USN | | | | | | | | | | |

Internal Assessment Test II Set-3 – Nov/Dec 2025

Sub:	Deep Learning					Sub Code:	BCS714A	Branch	ISE	
Date:	02.12.2025	Duration:	90 min's	Max Marks:	50	Sem/Sec:	VII/ A, B, C		OBE	

	Answer any FIVE questions	MARKS	CO	RBT
1	Explain the algorithms with adaptive learning rates AdaGrad, RMSProp, and Adam with suitable explanations.	10	CO2	L2
2	Explain the concept of L2 parameter regularization in deep learning in detail	10	CO2	L2
3	Discuss how Sparse Representations help in improving model efficiency and generalization.	10	CO2	L3
4	Write a note on Computing the Gradient in a Recurrent Neural Network Write a short note on Bidirectional Recurrent Neural Networks with examples.	7 3	CO4	L2
5	Explain the Encoder–Decoder Sequence-to-Sequence architecture with a neat diagram and its applications.	10	CO4	L2
6	What are Recursive Neural Networks? Describe their working principle with an example	10	CO4	L2

Faculty Signature CCI Signature HOD Signature

USN | | | | | | | | | | |

Internal Assessment Test II Set-3 – Nov/Dec 2025

Sub:	Deep Learning					Sub Code:	BCS714A	Branch	ISE	
Date:	02.12.2025	Duration:	90 min's	Max Marks:	50	Sem/Sec:	VII / A, B, C		OBE	

	Answer any FIVE questions	MARKS	CO	RBT
1	Explain the algorithms with adaptive learning rates AdaGrad, RMSProp, and Adam with suitable explanations.	10	CO2	L2
2	Explain the concept of L2 parameter regularization in deep learning in detail	10	CO2	L2
3	Discuss how Sparse Representations help in improving model efficiency and generalization.	10	CO2	L3
4	Write a note on Computing the Gradient in a Recurrent Neural Network Write a short note on Bidirectional Recurrent Neural Networks with examples.	7 3	CO4	L2
5	Explain the Encoder–Decoder Sequence-to-Sequence architecture with a neat diagram and its applications.	10	CO4	L2
6	What are Recursive Neural Networks? Describe their working principle with an example	10	CO4	L2

Faculty Signature CCI Signature HOD Signature

USN | | | | | | | | | | |

Internal Assessment Test II Set-3 – Nov/Dec 2025

Sub:	Deep Learning					Sub Code:	BCS714A	Branch	ISE
Date:	02.12.2025	Duration:	90 min's	Max Marks:	50	Sem/Sec:	VII/ A, B, C		OBE

Answer any FIVE questions

		MARKS	CO	RBT
1	Explain the algorithms with adaptive learning rates AdaGrad, RMSProp, and Adam with suitable explanations.	10	CO2	L2

- Learning rate is one of the most difficult and most important hyperparameters in neural networks.
- Deep networks have different sensitivity in different parameter directions → some directions require small steps, others can tolerate larger steps.
- Momentum helps but adds another hyperparameter.
- Motivation: instead of one global learning rate, use separate learning rates per parameter and adapt them during training.
- Idea: if a parameter's gradient keeps the same sign, the learning rate for that parameter should increase;
- if the gradient changes sign, learning rate should decrease.
- Early approach: Delta-bar-delta (Jacobs, 1988) — adaptive learning rate heuristic, works only for full-batch optimization.
- Modern methods: mini-batch-based adaptive learning algorithms that adjust learning rates automatically

AdaGrad

- AdaGrad adapts learning rates per parameter.
- Learning rate for each parameter is scaled by:
- Inverse square root of the sum of past squared gradients.

Effect:

- Parameters with large gradients → learning rate decreases fast.
- Parameters with small gradients → learning rate decreases slowly.

Result: more progress in gently sloped directions of parameter space.

- In convex optimization, AdaGrad has strong theoretical guarantees.
- In deep learning, AdaGrad can shrink learning rates too much, too early (due to accumulating squared gradients), slowing training.
- Works well for some deep models but not consistently for all.

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, perhaps 10^{-7}, for numerical stability

Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

RMSProp

- RMSProp (Hinton, 2012) is a modification of AdaGrad for non-convex optimization (like deep networks).
- Key change: instead of summing all past squared gradients (AdaGrad), RMSProp uses an exponentially decaying moving average of squared gradients.
- AdaGrad's learning rate shrinks too much because it accumulates all past gradients.
- Neural networks often move through many regions before reaching a local convex region.

