

## DL – Unit 4 (Recurrent Neural Network) – END-SEM PYQ Answers MAY-JUNE 2023

**Q3a** What is RNN? What is the need of RNN? Explain the working of RNN. [6 Marks]

### RNN — Recurrent Neural Network

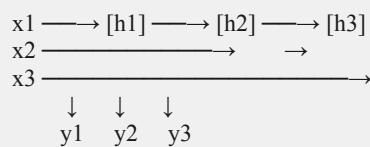
A Recurrent Neural Network (RNN) is a class of neural network specifically designed to process sequential or time-series data. Unlike a standard feedforward network where information flows only forward (input → hidden → output), an RNN has recurrent connections that allow information from previous time steps to feed back into the current computation. This gives the network a form of 'memory'.

#### Need for RNN

- **Sequential Data:** Language, speech, music, stock prices, and video all have an inherent order where the meaning of an element depends on what came before.
- **Variable-length Input/Output:** Standard networks require fixed-size inputs and outputs. RNNs handle sequences of arbitrary length.
- **Shared Parameters Across Time:** A single set of weights is applied at every time step, enabling efficient learning and generalization across different positions in the sequence.
- **Context Preservation:** The hidden state accumulates contextual information from all past inputs, allowing the network to 'remember' relevant history.

#### Architecture and Working

Unrolled RNN (across time steps  $t=1,2,3$ ):



At each time step  $t$ :

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

$$y_t = W_{hy} \cdot h_t + b_y$$

$h_t$  = hidden state (memory) at time  $t$

$x_t$  = input at time  $t$

$W_{hh}$  = recurrent weight matrix (hidden-to-hidden)

$W_{xh}$  = input weight matrix  $W_{hy}$  = output weight matrix

#### Training: Backpropagation Through Time (BPTT)

- The loss is computed at each time step and the gradients are back-propagated through the unrolled network.
- Because the same weights are reused, gradients are multiplied repeatedly — leading to the vanishing gradient problem for long sequences (gradients shrink to near-zero) or exploding gradients (gradients grow uncontrollably).
- Gradient clipping addresses exploding gradients; LSTM/GRU addresses vanishing gradients.

*Note: The vanishing gradient problem is the primary weakness of vanilla RNNs — they struggle to capture dependencies spanning more than ~10-20 time steps. LSTM was designed specifically to overcome this limitation.*

**Q3b** How LSTM and Bidirectional LSTM works.

[6 Marks]

### LSTM — Long Short-Term Memory

LSTM (Hochreiter & Schmidhuber, 1997) is an advanced RNN cell that solves the vanishing gradient problem by maintaining a separate cell state ( $C_t$ ) that acts as a 'conveyor belt' for information, controlled by three learned gating mechanisms.

#### LSTM Architecture — Gates

LSTM Cell at time step  $t$ :

Forget Gate:  $f_t = \sigma(W_f \cdot [h_{(t-1)}, x_t] + b_f)$   
 Input Gate:  $i_t = \sigma(W_i \cdot [h_{(t-1)}, x_t] + b_i)$   
 Candidate:  $\tilde{C}_t = \tanh(W_C \cdot [h_{(t-1)}, x_t] + b_C)$   
 Cell State:  $C_t = f_t \odot C_{(t-1)} + i_t \odot \tilde{C}_t$   
 Output Gate:  $o_t = \sigma(W_o \cdot [h_{(t-1)}, x_t] + b_o)$   
 Hidden State:  $h_t = o_t \odot \tanh(C_t)$

$\sigma$  = sigmoid,  $\odot$  = element-wise

multiplication

#### Gate Roles

- **Forget Gate ( $f_t$ ):** Decides what fraction of the previous cell state to erase. If  $f_t \approx 0$ , the past memory is forgotten; if  $f_t \approx 1$ , it is fully retained.
- **Input Gate ( $i_t$ ):** Decides which new information from the current input should be written into the cell state.
- **Candidate Cell ( $\tilde{C}_t$ ):** A proposed new memory computed from the current input and previous hidden state using  $\tanh$ .
- **Cell State Update:** The actual long-term memory  $C_t$  is a blend of the old memory (filtered by forget gate) and new information (filtered by input gate).
- **Output Gate ( $o_t$ ):** Decides which parts of the cell state to expose as the hidden state  $h_t$  (the short-term output/memory).

#### Bidirectional LSTM

A Bidirectional LSTM (BiLSTM) runs two separate LSTM layers over the same sequence — one in the forward direction (left to right) and one in the backward direction (right to left). The hidden states from both directions are concatenated at each time step, giving the model access to both past and future context.

Forward LSTM:  $x_1 \rightarrow [h1 \rightarrow] \rightarrow [h2 \rightarrow] \rightarrow [h3 \rightarrow]$   
 Backward LSTM:  $x_3 \rightarrow [h3 \leftarrow] \rightarrow [h2 \leftarrow] \rightarrow [h1 \leftarrow]$   
 Output at  $t=2$ :  $[h2 \rightarrow | h2 \leftarrow]$  (concatenation)

- **Use cases:** Named Entity Recognition, machine translation, sentiment analysis — anywhere where future context is available at training time.

