

Module 4

Sequence Modelling: Recurrent and Recursive Nets

Sequence modeling means building models that can understand, generate or predict data that comes in a sequence.

Example:

Audio $\rightarrow$ A spoken phrase $\rightarrow$ Convert to text

Recurrent Neural Networks (RNNs)

RNNs are designed for sequential data - like text, audio or time series.

Structure $\rightarrow$ Linear chain

Typical use case $\rightarrow$ Language, speech and time series

Example $\rightarrow$ Text Generation

Recursive Neural Networks

Recursive networks are similar in spirit but process hierarchical (tree-like) structures.

Structure $\rightarrow$ Tree

Typical use case $\rightarrow$ Syntax trees, structured data

Example $\rightarrow$ Sentence parsing.

Unfolding Computational Graphs

A computational graph is a diagram that represents how computation are performed in a neural network. Node represents variables or operations like inputs, outputs edge (arrows) represent the flow of data b/w operations

→ In recurrent neural networks, computations are repeated over time steps - the same operation happens again and again as new inputs arrive.

$$s(t) = f(s(t-1), \theta)$$

$s(t)$ → state at time t
 f → the function that updates the state
 θ : the parameters

→ Unfolding over time.

$$\begin{aligned}
 s(1) &= f(s(0), \theta) \\
 s(2) &= f(s(1), \theta) \\
 s(3) &= f(s(2), \theta)
 \end{aligned}$$

→ $s(t) = f(s(t-1), x(t); \theta)$ → core equation of an RNN.

$s(t)$ → state at time t
 $s(t-1)$ → the previous state
 $x(t)$ → the input at time step t
 $f(\cdot)$ → the ~~fun~~ function
 θ → the parameters

→ Hidden state Notation

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

$h(t)$ → Hidden state

→ $h(t) = g(t)(x(1), x(2), \dots, x(t); \theta)$

The hidden state at time t depends on all inputs up to time t

→ the function $g(t)$ represents the full computation

Unfolding gives us two big advantages:

1. Parameter Sharing and Generalization

No matter how long the input sequence is, the model always uses the same parameters θ and the same update rule.

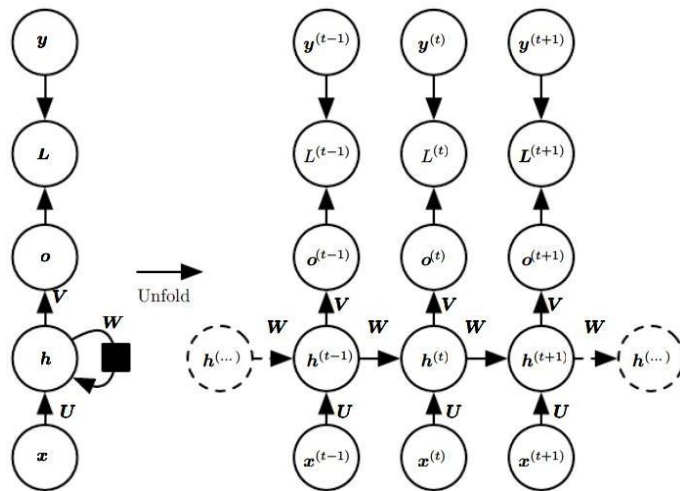
This allows the model to handle variable-length sequences without learning a new model for each possible length.

2. Efficient Learning

Sharing parameters means fewer parameters to train and better generalization.

The model learns consistent patterns (like word dependencies) that apply across positions in time.

Recurrent Neural Networks



Left Side: RNN With Recurrent Connections (Compact View).

The model receives an input sequence:

$x(1), x(2), x(3), \dots, x(t)$

At each time step t :

- Input $x(t)$ enters the RNN.
- Hidden state $h(t)$ is generated.
- Output $o(t)$ is produced.
- Loss $L(t)$ is computed using target $y(t)$.

Connections shown:

- $U \rightarrow$ Input-to-Hidden weight matrix ($x \rightarrow h$)
- $W \rightarrow$ Hidden-to-Hidden recurrent weight matrix ($h(t-1) \rightarrow h(t)$)
- $V \rightarrow$ Hidden-to-Output weight matrix ($h \rightarrow o$)

Right Side: Time-Unfolded Computational Graph

Each time step has its own nodes:

- $x(t) \rightarrow$ input at time t
- $h(t) \rightarrow$ hidden state at time t
- $o(t) \rightarrow$ output at time t

- $L(t) \rightarrow$ loss at time t

Parameter sharing:

