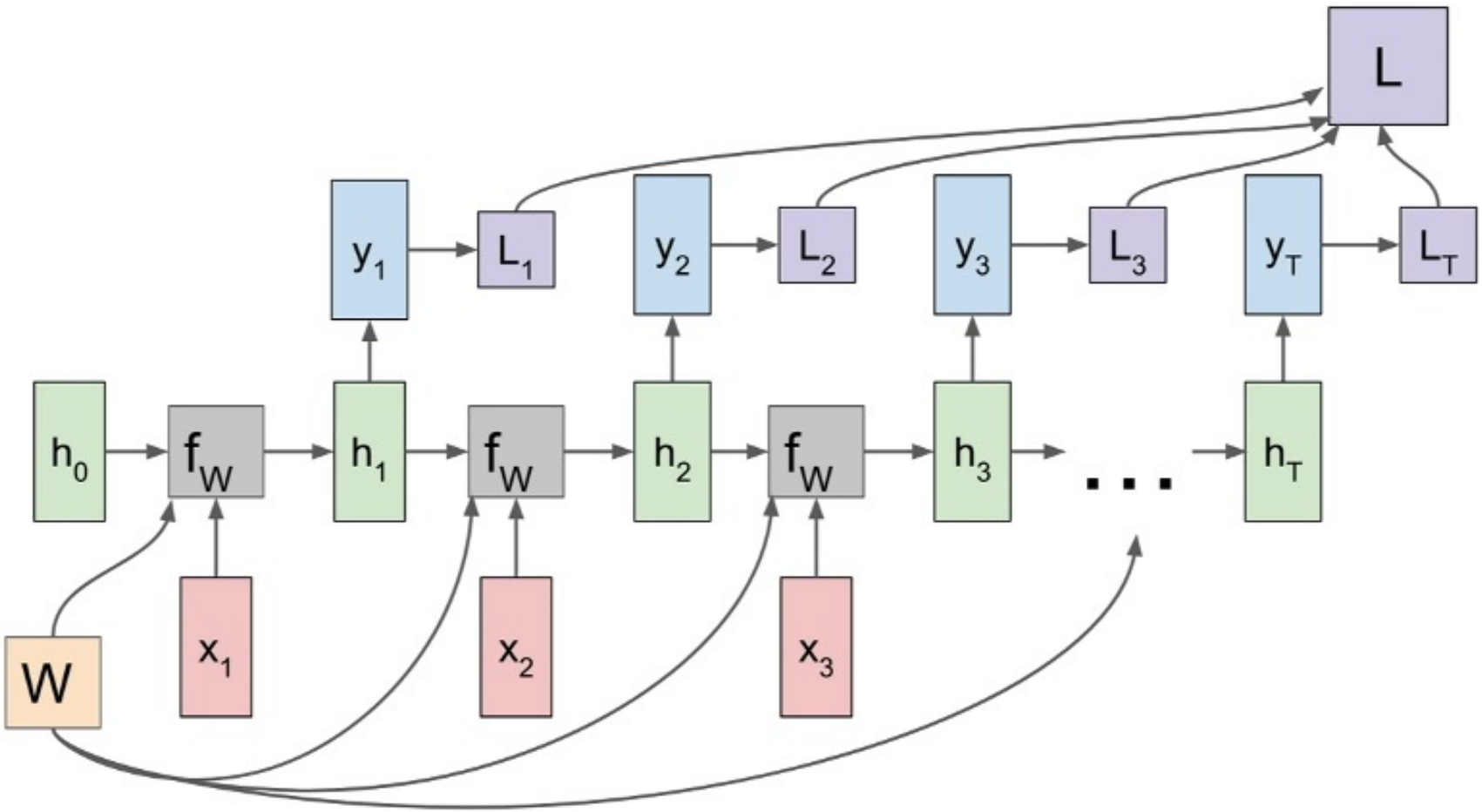
RNN: Computational Graph



RNN: Computational Graph

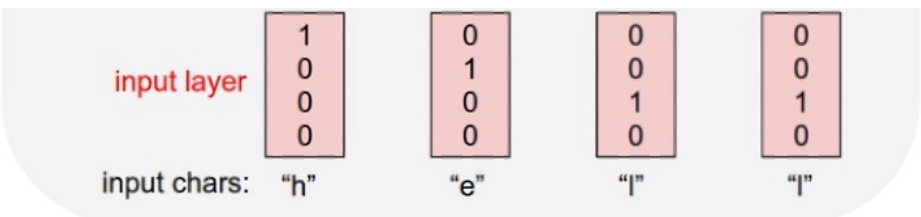


RNN: Computational Graph



Example: Language model

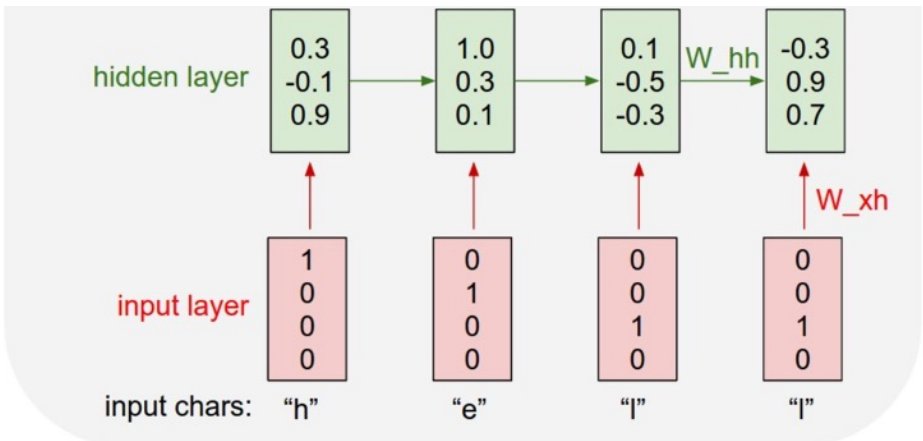
- Task: Predicting the next character
- Training sequence: “hello”
- Vocabulary:
  - [h, e, l, o]
- Encoding: One-hot



RNN: Predicting the next character

- Task: Predicting the next character
- Training sequence: "hello"
- Vocabulary:
  - [h, e, l, o]
- Encoding: One-hot
- Model:

$$h_t = \tanh\left(W_{hh} h_{t-1} + W_{xh} x_t\right)$$

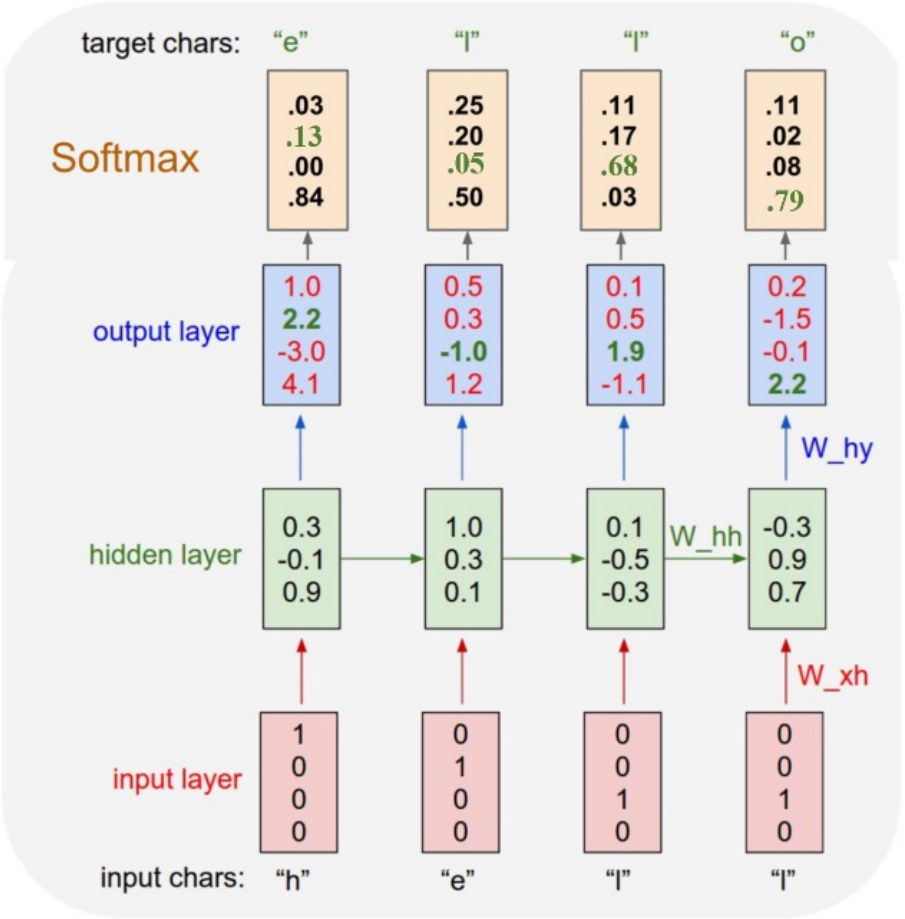


RNN: Predicting the next character

- Task: Predicting the next character
- Training sequence: “hello”
- Vocabulary:
  - [h, e, l, o]
- Encoding: One-hot
- Model:

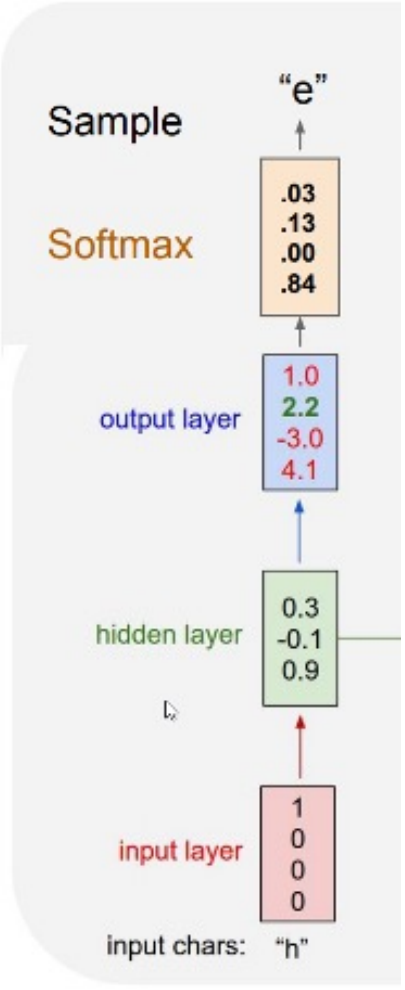
$$h_t = \tanh\left(W_{hh} h_{t-1} + W_{xh} x_t\right)$$

$$y_t = W_{hy} h_t$$



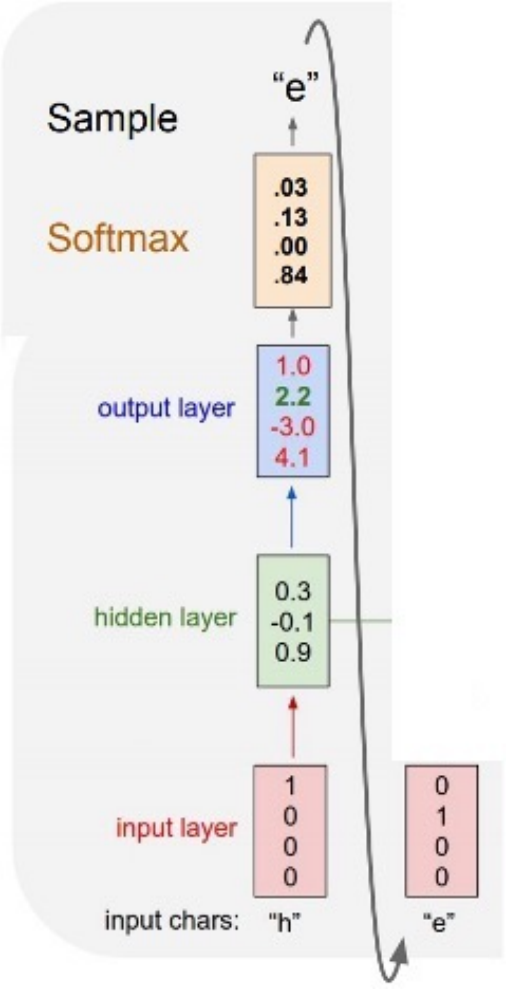
RNN: Predicting the next character

- Vocabulary:
  - [h, e, l, o]
- At test time, we can sample from the model to synthesize new text.



