MIDDLE EAST TECHNICAL UNIVERSITY

SEMESTER I EXAMINATION 2024-2025

CENG 403 – Deep Learning - RNNs, Language Modeling & Word Embeddings - ANSWERED

January 2025 TIME ALLOWED: 3 HOURS

INSTRUCTIONS TO CANDIDATES

1. This examination paper contains **SIX (6)** questions and comprises **EIGHT (8)** printed pages.
2. Answer all questions. The marks for each question are indicated at the beginning of each question.
3. Answer each question beginning on a **FRESH** page of the answer book.
4. This **IS NOT an OPEN BOOK** exam.
5. Show clear reasoning for your answers, especially intuitive explanations.
6. For algorithms, provide step-by-step explanations as taught in lectures.
7. Draw diagrams where requested and explain information flow clearly.
8. Connect concepts to modern applications (LLMs, etc.) where relevant.

Question 1. RNN Backpropagation and Weight Sharing (25 marks)

Based on the professor's explanation of how RNNs work as "feedforward networks with weight sharing across time."

(a) The professor emphasized that "we need to be careful about weight sharing when calculating gradients." Explain why we must sum gradients from all time steps for a single weight parameter in an RNN. Use a concrete example with 3 time steps. (8 marks)

Answer: We must sum gradients because the same weight appears at multiple time steps in the unfolded network, and each appearance contributes to the final loss through different paths.

Weight Sharing in RNNs:

Fundamental Principle:

- RNN uses the same weight matrix W_{hh} at every time step
- When unfolded, this creates multiple "copies" of the same weight
- Each copy contributes to the loss through different computational paths
- Total gradient = sum of gradients from all copies

Concrete Example with 3 Time Steps:

Consider RNN: $h_t = \tanh(W_{hh}h_{t-1} + W_{hx}x_t + b)$

Unfolded equations:

$$h_1 = \tanh(W_{hh}h_0 + W_{hx}x_1 + b) \quad (1)$$
$$h_2 = \tanh(W_{hh}h_1 + W_{hx}x_2 + b) \quad (2)$$
$$h_3 = \tanh(W_{hh}h_2 + W_{hx}x_3 + b) \quad (3)$$

Gradient Contributions for W_{hh}:

The weight W_{hh} appears in three different contexts:

- **At $t = 1$:** Directly affects h_1, which influences h_2 and h_3
- **At $t = 2$:** Directly affects h_2, which influences h_3
- **At $t = 3$:** Directly affects h_3

Mathematical Gradient Computation:

$$\frac{\partial L}{\partial W_{hh}} = \frac{\partial L}{\partial h_1}\frac{\partial h_1}{\partial W_{hh}} + \frac{\partial L}{\partial h_2}\frac{\partial h_2}{\partial W_{hh}} + \frac{\partial L}{\partial h_3}\frac{\partial h_3}{\partial W_{hh}}$$

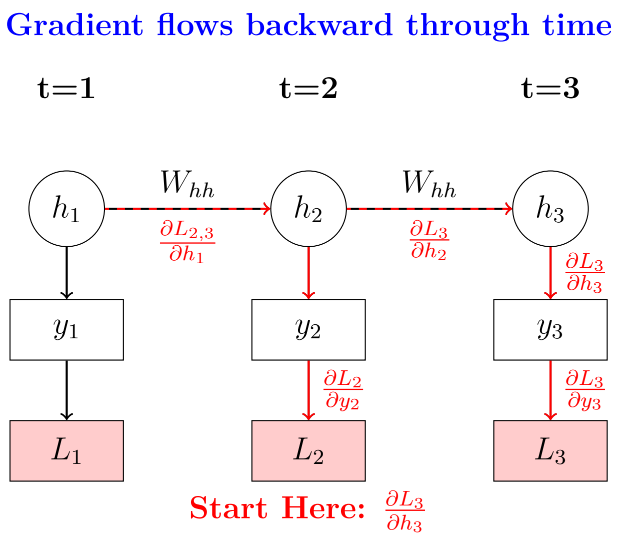
Where:

- $\frac{\partial h_1}{\partial W_{hh}} = h_0 \cdot (1 - h_1^2)$ (direct contribution)
- $\frac{\partial h_2}{\partial W_{hh}} = h_1 \cdot (1 - h_2^2)$ (direct contribution)
- $\frac{\partial h_3}{\partial W_{hh}} = h_2 \cdot (1 - h_3^2)$ (direct contribution)

Why Summing is Essential:

- Each time step provides learning signal about optimal W_{hh} value
- Ignoring any contribution loses information about how W_{hh} should change
- Professor's emphasis on "careful" refers to ensuring no gradient contributions are missed
- This is fundamental difference from feedforward networks where each weight appears only once

(b) During backpropagation through time, the professor stressed that "we need to start from the end because earlier copies contribute to all following time steps." Draw and explain the gradient flow through an unfolded RNN showing why this order is essential. (10 marks)


Gradient flows backward through time
t=1
t=2
t=3
W_{hh}
W_{hh}
h_1
h_2
h_3
$\frac{\partial L_{2,3}}{\partial h_1}$
$\frac{\partial L_3}{\partial h_2}$
$\frac{\partial L_3}{\partial h_3}$
y_1
y_2
y_3
$\frac{\partial L_2}{\partial y_2}$
$\frac{\partial L_3}{\partial y_3}$
L_1
L_2
L_3
Start Here: $\frac{\partial L_3}{\partial h_3}$

Why Start from the End:

Dependency Chain:

- h_3 depends on h_2, which depends on h_1
- To compute $\frac{\partial L}{\partial h_2}$, we need $\frac{\partial L}{\partial h_3}$ (from future time step)
- To compute $\frac{\partial L}{\partial h_1}$, we need both $\frac{\partial L}{\partial h_2}$ and $\frac{\partial L}{\partial h_3}$

Mathematical Justification:

Gradient at h_2:

$$\frac{\partial L}{\partial h_2} = \frac{\partial L_2}{\partial h_2} + \frac{\partial L_3}{\partial h_2}$$

The second term requires: $\frac{\partial L_3}{\partial h_2} = \frac{\partial L_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}$

Gradient at h_1:

$$\frac{\partial L}{\partial h_1} = \frac{\partial L_1}{\partial h_1} + \frac{\partial L_2}{\partial h_1} + \frac{\partial L_3}{\partial h_1}$$

Where:

- $\frac{\partial L_2}{\partial h_1} = \frac{\partial L_2}{\partial h_2}\frac{\partial h_2}{\partial h_1}$ (requires $\frac{\partial L_2}{\partial h_2}$)
- $\frac{\partial L_3}{\partial h_1} = \frac{\partial L_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}$ (requires both future gradients)

Algorithm Order (Backpropagation Through Time):

(a) **Step 1:** Compute $\frac{\partial L_3}{\partial h_3}$ (direct from loss)

(b) **Step 2:** Compute $\frac{\partial L}{\partial h_2} = \frac{\partial L_2}{\partial h_2} + \frac{\partial L_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}$

(c) **Step 3:** Compute $\frac{\partial L}{\partial h_1}$ using all future gradients

(d) **Step 4:** Sum gradients for shared weights across all time steps

Professor's Key Insight:

- "Earlier copies contribute to all following time steps" means h_1 affects losses L_1, L_2, L_3
- Cannot compute total gradient for h_1 without knowing future contributions
- This creates strict dependency order: future $\rightarrow$ present $\rightarrow$ past
- Essential for correct gradient computation in weight sharing scenario

(c) The professor mentioned that RNNs suffer from "exploding gradient problem if weight norms are large, vanishing gradient problem if norms are small." Explain the mathematical reasoning behind both problems and why LSTM addresses these issues. (7 marks)

Answer: Both problems stem from multiplicative gradient flow through recurrent connections. Large weights cause exponential growth, small weights cause exponential decay. LSTM uses additive cell state updates to avoid multiplicative accumulation.