*Note: GRU (Gated Recurrent Unit) is a simplified variant of LSTM with only two gates (reset and update) and no separate cell state. It has fewer parameters and often matches LSTM performance on smaller datasets.*

**Q3c** Explain Unfolding computational graphs with example.

[5 Marks]

## Unfolding Computational Graphs in RNNs

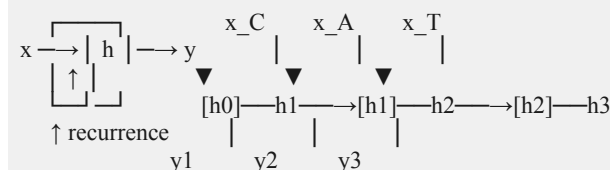
An RNN can be viewed as a compact recursive graph where the same function is applied at each time step sharing the same parameters. 'Unfolding' (or 'unrolling') means expanding this recursive graph across time into a standard feedforward-like DAG (Directed Acyclic Graph), making the flow of computation and gradients explicit and tractable.

### Why Unfold?

- **Gradient computation:** BPTT requires the unrolled graph to compute gradients at each time step.
- **Visualisation:** Unrolling clarifies how the network processes a sequence step-by-step.
- **Parallelism:** Modern frameworks (PyTorch, TensorFlow) represent RNNs as unrolled graphs for automatic differentiation.

### Example — Processing 'CAT' (3 characters)

Compact (Folded):      Unfolded (T=3 steps):



Same  $W, U, b$  used at every step — parameter sharing.

At each unfolded step, the same weight matrices  $W_{xh}$  and  $W_{hh}$  are applied, which is what 'parameter sharing across time' means. This makes RNNs capable of generalizing to sequences of any length with a fixed number of parameters.

*Note: Truncated BPTT is used in practice: instead of unrolling the full sequence (which can be thousands of steps), gradients are computed over a fixed window (e.g., 20 steps) to keep computation manageable.*

**Q4a** What are types of RNN (Recurrent Neural Network)? How to train RNN. [6 Marks]

### Types of RNN

- **One-to-One:** Standard neural network (no recurrence). Input: single vector  $\rightarrow$  Output: single vector. Example: image classification.
- **One-to-Many:** Single input produces a sequence of outputs. Example: image captioning (one image  $\rightarrow$  sequence of words).
- **Many-to-One:** A sequence of inputs produces a single output. Example: sentiment analysis (sequence of words  $\rightarrow$  positive/negative).
- **Many-to-Many (Synced):** Input and output sequences are aligned step-by-step. Example: video classification (label each frame), POS tagging.
- **Many-to-Many (Encoder-Decoder):** Input sequence is encoded into a context vector, which is decoded into an output sequence (can be different length). Example: machine translation, text summarisation.

### How to Train an RNN — BPTT

Step 1: Forward Pass  
For  $t = 1$  to  $T$ :

```

h_t = tanh(W_hh · h_(t-1) + W_xh · x_t + b_h)
y_t = softmax(W_hy · h_t + b_y)
L_t = CrossEntropy(y_t, target_t)
Total Loss L = Σ L_t

```

Step 2: Backward Pass (BPTT)

Unroll graph; compute  $\partial L / \partial W_{hh}$ ,  $\partial L / \partial W_{xh}$ ,  $\partial L / \partial W_{hy}$   
 Gradients flow backwards through time:  $t=T \dots t=1$

Step 3: Parameter Update  $W \leftarrow W - \eta \cdot \partial L / \partial W$  (SGD, Adam, RMSProp, etc.)

- **Vanishing Gradients:** In long sequences, the repeated multiplication of  $W_{hh}$  during BPTT can make gradients vanish. Solution: LSTM/GRU, gradient clipping.
- **Exploding Gradients:** Gradients can grow exponentially. Solution: Gradient clipping (cap gradient norm to a threshold, e.g., 5).

*Note: Modern training of RNNs uses Adam optimizer with gradient clipping. Teacher forcing (using the true target at each step rather than the model's own prediction) speeds up training for sequence-to-sequence models.*

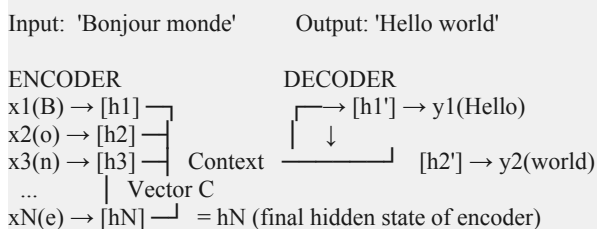
**Q4b** Explain Encoder-Decoder Sequence to Sequence architecture with its application.

**[6 Marks]**

### Encoder-Decoder (Seq2Seq) Architecture

The Encoder-Decoder architecture (Sutskever et al., 2014) is designed to handle Many-to-Many tasks where the input and output sequences may be of different lengths — which is impossible with a single RNN. It consists of two distinct RNN components.

#### Architecture



Encoder reads the full input and compresses it into C. Decoder generates output one token at a time, conditioned on C.