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
end while

Adam

Adam = *Adaptive Moments* (Kingma & Ba, 2014).

Combines ideas from **RMSProp + Momentum**.

Uses **first-order moment (mean)** of gradients (momentum) + **second-order moment (uncentered variance)**.

Applies **exponential moving averages** for both moments.

Includes **bias-correction terms** (important early in training).

- Corrects bias due to initialization at zero.

RMSProp also uses second-order moment but **without bias correction** → Adam better early stability.

Adam is **robust to hyperparameter choices**, but **learning rate sometimes needs tuning**.

Popular for deep learning because it **adapts learning rates per parameter** and **performs well in practice**.

2	Explain the concept of L2 parameter regularization in deep learning in detail • In **machine learning**, a **penalty** is an **extra cost added to the loss function** to discourage the model from becoming too complex or overfitting the training data. Penalties are common in **regularization** methods like: • **L1 penalty (Lasso)** → penalizes the absolute values of weights $\text{Penalty} = \lambda \sum \lvert w_i \rvert$ • **L2 penalty (Ridge)** → penalizes the squared values of weights $\text{Penalty} = \lambda \sum w_i^2$ • One of the simplest and most common kinds of parameter norm penalty: the L2 parameter norm penalty, commonly known as weight decay. • **L2 parameter norm penalty** means we measure the size of the weights using the **squared Euclidean norm** • looking at **weight decay regularization** (L2 regularization). The idea is to add a penalty proportional to the sum of squared weights. • Here, the model is simplified by: • **No bias terms** → so all parameters are just the weight vector w. • Using **L2 penalty** $\frac{\alpha}{2} w^\top w$.	10	CO2	L2

$$\tilde{J}(w; X, y) = \frac{\alpha}{2} w^\top w + J(w; X, y)$$

- $\tilde{J}(w; X, y)$ = **total objective function** (regularized loss)
- $J(w; X, y)$ = the **original loss function** (e.g., MSE, cross-entropy) measuring prediction error on training data
- $\frac{\alpha}{2} w^\top w$ = the **penalty term**
  - $w^\top w$ is the sum of squares of weights = $\|w\|_2^2$
  - α = regularization strength (hyperparameter)
  - The factor $\frac{1}{2}$ makes derivatives cleaner

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2} \boldsymbol{w}^\top \boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}),$$

with the corresponding parameter gradient

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}).$$

To take a single gradient step to update the weights, we perform this update

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon \left(\alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) \right).$$

Written another way, the update is:

$$\boldsymbol{w} \leftarrow (1 - \epsilon\alpha)\boldsymbol{w} - \epsilon \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}).$$

- The approximation Jˆ is given by

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + \frac{1}{2} (\boldsymbol{w} - \boldsymbol{w}^*)^\top \boldsymbol{H} (\boldsymbol{w} - \boldsymbol{w}^*)$$

- H is the Hessian matrix of J with respect to w evaluated at w∗.
- The minimum of Jˆ occurs where its gradient is equal to 0

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*)$$

- To study the effect of weight decay, we modify the above equation by adding the weight decay gradient. We use the variable w˜ to represent the location of the minimum

$$\alpha \tilde{\boldsymbol{w}} + \boldsymbol{H}(\tilde{\boldsymbol{w}} - \boldsymbol{w}^*) = 0 \qquad (7.8)$$

$$(\boldsymbol{H} + \alpha \boldsymbol{I}) \tilde{\boldsymbol{w}} = \boldsymbol{H} \boldsymbol{w}^* \qquad (7.9)$$

$$\tilde{\boldsymbol{w}} = (\boldsymbol{H} + \alpha \boldsymbol{I})^{-1} \boldsymbol{H} \boldsymbol{w}^*. \qquad (7.10)$$

Substitute $H = Q\Lambda Q^T$ into (7.10).

$$\tilde{\boldsymbol{w}} = (\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top + \alpha \boldsymbol{I})^{-1} \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top \boldsymbol{w}^* \qquad (7.11)$$

Factor out Q and Q^T.

$$= \left[\boldsymbol{Q}(\boldsymbol{\Lambda} + \alpha \boldsymbol{I}) \boldsymbol{Q}^\top \right]^{-1} \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top \boldsymbol{w}^* \qquad (7.12)$$

