UCS2723 DEEP LEARNING

Unit 3

Recurrent Neural Networks

Unfolding Graphs - RNN Design Patterns : Acceptor - Encoder - Transducer, Gradient Computation - Sequence Modeling Conditioned on Contexts - Bidirectional RNN - Sequence to Sequence RNN - Deep Recurrent Networks - Recursive Neural Networks - Long Term Dependencies ; Leaky Units : Skip connections and dropout ; Gated architecture : LSTM

* Introduction to Recurrent Neural Networks (RNNs)

→ RNNs are a family of neural networks for processing sequential data.

→ Specialized for sequences : $x^{(1)}, x^{(2)}, \ldots x^{(\tau)}$

→ Scales to long sequences and can handle variable - length sequences

→ Parameter sharing across time steps enables generalization

Parameter Sharing in RNNs

→ Parameters are shared across different parts of the model.

→ Allow model generalize across different sequence lengths

eg. extracting information from sentences with varying word positions.

→ This is different from feedforward networks that have separate parameters for each input feature.

Parameter Sharing in Convolutional Networks vs RNNs

→ Convolutional networks use parameter sharing across a 1-D temporal sequence.

→ RNNs share parameters in a different way, through a deep computational graph.

→ Convolution is shallow, RNNs involve recurrent connections for deeper processing.

* Computational Graphs and RNNs

→ A computational graph formalizes the structure of computations.

→ Unfolding a recurrent computation into a computational graph shows a repetitive structure

eg. a classical dynamical system: $s(t) = f(s(t-1); \theta)$

For $\gamma = 3$

$$s^{(3)} = f(s^{(2)}; \theta)$$

$$= f[f(s^{(1)}; \theta); \theta)$$

→ This unfolding results in a directed acyclic graph (DAG).

→ It allows parameter sharing across time steps.

→ If there is an external signal $x(t)$, then the equation becomes

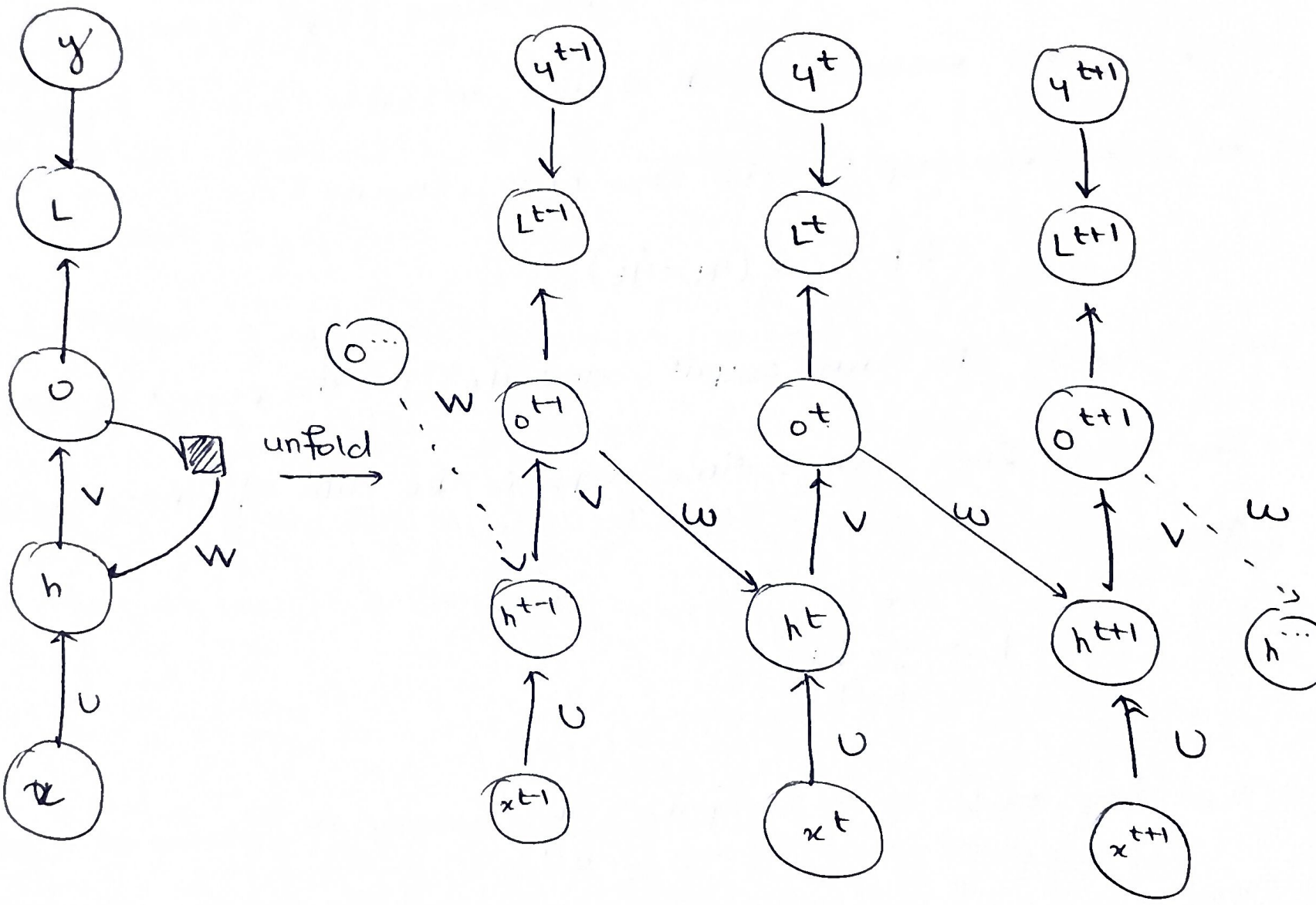
$$s^{(t)} = f[s^{(t-1)}; x^{(t)}; \theta)$$

* Architecture of RNNs and Unfolding

→ Includes a hidden state $h^{(t)}$ which represents a lossy summary of past inputs.

→ $h^{(t)}$ used for tasks like predicting the next word in language models

→ Consider an RNN and its unfolding.



* Advantages of Unfolding RNNs

→ maintains the same input size regardless of sequence length

→ uses the same transition function f across all time steps

→ Facilitates learning with fewer training examples

→ Helps generalize to unseen sequence length

* Training RNNs: Backpropagation through Time

→ Training RNNs requires calculating gradients of the loss function L with respect to the weights W_h and W_x.

→ To handle sequential data, we use a process called Backpropagation Through Time (BPTT)

→ BPTT 'unrolls' the RNN across time steps, treating it as a deep neural network with shared weights across layers

→ At each time step t, the loss L_t is computed:

$$L_t = \ell(y_t, \hat{y}_t)$$

where y_t is the true output, and $\hat{y}_t$ is the predicted output.

→ The total loss over T time steps is the sum of individual losses:

$$L = \sum_{t=1}^{T} L_t$$

→ The key task is to compute $\frac{\partial L}{\partial W}$ using the chain rule across t steps.

Gradient Computation

→ The gradient of the loss L with respect to the weights W involv summing over the contributions from all the time steps.

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

→ Using the chain rule, the gradient at time step t depends on the gradients from all the time steps:

$$\frac{\partial L_t}{\partial W} = \frac{\partial L_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W}$$

→ This recursive process continues back through time, hence it is called Back propagation Through Time.

* Challenges with BPTT

→ The recurrent structure of RNNs leads to challenges during BPTT, particularly:

(i) Exploding Gradients - Gradients became excessively large

(ii) Vanishing Gradients - Gradients shrink to near zero.

→ These issues occur due to repeated multiplication of the derivative of the recurrent weight matrix W_h over time

Vanishing Gradients

→ When W_h has small eigenvalues, $\frac{\partial h_t}{\partial h_{t-1}}$ tends to shrink

→ Over many time steps, this results in: $\frac{\partial L}{\partial W} \approx 0$

→ This means that there is no significant weight update, the network 'forget's' past dependencies.