#### Components

- **Encoder:** Processes the input sequence  $x_1 \dots x_N$  one token at a time. The final hidden state  $h_N$  (or sometimes all hidden states) forms the context vector C, a fixed-length summary of the entire input.
- **Context Vector:** Bridges encoder and decoder. It is the 'bottleneck' — all source information must pass through it. This limits performance for very long sequences (addressed by Attention Mechanism).
- **Decoder:** An RNN initialised with the context vector C. At each step it generates one output token, using both its current hidden state and the previously generated token as input.

#### Applications

- Machine Translation (English  $\rightarrow$  French, Hindi, etc.)
- Text Summarisation (long article  $\rightarrow$  short summary)

- Question Answering
- Chatbots and dialogue systems
- Image Captioning (CNN encoder + RNN decoder)
- Speech Recognition (audio features → text)

*Note: The key limitation of the basic Encoder-Decoder model is that all source information is compressed into a single fixed-size context vector. The Attention Mechanism (Bahdanau, 2015) solves this by allowing the decoder to 'look back' at all encoder hidden states at each decoding step, weighted by relevance. Transformers (2017) generalize this into self-attention across all positions, eliminating RNNs entirely and enabling parallelization.*

#### Q4c Differentiate between Recurrent and Recursive Neural Network.

[5 Marks]

Property	Recurrent vs Recursive Neural Network
Structure	Recurrent: Linear chain — processes sequences left-to-right (or step-by-step). Recursive: Tree structure — processes hierarchical data (trees, graphs).
Data Type	Recurrent: Sequential (time-series, text sentences, speech). Recursive: Hierarchical (parse trees, sentiment of nested phrases, program ASTs).
Parameter Sharing	Recurrent: Same weights applied at each time step of the sequence. Recursive: Same weights applied at each node of the tree regardless of depth.
Memory	Recurrent: Hidden state summarises all past inputs in the sequence. Recursive: Each node's representation is a function of its children's representations.
Gradient Flow	Recurrent: Gradients flow back through time (BPTT) — prone to vanishing gradient for long sequences. Recursive: Gradients flow up through tree levels — less severe but still present.
Examples	Recurrent: LSTM, GRU, vanilla RNN for NLP tasks. Recursive: Recursive Neural Tensor Network (RNTN) for sentiment analysis of parse trees.

## NOV-DEC 2023

#### Q3a Draw CNN architecture and explain its working.

[6 Marks]

**[REPEATED]** CNN architecture is covered comprehensively in Unit 3 (Q1b, May-June 2023). Refer to that document for the full layered diagram and explanation.

#### Q3b Explain the types of Recurrent Neural Network.

[6 Marks]

**[REPEATED]** Types of RNN are covered in Q4a of May-June 2023 (Unit 4). The five types — One-to-One, One-to-Many, Many-to-One, and both Many-to-Many variants — are detailed there with examples.

#### Q3c Justify RNN is better suited to treat sequential data than a feedforward neural network.

[5 Marks]

### RNN vs Feedforward Networks for Sequential Data

- **Fixed Input Size Limitation:** A standard feedforward network requires fixed-size input. To handle a sentence of variable length, you would need to pad or truncate, losing information.

RNN handles any sequence length naturally.

- **No Temporal Context:** A feedforward network treats each input independently with no memory of previous inputs. For sequential data, the meaning of a word or sensor reading depends on its history — which a feedforward network cannot capture.
- **Parameter Efficiency:** If a feedforward network were to handle a sequence of length  $T$  by having  $T$  sets of weights, it would have  $T \times$  (normal parameter count) parameters — impractical and unscalable. RNN shares one set of weights across all time steps.
- **Positional Sensitivity:** RNN is inherently sensitive to order because it processes elements sequentially. Shuffling the sequence produces a different hidden state sequence. A feedforward network with a flat input vector ignores order entirely.
- **Captures Long-Range Dependencies:** With LSTM/GRU cells, RNNs can model dependencies spanning many time steps (e.g., subject-verb agreement separated by a long clause in a sentence).

*Note: While RNNs are theoretically well-suited to sequential data, Transformer models (based purely on self-attention) now outperform RNNs for most NLP tasks because they process all positions in parallel and have a direct path between any two positions — avoiding the sequential bottleneck.*

**Q4a** Explain Recurrent Neural Network with its architecture.

[6 Marks]

**[REPEATED]** RNN architecture and working is covered in detail in Q3a of May-June 2023 (Unit 4). Refer to that section for mathematical equations, the unrolled diagram, and training via BPTT.

**Q4b** Draw and explain architecture for Long Short-Term Memory (LSTM).

[6 Marks]

**[REPEATED]** LSTM architecture is covered in Q3b of May-June 2023 (Unit 4). Refer to that section for the full gate equations (forget, input, output), cell state update formula, and Bidirectional LSTM.

**Q4c** Explain how the memory cell in the LSTM is implemented computationally? [5 Marks]

### LSTM Memory Cell — Computational Implementation

The key innovation of LSTM is the cell state  $C_t$  — a 'conveyor belt' that runs straight down the time sequence with only minor linear interactions, allowing gradients to flow through many time steps without vanishing.