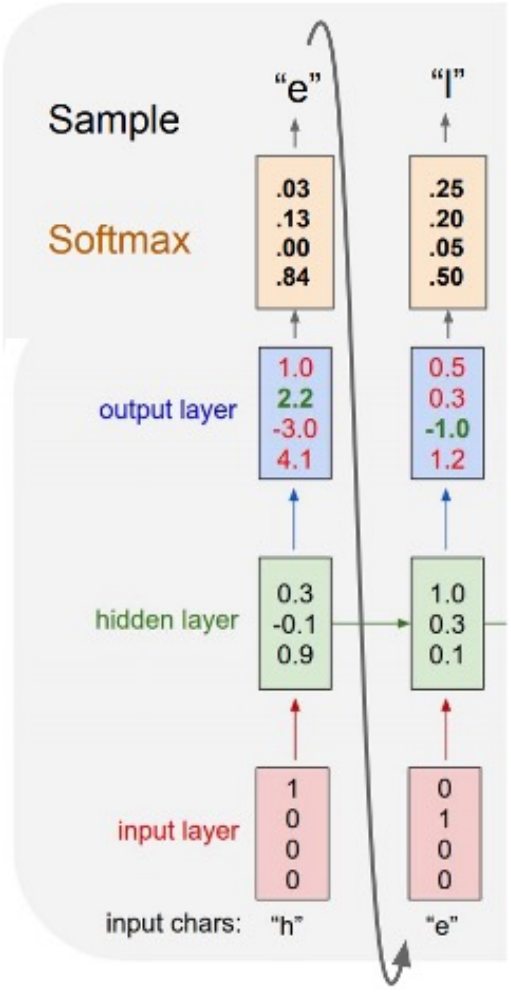
RNN: Predicting the next character

- Vocabulary:
  - [h, e, l, o]
- At test time, we can sample from the model to synthesize new text.



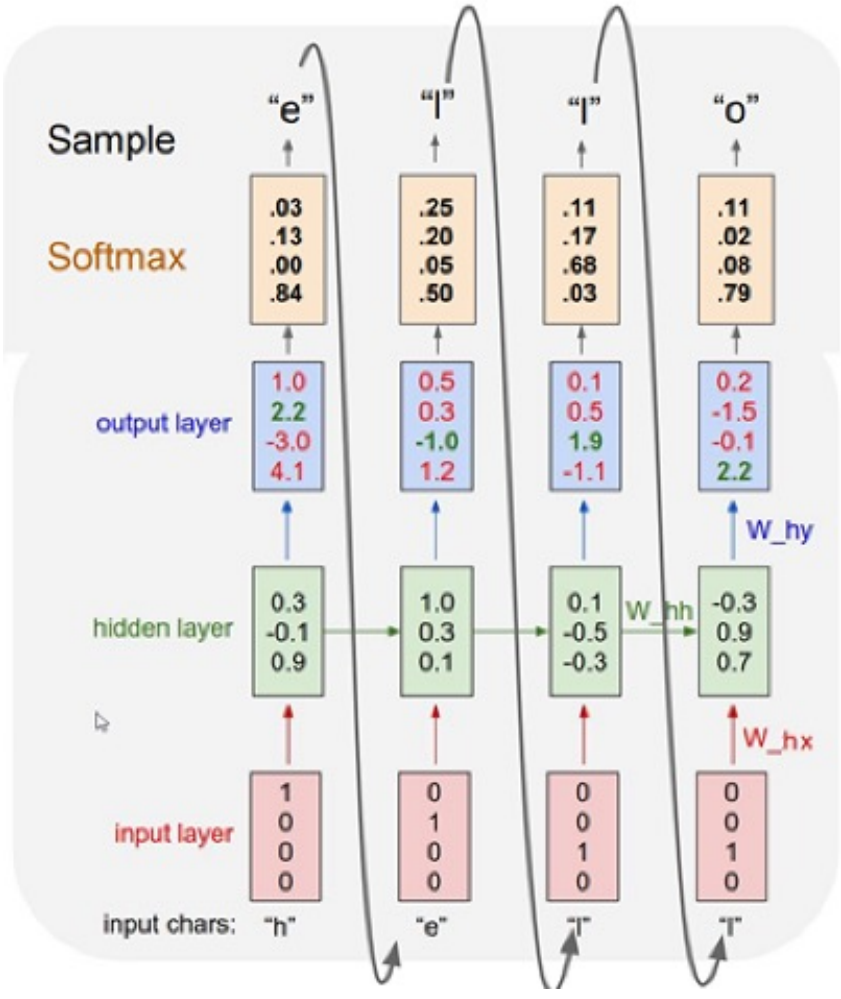
RNN: Predicting the next character

- Vocabulary:
  - [h, e, l, o]
- At test time, we can sample from the model to synthesize new text.



RNN: Predicting the next character

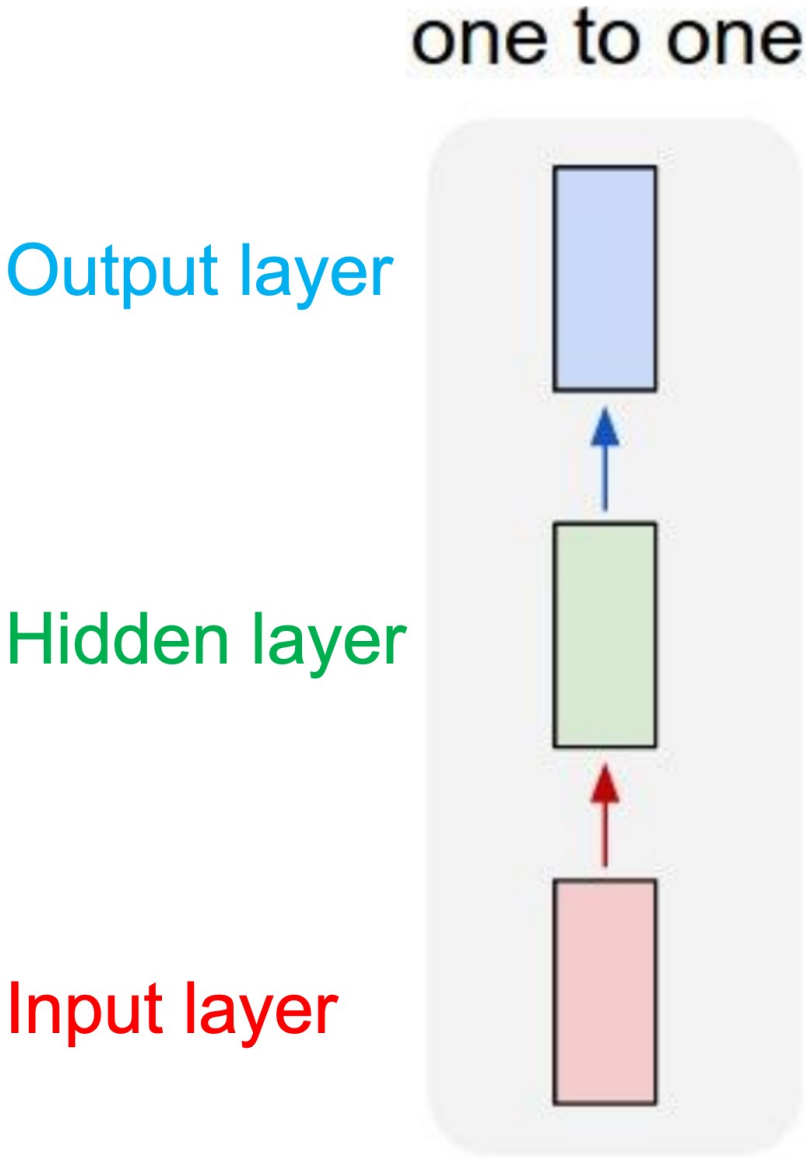
- Vocabulary:
  - [h, e, l, o]
- At test time, we can sample from the model to synthesize new text.



Outline

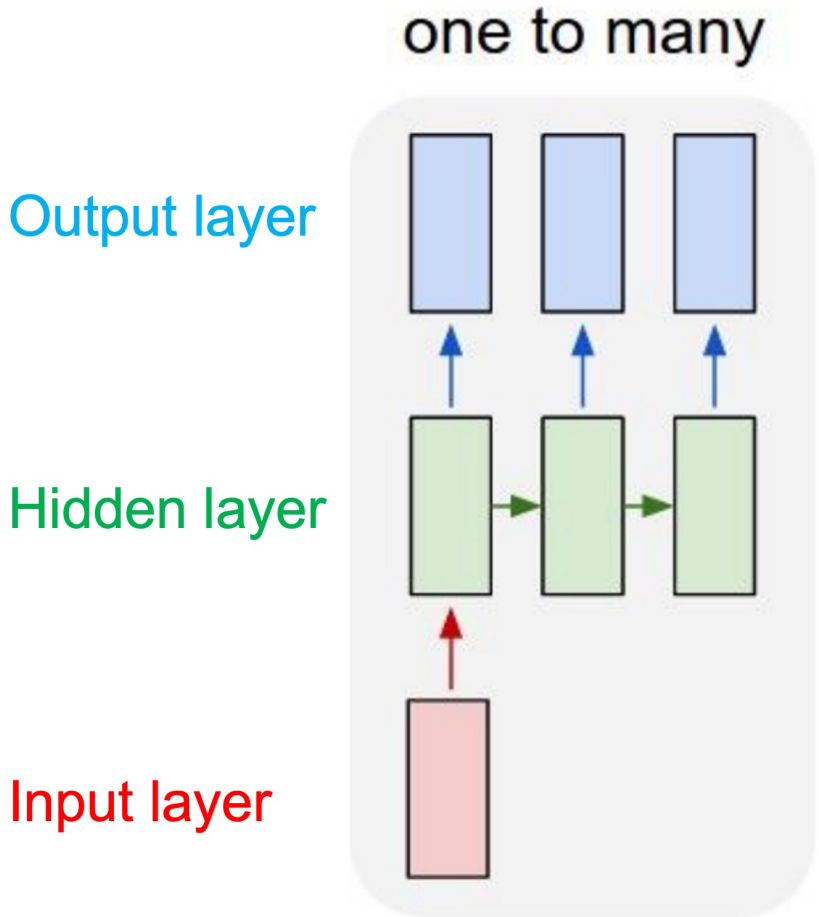
RNN: One-to-one

- Normal feed forward neural network
- One input - one output (classification or regression)