Exploding Gradients

→ When W_h has large eigenvalues, $\frac{\partial h_t}{\partial h_{t-1}}$ grows exponentially

→ Gradients become excessively large:

$$\frac{\partial L}{\partial W} \to \infty$$

→ Leads to unstable training, where weight updates become erratic

* Solutions to Gradient Problems

1. Gradient Clipping - Limits the magnitude of gradients during backpropagation

2. Use of LSTM / GRUs - Specialized architecture designed to maintain gradients over long sequences

3. Proper Weight Initialization - Ensures that eigen values of W_n remain within a stable range

4. Batch Normalization - Helps control gradients during training.

UCS2723 DEEP LEARNING

Unit 3

Recurrent Neural Networks

*Whiteboard Analogy for RNNs

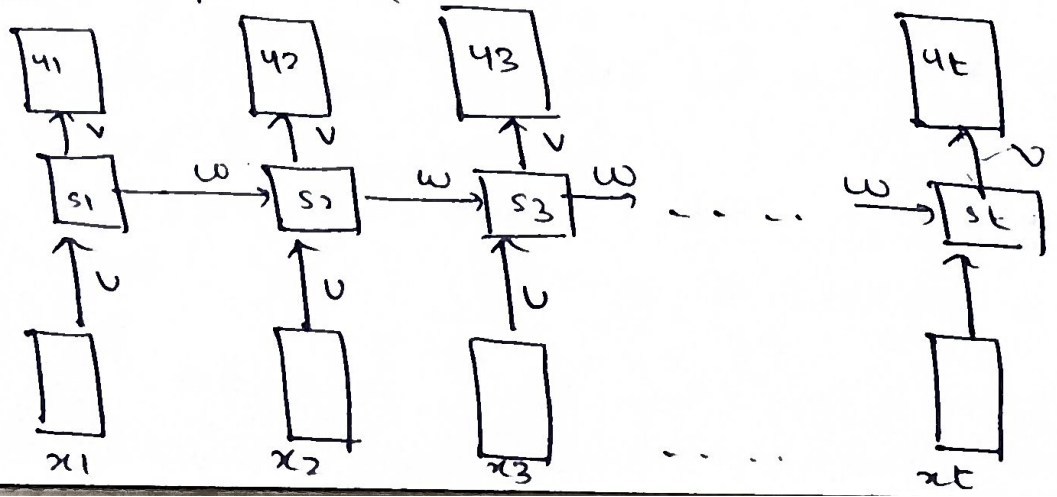
→ Imaging a whiteboard where you write down information at each time step. Over time the board gets filled up, and older information gets overwritten, making it difficult to retrieve the original content.

→ In RNNs, the information from earlier timesteps gets morphed or lost as new information is added, leading to difficulties in retaining long-term information.

→ The information stored in the state becomes progressively less useful for long-term dependencies, making it hard to trace back and assign responsibility to earlier time steps during backpropagation.

Formally,

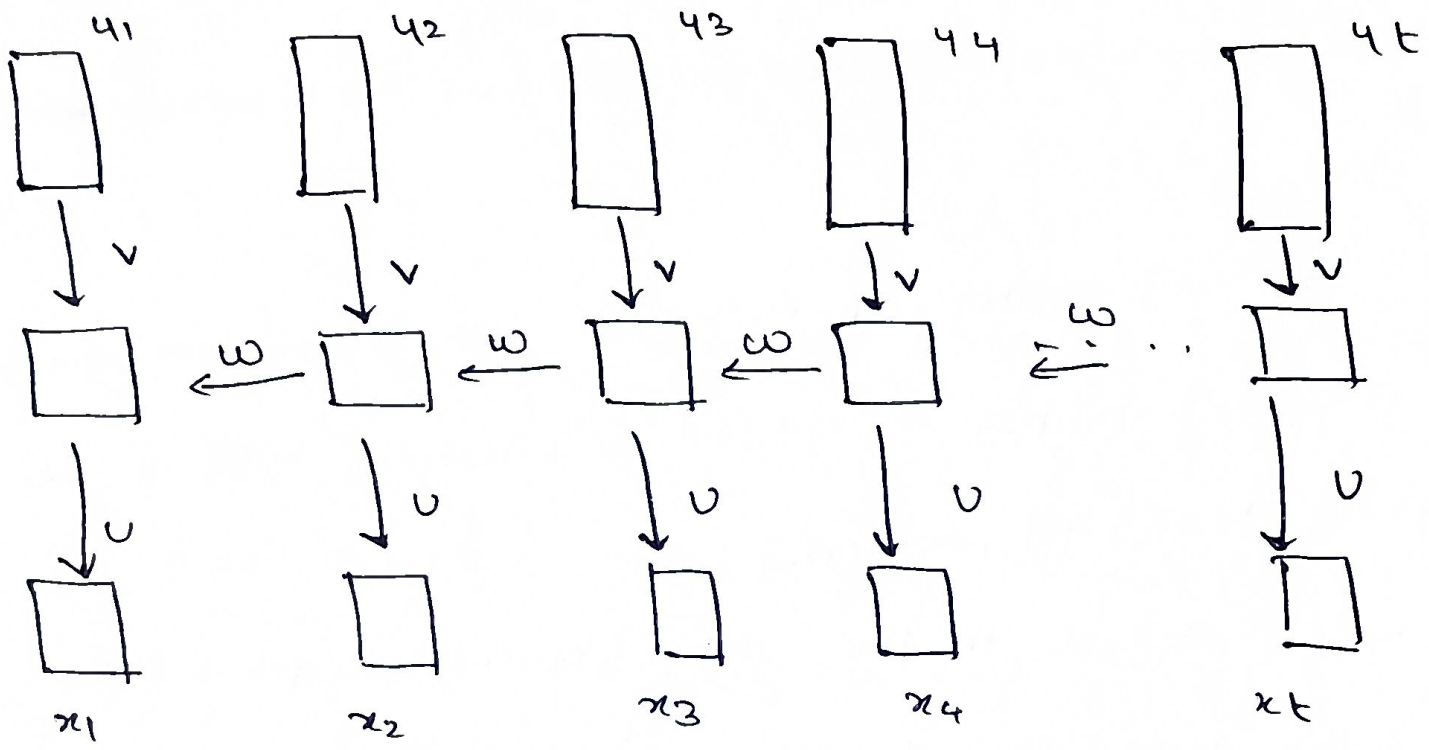
Consider the following RNN



→ Each state S_i of an RNN records information from all the previous timesteps

→ After t steps, the information stored at time step $t-k$ (for some $k < t$) gets completely morphed so much that it would be impossible to extract the original info at time step $t-k$.

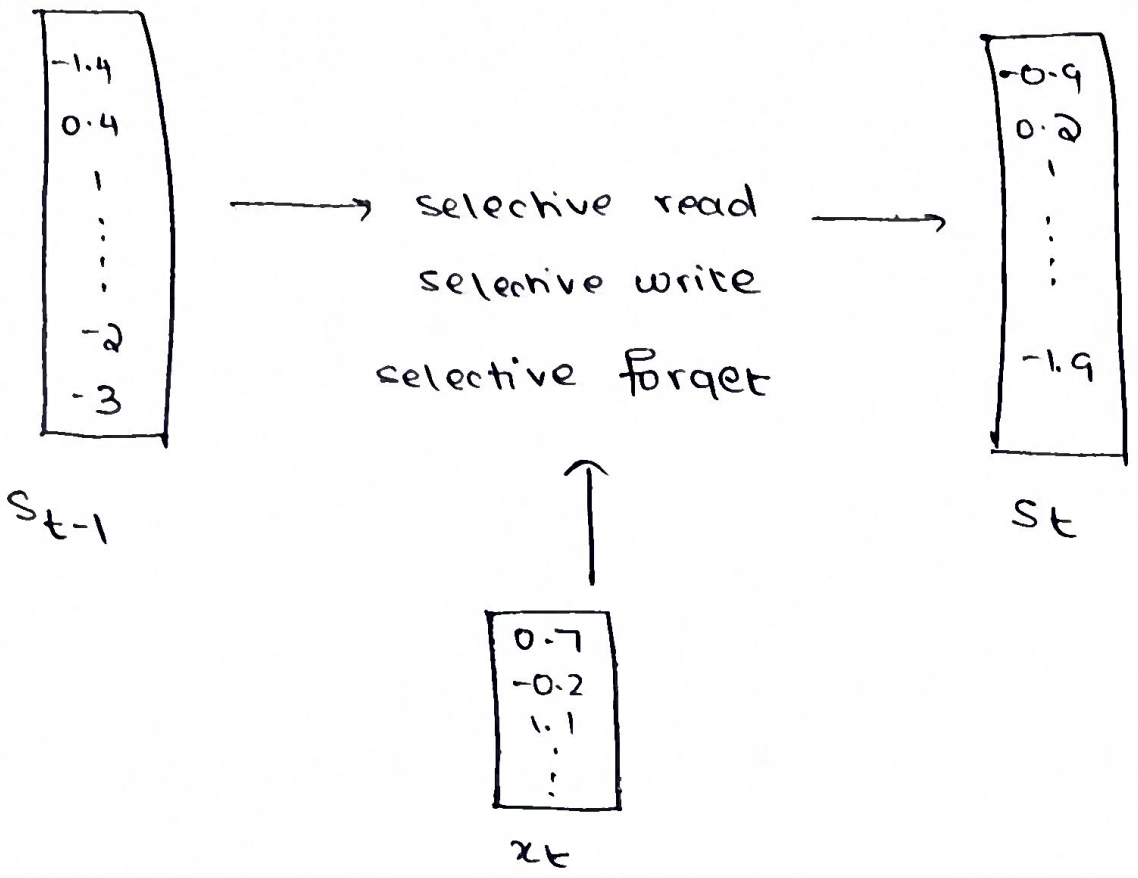
Now consider the RNN during backpropagation