#### Step-by-Step Computation

Inputs at time  $t$ :  $x_t$  (current input),  $h_{(t-1)}$  (previous hidden state)  
Concatenate:  $z = [h_{(t-1)} ; x_t]$  (dim = hidden\_size + input\_size)

1. Forget Gate:  $f_t = \sigma(W_f \cdot z + b_f)$  // what to erase from  $C$
2. Input Gate:  $i_t = \sigma(W_i \cdot z + b_i)$  // what to write
3. Candidate:  $\tilde{C}_t = \tanh(W_C \cdot z + b_C)$  // proposed new memory
4. Cell Update:  $C_t = f_t \odot C_{(t-1)} + i_t \odot \tilde{C}_t$
5. Output Gate:  $o_t = \sigma(W_o \cdot z + b_o)$  // what to expose
6. Hidden State:  $h_t = o_t \odot \tanh(C_t)$

$\odot$  = element-wise (Hadamard) product

$\sigma$  = sigmoid (outputs in  $[0,1]$  — act as soft gates)  $\tanh$  = squashes values to  $[-1,1]$

- **Gradient Flow Through Cell State:** The cell state update  $C_t = f_t \odot C_{(t-1)} + i_t \odot \tilde{C}_t$  is an additive (not multiplicative chain) operation through time. Backpropagating through addition

does not multiply gradients, preventing vanishing. The constant error carousel (CEC) is the key idea: the gradient of  $C_t$  with respect to  $C_{(t-1)}$  is simply  $f_t \approx 1$  (if the forget gate is open), meaning the cell state gradient flows unchanged.

*Note: In code (PyTorch): `torch.nn.LSTM(input_size, hidden_size, num_layers)`. The cell returns (output,  $(h_n, c_n)$ ) — both the hidden state  $h$  and the cell state  $c$  are maintained. For GRU: `torch.nn.GRU` returns only (output,  $h_n$ ) since GRU has no separate cell state.*

## MAY-JUNE 2024

**Q3a** Explain RNN with its types.

[6 Marks]

**[REPEATED]** RNN and its types are covered in Q3a and Q4a of May-June 2023 (Unit 4). Refer to those sections for the full explanation including the mathematical equations, unrolled diagram, and all five types with examples.

**Q3b** Explain in brief Encoder Decoder architecture.

[6 Marks]

**[REPEATED]** Encoder-Decoder (Seq2Seq) architecture is covered in Q4b of May-June 2023 (Unit 4). Refer to that section for the full architecture diagram, component descriptions, and applications.

**Q3c** Explain Different types of Deep Learning.

[5 Marks]

### Types of Deep Learning

- **Supervised Deep Learning:** Trained on labeled input-output pairs. The network learns a mapping from inputs to outputs by minimising a loss function. Examples: CNNs for image classification, RNNs for sequence prediction, Transformers for NLP.
- **Unsupervised Deep Learning:** Trained on unlabeled data to discover hidden structure. Examples: Autoencoders for dimensionality reduction/feature learning, GANs for generative modelling, Deep Belief Networks (DBNs).
- **Semi-Supervised Deep Learning:** Uses a small amount of labeled data combined with a large amount of unlabeled data. The network leverages the unlabeled data to learn better representations.
- **Self-Supervised Deep Learning:** A form of unsupervised learning where the network generates its own labels from the data structure (e.g., predicting the next word, predicting masked image patches). Examples: BERT, GPT, SimCLR, DINO.
- **Reinforcement Deep Learning:** An agent learns by interacting with an environment to maximise cumulative reward. Deep neural networks are used as function approximators. Examples: Deep Q-Networks (DQN), A3C, PPO, AlphaGo.
- **Transfer Learning / Fine-tuning:** A model pre-trained on a large dataset (e.g., ImageNet, Wikipedia) is fine-tuned on a smaller task-specific dataset. Drastically reduces data and compute requirements.

*Note: The dominant paradigm in modern deep learning is self-supervised pre-training followed by supervised fine-tuning. This has led to foundation models (GPT, BERT, SAM, CLIP) that achieve state-of-the-art across dozens of tasks with minimal task-specific data.*

**Q4a** WSN on Performance Matrices.

[6 Marks]



## Performance Metrics / Matrices for Deep Learning Models

Performance metrics quantify how well a model generalises to unseen data. The choice of metric depends on the task (classification, regression, object detection, generation, etc.).

### Classification Metrics

- **Accuracy:** Proportion of correct predictions:  $(TP + TN) / (TP + TN + FP + FN)$ . Simple but misleading for imbalanced datasets.
- **Precision:**  $TP / (TP + FP)$ . Of all positive predictions, how many were correct? Important when false positives are costly (e.g., spam filtering).
- **Recall (Sensitivity):**  $TP / (TP + FN)$ . Of all actual positives, how many were found? Critical when false negatives are costly (e.g., cancer detection).
- **F1-Score:** Harmonic mean of Precision and Recall:  $2 \times P \times R / (P + R)$ . Balances both; used when classes are imbalanced.
- **Confusion Matrix:** A table showing TP, TN, FP, FN for all classes. Provides a complete picture of classification performance.
- **ROC-AUC:** Area Under the ROC curve. Measures the trade-off between true positive rate and false positive rate across all thresholds. AUC = 1 is perfect; AUC = 0.5 is random.