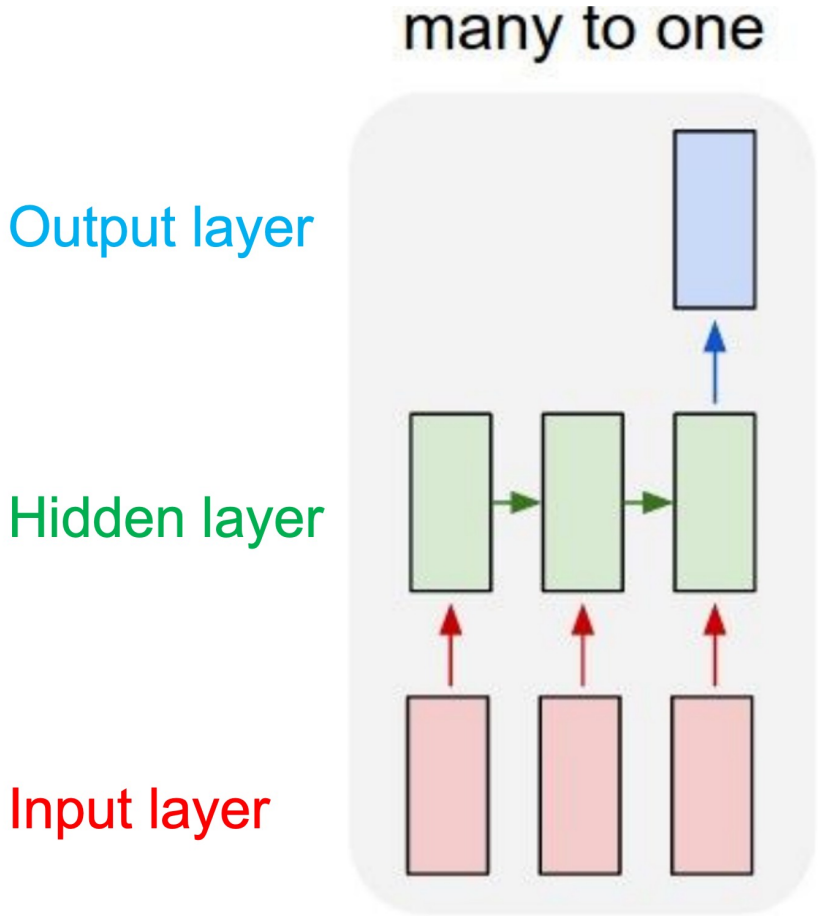
RNN: One-to-many

- Example: Image Captioning
- Input: One Image
- Output: A sequence of words



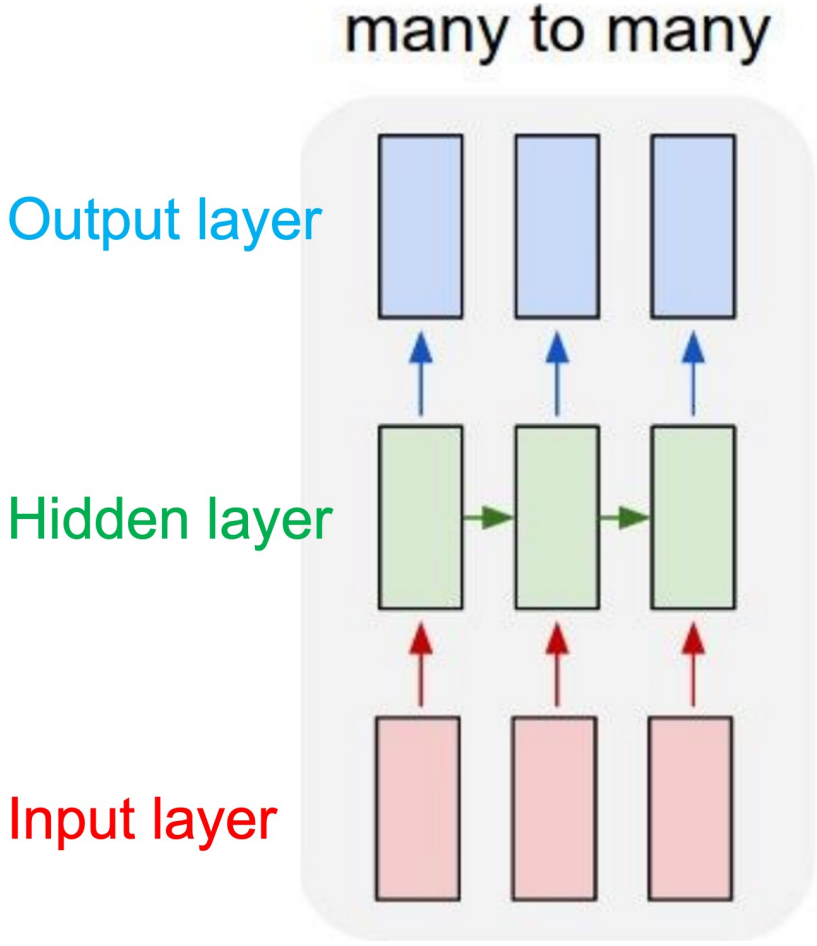
RNN: Many-to-one

- Example: Sentiment Analysis
- Input: A series of words/characters
- Output: A sentiment category



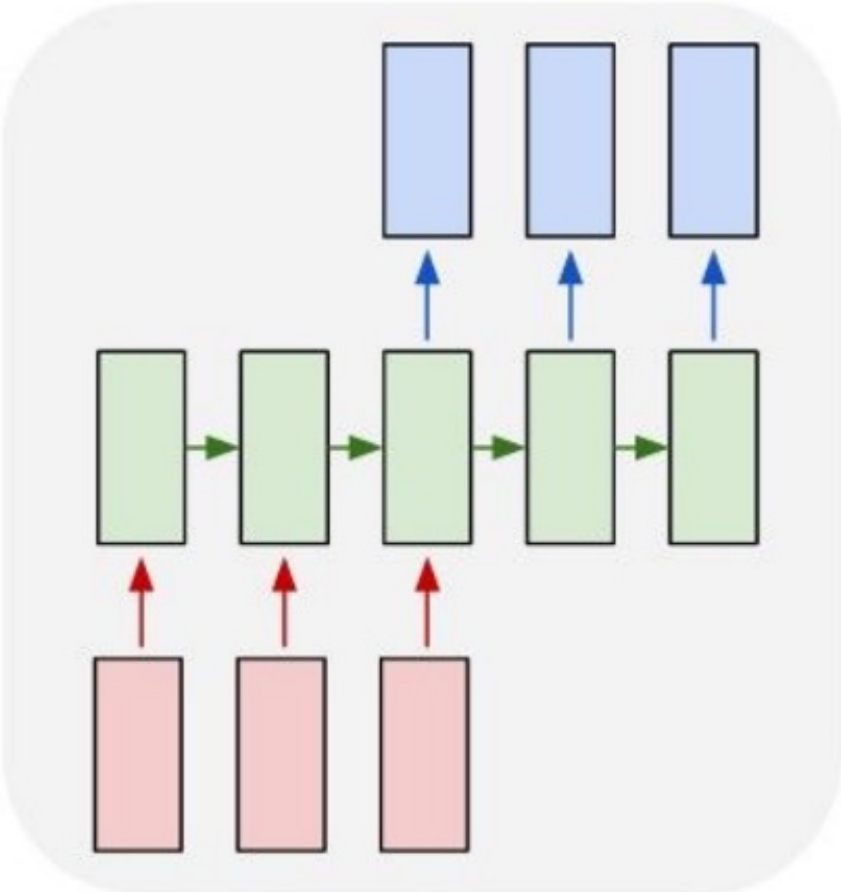
RNN: Many-to-many

- Example: Part-of-speech (POS) tagging / grammatical tagging
- Input: A sequence of words
- Output: A category for each word



RNN: Many-to-many (encoder-decoder)

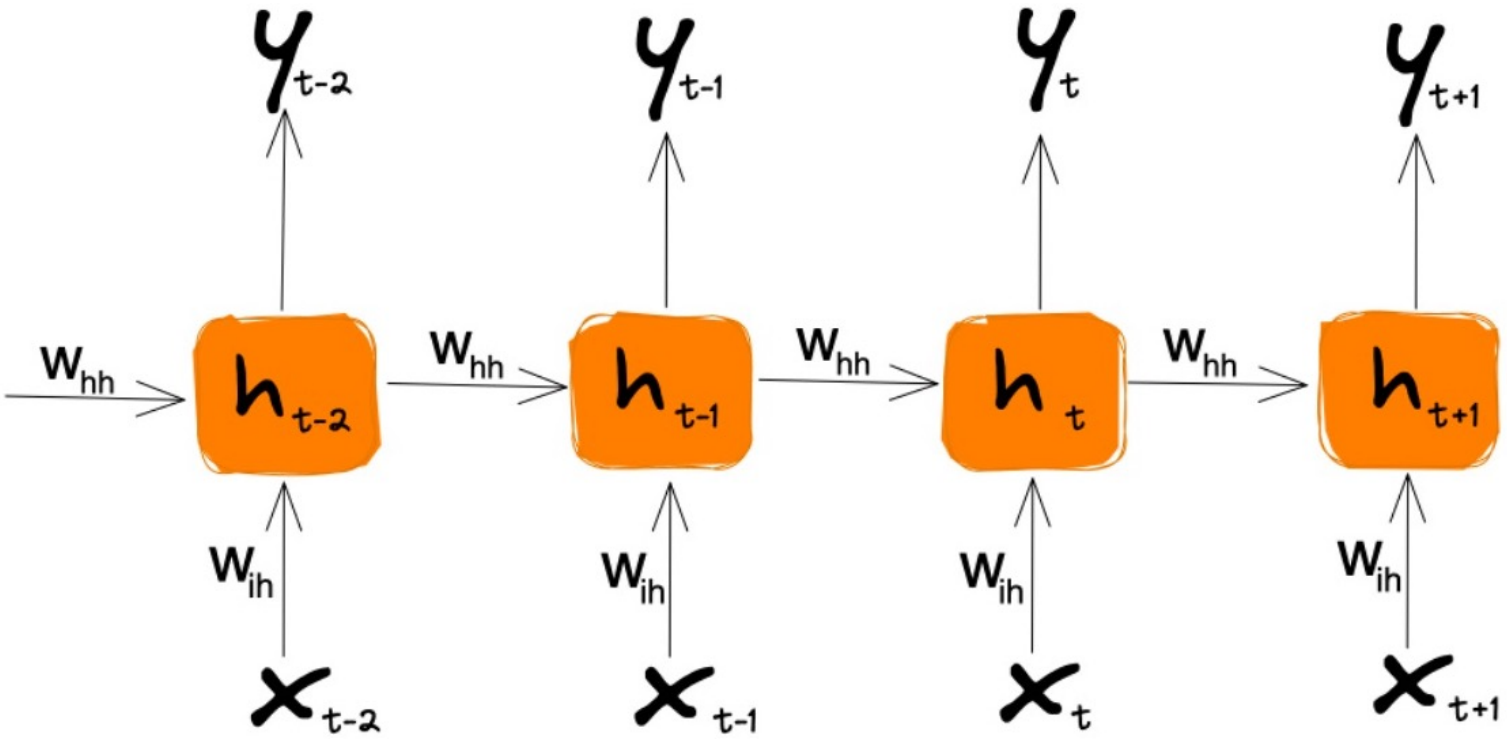
- Example: Neural Machine Translation
- Note that the inputs and outputs can have different lengths here.