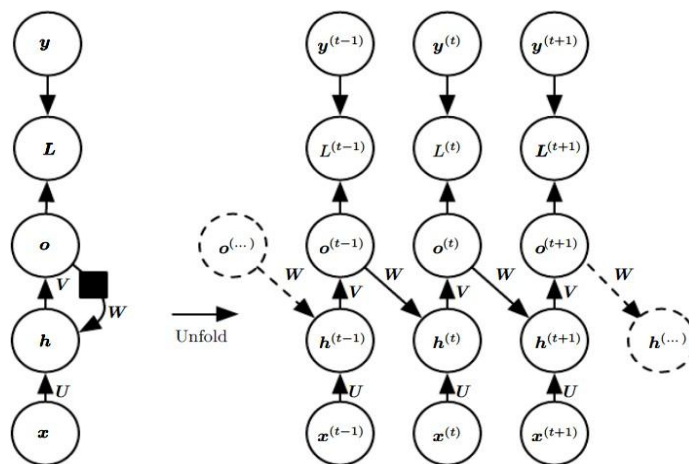
- The same U, W, V are used for *all* time steps.
- The network is not creating new weights for each time step.

For each output $o(t)$:

- Apply soft max $\rightarrow \hat{y}(t)$
- Compare $\hat{y}(t)$ with target $y(t)$
- Compute loss $L(t)$

Total loss = sum of losses at each time step:

$$L = \sum L(t)$$



Left Side: Circuit Diagram (Compact View).

At each time step t :

- Input: $x(t)$
- Hidden activations: $h(t)$
- Output: $o(t)$
- Target: $y(t)$
- Loss: $L(t)$
- There is no direct connection from $h(t-1) \rightarrow h(t)$
- Instead, output $o(t-1)$ is connected to hidden layer $h(t)$.

The model depends on past outputs, not past hidden states.

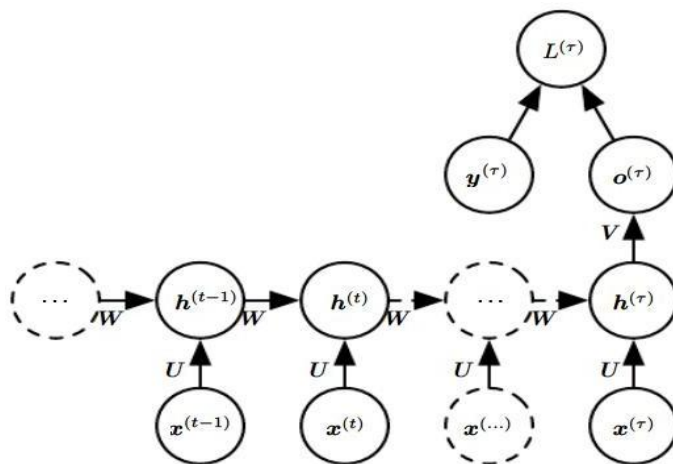
Right Side: Time-Unfolded Graph.

$$o(t-1) \rightarrow h(t) \rightarrow o(t) \rightarrow h(t+1)$$

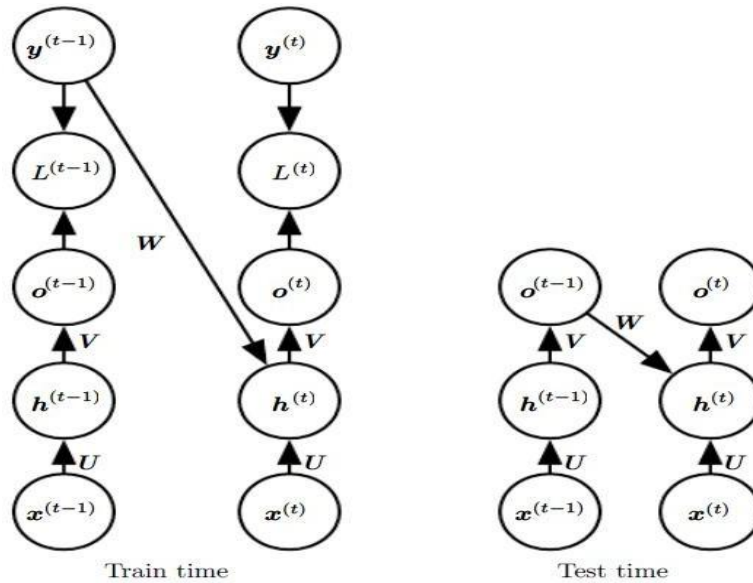
- Each $h(t)$ receives:
 - input $x(t)$
 - previous output $o(t-1)$
- There is no arrow from $h(t-1) \rightarrow h(t)$
At each time step:
- $o(t) \rightarrow \text{soft max} \rightarrow \text{predicted } \hat{y}(t)$
- Compare with true target $y(t)$
- Compute $L(t)$

Losses at each time step are independent.