### Regression Metrics

- **MSE (Mean Squared Error):** Average of squared differences between predictions and targets. Sensitive to outliers.
- **MAE (Mean Absolute Error):** Average of absolute differences. More robust to outliers.
- **R<sup>2</sup> (Coefficient of Determination):** Proportion of variance explained by the model.  $R^2 = 1$  is perfect fit.

### Deep Learning Specific Metrics

- **Perplexity:** Measure of how well a language model predicts a sequence. Lower is better.
- **BLEU Score:** Measures overlap between generated text and reference text. Used for machine translation.
- **mAP (mean Average Precision):** Standard metric for object detection tasks.
- **IoU (Intersection over Union):** Measures overlap between predicted and ground-truth bounding boxes/segmentation masks.

*Note: WSN here likely refers to 'Write Short Note' (a common instruction format in SPPU exams). Key insight: always choose metrics appropriate to the task and data distribution — accuracy alone is insufficient for imbalanced classification problems.*

**Q4b** Compare implicit and explicit memory.

**[6 Marks]**

### Implicit vs Explicit Memory in Neural Networks

Property	Implicit vs Explicit Memory
Definition	Implicit: Memory encoded in network weights through training (learned associations). Explicit: Directly readable/writable memory storage, separate from weights (e.g., external memory matrix).
Location	Implicit: Distributed across all weight matrices $W$ of the network. Explicit: Stored in a dedicated memory bank or external data structure.



Access	Implicit: Accessed indirectly through forward computation — you cannot directly read a specific 'fact' from weights. Explicit: Addressed directly by a query vector (e.g., Neural Turing Machine uses content/location-based addressing).
Capacity	Implicit: Limited by network size; interference between stored facts occurs for large knowledge bases. Explicit: Can store arbitrary amounts; scales independently of network size.
Persistence	Implicit: Persistent across all inputs — knowledge is baked into weights permanently until retrained. Explicit: Can be dynamically updated per input sequence at inference time.
Flexibility	Implicit: Inflexible — updating one fact may distort others (catastrophic forgetting). Explicit: Flexible — individual memory slots can be updated without affecting others.
Examples	Implicit: Standard RNN hidden state, CNN feature representations, Transformer attention weights. Explicit: Neural Turing Machine (NTM), Differentiable Neural Computer (DNC), Memory Networks.

*Note: The RNN hidden state ( $h_t$ ) is often described as implicit short-term memory — it encodes contextual information about the current sequence but is not directly addressable. The LSTM cell state is a refined form of implicit memory with better gradient flow. Explicit memory architectures (NTM, DNC) are still an active research area, aiming to give neural networks more computer-like memory access.*

**Q4c** What are default baseline models? Explain in brief.

**[5 Marks]**

### Default Baseline Models in Deep Learning

A baseline model is the simplest, most naive model against which all other (more complex) models are compared. It establishes the minimum performance level that any 'intelligent' model must beat to justify its complexity.

#### Common Default Baselines

- **Random Classifier:** Predicts a random class — usually proportional to class frequency. For a balanced 10-class problem, baseline accuracy = 10%.
- **Majority Class Classifier:** Always predicts the most frequent class. Simple but effective on highly imbalanced datasets. For a 90% positive class, this gives 90% accuracy — misleadingly high.
- **Mean/Median Predictor (Regression):** Always predicts the mean (or median) of the training targets. Baseline MSE = variance of the target distribution.
- **Linear/Logistic Regression:** A simple linear model. If a deep learning model cannot beat logistic regression, the problem or data needs revisiting.
- **Previous Value Predictor (Time-series):** Predicts the next value as the last observed value (random walk). Surprisingly difficult to beat for financial data.
- **Pre-trained Model Zero-shot:** Use a foundation model (e.g., GPT, CLIP) directly without fine-tuning as a baseline for new tasks.

#### Why Baselines Matter

- They reveal whether your complex model is actually learning something useful.
- They help set expectations and justify the cost of training large models.
- They expose data quality issues — if a simple baseline beats a complex model, the features or

labels are likely problematic.

*Note: A well-known deep learning engineering principle (Karpathy's 'recipe'): always start with the simplest possible model, overfit one batch, then gradually add complexity. This disciplined baselining prevents wasting weeks on debugging a complex model when the data pipeline itself is broken.*

## MAY-JUNE 2025

**Q3a** Explain recursive neural network.

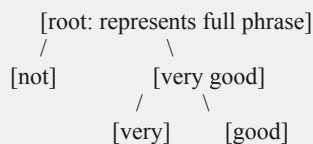
**[6 Marks]**

### Recursive Neural Network

A Recursive Neural Network (RecNN) is a generalisation of Recurrent Neural Networks from linear sequences to tree-structured or graph-structured data. Instead of applying the same function along a temporal chain (as RNNs do), a RecNN applies the same parameterised function recursively at each node of a tree, computing each node's representation as a function of its children's representations — bottom-up.

### Architecture Diagram

Example: Sentiment of the phrase 'not very good'



At each internal node:  $h = \tanh(W \cdot [h_{\text{left}} ; h_{\text{right}}] + b)$

Same  $W, b$  shared across ALL nodes regardless of depth. Root vector  $h_{\text{root}} \rightarrow \text{classifier} \rightarrow \text{sentiment label}$ .