Outline

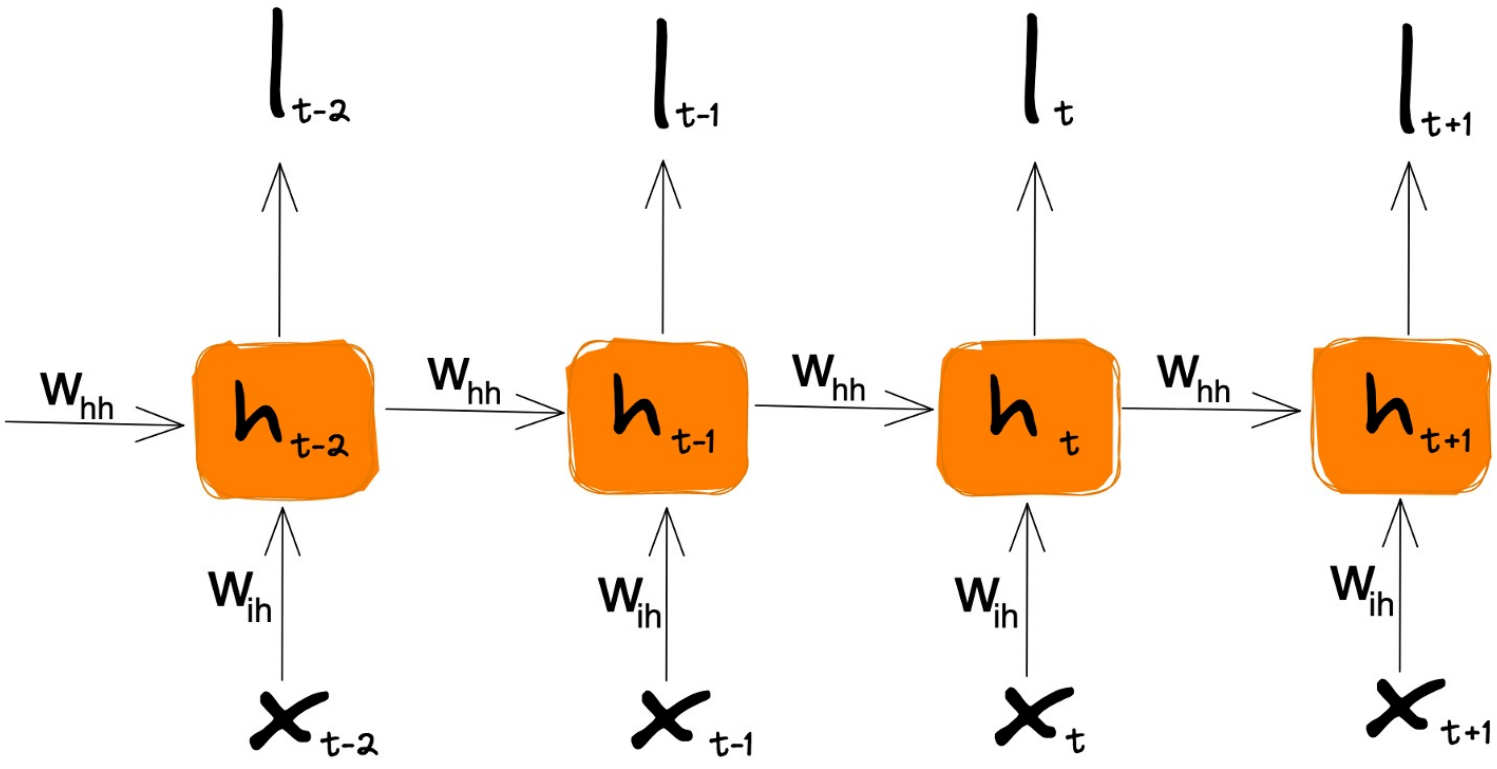
Training RNNs: Backpropagation Through Time

- RNNs are trained using Backpropagation through time (BPTT).
- BPTT steps:
  - Unroll the RNN in the time dimension (. . . t-2, t-1, t, t+1. . .).
  - Apply backpropagation on the unrolled RNN by treating it as a regular feed-forward network.



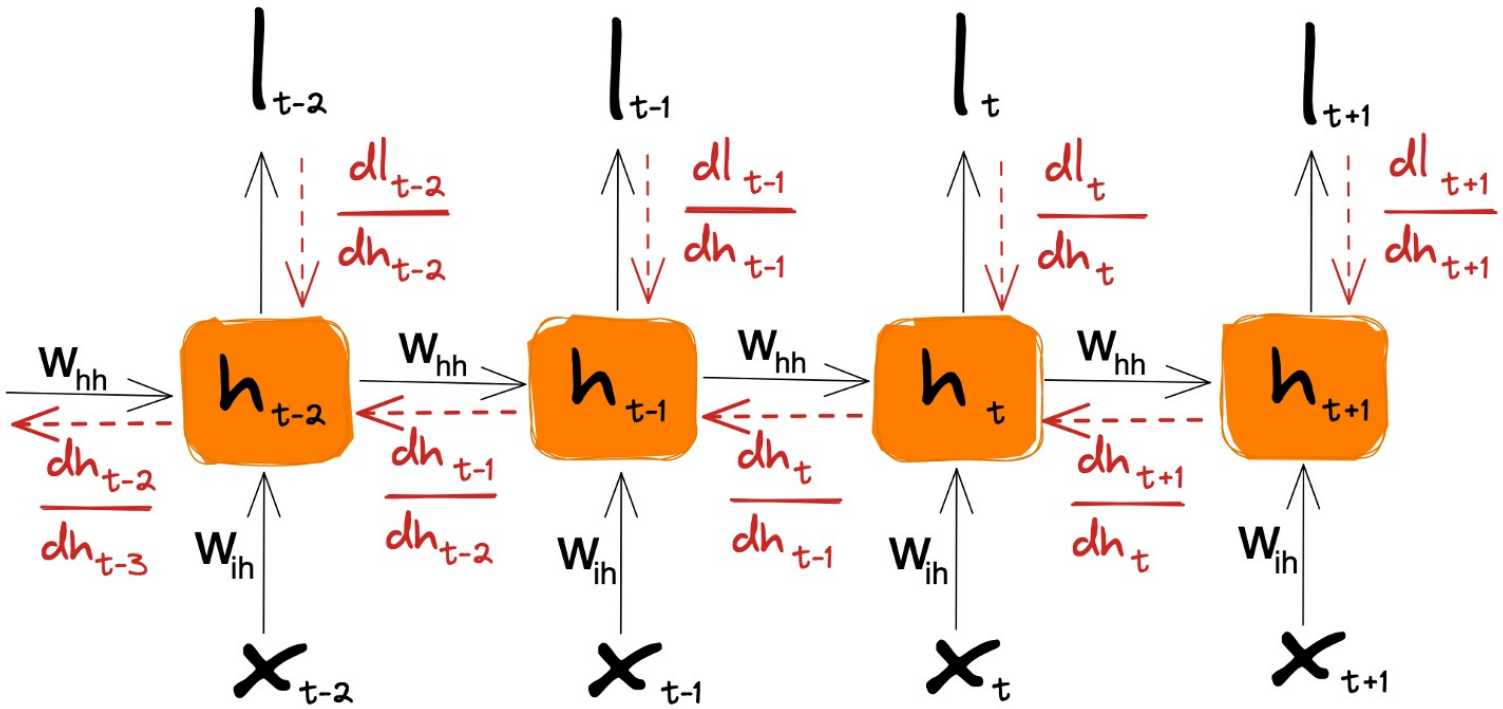
Training RNNs: Backpropagation Through Time

Compute the loss at each time step



Training RNNs: Backpropagation Through Time

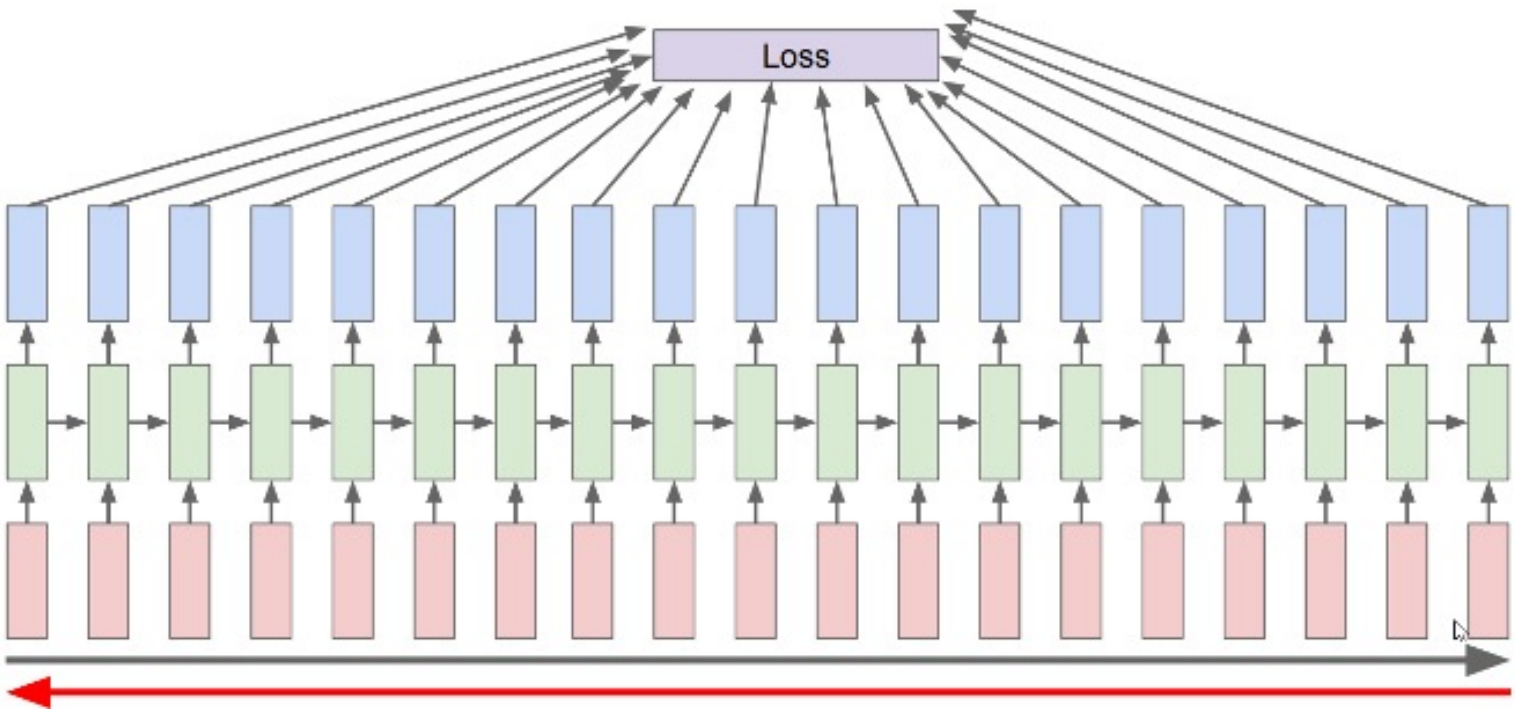
- The gradients are then backpropagated, as shown through the red arrows.
- Notice how weights at a time step k receive gradient from all the time steps t where t $>$ k, i.e., all the time steps ahead of k.