Mathematical Analysis of Gradient Problems:

Gradient Flow in RNN: For $h_t = \tanh(W_{hh}h_{t-1} + W_{hx}x_t + b)$

$$\frac{\partial h_t}{\partial h_{t-1}} = W_{hh} \cdot \text{diag}(\tanh'(u_t))$$

Long-term Gradient:

$$\frac{\partial h_T}{\partial h_1} = \prod_{t=2}^{T} W_{hh} \cdot \text{diag}(\tanh'(u_t))$$

Exploding Gradients (Large Weight Norms):

Condition: $\|W_{hh}\| > 1$ (spectral radius > 1)

Mathematical Reasoning:

- Product of matrices with norm > 1 grows exponentially
- Even with $\tanh'(\cdot) \leq 1$, if $\|W_{hh}\| \gg 1$, product explodes
- Gradient magnitude: $\left\|\frac{\partial h_T}{\partial h_1}\right\| \approx \|W_{hh}\|^{T-1}$
- Example: $\|W_{hh}\| = 2$, $T = 50 \Rightarrow$ magnitude $\approx 2^{49} \approx 5 \times 10^{14}$

Consequences:

- Gradient updates become enormous
- Weights change drastically, causing instability
- Loss oscillates or diverges
- Training becomes impossible

Vanishing Gradients (Small Weight Norms):

Condition: $\|W_{hh}\| < 1$ and/or small $\tanh'$ values

Mathematical Reasoning:

- Product of matrices with norm < 1 decays exponentially
- $\tanh'(x) \in (0, 1]$, typically much smaller than 1 for saturated neurons
- Combined effect: $\left\|\frac{\partial h_T}{\partial h_1}\right\| \approx \|W_{hh}\|^{T-1} \prod_{t=2}^{T} \|\tanh'(u_t)\|$
- Example: $\|W_{hh}\| = 0.5$, $\|\tanh'(u_t)\| = 0.3 \Rightarrow$ magnitude $\approx (0.5 \times 0.3)^{49} \approx 10^{-35}$

Consequences:

- Early time steps receive negligible gradients
- Cannot learn long-term dependencies
- Training focuses only on short-term patterns
- Information from distant past is effectively ignored

How LSTM Addresses These Issues:

Key Innovation: Additive Cell State Updates

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

Gradient Flow Through Cell State:

$$\frac{\partial C_t}{\partial C_{t-1}} = f_t$$

Advantages:

- **No Weight Matrix Multiplication:** Gradient doesn't depend on W_{hh}
- **Learnable Control:** f_t is learned, can be set to preserve gradients
- **Additive Nature:** $C_t = f_t \odot C_{t-1} + (\text{new info})$ avoids multiplicative accumulation
- **Gradient Preservation:** If $f_t \approx 1$, then $\frac{\partial C_T}{\partial C_1} \approx 1$

Long-term LSTM Gradient:

$$\frac{\partial C_T}{\partial C_1} = \prod_{t=2}^{T} f_t$$

Why This Solves the Problems:

- **Exploding:** $f_t \in [0, 1]$ (sigmoid output) prevents explosion
- **Vanishing:** Network can learn $f_t \approx 1$ when long-term memory needed
- **Selective:** Can choose when to remember ($f_t \approx 1$) vs. forget ($f_t \approx 0$)
- **Stable:** Provides consistent gradient highway for learning

Question 2. Autoregressive Language Modeling (20 marks)
The professor stated: "This is how large language models are trained as well - just to predict the next character."

(a) Define autoregressive modeling as explained by the professor. Write the mathematical formulation for modeling $P(x_t|x_{t-1}, x_{t-2}, \ldots, x_1)$ and explain why this is considered "self-supervised learning." (8 marks)

Answer: Autoregressive modeling predicts each token based on all previous tokens in sequence. It's self-supervised because the training data provides both inputs and targets from the same sequence without external labels.