- **Leaf nodes:** Each leaf (word/terminal) is initialised with a word embedding (e.g., from GloVe or Word2Vec).
- **Internal nodes:**  $h_{\text{node}} = \tanh(W \cdot [h_{\text{left}} ; h_{\text{right}}] + b)$ . The same weight matrix  $W$  is reused at every node in the tree regardless of position or level — this is what 'recursive' means.
- **Root node:** Captures the meaning of the full phrase and is passed to a softmax classifier.

### Key Differences from Recurrent Neural Network

Property	RecNN vs RNN
Structure	RecNN: Tree/graph — hierarchical, branching. RNN: Linear chain — strictly sequential.
Input type	RecNN: Parse trees, syntax trees, nested expressions. RNN: Ordered sequences — text, time-series.
Compositionality	RecNN: Explicitly models that meaning of a phrase = $f(\text{meanings of sub-phrases})$ . RNN: Captures temporal context, not hierarchical composition.
Example use	RecNN: Sentiment of 'not bad at all' via parse tree. RNN: Next-word prediction in a sentence.

- **Application — RNTN (Recursive Neural Tensor Network):** Stanford's Socher et al. showed that RecNNs can correctly analyse sentiment at every sub-phrase level, capturing negations like 'not very good' that flat bag-of-words models miss entirely.

*Note: Recursive Neural Networks have been largely superseded by Transformer-based models (BERT, GPT) that implicitly learn hierarchical structure through multi-head self-attention without requiring an explicit parse tree as input. However, RecNNs remain foundational for understanding compositionality in neural representations.*

**Q3b** Explain the LSTM in RNN.

[6 Marks]

**[REPEATED]** LSTM is covered in Q3b of May-June 2023 with full gate equations (forget, input, output), cell state update, ASCII architecture, and Bidirectional LSTM. See also ND2023 Q4b and Q4c for computational implementation of the memory cell.

**Q3c** Explain in brief about working of RNN.

[5 Marks]

**[REPEATED]** RNN working is in Q3a of May-June 2023: recurrence equation  $h_t = \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t + b)$ , unrolled diagram, need for RNNs, and BPTT training.

**Q4a** Difference between CNN vs RNN.

[6 Marks]

### CNN vs RNN — Detailed Comparison

Property	CNN vs RNN
Primary data	CNN: Grid-structured spatial data — images (2D), audio spectrograms. RNN: Sequential/temporal data — text, speech, time-series.
Core operation	CNN: Learnable filters slide over input performing discrete cross-correlation. RNN: Hidden state update $h_t = f(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t)$ — recurrence.
Memory	CNN: No cross-sample memory — all spatial positions processed simultaneously within a fixed receptive field. RNN: Explicit sequential memory — $h_t$ encodes all past inputs $x_1 \dots x_t$ .
Parameter sharing	CNN: Spatial — same filter applied to every location in the input map. RNN: Temporal — same weight matrices applied at every time step.
Input length	CNN: Fixed spatial dimensions (images resized). RNN: Variable-length sequences handled naturally.
Parallelism	CNN: Fully parallelisable across all spatial positions. RNN: Inherently sequential — step $t$ requires $h_{t-1}$ .
Gradient stability	CNN: Stable — fixed depth. RNN: Vanishing/exploding gradients for long sequences (LSTM mitigates this).
Typical tasks	CNN: Classification, detection, segmentation. RNN/LSTM: Translation, language modelling, speech recognition.
Hybrid use	CNNs extract spatial features from frames; LSTMs process the resulting sequence — used in video captioning and activity recognition.

*Note: The line between CNN and RNN has blurred: Temporal Convolutional Networks use dilated 1-D convolutions for sequence modelling without recurrence; Transformers use self-attention, replacing both. In practice, choose CNN for spatial structure, RNN for temporal dependencies, and Transformers when you need both and have the compute budget.*

**Q4b** What are the challenges of long term dependencies?

[6 Marks]

### Challenges of Long-Term Dependencies

- **1. Vanishing Gradients:** During BPTT, the gradient at step  $t-k$  involves multiplying  $W_{hh}$  by itself  $k$  times. If the spectral radius of  $W_{hh}$  is less than 1,  $(W_{hh})^k \rightarrow 0$  exponentially, making gradients at early time steps negligibly small. The network effectively cannot learn from inputs more than  $\sim 10-20$  steps ago.

$\partial L / \partial h_{t-k} = \partial L / \partial h_t \cdot (W_{hh})^k \cdot \dots$  If  $|\text{eigenvalues}(W_{hh})| < 1$ : shrinks to zero exponentially as  $k$  grows.

- **2. Exploding Gradients:** If spectral radius  $> 1$ , gradients grow exponentially — NaN values and training divergence. Fixed with gradient clipping: clip gradient norm to threshold  $T$  (e.g.,  $T = 5$ ).
- **3. Memory Decay:** The hidden state  $h_t$  must simultaneously encode all recent context. Recent information overwrites older information at each step — the RNN acts as lossy compression of history, making selective long-range retention unreliable.
- **4. Bridging Long Gaps:** Even with LSTM, the network must learn which past inputs matter for the current output — a difficult credit assignment problem over long horizons (e.g., subject-verb agreement across 50 words).
- **5. Sequential Bottleneck:** Information from  $t=1$  must be passed through  $T$  hidden state transformations to reach  $t=T$ . Each transformation loses some information, so even LSTM's cell state degrades over sequences of hundreds of steps.
- **6. Computational Depth During Training:** BPTT for a  $T$ -length sequence requires  $T$  sequential matrix multiplications — cannot be parallelised across time, slowing training significantly.
- **7. Hyperparameter Sensitivity:** The forget gate bias initialisation in LSTMs critically determines effective memory span. Poor initialisation (defaulting to forget) prevents the model from ever learning long-range dependencies.