Training RNNs: Backpropagation Through Time

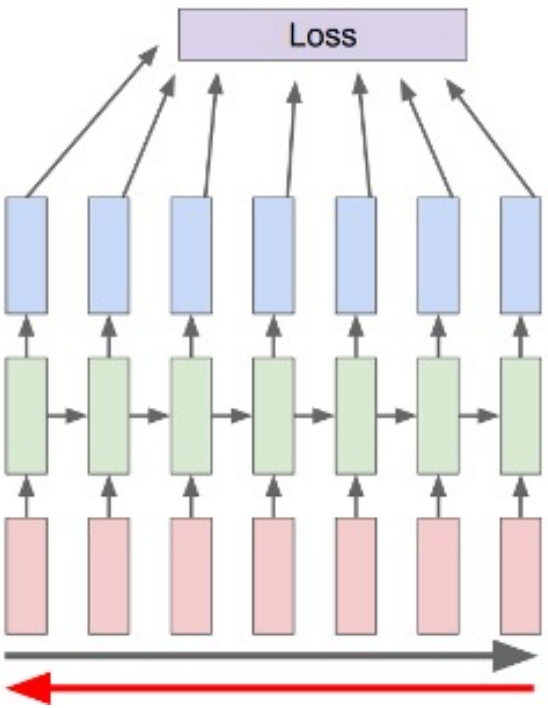
Challenge:

- To calculate gradients, you must keep the internal states of RNN in the memory until you get the backpropagating gradient.
- Then, what if you are reading a book with 100 million characters?



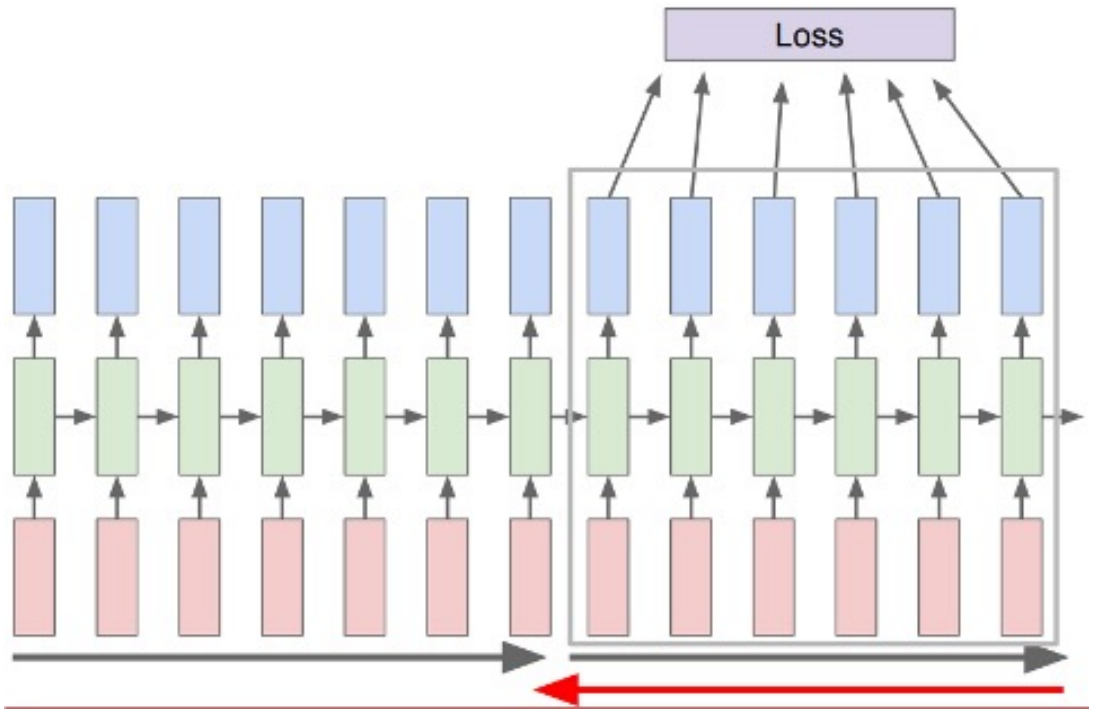
Training RNNs: Backpropagation Through Time

- The solution is to stop at some point and update the weight as if we were done. This algorithm is called Truncated BPTT, as we truncate the unrolled RNN and apply BPTT on what we get.
- Here, BPTT is applied to whatever has been computed so far.



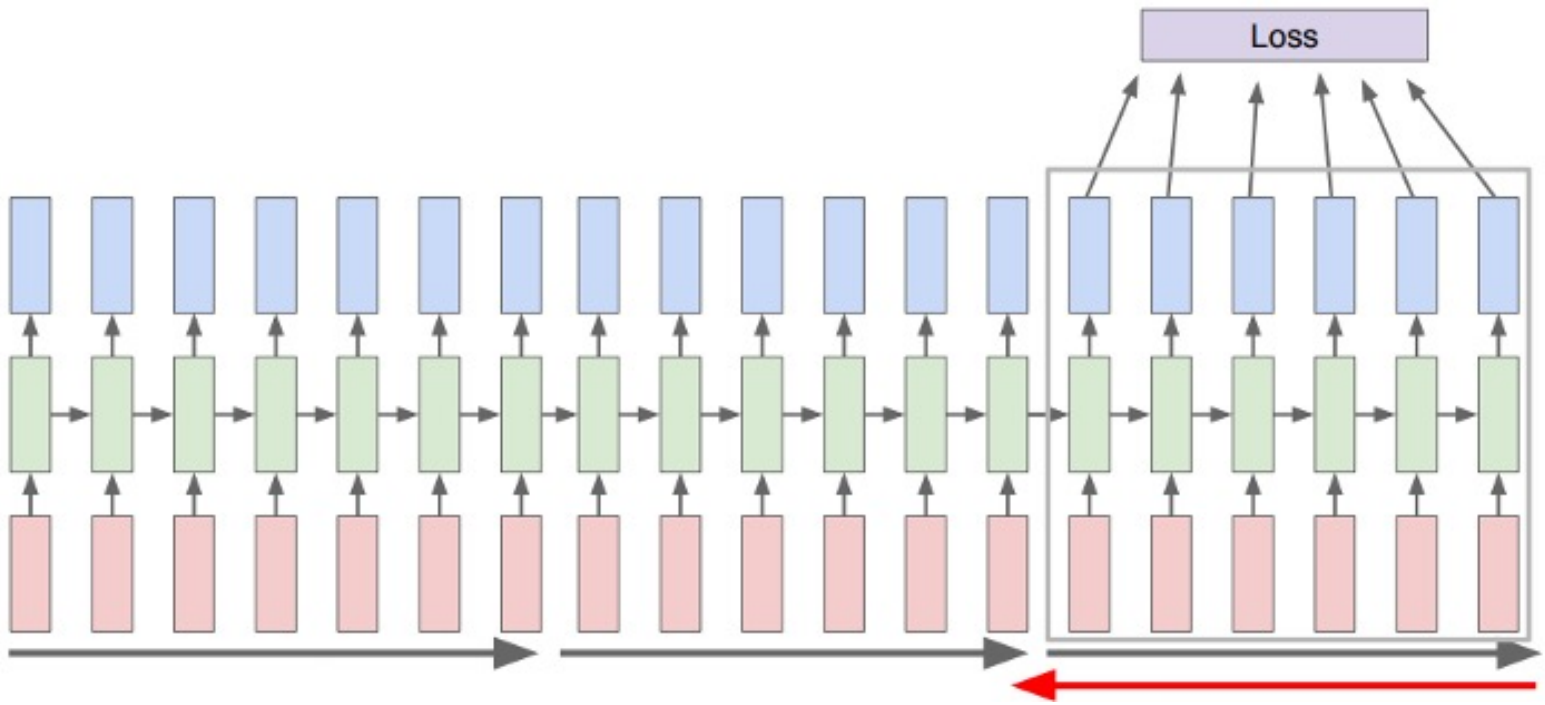
Training RNNs: Backpropagation Through Time

- We unroll the RNN further and again stop after a few time steps.
- The BPTT is now applied on the newly unrolled steps (and not on the previous ones).



Training RNNs: Backpropagation Through Time

- And this continues until the RNN has been completely unrolled for all the time steps.



Training RNNs: Backpropagation Through Time

- Advantages:
  - Reduces the memory requirement
  - Faster parameter updates
- Disadvantage:
  - Hard to capture dependencies longer than the truncated length.

Outline

Challenges in training RNNs

- Exploding gradients
- Vanishing gradients
- Preserve long-range dependencies

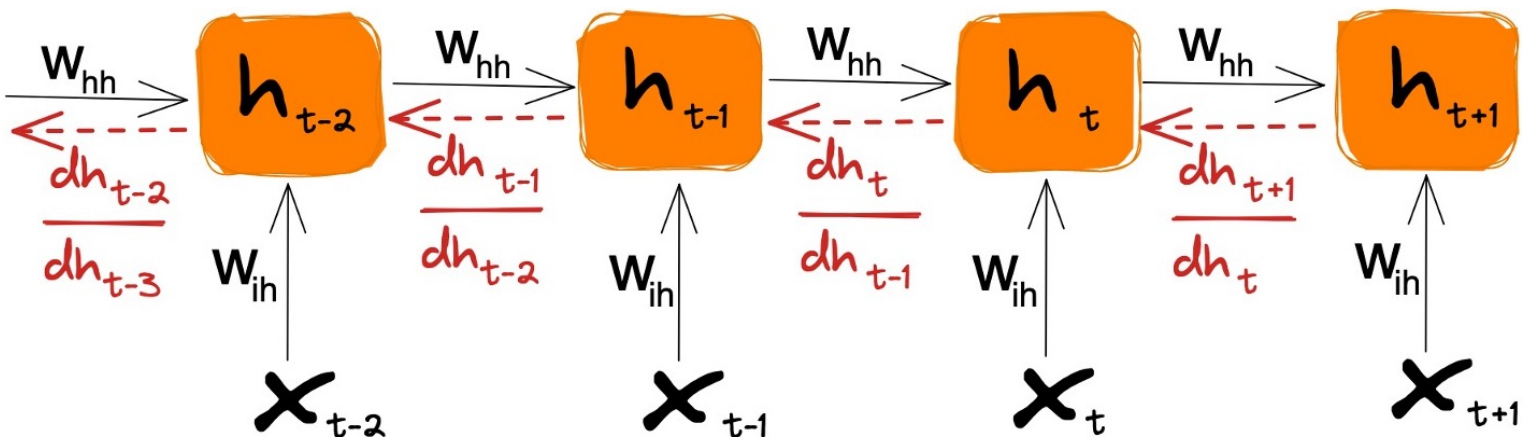
Challenges: Exploding or vanishing gradients

The basic intuition behind vanishing or exploding gradients is pretty simple:

- If you take a real number < 1 and keep multiplying it by itself, the result will approach 0.
  - E.g. $(0.7)^{10} \approx 0.028$ and $(0.7)^{100} \approx 0$
- If you take a real number > 1 and keep multiplying it by itself, the result will approach ∞.
  - E.g. $(7)^{10} \approx 282475249$ and $(7)^{100} \approx 3.23\text{x}10^{84}$

Challenges: Exploding or vanishing gradients

- Notice how gradients pass through $\mathbf{w_{hh}}$ at every time step.
- Let's see mathematically how gradients for $\mathbf{w_{hh}}$ flow from a *time step* t to a *time step* k.