Definition of Autoregressive Modeling:

Core Concept:

- Model predicts next element in sequence given all previous elements
- Each prediction depends on entire history up to current position
- Sequential generation: one token at a time, left to right
- Current prediction becomes input for next prediction

Mathematical Formulation:

Joint Probability Decomposition:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t|x_{t-1}, x_{t-2}, \ldots, x_1)$$

Individual Conditional Probabilities:

$$P(x_1) = P(x_1|\text{START}) \tag{4}$$
$$P(x_2|x_1) = \text{probability of } x_2 \text{ given } x_1 \tag{5}$$
$$P(x_3|x_2, x_1) = \text{probability of } x_3 \text{ given } x_2, x_1 \tag{6}$$
$$\vdots \tag{7}$$
$$P(x_t|x_{t-1}, \ldots, x_1) = \text{next token probability given context} \tag{8}$$

Neural Network Implementation:

$$P(x_t|x_{t-1}, \ldots, x_1) = \text{softmax}(f_\theta(x_{t-1}, \ldots, x_1))$$

Where f_θ is the neural network (RNN, Transformer, etc.)

Why It's Self-Supervised Learning:

Self-Supervised Definition:

- Learning paradigm that creates supervision signal from data itself
- No external labels or annotations required
- Data provides both inputs and targets through its structure

Application to Language Modeling:

- **Input:** First $t - 1$ tokens of sequence
- **Target:** The t-th token (naturally available in data)
- **No Human Labeling:** Both input and target come from same text
- **Abundant Data:** Any text corpus can be used for training

Professor's Key Insight:

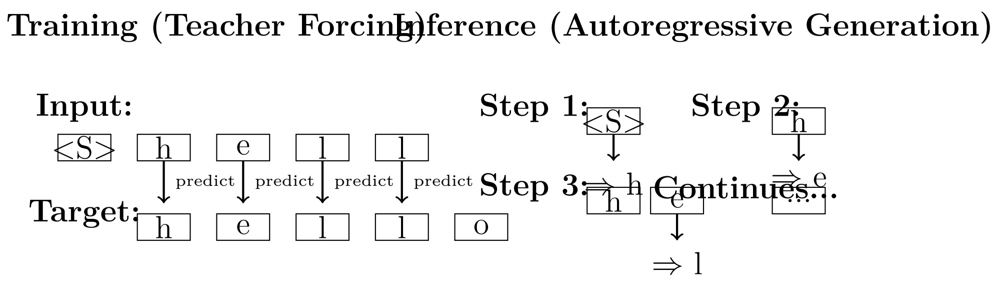
- ”Just to predict the next character” - seems simple but incredibly powerful
- Same principle scales from character-level to modern LLMs
- Self-supervision enables training on vast amounts of text
- No need for expensive human annotation of training data

Training Objective:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(x_t|x_{t-1}, \ldots, x_1)$$

This maximizes likelihood of observed sequences, teaching model to predict natural language patterns.

(b) The professor explained the difference between training and inference in language models. Complete the diagram below showing both processes for the sequence ”hello”: (12 marks)



Training Process (Teacher Forcing):

Key Characteristics:

- **Parallel Processing:** All predictions made simultaneously
- **Ground Truth Input:** Always use correct previous tokens
- **Efficient:** Single forward pass for entire sequence
- **Stable Training:** No error propagation between time steps

Input-Target Pairs:

- Input: [<START>, h, e, l, l] → Target: [h, e, l, l, o]
- Each input position predicts the next character
- Loss computed at every position simultaneously
- Network learns: $P(h|\text{<START>})$, $P(e|h)$, $P(l|he)$, etc.