### Solutions

- **LSTM/GRU:** Gated architectures with additive cell state updates — gradients flow unchanged via the constant error carousel.
- **Attention Mechanism:** Decoder directly accesses any encoder hidden state, bypassing the sequential bottleneck.
- **Transformers:** Self-attention gives  $O(1)$  path length between any two positions — the definitive solution to long-range dependencies.

*Note: The Transformer was specifically designed around this problem. By computing all pairwise attention scores simultaneously, the path length between positions  $i$  and  $j$  is always exactly 1, regardless of how far apart they are in the sequence.*

**Q4c** Explain Encoder-Decoder RNN model.

[5 Marks]

**[REPEATED]** Encoder-Decoder (Seq2Seq) is in Q4b of May-June 2023 with architecture diagram, encoder/context-vector/decoder explanations, teacher forcing, and applications (translation, summarisation, captioning, chatbots).

### NOV-DEC 2025

**Q3a** How is the computational graph of an RNN different from that of a feedforward neural network?

[6 Marks]

## Computational Graph: RNN vs Feedforward Neural Network

A computational graph represents a mathematical computation as a directed graph where nodes are operations and edges are data tensors. Comparing the graphs of feedforward networks and RNNs reveals why RNNs are more powerful for sequences but harder to train.

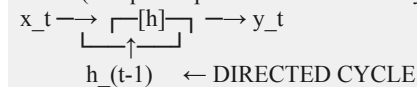
### Feedforward Network Graph

$x \rightarrow [\text{Linear } W1] \rightarrow [\text{ReLU}] \rightarrow [\text{Linear } W2] \rightarrow [\text{Softmax}] \rightarrow y$

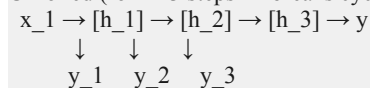
- Directed Acyclic Graph (DAG): no cycles.
- Fixed depth — always the same number of operations.
- No parameter sharing across layers ( $W1 \neq W2$ ).
- Gradient flows back through a fixed finite number of paths.
- One forward pass = one fixed-size input processed.

### RNN Computational Graph

Folded (compact representation — has a cycle):



Unrolled (for  $T=3$  steps — breaks cycle into a deep DAG):



- Parameter sharing: same  $W_{hh}$ ,  $W_{xh}$ ,  $W_{hy}$  at EVERY step.
- Depth =  $T$  (sequence length) — variable.
- Gradients must flow back  $T$  steps (BPTT).

### Comparison Table

Property	Feedforward vs RNN Graph
Cycle	Feedforward: DAG — no cycles. RNN (folded): has directed cycle; unrolled: deep DAG of depth $T$ .
Parameter sharing	Feedforward: each layer has independent weights. RNN: $W_{hh}$ , $W_{xh}$ shared identically across all $T$ time steps.
Depth	Feedforward: fixed by architecture. RNN: equals sequence length $T$ — varies at runtime.
Hidden state	Feedforward: none — inputs processed independently. RNN: $h_t$ persists and accumulates context across the sequence.
Gradient issues	Feedforward: stable, bounded depth. RNN: gradients multiplied $T$ times — vanishing/exploding for large $T$ .
Parallelism	Feedforward: fully parallel. RNN: inherently sequential — step $t$ requires $h_{(t-1)}$ .

*Note: The directed cycle in the folded RNN graph is what gives RNNs their sequential memory — but this same cycle is why training RNNs requires unrolling (BPTT) rather than simple backpropagation. PyTorch and TensorFlow handle this automatically by dynamically unrolling the graph for each input sequence during the forward pass.*

**Q3b** List the types of RNN and explain LSTM three gates.

**[6 Marks]**

**[REPEATED]** RNN types (one-to-one through many-to-many) are in Q4a of May-June 2023. LSTM three gates (forget, input, output) with full equations and diagram are in Q3b of May-June 2023 and

ND2023 Q4b.

**Q3c** What is Encoder-Decoder architecture, and how does it work in sequence-to-sequence learning? [6 Marks]

**[REPEATED]** *Encoder-Decoder (Seq2Seq) is in Q4b of May-June 2023 with full diagram, component explanations, context vector, and applications.*

**Q4a** What are limitations of Bidirectional RNNs, and how do they differ from standard RNNs? [6 Marks]

### Bidirectional RNNs vs Standard RNNs

A Bidirectional RNN (BiRNN) runs two separate RNN layers over the same sequence — one forward (left to right) and one backward (right to left). The hidden states from both directions are concatenated at each position, giving the model access to both past and future context simultaneously. While this improves accuracy on many offline tasks, it comes with important limitations.