→ It is very hard to assign the responsibility of the error caused at time step t to the events that occurred at time step $t-k$.

→ In order to overcome this limitation of RNNs, which have a finite state size, LSTMs can be utilized which make use of the properties of selective read, selective write and selective forget.

Consider the following LSTM



→ s_{t-1} is computed. Provide new information x_t to compute the new state s_t.

→ Use selective write, selective read and selective forget so that only important info is retained in s_t

A. **Selective Write Mechanism**

→ In LSTM, the output gate decides what fraction of the previous state s_{t-1} should be passed onto the next state.

→ Instead of passing the entire state, LSTM allows selective writing of only certain portions of the current state to the next state.

→ A vector o_{t-1} is introduced, which is multiplied with the current state s_{t-1} to determine what fraction of each

element is passed on.

→ The LSTM needs to learn the value of o_{t-1}.

This is done as follows:

$$o_{t-1} = \sigma(W_o h_{t-2} + V_o x_{t-1} + b_o)$$

$$h_{t-1} = o_{t-1} \odot \tanh(s_{t-1})$$

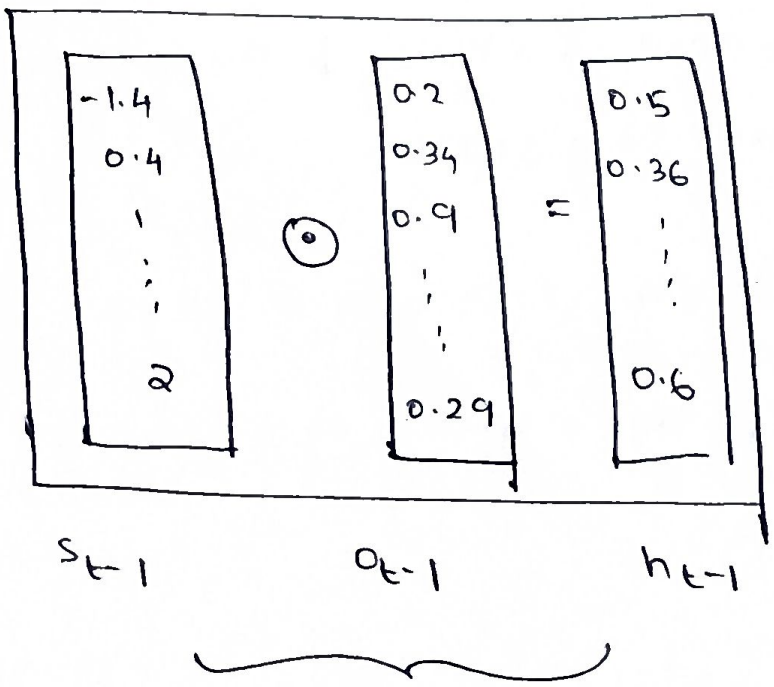
where:

→ The hidden state h_{t-2} is used to compute the output gate o_{t-1}

→ W_o, V_o, b_o are parameters to be learned along with the existing parameters W, U, V.

→ σ = sigmoid function - to ensure that the values are between 0 and 1.

→ $\odot$ is the element-wise Hadamard product



B. Selective Read Mechanism

→ A new temporary state $\tilde{s}_t$ is calculated using the previous hidden state h_{t-1}, and the new input x_t. This state captures the updated information at time step t.

→ However, we may not want to use all the information in $\tilde{s}_t$. To handle this, the input gate (i_t) decides how much of the new information should be used.

→ The new state $\tilde{s}_t$ is computed as

$$\tilde{s}_t = \tanh(Wh_{t-1} + Ux_t + b)$$

To use only some info. From this, the input gate (i_t) is defined as:

$$i_t = \sigma(w_i h_{t-1} + U_i x_t + b_i)$$

The input gate selectively reads the state information

$$s_t = \tilde{s}_t \odot i_t$$

Here, W, U, b are parameters for the state $\tilde{s}_t$.

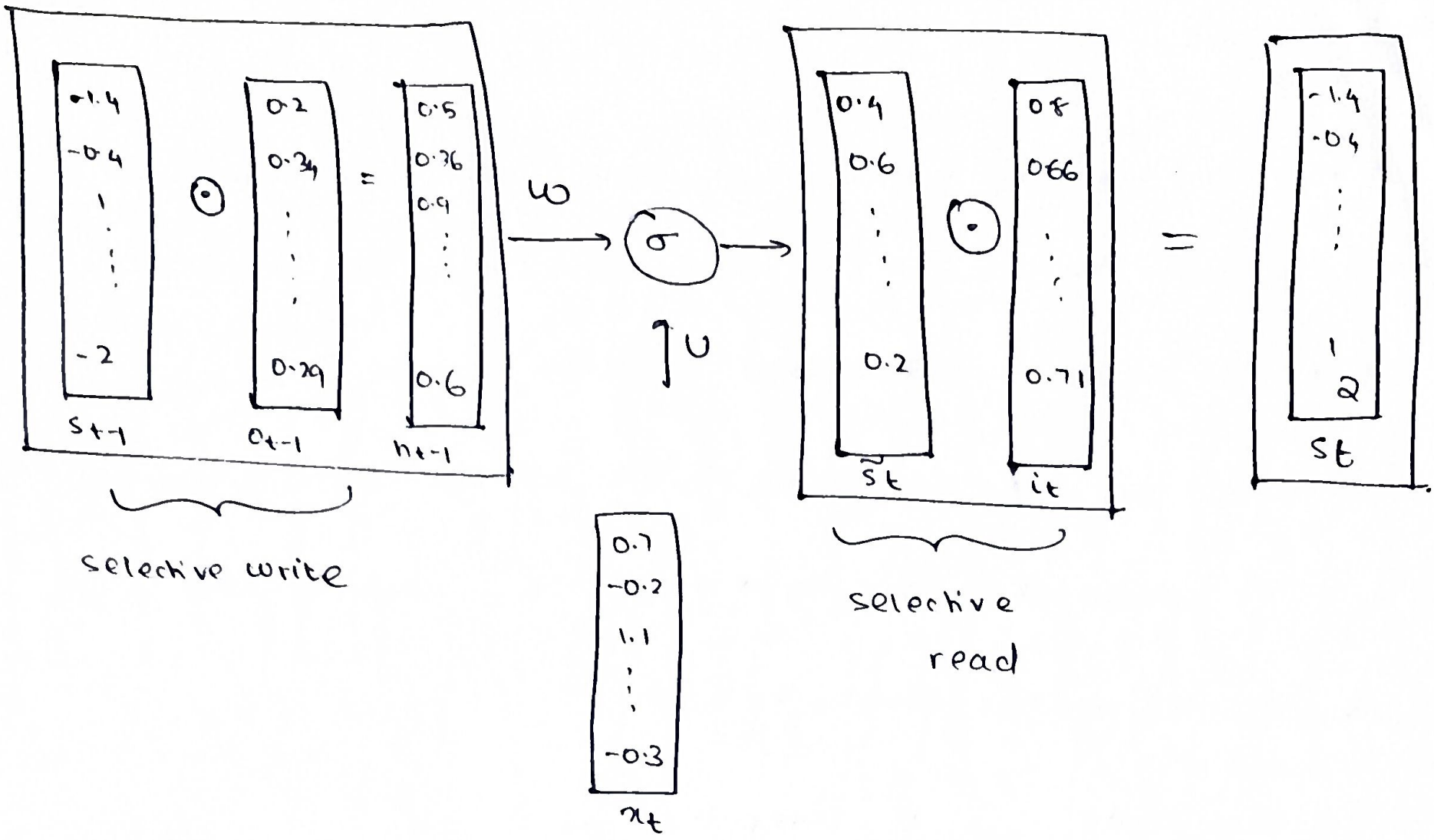
W_i, U_i, b_i are parameters for the input gate i_t