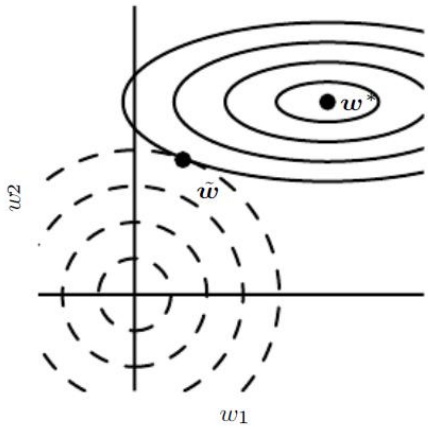
Use the identity: $(QAQ^T)^{-1} = QA^{-1}Q^T$

$$= \boldsymbol{Q}(\boldsymbol{\Lambda} + \alpha \boldsymbol{I})^{-1} \boldsymbol{\Lambda} \boldsymbol{Q}^\top \boldsymbol{w}^*. \qquad (7.13)$$

Here, Each component along eigenvector i is scaled by:

$$\frac{\lambda_i}{\lambda_i + \alpha}$$

- If $\lambda_i \gg \alpha \rightarrow$ scaling $\approx 1 \rightarrow$ **kept**
- If $\lambda_i \ll \alpha \rightarrow$ scaling $\approx 0 \rightarrow$ **shrunk**

	•  An illustration of the effect of L2 (or weight decay) regularization on the value of the optimal w the sum of squared errors: $$(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y})^\top(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y}). \quad (7$$ When we add L^2 regularization, the objective function changes to $$(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y})^\top(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y})+\frac{1}{2}\alpha\boldsymbol{w}^\top\boldsymbol{w}. \quad (7$$ This changes the normal equations for the solution from $$\boldsymbol{w}=(\boldsymbol{X}^\top\boldsymbol{X})^{-1}\boldsymbol{X}^\top\boldsymbol{y} \quad (7$$ to			
3	Discuss how Sparse Representations help in improving model efficiency and generalization. • Weight decay penalizes model parameters directly. • Another approach: penalize neural activations to encourage sparse activations. • This enforces sparsity in representations, not necessarily in weights. • L1 regularization → sparse parameters (many weights become zero). • Representational sparsity → many units/activations become zero. These two types of sparsity are different: Parameter sparsity = sparse weights Representation sparsity = sparse feature activations • A simplified view of this distinction can be illustrated in the context of linear regression $$\begin{bmatrix}18\\5\\15\\-9\\-3\end{bmatrix}=\begin{bmatrix}4&0&0&-2&0&0\\0&0&-1&0&3&0\\0&5&0&0&0&0\\1&0&0&-1&0&-4\\1&0&0&0&-5&0\end{bmatrix}\begin{bmatrix}2\\3\\-2\\-5\\1\\4\end{bmatrix} \quad (7.46)$$ $\boldsymbol{y}\in\mathbb{R}^m$ $\boldsymbol{A}\in\mathbb{R}^{m\times n}$ $\boldsymbol{x}\in\mathbb{R}^n$ $$\begin{bmatrix}-14\\1\\19\\2\\23\end{bmatrix}=\begin{bmatrix}3&-1&2&-5&4&1\\4&2&-3&-1&1&3\\-1&5&4&2&-3&-2\\3&1&2&-3&0&-3\\-5&4&-2&2&-5&-1\end{bmatrix}\begin{bmatrix}0\\2\\0\\0\\-3\\0\end{bmatrix} \quad (7.47)$$ $\boldsymbol{y}\in\mathbb{R}^m$ $\boldsymbol{B}\in\mathbb{R}^{m\times n}$ $\boldsymbol{h}\in\mathbb{R}^n$	10	CO2	L3

	• Two cases:
	  • **Sparse parameters** → few non-zero weights.
	  • **Sparse representation** → representation h of input x has many zero values.
	• Here, focus is on **sparse representation**:
	  $h = f(x)$ represents input but with many zeros.
	• **Representational regularization** encourages sparsity in **activations** rather than weights.
	• Implemented similarly to parameter regularization, using a **norm penalty** on h.
	• Regularized loss:
	$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\,\Omega(h)$$
	• $\alpha \geq 0$ controls strength of sparsity penalty (larger α = more sparsity).
	