Teacher Forcing and Networks with Output Recurrence



Time-unfolded recurrent neural network with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing. There might be a target right at the end (as depicted here) or the gradient on the output $o(t)$ can be obtained by back-propagating from further downstream modules.



Teacher forcing is a training technique that is applicable to RNNs that have connections from their output to their hidden states at the next time step. *(Left)* At train time, we feed the *correct output* $y^{(t)}$ drawn from the train set as input to $h^{(t+1)}$. *(Right)* When the model is deployed, the true output is generally not known. In this case, we approximate the correct output $y^{(t)}$ with the model's output $o^{(t)}$, and feed the output back into the model.

likelihood criterion is

$$\begin{aligned} & \log p \left(\mathbf{y}^{(1)}, \mathbf{y}^{(2)} \mid \mathbf{x}^{(1)}, \mathbf{x}^{(2)} \right) \\ &= \log p \left(\mathbf{y}^{(2)} \mid \mathbf{y}^{(1)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)} \right) + \log p \left(\mathbf{y}^{(1)} \mid \mathbf{x}^{(1)}, \mathbf{x}^{(2)} \right) \end{aligned}$$

Computing the Gradient in a Recurrent Neural Network

1) Computational Graph

→ parameters U, V, W, b, c

→ Time-indexed nodes: $x(t), u(t), o(t), L(t)$

2) Loss Gradient Initialization

$$L = \sum_t L(t)$$

start with:

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

3) Output Gradient Softmax + Cross-Entropy.

$$(\nabla o(t))_t = y(t) - l_t = y(t)$$

$o(t)$ is input to softmax → gives predicted probabilities.

A) Backpropagate from Final Time Step T

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

$t=T$, hidden state $h(T)$ only affects output $o(T)$:

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

B) Recursive Gradient for Hidden States

$$\nabla_{h(t)} L = \left(\frac{\partial h(t+1)}{\partial h(t)} \right)^T \nabla_{h(t+1)} L + \left(\frac{\partial o(t)}{\partial h(t)} \right)^T \nabla_{o(t)} L$$

output $o(t)$

Next hidden state $h(t+1)$

C) Apply Activation Derivatives

$$h(t) = \tanh(\cdot)$$

$$\frac{\partial h(t+1)}{\partial h(t)} = W^T \cdot \text{diag}(1 - h(t+1)^2)$$

Final recursive gradient:

$$\nabla_{h(t)} L = W^T \cdot \text{diag}(1 - h(t+1)^2) \cdot \nabla_{h(t+1)} L + V^T \cdot \nabla_{o(t)} L$$

Gradient Equations① Bias for output layer c

$$\nabla_c L = \sum_t \nabla_{o(t)} L$$

$o(t) = V h(t) + c$, the derivative w.r.t c is just the gradient on $o(t)$

② Bias for hidden layer b

$$\nabla_b L = \sum_t \text{diag}(1 - u(t)^2) \cdot \nabla u(t)$$

→ $u(t) = \tanh(\cdot)$

→ Applied element-wise a diagonal matrix

③ Output Weights V

$$\nabla_V L = \sum_t \nabla u(t) \cdot u(t)^T$$

→ Standard outer product gradient on output times hidden state.

④ Hidden-to-Hidden Weights W

$$\nabla_W L = \sum_t \text{diag}(1 - u(t)^2) \cdot \nabla u(t) \cdot u(t-1)^T$$

→ Each time step contributes based on how $u(t)$ depends on $u(t-1)$

⑤ Input-to-Hidden Weights U

$$\nabla_U L = \sum_t \text{diag}(1 - u(t)^2) \cdot \nabla u(t) \cdot x(t)^T$$

→ Similar to W , but uses input $x(t)$ instead of previous hidden state.

Recurrent Networks as Directed Graphical Models

predictive log-likelihood objective.