σ = sigmoid function

$\odot$ = element-wise Hadamard product

h_{t-1} = hidden state from previous time step

x_t = i/p at current time step



c. **Selective Forget Mechanism**

→ The forget gate controls which parts of the previous state s_{t-1} should be forgotten to make room for new information.

→ This gate is critical for preventing the accumulation of irrelevant information over time, ensuring that only useful information is kept.

→ First, combine s_{t-1} and $\tilde{s}_t$ to get the new state

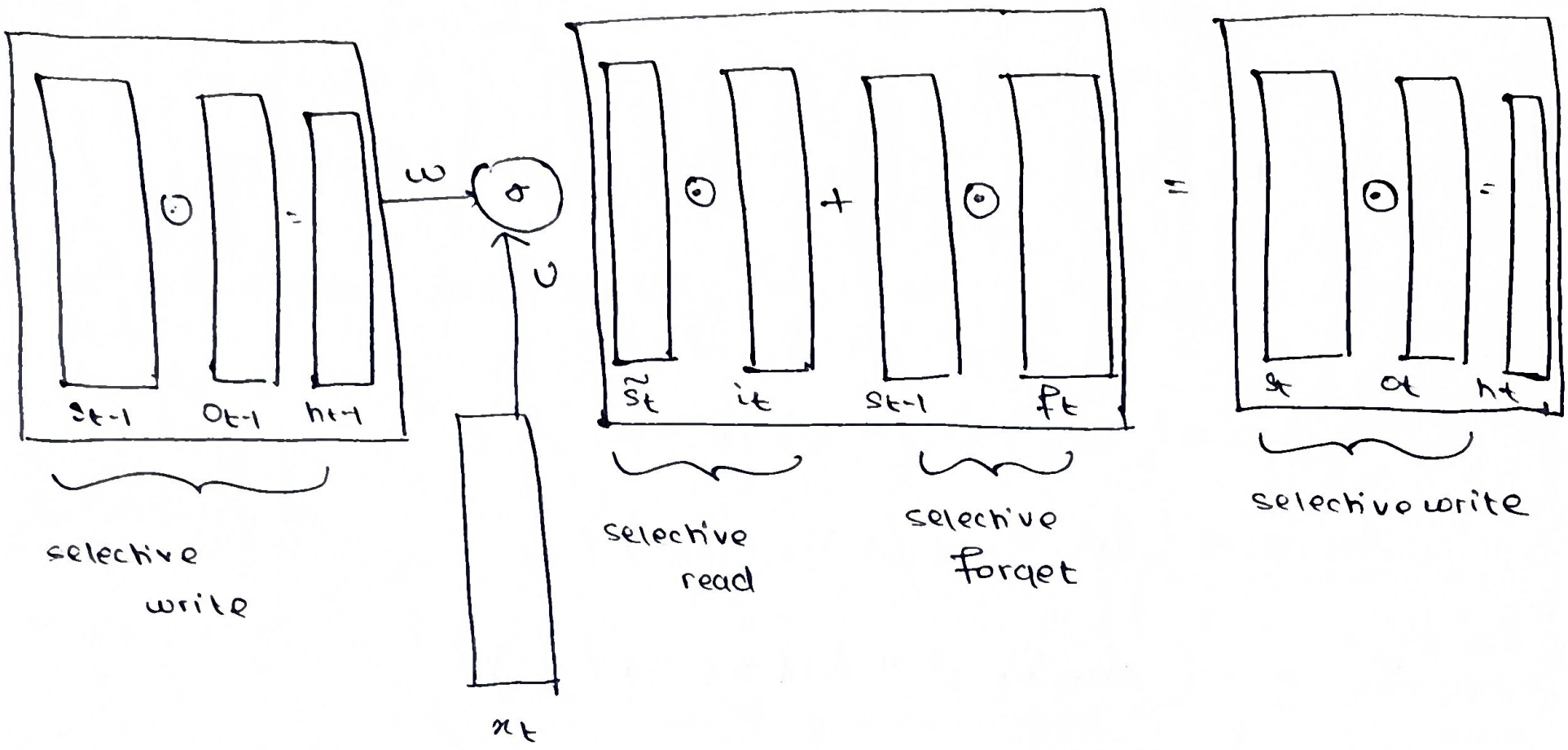
$$s_t = s_{t-1} + i_t \odot \tilde{s}_t$$

→ But, we don't want to use the whole of s_{t-1} and might want to forget some portion of it

→ The <u>forget gate</u> works as follows:

$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f)$$

Now the architecture looks like:



∴ The full set of equations for LSTM are as follows:

1. Previous state : s_{t-1}

2. Output gate : $o_{t-1} = \sigma(W_o h_{t-2} + U_o x_{t-1} + b_o)$

3. Selectively write : $h_{t-1} = o_{t-1} \odot \tanh(s_{t-1})$

4. Current (temporary) state : $\tilde{s}_t = \tanh(W h_{t-1} + U x_t + b)$

5. Input Gate : $i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$

6. Selective Read $= i_t \odot \tilde{s}_t$

7. New State (Temporary) : $s_t = s_{t-1}$ ⊙ i_t

8. Forget Gate : $f_t = \sigma(w_f h_{t-1} + U_f x_t + b_f)$

9. Output State : $s_t = f_t \odot s_{t-1} + i_t \odot \tilde{s}_t$

10. Output of LSTM : $rnn_{out} = h_t = o_t \odot \tanh(s_t)$

Gates

$$o_t = \sigma(w_o h_{t-1} + U_o x_t + b_o)$$

$$i_t = \sigma(w_i h_{t-1} + U_i x_t + b_i)$$

$$f_t = \sigma(w_f h_{t-1} + U_f x_t + b_f)$$

States

$$\tilde{s}_t = \tanh(w h_{t-1} + U x_t + b)$$

$$s_t = f_t \odot s_{t-1} + i_t \odot \tilde{s}_t$$

$$h_t = o_t \odot \tanh(s_t)$$

$$rnn_{out} = h_t$$

* BiLSTM

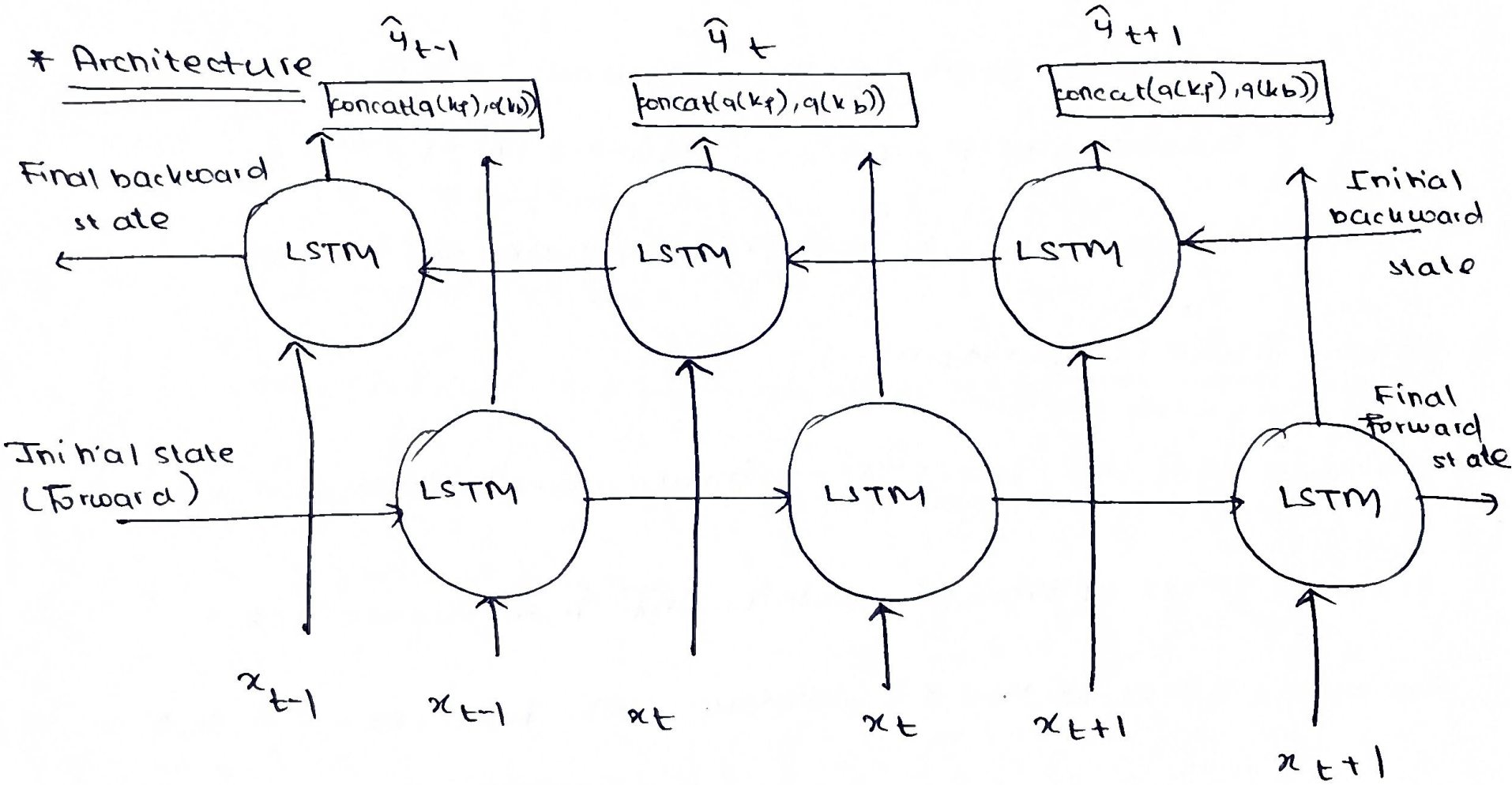
→ BiLSTM = Bidirectional Long Short Term Memory

→ an extension of the traditional LSTM architecture

→ can process data in both forward and backward directions

→ allows it to capture both past and future context, which is useful for tasks where context from both sides of a word or sequence matters.

* Architecture



→ has two LSTM layers running in parallel

→ Forward layer: process sequence L → R

process sequence R → L

→ Outputs are combined to form the final representation.

* Working Principle

1. Input Sequence Processing

→ Input processed in both directions simultaneous

The → cat → sat → on → the → mat

mat → the → on → sat → cat → The

2. Dual LSTM Layers

→ Two LSTM layers

Forward LSTM : process sequence L → R

Backward LSTM : process sequence R → L

→ Each LSTM maintains its own hidden state and cell state

3. Hidden State Computation

→ At each time step, both LSTMs compute their hidden states:

Forward LSTM at time t: contains info from words 1 to t

Backward LSTM at time t: contains info from words n to t

(n = sequence length)

4. Combining Outputs

→ Outputs from both directions combined for final output.

→ Common combination methods:

Concatenation : $[h_{forward}, h_{backward}]$

Element-wise sum : $h_{forward} + h_{backward}$

Element-wise product : $h_{forward} * h_{backward}$

5. Context Representation

→ Combined output represents word context considering the past & future

eg. For 'sat' in 'The cat sat on the mat'

Forward LSTM provides context of 'The cat'

Backward LSTM provides context of 'the mat'

→ Results in rich, bidirectional context representation.

6. Output Layer

→ Combined representation fed into final output layer

→ Output layer depends on task:

(i) Classification : Dense layer with softmax activation

(ii) Sequence Labelling : Time distributed dense - layer

(iii) Machine Translation : Another LSTM for decoding

* Features of BiLSTM

1. Parallelization - forward & backward passes computed in parallel
2. Information flow - each direction captures different sequence aspects
3. Variable Sequence Length - Handles variable length inputs naturally
4. Gradient Flow - Gradients flow in both directions during training
5. Feature Extraction : extracts complex features from full context

* BiLSTM vs. LSTM

Pros	Cons
→ captures bidirectional context	→ Increased computational complexity
→ improved performance	→ Higher memory usage
→ better for tasks requiring full context (eg. NER, Machine Translation)	→ Not suitable for real-time applications - requires full sequence before processing
	→ potential overfitting (on small datasets)

* When to use BiLSTM

1. Tasks requiring full context understanding
2. Sufficient computational resources available
3. Large enough dataset to prevent overfitting
4. Examples:

   For (i) sentiment analysis
   (ii) NER
   (iii) Machine Translation
   (iv) Speech Recognition

* Leaky Units

→ Leaky units are designed to address the vanishing gradient problem

→ They allow a small amount of information to 'leak' through each layer.

→ Two main kinds of leaky units:

(i) Skip connections

(ii) Dropout

A. Skip connections

→ also known as residual connections

→ allows info to skip one or more layers

→ Formula: $y = F(x) + x$

→ Benefits:

(i) Mitigate vanishing gradients

(ii) Enable training of very deep networks

(iii) Improve information flow

B. Dropout

→ regularization technique to prevent overfitting

→ randomly drops out a % of neurons during training

→ Formula: $y = (r * w) \wedge x$ r = binary mask

→ Benefits:

(i) reduces overfitting

(ii) Improves generalization

(iii) Acts as an ensemble of diff. networks

* Gated Architectures

(i) Long Short - Term Memory (LSTM)

(ii) Gated Recurrent Unit (GRU)

→ Gated architectures are designed to handle long-term dependencies in sequential data

→ Use gating mechanisms to control information flow

A. LSTM

→ a kind of RNN

→ designed to handle long-term dependencies

→ Key components :

(i) cell gate

(ii) input gate

(iii) forget gate

(iv) output gate

(i) **Cell State** - long-term memory of the network

(ii) **Input Gate** - controls what new information to add to cell state

(iii) **Forget Gate** : - decides what info to discard from cell state

(iv) **Output Gate** : - Determines what to output based on cell state

Advantages of LSTM - effectively handles long-term dependencies
- mitigates vanishing and exploding gradients problems
- selective memory retention & forgetting
- widely used in sequence modelling tasks
  - NLP
  - speech Recognition
  - Time series Analysis

* Gated Recurrent Units (GRUs)

→ simplified version of LSTM

→ combines the forget and input gates into a single (update gate)

→ merges cell state and hidden state

Key Components of GRU

1. **Update Gate (z_t)**

→ decides how much of the past information should be kept and how much of the current input should be used to update the hidden state

→ combines the roles of LSTM's forget gate and input gate.