	• Applying a **penalty on the representation** h can create **representational sparsity**.
	• Common penalty: **L1 norm on activations**
	$$\Omega(h) = \|h\|_1 = \sum_i |h_i|$$
	• **Other sparsity-inducing penalties**:
	  • Student-t prior on h (Olshausen & Field, 1996; Bergstra, 2011)
	  • **KL-divergence penalty** (especially when activations are between 0 and 1) (Larochelle & Bengio, 2008)
	• Some methods enforce **sparse average activation**:
	$$\frac{1}{m}\sum_i h^{(i)}$$
	Target small value, e.g., **0.01** (Lee et al., 2008; Goodfellow et al., 2009)
	• **Hard constraint approach** (instead of soft penalty):
	  • e.g., **Orthogonal Matching Pursuit (OMP)**
	  • Choose h with fewer than k non-zero elements:
	$$\arg\min_{h,\,\|h\|_0<k} \|x - Wh\|^2$$			

4	Write a note on Computing the Gradient in a Recurrent Neural Network • Computing the gradient through a recurrent neural network is straightforward. • Gradients obtained by back-propagation may then be used with any general-purpose gradient-based techniques to train an RNN. • We start the recursion with the nodes immediately preceding the final loss $$\frac{\partial L}{\partial L^{(t)}} = 1.$$ The gradient $\nabla_{\boldsymbol{o}^{(t)}} L$ on the outputs at time step t, for all i, t, is as $$(\nabla_{\boldsymbol{o}^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}}\frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i,y^{(t)}}. \qquad (10.18)$$ We work our way backwards, starting from the end of the sequence. At the final time step τ, $\boldsymbol{h}^{(\tau)}$ only has $\boldsymbol{o}^{(\tau)}$ as a descendent, so its gradient is simple: $$\nabla_{\boldsymbol{h}^{(\tau)}} L = \boldsymbol{V}^\top \nabla_{\boldsymbol{o}^{(\tau)}} L. \qquad (10.19)$$	7 3	CO4	L2

We can then iterate backwards in time to back-propagate gradients through time, from $t = \tau - 1$ down to $t = 1$, noting that $\boldsymbol{h}^{(t)}$ (for $t < \tau$) has as descendents both $\boldsymbol{o}^{(t)}$ and $\boldsymbol{h}^{(t+1)}$. Its gradient is thus given by

$$\nabla_{\boldsymbol{h}^{(t)}} L = \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}}\right)^{\top} (\nabla_{\boldsymbol{h}^{(t+1)}} L) + \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\right)^{\top} (\nabla_{\boldsymbol{o}^{(t)}} L) \tag{10.20}$$

$$= \boldsymbol{W}^{\top} (\nabla_{\boldsymbol{h}^{(t+1)}} L) \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right) + \boldsymbol{V}^{\top} (\nabla_{\boldsymbol{o}^{(t)}} L) \tag{10.21}$$

Using this notation, the gradient on the remaining parameters is given by:

$$\nabla_{\boldsymbol{c}} L = \sum_t \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}}\right)^{\top} \nabla_{\boldsymbol{o}^{(t)}} L = \sum_t \nabla_{\boldsymbol{o}^{(t)}} L \tag{10.22}$$

$$\nabla_{\boldsymbol{b}} L = \sum_t \left(\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}^{(t)}}\right)^{\top} \nabla_{\boldsymbol{h}^{(t)}} L = \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \nabla_{\boldsymbol{h}^{(t)}} L \tag{10.23}$$

$$\nabla_{\boldsymbol{V}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}}\right) \nabla_{\boldsymbol{V}} o_i^{(t)} = \sum_t (\nabla_{\boldsymbol{o}^{(t)}} L)\, \boldsymbol{h}^{(t)\top} \tag{10.24}$$

$$\nabla_{\boldsymbol{W}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)} \tag{10.25}$$

$$= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) (\nabla_{\boldsymbol{h}^{(t)}} L)\, \boldsymbol{h}^{(t-1)\top} \tag{10.26}$$

$$\nabla_{\boldsymbol{U}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{\boldsymbol{U}^{(t)}} h_i^{(t)} \tag{10.27}$$

$$= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) (\nabla_{\boldsymbol{h}^{(t)}} L)\, \boldsymbol{x}^{(t)\top} \tag{10.28}$$

Write a short note on Bidirectional Recurrent Neural Networks with examples.

The recurrent networks discussed so far are **causal** models.

This means the hidden state at time t uses only **past inputs and the current input**:

$$x(1), \ldots, x(t-1), x(t)$$

Some RNNs also use **past output values** y (when available) to influence the current state.

Future information is not used — the model only depends on past and present time steps.

In many tasks (speech, handwriting), the current prediction $y(t)$ depends on **both past and future inputs**.

Examples:

- Speech recognition – future phonemes/words help disambiguate current sound
- Handwriting recognition – future strokes help interpret current stroke

Bidirectional RNNs (Schuster & Paliwal, 1997) solve this by using:

- One RNN scanning **forward in time**
- One RNN scanning **backward in time**

Output at time t uses **both past and future context**.