→ the conditional probability of the next output given past inputs

$$\log p(y(t) | x(1), \dots, x(t))$$

→ The model also uses past outputs as inputs:

$$\log p(y(t) | x(t), \dots, x(1), y(1), \dots, y(t-1))$$

→ The model to capture dependencies across both inputs and outputs

Modelling a sequence without inputs

$$P(Y) = \prod_{t=1}^T P(y(t) | y(1), \dots, y(t-1))$$

→ RNN modelling a sequence of scalars $Y = \{y(1), \dots, y(T)\}$

Negative Log-Likelihood Loss

$$L = \sum_t L(t)$$

To minimize the negative log-likelihood

where,

$$L(t) = -\log p(y(t) | y(1), \dots, y(t-1))$$

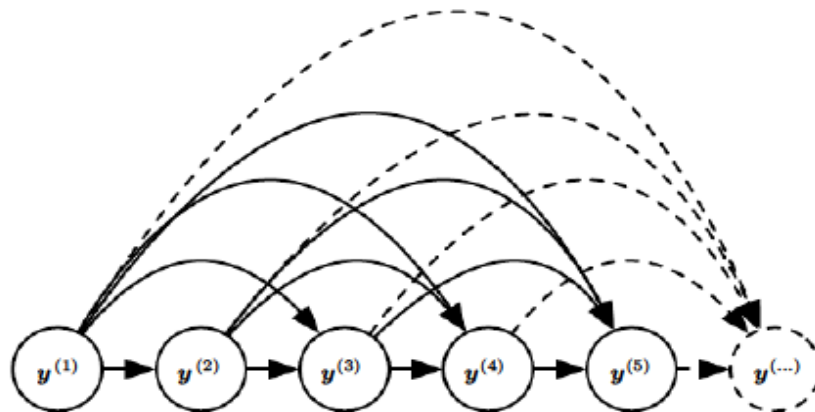


Figure 10.7: Fully connected graphical model for a sequence $y^{(1)}, y^{(2)}, \dots, y^{(t)}, \dots$: every past observation $y^{(i)}$ may influence the conditional distribution of some $y^{(t)}$ (for $t > i$), given the previous values. Parametrizing the graphical model directly according to this graph (as in equation 10.6) might be very inefficient, with an ever growing number of inputs and parameters for each element of the sequence. RNNs obtain the same full connectivity but efficient parametrization, as illustrated in figure 10.8.

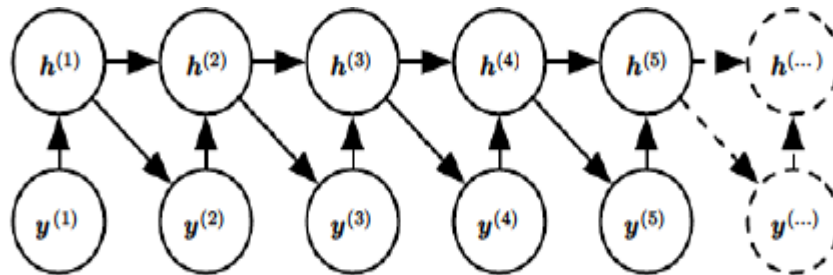


Figure 10.8: Introducing the state variable in the graphical model of the RNN, even though it is a deterministic function of its inputs, helps to see how we can obtain a very efficient parametrization, based on equation 10.5. Every stage in the sequence (for $h^{(t)}$ and $y^{(t)}$) involves the same structure (the same number of inputs for each node) and can share the same parameters with the other stages.

Modeling Sequences Conditioned on Context with RNNs

Consider a sequence of scalar variables:

$$\log p(\mathbf{y}^{(t)} \mid \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}),$$

Text Generation

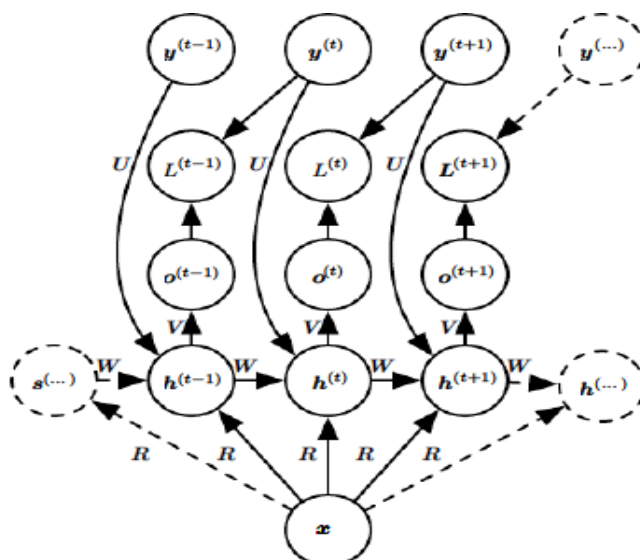
"The cat sleeps."