Challenges: Exploding or vanishing gradients

- Let the hidden state at time step t be ht and loss be L.
- Recall the equation for calculating ht is

$$\mathrm{h_t} = \sigma\left(\mathrm{W_{hh}}\mathrm{h_{t-1}} + \mathrm{W_{ih}}\mathrm{x_t} + \mathrm{b}\right)$$

- We're interested in calculating the gradients received by $\mathrm{W_{hh}}$ at time step k. Using the chain rule, we have

$$\frac{\mathrm{dL}}{\mathrm{dW_{hh}^{(k)}}} = \frac{\mathrm{dL}}{\mathrm{dh_t}}\frac{\mathrm{dh_t}}{\mathrm{dh_{t-1}}}\frac{\mathrm{dh_{t-1}}}{\mathrm{dh_{t-2}}}\cdots\frac{\mathrm{dh_{k+1}}}{\mathrm{dh_k}}\frac{\mathrm{dh_k}}{\mathrm{dW_{hh}^{(k)}}} \tag{1}$$

$$\frac{\mathrm{dL}}{\mathrm{dW_{hh}^{(k)}}} = \frac{\mathrm{dL}}{\mathrm{dh_t}}\prod_{\mathrm{i=k+1}}^{\mathrm{t}}\frac{\mathrm{dh_i}}{\mathrm{dh_{i-1}}}\frac{\mathrm{dh_k}}{\mathrm{dW_{hh}^{(k)}}} \tag{2}$$

$$\prod_{\mathrm{i=k+1}}^{\mathrm{t}}\frac{\mathrm{dh_i}}{\mathrm{dh_{i-1}}} = \prod_{\mathrm{i=k+1}}^{\mathrm{t}}\mathrm{diag}\left(\sigma'(\mathrm{h_{i-1}})\right)\mathrm{W_{hh}^T} \tag{3}$$

  - where σ' is the derivative of σ, **diag** converts vector to a diagonal matrix and $\mathrm{W^T}$ represents the transpose of W.

Challenges: Exploding or vanishing gradients

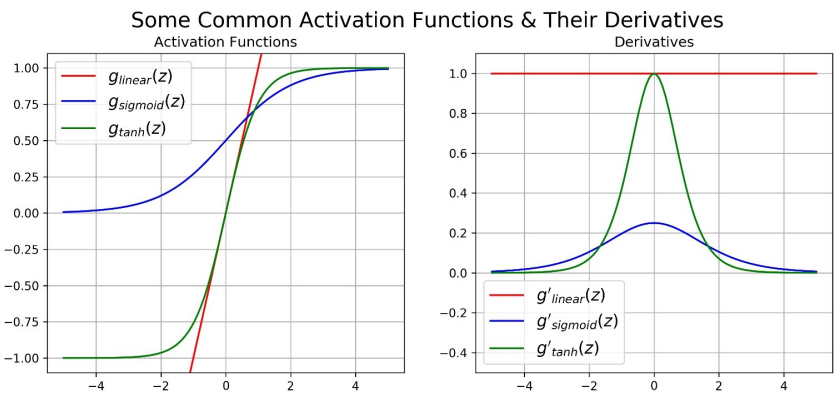
$$\frac{\mathrm{dL}}{\mathrm{dW}_{\mathrm{hh}}^{(\mathrm{k})}} = \frac{\mathrm{dL}}{\mathrm{dh_t}} \prod_{\mathrm{i=k+1}}^{\mathrm{t}} \frac{\mathrm{dh_i}}{\mathrm{dh_{i-1}}} \frac{\mathrm{dh_k}}{\mathrm{dW}_{\mathrm{hh}}^{(\mathrm{k})}} \tag{2}$$

$$\prod_{\mathrm{i=k+1}}^{\mathrm{t}} \frac{\mathrm{dh_i}}{\mathrm{dh_{i-1}}} = \prod_{\mathrm{i=k+1}}^{\mathrm{t}} \mathrm{diag}\left(\sigma'(\mathrm{h_{i-1}})\right) \mathrm{W}_{\mathrm{hh}}^{\mathrm{T}} \tag{3}$$

- Equations (2) and (3) tell us that the gradient from the loss function at time step t is multiplied by σ' and the matrix $\mathrm{W_{hh}}$ (t-k) times to get the gradients received by $\mathrm{W_{hh}}$ at time step k.
- For simplicity, let's replace $\mathrm{W}_{\mathrm{hh}}^{\mathrm{T}}$ with W and $\mathrm{diag}\left(\sigma'(\mathrm{h_{i-1}})\right)$ with σ' (we ignore $\mathrm{h_{i-1}}$ for now).
- The product (equation (3)) then becomes $\sigma'^{\mathrm{n}}\mathrm{W}^{\mathrm{n}}$ for BPTT through n time steps.
- Let's analyse both the terms in the product.

Challenges: Exploding or vanishing gradients

- σ'^{n} will lead to vanishing gradients if $\sigma' < 1$ for a lot of $\mathrm{h_{i-1}}$ values and exploding gradients if it is > 1 (recall the basic intuition).
- For $\sigma =$ sigmoid or tanh, the derivatives have an upper bound of 0.25 and 1, respectively, so they can't lead gradients to explode but can help gradients vanish.



Challenges: Exploding or vanishing gradients (not relevant for the exam)

- To analyse W, we will assume that its eigendecomposition exists. You should be familiar with it from MAT1110 / MAT1120.
- Decompose square matrix W as $\mathrm{W} = \mathrm{U}\Sigma\mathrm{U}^{-1}$ where U's columns are the eigenvectors of W and Σ is a diagonal matrix whose diagonal elements are the corresponding eigenvalues.

Challenges: Exploding or vanishing gradients (not relevant for the exam)

- $\mathrm{W}^{\mathrm{n}} = (\mathrm{U}\Sigma\mathrm{U}^{-1})(\mathrm{U}\Sigma\mathrm{U}^{-1})(\mathrm{U}\Sigma\mathrm{U}^{-1})\ldots(\mathrm{U}\Sigma\mathrm{U}^{-1})$
- $\mathrm{W}^{\mathrm{n}} = \mathrm{U}\Sigma\mathrm{I}\Sigma\mathrm{I}\Sigma\mathrm{I}\ldots\Sigma\mathrm{U}^{-1}$ because $\mathrm{U}\mathrm{U}^{-1} = \mathrm{I}$ (I=Identity matrix) by definition
- $\mathrm{W}^{\mathrm{n}} = \mathrm{U}\Sigma\Sigma\Sigma\ldots\Sigma\mathrm{U}^{-1} = \mathrm{U}\Sigma^{\mathrm{n}}\mathrm{U}^{-1}$
- What this means is that the growth or decay of W's multiplication with itself is dictated by its eigenvalues.
- If the spectral radius (magnitude of the eigenvalue with the highest absolute value) of W is > 1, i.e., $\mathbf{max(np.abs(\Sigma))} > \mathbf{1}$, then the matrix will explode, and if it's < 1, the matrix will decay.

Challenges: Exploding or vanishing gradients (not relevant for the exam)

$$W^n = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix}^n \begin{bmatrix} d_1 & e_1 & f_1 \\ d_2 & e_2 & f_2 \\ d_3 & e_3 & f_3 \end{bmatrix}$$

$$W^n = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \begin{bmatrix} \lambda_1^n & 0 & 0 \\ 0 & \lambda_2^n & 0 \\ 0 & 0 & \lambda_3^n \end{bmatrix} \begin{bmatrix} d_1 & e_1 & f_1 \\ d_2 & e_2 & f_2 \\ d_3 & e_3 & f_3 \end{bmatrix}$$

$$W^n = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \begin{bmatrix} \lambda_1^n d_1 & \lambda_1^n e_1 & \lambda_1^n f_1 \\ \lambda_2^n d_2 & \lambda_2^n e_2 & \lambda_2^n f_2 \\ \lambda_3^n d_3 & \lambda_3^n e_3 & \lambda_3^n f_3 \end{bmatrix}$$

If only $|\lambda_1| > 1$ (and $|\lambda_2|, |\lambda_3| < 1$) then $\lambda_2^n = \lambda_3^n = 0$

Challenges: Exploding or vanishing gradients (not relevant for the exam)

$$W^n = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \begin{bmatrix} \lambda_1^n d_1 & \lambda_1^n e_1 & \lambda_1^n f_1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$W^n = \begin{bmatrix} \lambda_1^n d_1 a_1 & \lambda_1^n e_1 a_1 & \lambda_1^n f_1 a_1 \\ \lambda_1^n d_1 a_2 & \lambda_1^n e_1 a_2 & \lambda_1^n f_1 a_2 \\ \lambda_1^n d_1 a_3 & \lambda_1^n e_1 a_3 & \lambda_1^n f_1 a_3 \end{bmatrix}$$

All the gradients explode even if just one eigenvalue's magnitude is greater than one.