Highly successful in **speech, handwriting, and bioinformatics** applications.

Unlike fixed window models, they **don't need a fixed lookahead window**.

Can be extended to **2D**, using 4 directions (up, down, left, right) for image data.

Compared to CNNs, bidirectional RNNs:

- are **more expensive**
- but model **long-range interactions** across the input.

5	Explain the Encoder–Decoder Sequence-to-Sequence architecture with a neat diagram and its applications.	10	CO4	L2

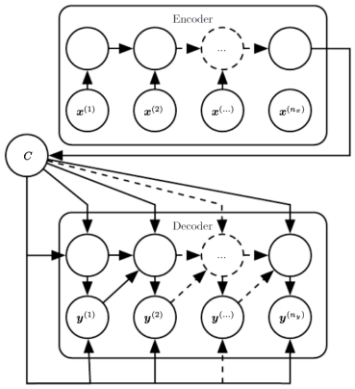


Figure 10.12: Example of an encoder-decoder or sequence-to-sequence RNN architecture, for learning to generate an output sequence $(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n_y)})$ given an input sequence $(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(n_x)})$. It is composed of an encoder RNN 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.

We train RNNs to map an **input sequence to an output sequence** of **different lengths**.

- Applications: **speech recognition, machine translation, question answering**.
- Input sequence is called the **context**.
- Model learns a **context representation CCC** that summarizes the input sequence X.
- CCC can be a **vector** or a **sequence of vectors** representing the input information.

The input sequence is denoted as

$$X = (x(1), x(2), \ldots, x(n_x))$$

where $x(t)$ is the input at time step t, and n_x is the input sequence length.

Seq2Seq (Encoder–Decoder) RNN architecture introduced by **Cho et al. (2014)** and **Sutskever et al. (2014)** for variable-length input/output mapping.

Encoder RNN reads input sequence and produces **context vector** C (usually last hidden state).

Decoder RNN generates the output sequence conditioned on the **context vector** C.

Notation:

Input length = n_x, Output length = n_y; n_x **and** n_y **can differ**.

Model trained to maximize:

$$\log P(y(1), ..., y_{n_y} \mid x(1), ..., x_{n_x})$$

Context C used as:

- **initial hidden state**, or
- **input at every decoder step** (or both).

Encoder and decoder **can have different hidden sizes**.

Limitation: Fixed-size context struggles with **long sequences**.

Bahdanau et al. (2015):

- Convert C from **single vector to sequence**.
- Introduce **attention mechanism** to select relevant parts of encoder outputs for each decoder step.

6	What are Recursive Neural Networks? Describe their working principle with an example	10	CO4	L2

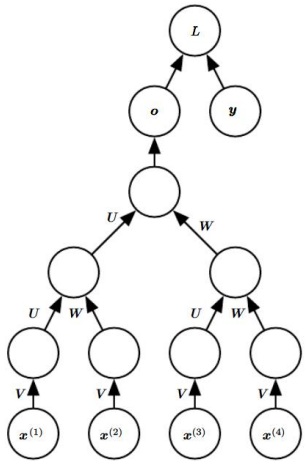


Figure 10.14: A recursive network has a computational graph that generalizes that of the recurrent network from a chain to a tree. A variable-size sequence $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$ can be mapped to a fixed-size representation (the output $\boldsymbol{o}$), with a fixed set of parameters (the weight matrices $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$). The figure illustrates a supervised learning case in which some target $\boldsymbol{y}$ is provided which is associated with the whole sequence.

Recursive neural networks (RecNNs) generalize RNNs by using a **tree-structured computational graph** instead of a chain.

Introduced by **Pollack (1990)**; applied in **reasoning**, **structured data**, **NLP**, and **vision** (Socher et al., 2011–2013).

Advantage: for sequence length τ, depth reduces from τ (RNN) to $O(\log \tau)$ → helps with **long-term dependencies**.

Key challenge: **how to choose the tree structure**.

- Option: fixed tree (e.g., **balanced binary tree**)
- Option: tree determined by **external parser** (e.g., syntactic parse tree for sentences)
- Ideal: model **learns tree structure automatically**

Variants exist: inputs/targets at tree nodes; computation at nodes can be **beyond simple affine + nonlinearity**.

Example: **tensor & bilinear forms** (Socher 2013) to model relational structure between embeddings.

Faculty Signature CCI Signature HOD Signature