The RNN models:

$$P(\text{"The cat sleeps."}) = P(\text{"The"}) \times P(\text{"cat"} \mid \text{"The"}) \times P(\text{"sleeps"} \mid \text{"The cat"})$$

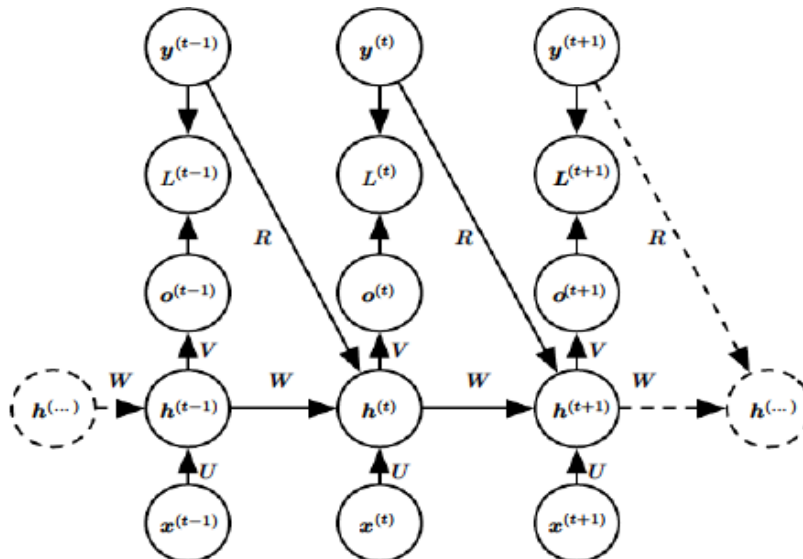
Each conditional term corresponds to one step of the RNN:

- $P(y^{(1)})$: predict first word
- $P(y^{(2)} \mid y^{(1)})$: predict second word given first
- $P(y^{(3)} \mid y^{(1)}, y^{(2)})$: predict third word given previous two



An RNN that maps a fixed-length vector x into a distribution over sequences Y . This RNN is appropriate for tasks such as image captioning, where a single image is used as input to a model that then produces a sequence of words describing the image. Each element

$y(t)$ of the observed output sequence serves both as input (for the current time step) and, during training, as target (for the previous time step).



A conditional recurrent neural network mapping a variable-length sequence of x values into a distribution over sequences of y values of the same length. The RNN contains connections from the previous output to the current state.

These connections allow this RNN to model an arbitrary distribution over sequences of y given sequences of x of the same length. Able to represent distributions in which the y values are conditionally independent from each other given the x values.

Bidirectional RNNs

Recurrent Neural Network (RNN) processes sequences in one direction — forward in time.

At each time step t :

$$h(t) = f(h(t-1), x(t))$$

and the output (prediction) $y(t)$ depends on the current and past inputs:

$$y(t) = g(h(t))$$

A Bidirectional RNN has two RNNs:

A forward RNN that reads the sequence from start $\rightarrow$ end
(captures information from the past)

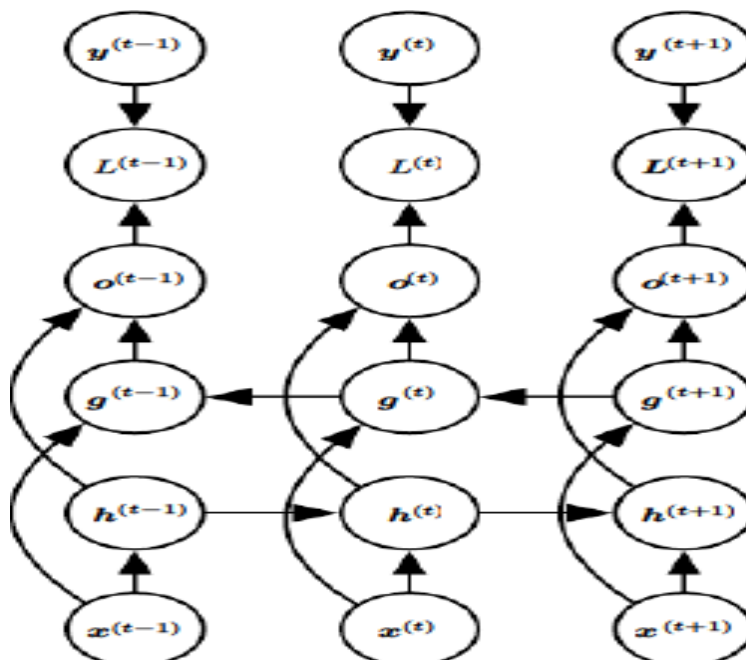
A backward RNN that reads the sequence from end $\rightarrow$ start
(captures information from the future)

At each time step t , we have two hidden states:

$h(t)$ (forward state), $g(t)$ (backward state)

These two are combined (e.g., concatenated or added):

$$o(t) = W_o[h(t); g(t)] + b$$



Bidirectional RNN processes a sequence.