Exploding gradients: Gradient clipping

- To avoid exploding gradients, clip them if they are larger than a threshold.
- Two common approaches: clipping-by-value or clipping-by-norm.
- Let **g** be a single gradient and **gg** be a vector of gradients
- **Clipping-by-value:** If $\mathrm{g} > \text{threshold}$, $\mathrm{g} = \text{threshold}$.
- **Clipping-by-norm:** If $\mathrm{l}^2\text{norm}(\mathrm{gg}) > \text{threshold}$, for each g in gg apply

$$\mathrm{g} = \mathrm{g} * \frac{\text{threshold}}{\mathrm{l}^2\text{norm}(\mathrm{gg})}$$

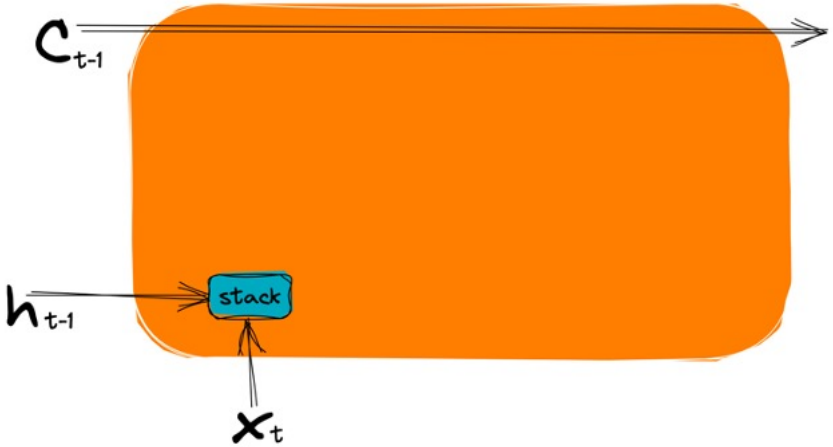
Orthogonal Initialisation

- One way to partially prevent gradients from vanishing or exploding at the beginning of the training is to initialise the matrix with orthogonal matrices.
- Orthogonal matrices have a nice property: all their eigenvalues are 1. Therefore, W^n neither explodes nor decays.
- But as BPTT updates the weight matrices over and over, the matrices might not remain orthogonal.

Outline

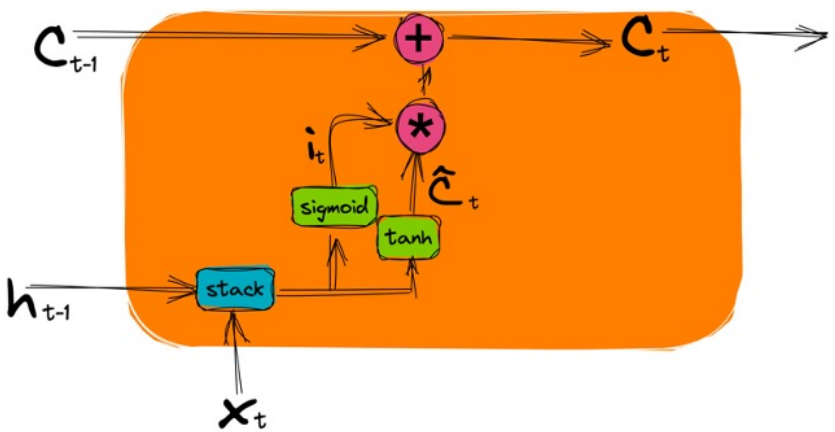
LSTM Networks

- One way to prevent vanishing gradients and preserve long-term dependencies is to use LSTM (Long Short Term Memory) networks.
- The main idea in LSTM's design is to maintain a cell state C which serves as memory. Each node (an LSTM cell) passes this cell state to itself across all the time steps.
- The other components are mostly gates that modify the cell state and generate output of the cell using the cell state.



LSTM Networks: Input Gate

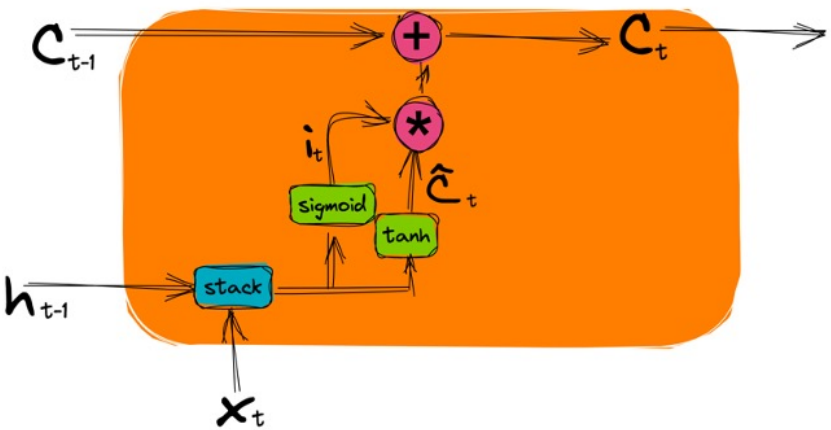
- Input gate controls what fraction of the new memory is added to the old one.
- To modify the cell state, first a new candidate cell state is generated as $\hat{\mathbf{C}} = \mathbf{tanh}\left(\mathbf{W_c}\left[\mathbf{h_{t-1}x_t}\right] + \mathbf{b_c}\right)$
- Then the input gate decides how much of this new candidate should be added to the memory. Input gate's value is calculated as $\mathbf{i_t} = \mathbf{sigmoid}(\mathbf{W_i}\left[\mathbf{h_{t-1}x_t}\right] + \mathbf{b_i})$



LSTM Networks: Output Gate

- The cell state of LSTM is internal. Other layers and cells don't have access to it. It's NOT cell's output.
- To get the output, first a candidate output is generated by simply passing the cell state through tanh: $\mathbf{\hat{h}_t = tanh(C_t)}$
- Then the output gate decides how much of this value are we going to output. Output gate's value is calculated as

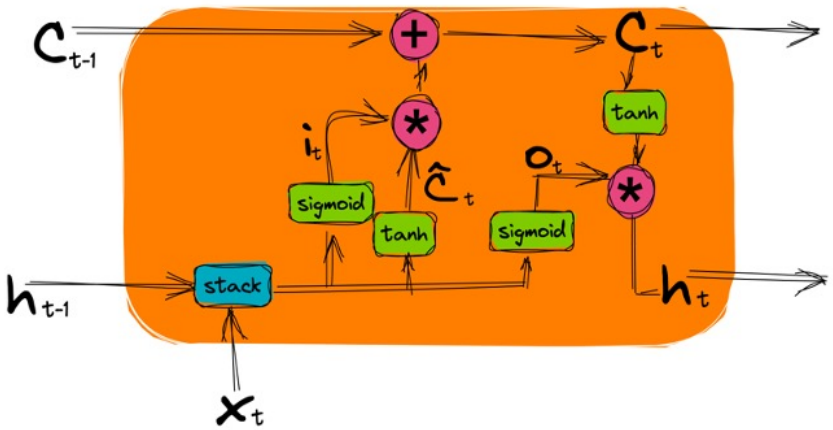
$$\mathbf{o_t = sigmoid(W_o\,[h_{t-1}x_t] + b_o)}$$



LSTM Networks: Output Gate

- $\hat{\mathbf{h}}_\mathbf{t} = \mathbf{tanh}(\mathbf{C_t})$
- $\mathbf{o_t} = \mathbf{sigmoid}(\mathbf{W_o}\,[\mathbf{h_{t-1}x_t}] + \mathbf{b_o})$
- The output of the cell is then

$$\mathbf{h_t} = \mathbf{o_t} \odot \hat{\mathbf{h}}_\mathbf{t} = \mathbf{o_t} \odot \mathbf{tanh}(\mathbf{C_t})$$

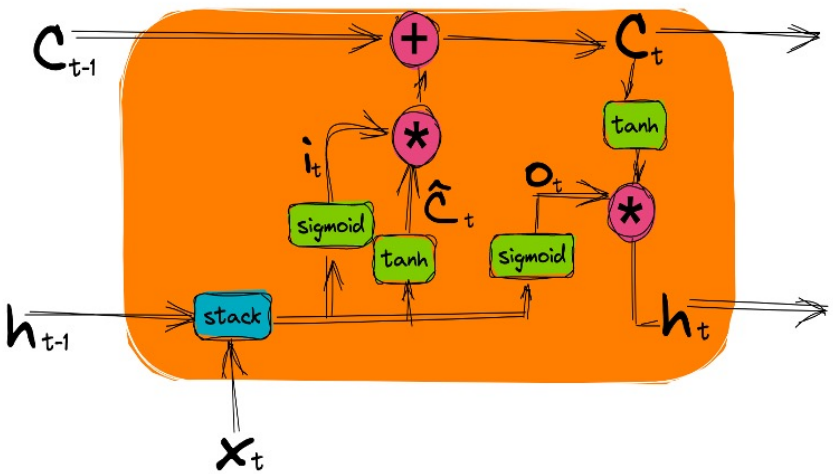


LSTM Networks: First (original) Formulation

- In the first LSTM paper, the cell state was updated as

$$\mathbf{C_t} = \mathbf{C_{t-1}} + \mathbf{i_t} \odot \hat{\mathbf{C}}$$

- This formulation ensured that the gradients could flow through the cell state without vanishing (or exploding) because $\frac{\mathrm{dC_t}}{\mathrm{dC_{t-1}}} = 1$
- So, $\frac{\mathrm{dL}}{\mathrm{dC_k}} = \frac{\mathrm{dL}}{\mathrm{dh_t}} \frac{\mathrm{dh_t}}{\mathrm{dC_t}} \frac{\mathrm{dC_t}}{\mathrm{dC_{t-1}}} \frac{\mathrm{dC_{t-1}}}{\mathrm{dC_{t-2}}} \cdots \frac{\mathrm{dC_{k+1}}}{\mathrm{dC_k}} = \frac{\mathrm{dL}}{\mathrm{dh_t}} \frac{\mathrm{dh_t}}{\mathrm{dC_t}} * 1 * 1 \cdots * 1$



LSTM Networks: Forget Gate

- LSTM's formulation with the forget gate(added a few years later) is the standard formulation.
- Forget gate decides how much of the old cell state to erase. This allows LSTM to forget whatever it finds useless. Just like other gates, forget gate is calculated as
  $\mathbf{f_t} = \mathbf{sigmoid}(\mathbf{W_f}\,[\mathbf{h_{t-1}}\mathbf{x_t}] + \mathbf{b_f})$
- The cell state in this standard formulation is then calculated as
  $$\mathbf{C_t} = \mathbf{f_t} \odot \mathbf{C_{t-1}} + \mathbf{i_t} \odot \hat{\mathbf{C}}$$
- The derivative of the cell state is not 1 anymore, but practically, LSTM with a forget gate performs better than the one without. In fact, the forget gate has been found to be the most important gate in LSTM.
  - Greff, Klaus, et al. "LSTM: A search space odyssey." IEEE transactions on neural networks and learning systems
  - Jozefowicz, Rafal, Wojciech Zaremba, and Ilya Sutskever. "An empirical exploration of recurrent network architectures." ICML 2015