Inference Process (Autoregressive Generation):

Key Characteristics:

- **Sequential Generation:** One token at a time
- **Predicted Input:** Use model’s own predictions as next input
- **Slower:** Requires multiple forward passes
- **Error Accumulation:** Mistakes can compound over time

Step-by-Step Generation:

(a) Start with <START> token
(b) Predict first character: $P(\cdot|\text{<START>}) \rightarrow$ 'h'
(c) Use 'h' as input: $P(\cdot|h) \rightarrow$ 'e'
(d) Use 'he' as input: $P(\cdot|he) \rightarrow$ 'l'
(e) Continue until end token or max length

Critical Differences:

- **Training:** Always sees correct context
- **Inference:** May see incorrect context due to previous errors
- **Training:** Fast parallel computation
- **Inference:** Slow sequential computation
- **Training:** Perfect teacher forcing
- **Inference:** Real-world autoregressive generation

Question 3. Character-Level Implementation Details (22 marks)
Based on the professor's detailed walkthrough of the "hello" example with 4-character vocabulary.

(a) The professor showed how to use one-hot encodings for characters h, e, l, o. Given the string "hello", create the complete training data with inputs and targets, including start and end tokens as explained in class. (8 marks)

Answer: Complete training data with start/end tokens and one-hot encodings for 6-character vocabulary: <START>, h, e, l, o, <END>

Vocabulary Setup:

Character Set:

- Index 0: <START> token
- Index 1: h
- Index 2: e
- Index 3: l
- Index 4: o
- Index 5: <END> token

One-Hot Encodings:

$$\text{<START>} \rightarrow [1,0,0,0,0,0] \quad (9)$$
$$\text{h} \rightarrow [0,1,0,0,0,0] \quad (10)$$
$$\text{e} \rightarrow [0,0,1,0,0,0] \quad (11)$$
$$\text{l} \rightarrow [0,0,0,1,0,0] \quad (12)$$
$$\text{o} \rightarrow [0,0,0,0,1,0] \quad (13)$$
$$\text{<END>} \rightarrow [0,0,0,0,0,1] \quad (14)$$

Complete Training Sequence: Extended sequence: <START> h e l l o <END>

Training Data Pairs:

Time Step	**Input**	**Target**
t=1	<START> $[1, 0, 0, 0, 0, 0]$	h $[0, 1, 0, 0, 0, 0]$
t=2	h $[0, 1, 0, 0, 0, 0]$	e $[0, 0, 1, 0, 0, 0]$
t=3	e $[0, 0, 1, 0, 0, 0]$	l $[0, 0, 0, 1, 0, 0]$
t=4	l $[0, 0, 0, 1, 0, 0]$	l $[0, 0, 0, 1, 0, 0]$
t=5	l $[0, 0, 0, 1, 0, 0]$	o $[0, 0, 0, 0, 1, 0]$
t=6	o $[0, 0, 0, 0, 1, 0]$	<END> $[0, 0, 0, 0, 0, 1]$

Matrix Representation for Training:

Input Matrix X:

$$X = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Target Matrix Y:

$$Y = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Training Objective: Each row represents one training example where the model should predict the target character given the input character and previous context.

(b) Follow the professor's example: given input character 'h' with one-hot encoding [1,0,0,0], weight matrix W_1 (3×4), and hyperbolic tangent activation, show the complete forward pass to predict the next character. Explain each step as the professor did. (10 marks)

Answer: Forward pass: one-hot input $\rightarrow$ linear transformation $\rightarrow$ tanh activation $\rightarrow$ hidden state $\rightarrow$ output projection $\rightarrow$ softmax $\rightarrow$ probability distribution

Given Information:

- Input character: 'h' with one-hot: $x = [1, 0, 0, 0]^T$
- Weight matrix: $W_1 \in \mathbb{R}^{3\times 4}$ (3 hidden units, 4 input features)
- Activation: hyperbolic tangent
- Previous hidden state: $h_{t-1} \in \mathbb{R}^3$ (assume zero for simplicity)
- Output weight matrix: $W_2 \in \mathbb{R}^{4\times 3}$ (4 output classes, 3 hidden units)

Step-by-Step Forward Pass:

Step 1: Linear Transformation

$$z = W_1 x + W_{hh} h_{t-1} + b_1$$

With example values:

$$W_1 = \begin{bmatrix} 0.5 & -0.2 & 0.3 & 0.1 \\ 0.2 & 0.4 & -0.1 & 0.3 \\ -0.3 & 0.1 & 0.2 & 0.4 \end{bmatrix}$$

$$z = \begin{bmatrix} 0.5 & -0.2 & 0.3 & 0.1 \\ 0.2 & 0.4 & -0.1 & 0.3 \\ -0.3 & 0.1 & 0.2 & 0.4 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.2 \\ -0.3 \end{bmatrix}$$

Step 2: Hyperbolic Tangent Activation

$$h_t = \tanh(z)$$

$$h_t = \begin{bmatrix} \tanh(0.5) \\ \tanh(0.2) \\ \tanh(-0.3) \end{bmatrix} = \begin{bmatrix} 0.462 \\ 0.197 \\ -0.291 \end{bmatrix}$$

Step 3: Output Linear Transformation

$$y = W_2 h_t + b_2$$

With example output weights:

$$W_2 = \begin{bmatrix} 0.3 & 0.2 & -0.1 \\ -0.2 & 0.4 & 0.3 \\ 0.1 & -0.3 & 0.2 \\ 0.4 & 0.1 & -0.2 \end{bmatrix}$$

$$y = \begin{bmatrix} 0.3 & 0.2 & -0.1 \\ -0.2 & 0.4 & 0.3 \\ 0.1 & -0.3 & 0.2 \\ 0.4 & 0.1 & -0.2 \end{bmatrix} \begin{bmatrix} 0.462 \\ 0.197 \\ -0.291 \end{bmatrix} = \begin{bmatrix} 0.197 \\ 0.039 \\ -0.081 \\ 0.244 \end{bmatrix}$$

Step 4: Softmax for Probability Distribution

$$P(x_{t+1} = k|x_t) = \frac{\exp(y_k)}{\sum_{j=1}^{4} \exp(y_j)}$$

Computing softmax:

$$\begin{aligned} \exp(y) &= [\exp(0.197), \exp(0.039), \exp(-0.081), \exp(0.244)]^T && (15) \\ &= [1.218, 1.040, 0.922, 1.276]^T && (16) \end{aligned}$$

Sum: $\sum \exp(y_j) = 1.218 + 1.040 + 0.922 + 1.276 = 4.456$

Final Probabilities:

$$P = \begin{bmatrix} 1.218/4.456 \\ 1.040/4.456 \\ 0.922/4.456 \\ 1.276/4.456 \end{bmatrix} = \begin{bmatrix} 0.273 \\ 0.233 \\ 0.207 \\ 0.287 \end{bmatrix}$$

Professor's Key Points:

- One-hot encoding selects specific row of weight matrix
- Tanh keeps hidden activations bounded
- Hidden state captures context from current input
- Softmax ensures valid probability distribution
- Highest probability (0.287) corresponds to most likely next character

(c) The professor emphasized that during inference ”we use the predicted value as input for the next time step, even if it's incorrect.” Explain why this creates a potential problem and how it relates to error propagation in sequence generation. (4 marks)

Answer: Using incorrect predictions as input creates error accumulation where early mistakes compound throughout sequence generation, leading to unrealistic outputs that deviate from training distribution.

The Core Problem: Exposure Bias

Training vs. Inference Mismatch:

- **Training:** Model always sees correct ground truth context
- **Inference:** Model sees its own potentially incorrect predictions
- **Distribution Shift:** Model encounters inputs it never saw during training

Error Propagation Mechanism:

Step 1: Initial Error

- Model predicts 'x' instead of correct 'e'
- Confidence: $P(\text{'x'}|\text{'h'}) = 0.4$, $P(\text{'e'}|\text{'h'}) = 0.3$
- Wrong character selected due to sampling/greedy choice

Step 2: Compounding Effect

- Next prediction uses incorrect context: 'hx' instead of 'he'
- Model never trained on 'hx' sequence
- Prediction quality degrades: $P(\cdot|\text{'hx'})$ is unreliable