It consists of two RNN layers running in opposite directions through time:

The forward RNN (bottom row with $h(t)$) moves left $\rightarrow$ right, capturing past information.

The backward RNN (middle row with

$g(t)$) moves right $\rightarrow$ left, capturing future information.

At each time step t :

The input is $x(t)$

The outputs from the two RNNs are $h(t)$ and $g(t)$

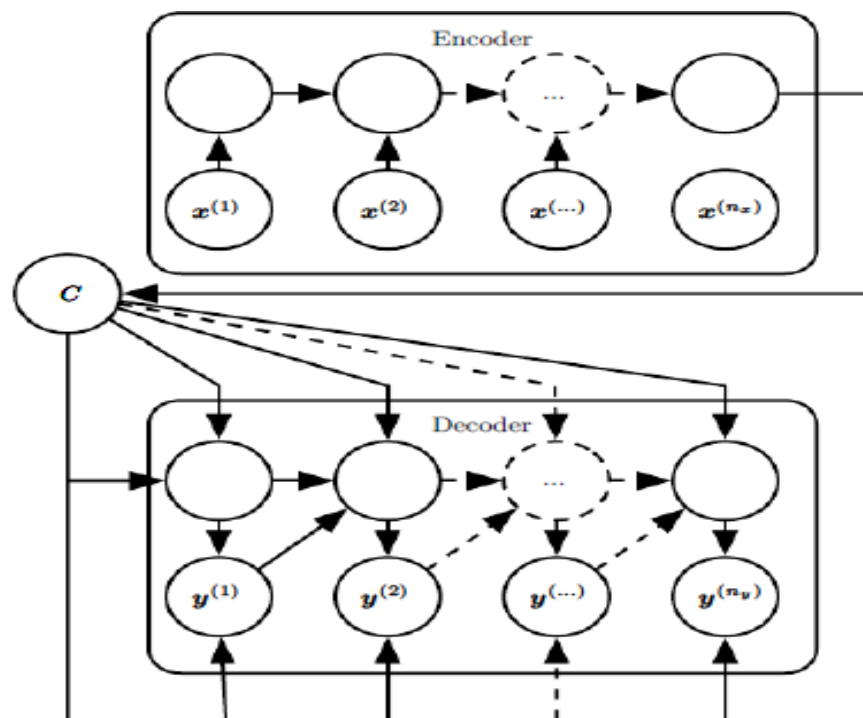
These are combined to produce the final output $o(t)$

The loss at that step is $L(t)$, which depends on $o(t)$ and the target $y(t)$.

Encoder-Decoder Sequence-to-Sequence Architectures

Input sequence of length $n_x \rightarrow$ Output sequence of length n_y .

The sequence-to-sequence (seq2seq) RNN model structure — sometimes also called the encoder-decoder architecture.



Example of an encoder-decoder or sequence-to-sequence RNN architecture,

for learning to generate an output sequence $(y(1), \dots, y(n_y))$ given an input sequence

$(x(1), x(2), \dots, x(n_x))$. It is composed of an encoder RNN that reads the input sequence

and a decoder RNN that generates the output sequence (or computes the probability of a

given output sequence). The final hidden state of the encoder RNN is used to compute a generally fixed-size context variable C which represents a semantic summary of the input sequence and is given as input to the decoder RNN.

It is composed of two main parts:

Encoder (top box) – reads and summarizes the input sequence

Decoder (bottom box) – generates the output sequence using the encoded summary.

Encoder RNN

Takes in the input sequence:

$$x^{(1)}, x^{(2)}, \dots, x^{(T_x)}$$

Each input token is processed step by step using a recurrent cell (like LSTM or GRU).

The hidden state updates as:

$$h_t = f(h_{t-1}, x^{(t)})$$

After reading the last token, the final hidden state (or a function of all hidden states) becomes the context vector C .

$$C = h_{T_x}$$

This C is a compressed representation — it captures the semantic meaning of the entire input sequence.

Decoder RNN

Starts with the context vector C .