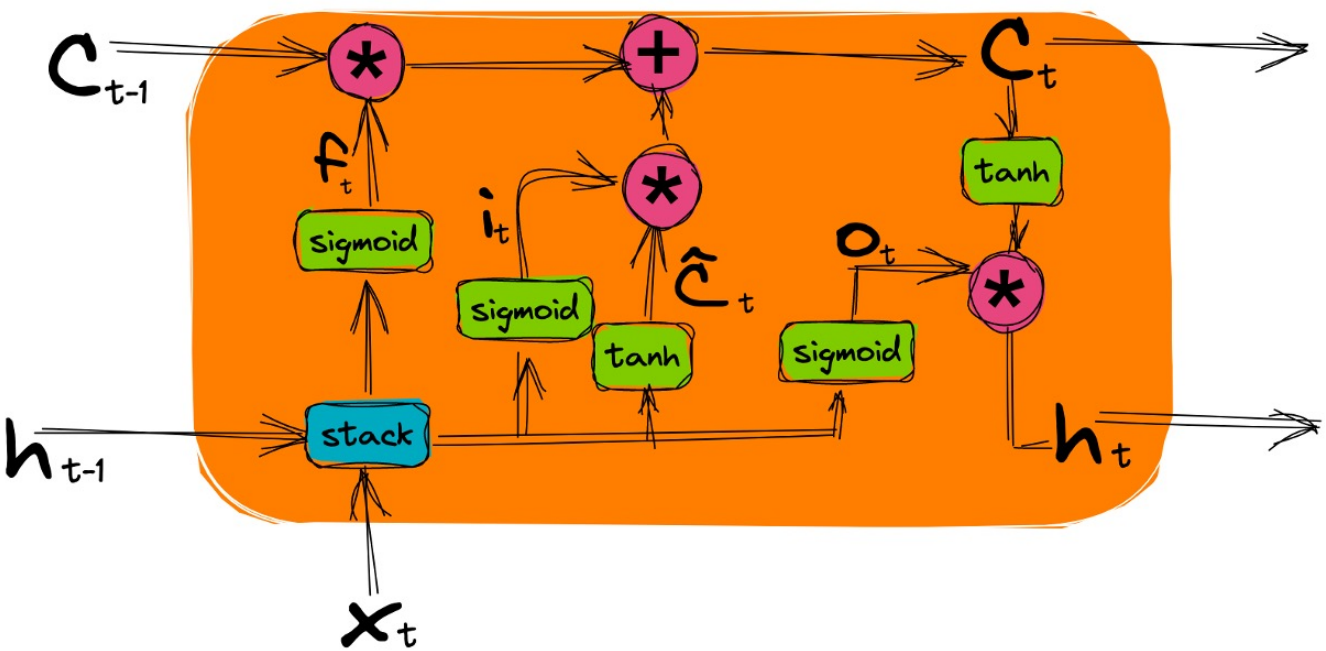
LSTM Networks: Forget Gate Initialisation

- Since the forget gate can interrupt gradient flow through the cell state, a common trick in initialising the forget gate is to set its bias to a positive value greater than 1.
- This results in the forget gate being open (having value 1) and gradients flowing through the cell state uninterrupted initially.

Standard LSTM Network

- $i_t = \text{sigmoid}(W_i \, [h_{t-1} x_t] + b_i)$
- $o_t = \text{sigmoid}(W_o \, [h_{t-1} x_t] + b_o)$
- $f_t = \text{sigmoid}(W_f \, [h_{t-1} x_t] + b_f)$
- $\hat{C} = \text{sigmoid}(W_c \, [h_{t-1} x_t] + b_c)$

- $\mathbf{C_t = f_t \odot C_{t-1} + i_t \odot \hat{C}}$
- $\mathbf{h_t = o_t \odot \tanh(C_t)}$

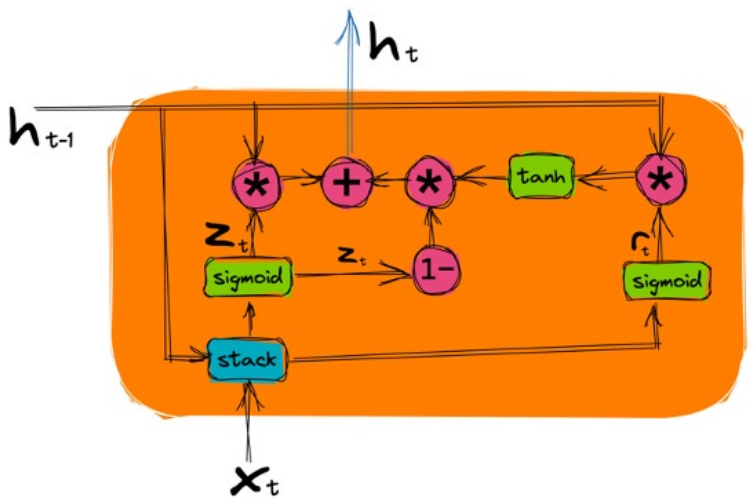


LSTM Success Highlights (not relevant for the exam)

- 2015: Google started using LSTMs for speech recognition. Google blog link
- 2016: Google started using LSTMs for translation. Google blog link
- 2017: Facebook started using LSTMs for translation. Facebook blog link
- 2017-2018: OpenAI Five's heroes in Dota2 were LSTM networks. OpenAI blog link
- 2019: DeepMind's AlphaStar for Starcraft II uses LSTM at the core. DeepMind blog link
- And many more.

GRU: Gated Recurrent Unit

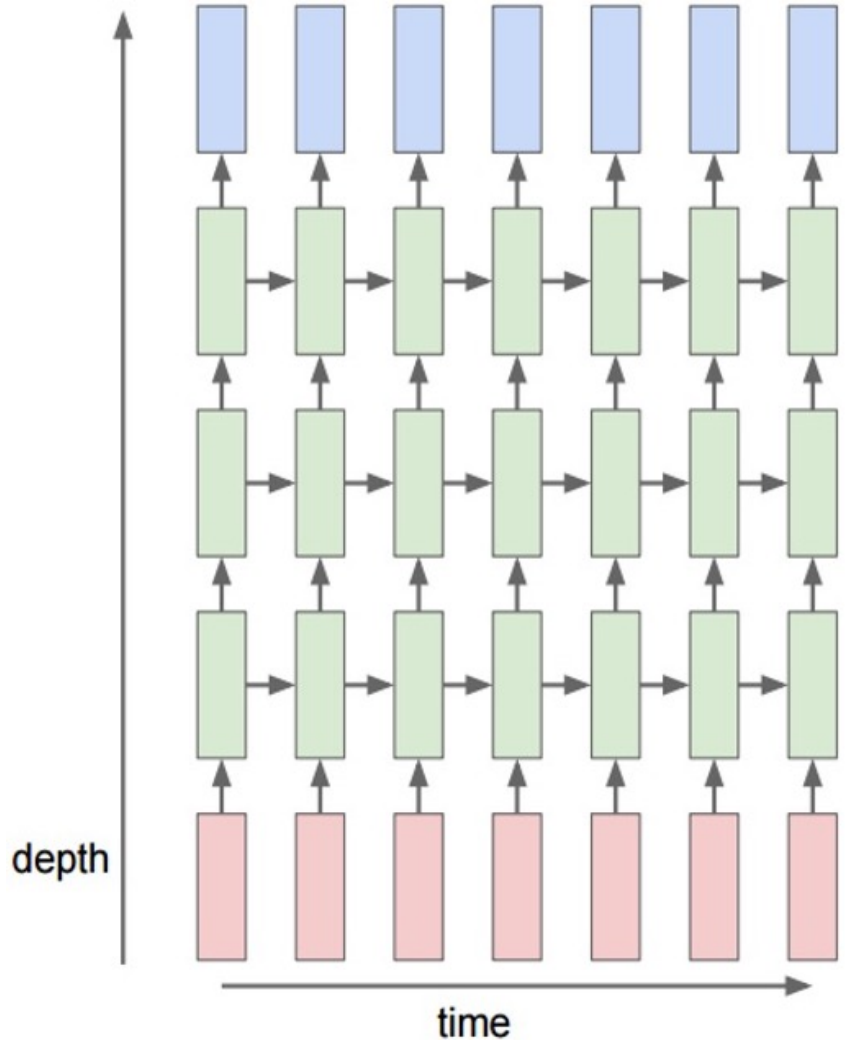
- It's a variant of LSTM. It's simpler and smaller (fewer parameters) than LSTM.
- It combines the input and forget gates into a single update gate.
- It also merges the cell state C and the hidden state h.
- **Update gate:** $z_t = \text{sigmoid}(W_z [h_{t-1} x_t] + b_z)$
- **Reset gate:** $r_t = \text{sigmoid}(W_r [h_{t-1} x_t] + b_r)$
- **Proposed activation:** $\hat{h_t} = \tanh(W_h [(r_t \odot h_{t-1}) x_t] + b_h)$
- **Final output activation:** $h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \hat{h_t}$



Outline

Multi-layer Recurrent Neural Networks

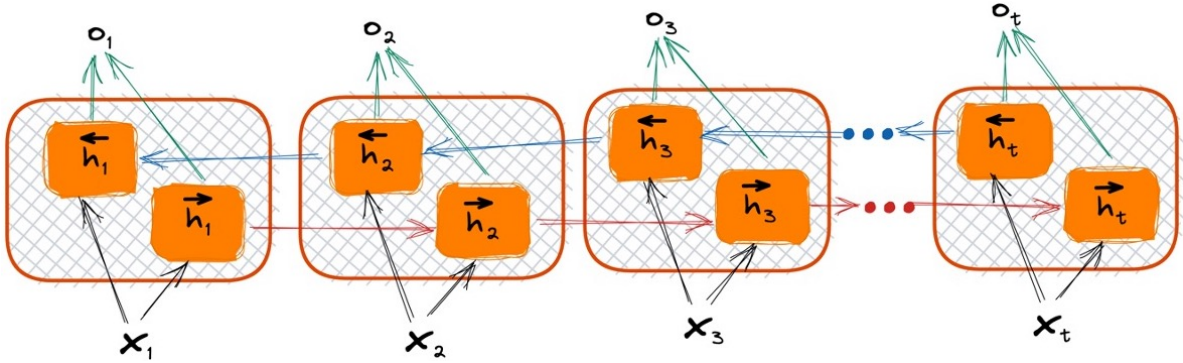
- Multi-layer RNNs can be used to enhance model complexity.
- Stacking layers creates higher level feature representation.
- Normally, only 2 or 3 layers deep. More complex relationships are learnt in the time dimension.



Bidirectional RNNs

- For many applications, looking at the "future" tokens can be helpful.
- Bidirectional RNNs traverse the input in both directions.
- Concatenate the hidden states from both directions.
- Hidden blocks in each direction can be vanilla RNNs, LSTMs, GRUs etc.
- Speech recognition, handwriting recognition, etc.

- $\overrightarrow{h_t} = \mathrm{f}(\overrightarrow{W_{hx}}\mathrm{x_t} + \overrightarrow{W_{hh}}\overrightarrow{h_{t-1}})$
- $\overleftarrow{h_t} = \mathrm{f}(\overleftarrow{W_{hx}}\mathrm{x_t} + \overleftarrow{W_{hh}}\overleftarrow{h_{t+1}})$
- $\mathrm{h_t} = \left[\overrightarrow{h_{t-1}}, \overleftarrow{h_{t+1}}\right]$
- $\mathrm{y_t} = \mathrm{g}(\mathrm{W_{hy}} + \mathrm{b})$



Outline

Additional Resources

- A blog post explaining LSTMs - https://colah.github.io/posts/2015-08-Understanding-LSTMs/
- Stanford's lecture on RNNs: cs231n: Lecture 10 — Recurrent Neural Networks
- An awesome blog post from Andrej Karpathy on RNNs and Language Modelling: The Unreasonable Effectiveness of Recurrent Neural Networks