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

2. **Reset Gate (r_t)**

→ controls how much of the previous input should be forgotten

→ important where only recent inputs are relevant

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

3. **Current Memory Content ($\tilde{h}_t$)**

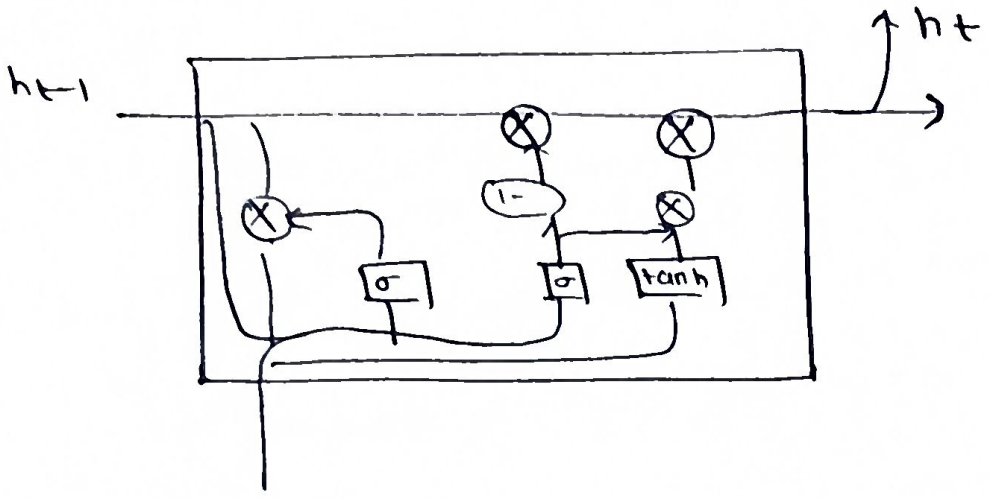
→ represents the candidate new hidden state for the time step t, incorporating the current input & a gated version of the previous hidden state

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

4. Final Hidden State (h_t)

→ combines the influence of past info (h_{t-1}) and new information (h_t)

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



* GRU vs LSTM: Advantages and Comparisons

Advantages

→ simpler architecture than LSTM

→ faster to train, less data to generalize

→ sometimes outperforms LSTM

Comparison

→ GRU has 2 gates (update & reset), vs LSTM's 3 gates

→ GRU doesn't have a separate cell state

→ GRU is generally faster, LSTM can be more powerful for some tasks

* Sequence-to-Sequence RNNs

→ In order to learn to generate an output sequence $(y(1), y(2) \ldots y(n_y))$ given an input sequence $(x(1), x(2), \ldots x(n_x))$ - an RNN can be used.

→ It is an encoder-decoder model, where the encoder RNN reads the input sequence, and a decoder RNN generates the output sequence.

Formally, given $y_1, y_2 \ldots y_{t-1}$, we want to find:

$$y^* = \text{argmax}\, P(y_t \mid y_1, y_2 \ldots y_{t-1})$$

$P(y_t \mid y_1, y_2 \ldots y_{t-1})$ is rewritten as:

$$P(y_t = j \mid y_1^{t-1}) = \text{softmax}(V s_t + c)_j$$

where $j \in V$

V = vocabulary

Encoder RNN

→ The final hidden state of the encoder RNN is used to compute a fixed size context variable c, which represents a semantic summary of the input sequence and is given as input to the decoder RNN.

Context Vector C → The context vector is a compressed representation of the input sequence, capturing all the information needed to generate the output sequence.

Decoder RNN → The decoder is another RNN that generates the output sequence $y = (y^{(1)}, y^{(2)} \dots, y^{(n_y)})$, where n_y is the length of the output sequence.

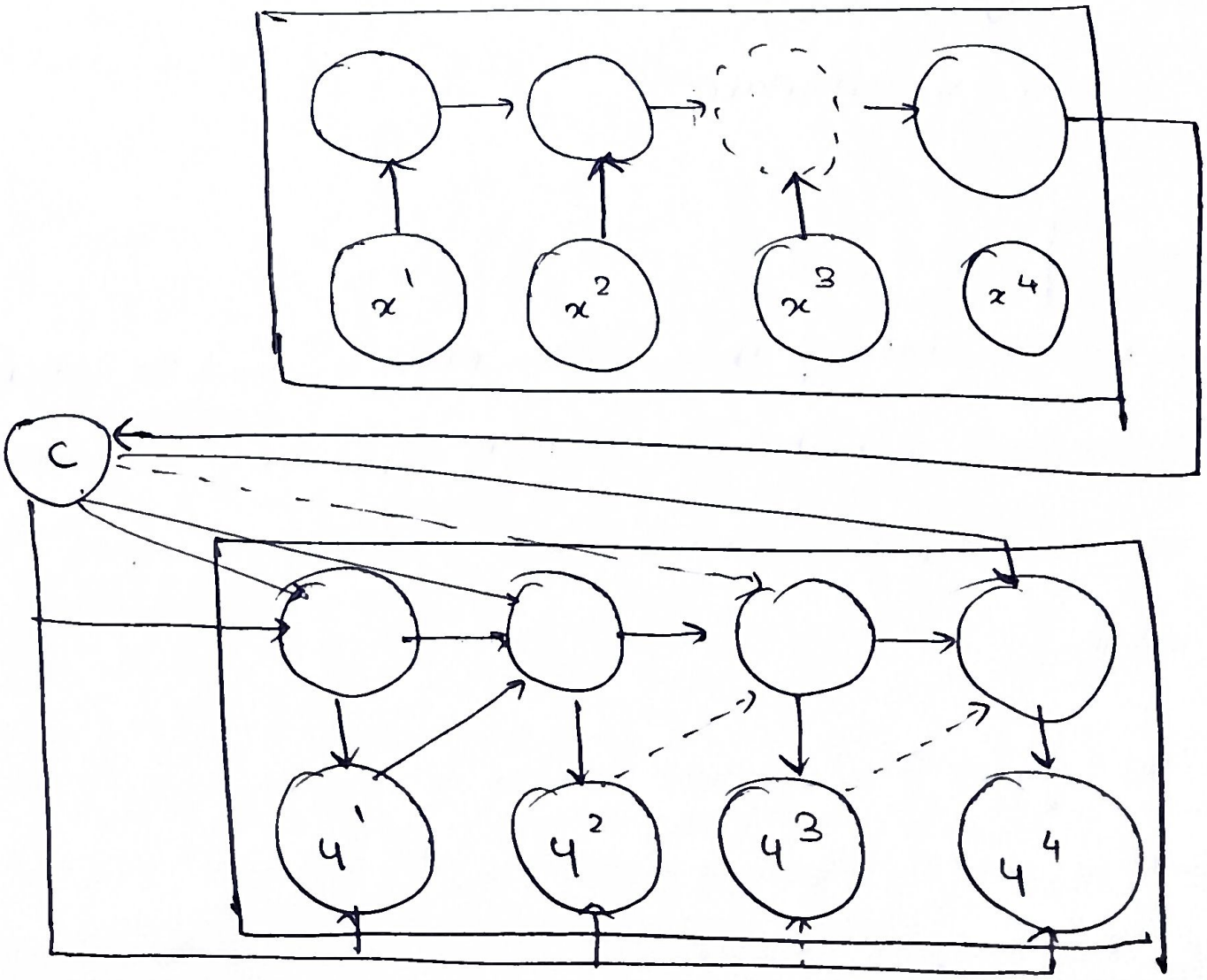
→ The decoder is initialized with the context vector C and generates one token at a time, conditioning on its previous inputs.

***

→ The key feature of seq-to-seq architecture is that the input and output sequences can have different lengths, unlike earlier architectures where the lengths were constrained to be the same.