#### Key Differences

Property	Standard RNN vs BiRNN
Context	Standard: past context only (causal). BiRNN: past AND future context at every position.
Architecture	Standard: one RNN chain. BiRNN: two independent RNN chains (forward + backward) whose outputs are concatenated.
Output dimension	Standard: hidden_size d. BiRNN: 2d.
Real-time use	Standard: can process streaming data step-by-step. BiRNN: requires the FULL sequence before processing — cannot stream.
Typical tasks	Standard: language generation, live speech. BiRNN: NER, sentiment analysis, offline speech recognition.

#### Limitations of BiRNN

- **No real-time / online processing:** The backward pass must start at the end of the sequence — the entire input must be available before any output can be produced. This rules out live chatbots, real-time speech-to-text, or any streaming application.
- **Double parameters and compute:** Two RNN chains means  $\sim 2\times$  parameters,  $2\times$  memory, and  $2\times$  forward/backward computation time.
- **Double BPTT complexity:** Gradients flow back through both chains independently, increasing debugging difficulty and training time.
- **Information leakage in generation:** In autoregressive models (predict the next word), BiRNNs are unusable — giving access to future tokens would make the task trivial and the model useless for generation.
- **Still limited on very long sequences:** BiRNNs still suffer from vanishing gradients across each chain for sequences of hundreds of steps. Attention is still needed for reliable long-range access.

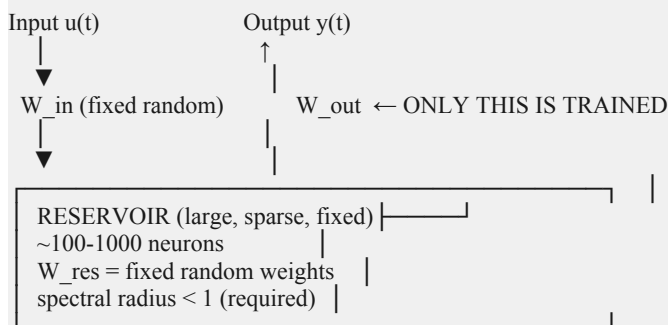
*Note: BERT (2018) achieves true bidirectionality using Transformer self-attention — every token attends to every other token simultaneously, in  $O(1)$  path length, fully in parallel at training time. This is why BERT massively outperformed BiLSTM-based models on every NLP benchmark when it was released.*

**Q4b Explain any Seven Challenges of Long-Term Dependencies.****[6 Marks]**

**[REPEATED]** Seven challenges of long-term dependencies are covered in Q4b of May-June 2025 (this document) with detailed explanations: vanishing gradients, exploding gradients, memory decay, bridging long gaps, sequential bottleneck, computational depth, and hyperparameter sensitivity.

**Q4c How Echo State Network Differ from Traditional RNNs?****[6 Marks]****Echo State Networks vs Traditional RNNs**

An Echo State Network (ESN) belongs to the family of Reservoir Computing — a paradigm where the internal recurrent dynamics of the network (the 'reservoir') are fixed and random, and only the readout (output) weights are trained. This is the fundamental departure from traditional RNNs where all weights are learned.

**ESN Architecture**

State update:  $x(t) = (1-\alpha)x(t-1) + \alpha \cdot \tanh(W_{\text{res}} \cdot x(t-1) + W_{\text{in}} \cdot u(t))$  Output:  $y(t) = W_{\text{out}} \cdot x(t)$  ← ridge regression, trivial to solve

**Comparison Table**

Property	Traditional RNN vs ESN
What is trained	Traditional RNN: all weights ( $W_{\text{xh}}$ , $W_{\text{hh}}$ , $W_{\text{hy}}$ ) via BPTT. ESN: only $W_{\text{out}}$ via linear regression — everything else is fixed.
Training algorithm	Traditional RNN: BPTT — expensive, requires storing all hidden states and backpropagating through T steps. ESN: simple ridge regression on collected reservoir states — one algebraic solve.
Vanishing gradients	Traditional RNN: major problem for long sequences. ESN: not applicable — no gradient-based training of recurrent weights.
Reservoir design	Traditional RNN: architecture learned end-to-end. ESN: reservoir must satisfy the echo state property: spectral radius of $W_{\text{res}} < 1$ to ensure the state forgets its initial conditions.
Expressiveness	Traditional LSTM: learns complex task-specific dynamics. ESN: limited by fixed random reservoir — cannot adapt internal representations.
Computational cost	Traditional RNN: $O(T \cdot n^2)$ per sequence for BPTT. ESN: $O(T \cdot n)$ for state collection + $O(n^3)$ once for linear solve — far cheaper for moderate n.
Use cases	Traditional RNN: complex NLP tasks. ESN: chaotic time series prediction, real-time signal processing, embedded systems where training speed is critical.

*Note: The Echo State Property is the theoretical foundation of ESNs: the reservoir must asymptotically 'echo' (reflect) the history of inputs rather than amplifying them indefinitely. This is guaranteed when the spectral radius (largest eigenvalue magnitude) of  $W_{\text{res}}$  is less than 1. ESNs were developed by Herbert Jaeger (2001) and have found practical use in speech recognition and robot control where ultra-fast training is essential.*