Generates the output sequence step by step:

$y^{(1)}, y^{(2)}, \dots, y^{(T_y)}$

Each time step uses:

- Previous hidden state s_{t-1}
- Previously generated output $y^{(t-1)}$
- Context vector C

$$s_t = f(s_{t-1}, y^{(t-1)}, C)$$

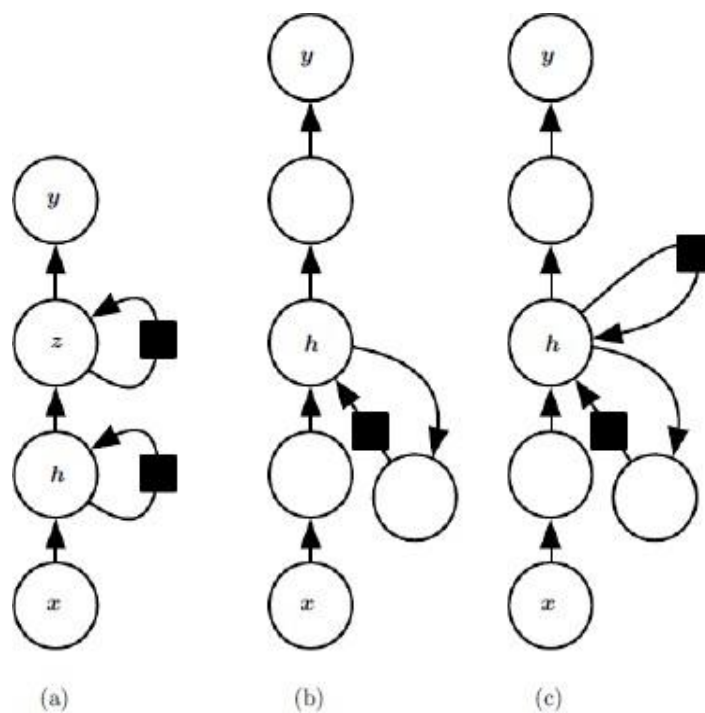
Then the decoder predicts:

$$\hat{y}^{(t)} = g(s_t)$$

Deep Recurrent Networks

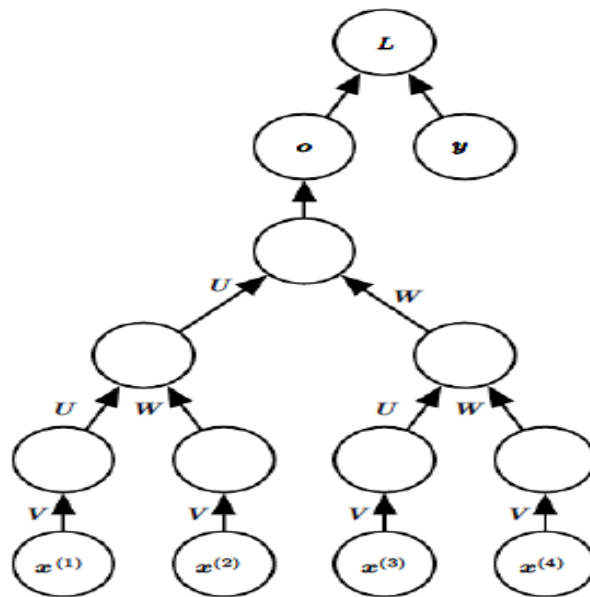
The computation in most RNNs can be decomposed into three blocks of parameters and associated transformations:

1. from the input to the hidden state,
2. from the previous hidden state to the next hidden state, and
3. from the hidden state to the output.



A recurrent neural network can be made deep in many ways, The hidden recurrent state can be broken down (a) into groups organized hierarchically. (b)Deeper computation (e.g., an MLP) can be introduced in the input-to hidden, hidden-to-hidden and hidden-to-output parts. This may lengthen the shortest path linking different time steps. (c)The path-lengthening effect can be mitigated by introducing skip connections.

Recursive Neural Networks



A recursive network has a computational graph that generalizes that of the recurrent network from a chain to a tree. A variable-size sequence $x(1), x(2), \dots, x(t)$ can be mapped to a fixed-size representation (the output o), with a fixed set of parameters (the weight matrices U, V, W). The figure illustrates a supervised learning case in which some target y is provided which is associated with the whole sequence.

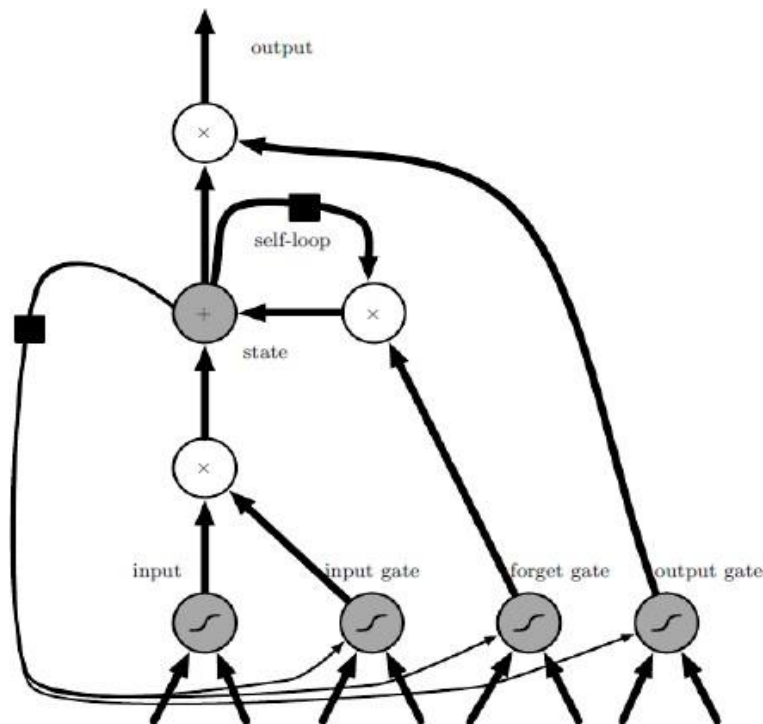
A binary tree of input vectors $x(1), x(2), \dots$ at the leaves.