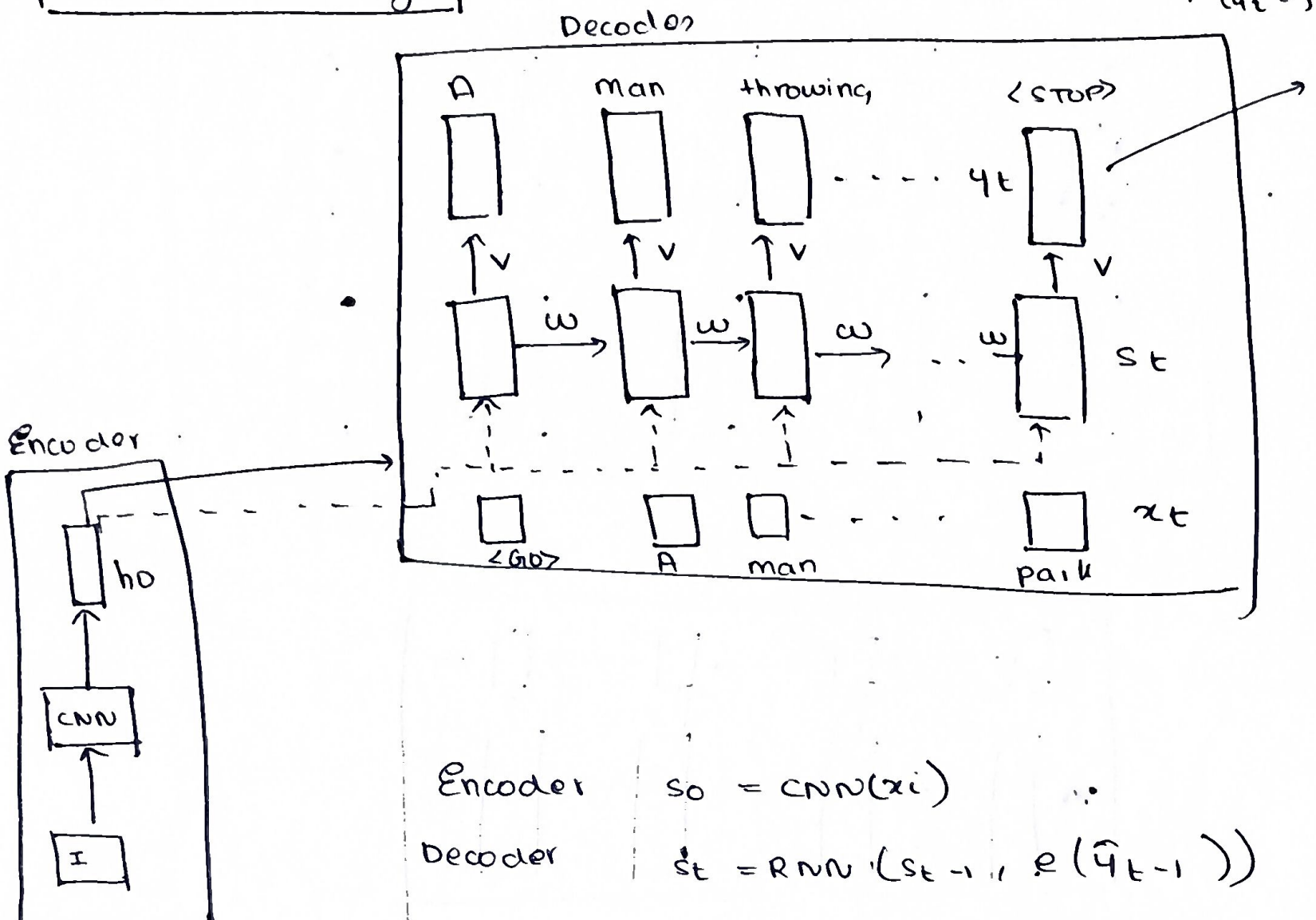
→ The RNNs are trained jointly to maximize the probability of the output sequence given the input sequence. The training objective is:

$$\max_{\theta} E_{(x,y)} [\log P_{\theta}(y|x)]$$



* Seq to Seq Architectures Applications

A. Image Captioning



$P(y_t = j | y_1^{t-1}, I)$

Encoder $s_0 = CNN(x_i)$

Decoder $s_t = RNN(s_{t-1}, e(\hat{y}_{t-1}))$

$$P(y_t | y_1^{t-1}, I) = \text{softmax}(V s_t + b)$$

Parameters = U_{dec}, V, W_{dec}, W_{conv}, b

Loss :

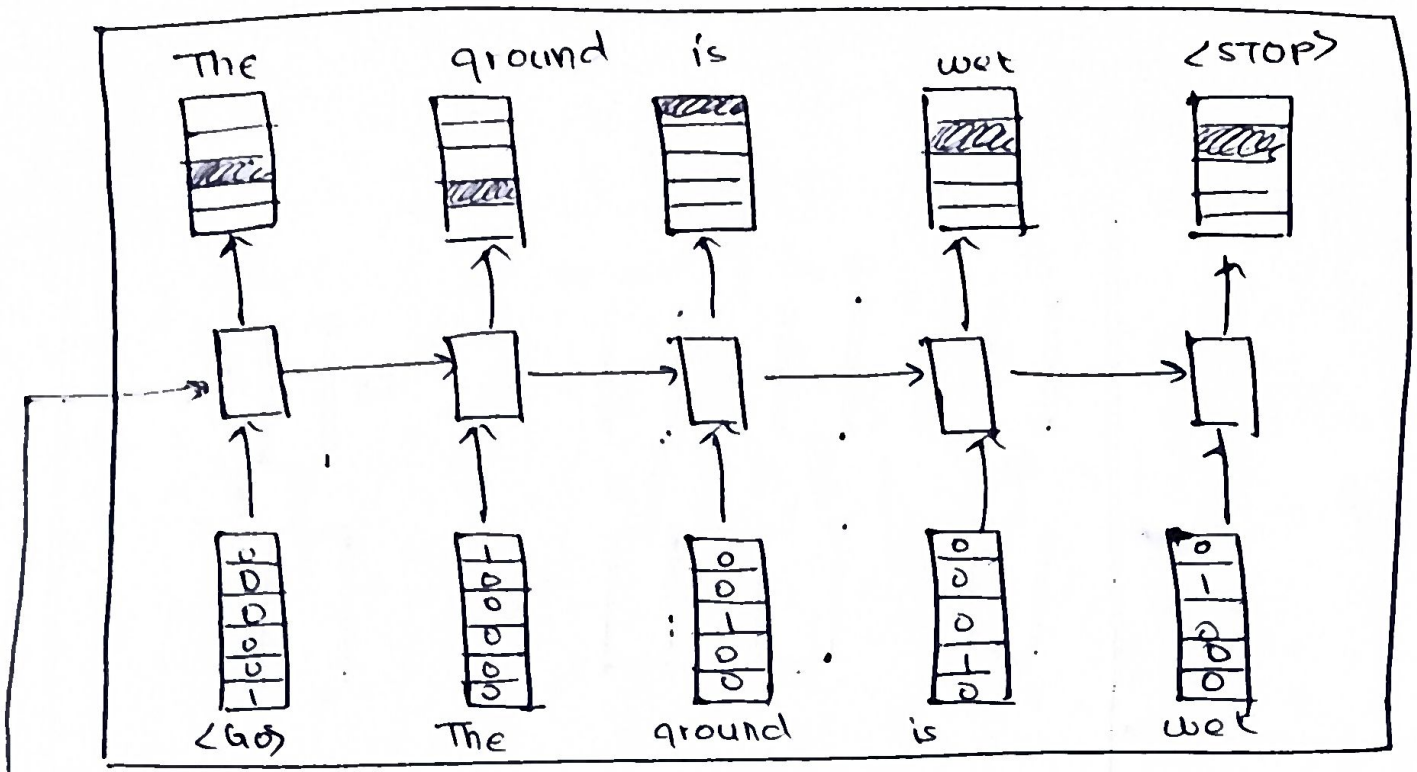
$$L(\theta) = \sum_{i=1}^{T} \ell_t(\theta) = -\sum_{t=1}^{T} \log P(y_t = \ell_t | y_1^{t-1}, I)$$

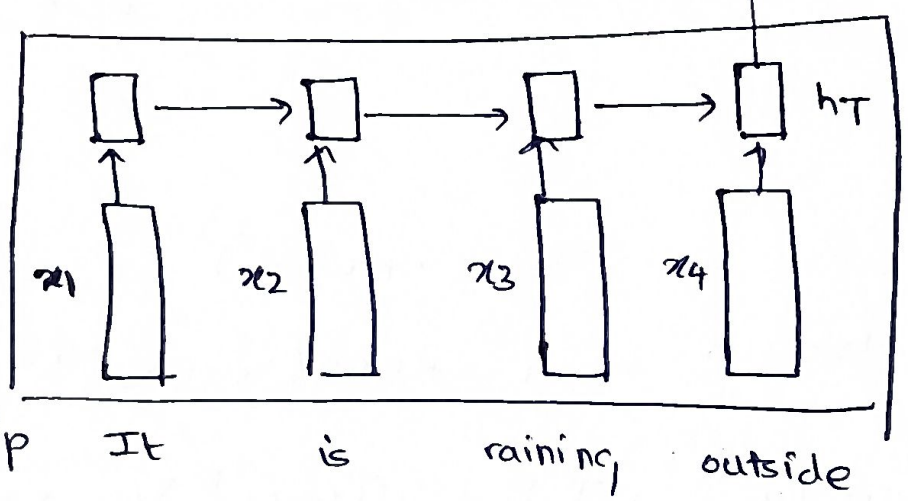
Algorithm : Gradient descent with backpropagation

B. Textual Entailment

→ Infer something from a piece of text

Peter snores ⇒ A man is sleeping.





C. Machine Translation

D. Machine Transliteration

Use the same architecture

Encoder :

$$h_t = RNN(h_{t-1}, x_{it})$$

Decoder :

$$s_0 = h_T$$

$$s_t = RNN(s_{t-1}, e(\hat{y}_{t-1}))$$

$$P(y_t | y_1^{t-1}, x) = \text{softmax}(V s_t + b)$$

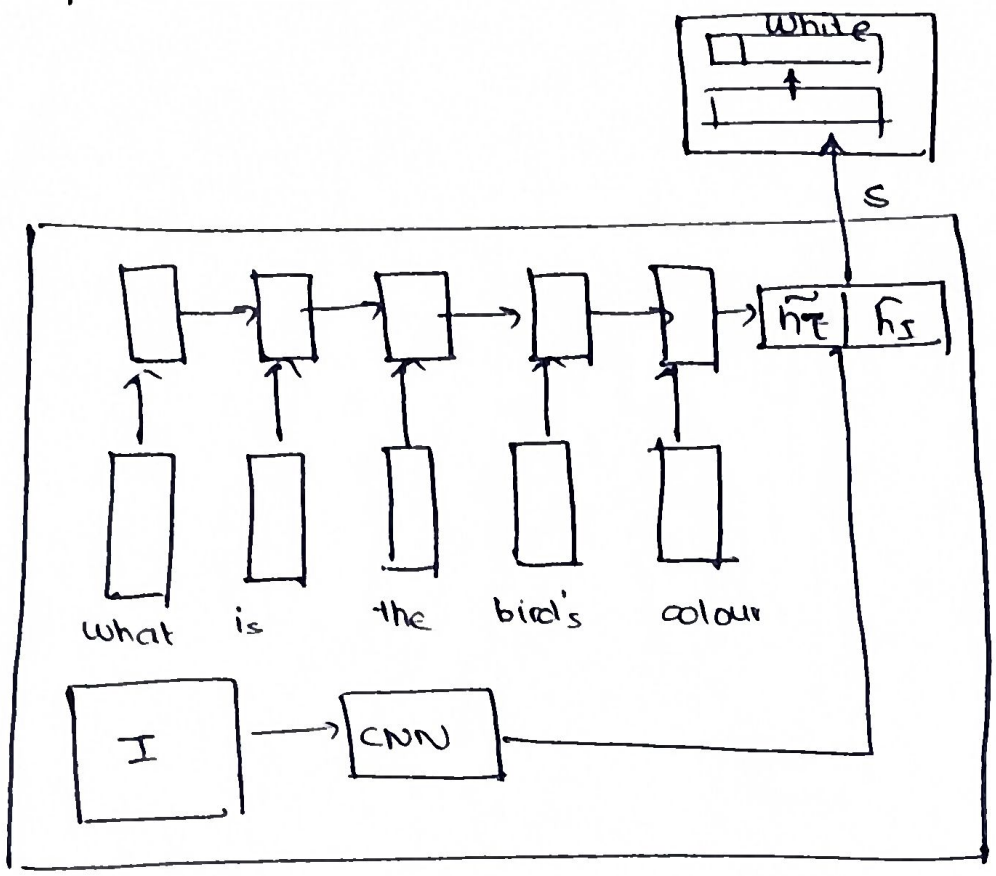
Parameters : $U_{dec}, V, W_{dec}, U_{enc}, W_{enc}, b$

Loss :

$$\mathcal{L}(\theta) = \sum_{i=1}^{T} L_t(\theta) = -\sum_{t=1}^{T} \log P(y_t = l_t | y_1^{t-1}, x)$$

Algorithm : Gradient descent with backpropagation

E. **Visual Question and Answering**



Data $\{x_i = \{I, q\}, y_i\}_{i=1}^{N}$

y_i = Answer

Encoder:

$$\hat{h}_I = CNN(I)$$

$$\tilde{h}_t = RNN(\tilde{h}_{t-1}, q_{it})$$

$$s = [\tilde{h}_T ; \hat{h}_I]$$

Decoder:

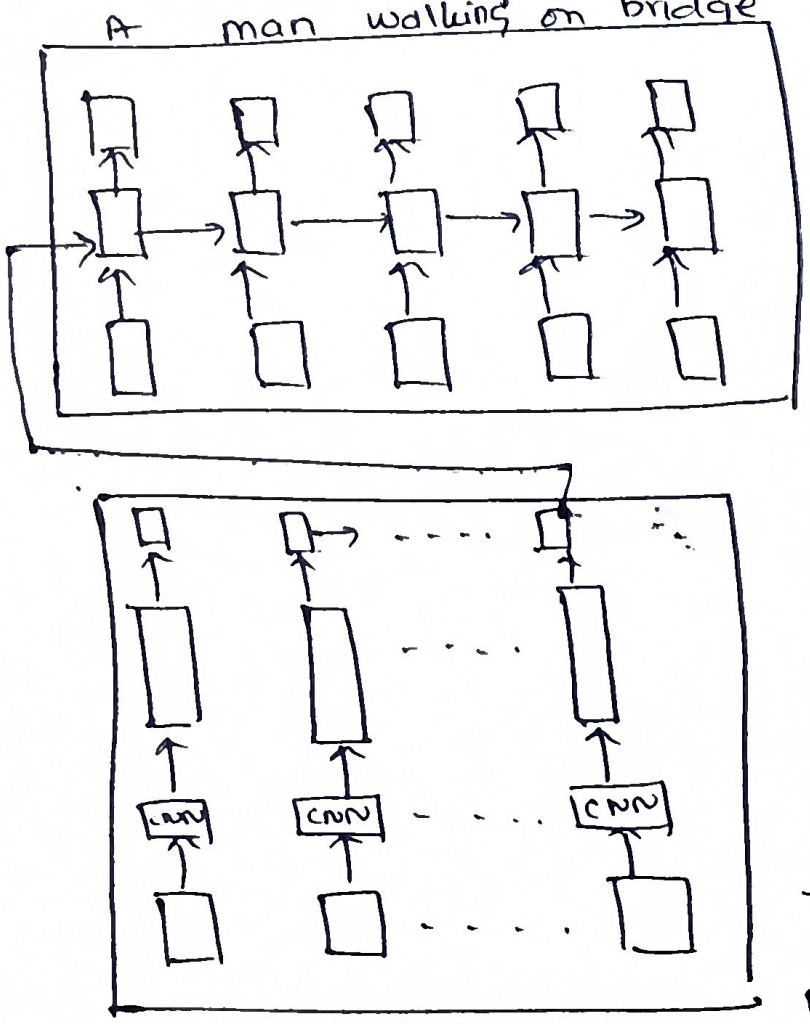
$$P(y \mid q, I) = softmax(V s + b)$$

Parameters: $V, b, U_q, W_q, W_{conv}, b$

Loss: $\mathcal{L}(\theta) = -\log P(y = l \mid I, q)$

Algorithm: Gradient descent w/ backpropagation

F. **Video Captioning**



Data: $\{x_i = \text{video}, y_i = \text{desc}_i\}_{i=1}^{N}$

Encoder:

$$h_t = RNN(h_{t-1}, CNN(x_{it}))$$

Decoder:

$$s_0 = h_T$$

$$s_t = RNN(s_{t-1}, e(\hat{y}_{t-1}))$$

$$P(y_t \mid y_1^{t-1}, x) = softmax(V s_t + b)$$

Parameters = $U_{dec}, W_{dec}, V, b, W_{conv}, U_{enc}, W_{enc}, b$

$$loss = -\sum_{t=1}^{T} \log P(y_t = l_t \mid y_1^{t-1}, x)$$

Algo: Grad descent w/ backprop