Each internal node computes a representation from its children (same function & shared parameters).

The root node produces the final output o (or some nodes produce targets y).

Parameters (U, V, W or a composition function) are shared across all nodes.

The Long Short-Term Memory and Other Gated RNNs



A recurrent cell that adds gate units (sigmoids) to control flow of information into, out of and through a memory/state vector.

The two most common gated RNNs: LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit).

Gates output values in $[0,1]$ and are multiplied elementwise to control contributions.

Maintain a cell state / memory that can accumulate useful information.

Use an input gate to decide what new information to write.

Use a forget gate to decide what old information to remove.

Use an output gate to decide what part of the memory to expose as hidden output.

Because some paths (cell $\rightarrow$ cell) are (nearly) linear and gated, gradients can pass long distances stably.

Cell state c : main memory vector (has a linear self-loop).

Input gate : controls how much new info to add to c .

Forget gate : controls how much of previous $c^{(t-1)}$ to keep.

Output gate : controls the exposed hidden state $h^{(t)}$.

Candidate input proposed new content (typically $\tanh$).

Gates use sigmoid nonlinearity; candidate uses tanh (or other squashing).

LSTM

The Long Short-Term Memory and other Gated RNNs

① Forget Gate $f_i^{(t)}$: controls how much of the previous state $s_i^{(t-1)}$ to retain.

$$f_i^{(t)} = \sigma \left(b_f^i + \sum_j U_f^{i,j} x_j^{(t)} + \sum_j W_f^{i,j} h_j^{(t-1)} \right)$$

② Input Gate $g_i^{(t)}$: controls how much new information to write into the cell.

$$g_i^{(t)} = \sigma \left(b_g^i + \sum_j U_g^{i,j} x_j^{(t)} + \sum_j W_g^{i,j} h_j^{(t-1)} \right)$$

③ Cell state update $s_i^{(t)}$: Combines retained memory & new input.

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + q_i^{(t)} \tanh(b_i + \sum_j U_{ij}^{(t)} x_j^{(t)} + \sum_j W_{ij}^{(t)} h_j^{(t-1)})$$

④ Output gate $q_i^{(t)}$: Controls how much of the cell state to expose as output.

$$q_i^{(t)} = \sigma(b_i + \sum_j U_{ij}^{(t)} x_j^{(t)} + \sum_j W_{ij}^{(t)} h_j^{(t-1)})$$

$$h_i^{(t)} = \tanh(s_i^{(t)}) \cdot q_i^{(t)}$$

Other Gated RNNs

① The GRU updated Recursion

$$h_i^{(t)} = u_i^{(t)} h_i^{(t-1)} + (1 - u_i^{(t)}) \tanh[b_i + \sum_j U_{ij} x_j^{(t)} + \sum_j W_{ij} (r_j^{(t)} h_j^{(t-1)})]$$

$h_i^{(t)} \rightarrow$ new hidden state

$h_i^{(t-1)} \rightarrow$ the previous hidden state

$u_i^{(t)} \rightarrow$ update gate

$r_j^{(t)} \rightarrow$ reset gate

$x_j^{(t)} \rightarrow$ current input at time t .

$U, W, b \rightarrow$ weight matrices and biases

② The Gated Equation

Update gate $u_i^{(t)}$ -

$$u_i^{(t)} = \sigma(b_i^u + \sum_j U_{ij}^u x_j^{(t)} + \sum_j W_{ij}^u h_j^{(t-1)})$$

Reset gate $r_j^{(t)}$

$$r_j^{(t)} = \sigma(b_j^r + \sum_j U_{ij}^r x_j^{(t)} + \sum_j W_{ij}^r h_j^{(t-1)})$$

==