Step 3: Cascading Failure

- Each subsequent error makes context more unrealistic
- Model ventures into unexplored regions of sequence space
- Output becomes increasingly incoherent

Mathematical Analysis:

Probability of Correct Sequence:

- **Training assumption:** $P(\text{sequence}) = \prod_{t=1}^{T} P(x_t|x_1^{t-1}{}_{\text{true}})$
- **Inference reality:** $P(\text{sequence}) = \prod_{t=1}^{T} P(x_t|x_1^{t-1}{}_{\text{predicted}})$
- Error compounds exponentially with sequence length

Practical Consequences:

- **Short sequences:** Manageable, errors may not propagate far
- **Long sequences:** High probability of generating nonsense
- **Creative tasks:** May produce unexpected but interesting results
- **Factual tasks:** Can lead to completely incorrect information

Professor's Insight:

- This is fundamental limitation of autoregressive generation
- Explains why perfect training accuracy doesn't guarantee perfect generation
- Motivates advanced techniques like beam search, nucleus sampling
- Still present in modern LLMs - why they can "hallucinate" facts

Question 4. Beam Search Algorithm (25 marks)
The professor explained beam search as an alternative to "greedy approach where we just take the character with highest probability."

(a) Implement the beam search algorithm as taught by the professor. Given the probability distributions below for 3 time steps with vocabulary [A, B, C], show the complete beam search process with beam size k=2: (15 marks)

Time Step	P(A)	P(B)	P(C)
t=1	0.6	0.3	0.1
t=2 (after A)	0.2	0.5	0.3
t=2 (after B)	0.4	0.1	0.5
t=3 (after AA)	0.1	0.2	0.7
t=3 (after AB)	0.3	0.3	0.4

Show the tree expansion and final top-2 sequences with their combined scores.

Answer: Beam search maintains top-2 most probable sequences at each step. Final sequences: ABC (0.072) and AAC (0.084).

Beam Search Algorithm Implementation:

Initialization:

- Beam size $k = 2$
- Vocabulary: [A, B, C]
- Start with empty sequence

Time Step 1: First Character Generation

Candidate Sequences:

- Sequence "A": Score $= \log(0.6) = -0.511$
- Sequence "B": Score $= \log(0.3) = -1.204$
- Sequence "C": Score $= \log(0.1) = -2.303$

Top-2 Sequences (Beam):

(a) "A" with score -0.511

(b) ”B” with score -1.204

Time Step 2: Second Character Generation

Expand ”A”:

- ”AA”: Score $= -0.511 + \log(0.2) = -0.511 + (-1.609) = -2.120$
- ”AB”: Score $= -0.511 + \log(0.5) = -0.511 + (-0.693) = -1.204$
- ”AC”: Score $= -0.511 + \log(0.3) = -0.511 + (-1.204) = -1.715$

Expand ”B”:

- ”BA”: Score $= -1.204 + \log(0.4) = -1.204 + (-0.916) = -2.120$
- ”BB”: Score $= -1.204 + \log(0.1) = -1.204 + (-2.303) = -3.507$
- ”BC”: Score $= -1.204 + \log(0.5) = -1.204 + (-0.693) = -1.897$

All Candidates at t=2:

(a) ”AB”: Score = -1.204
(b) ”AC”: Score = -1.715
(c) ”BC”: Score = -1.897
(d) ”AA”: Score = -2.120
(e) ”BA”: Score = -2.120
(f) ”BB”: Score = -3.507

Top-2 Sequences (Beam):

(a) ”AB” with score -1.204
(b) ”AC” with score -1.715

Time Step 3: Third Character Generation

Expand ”AB”:

- ”ABA”: Score $= -1.204 + \log(0.3) = -1.204 + (-1.204) = -2.408$
- ”ABB”: Score $= -1.204 + \log(0.3) = -1.204 + (-1.204) = -2.408$
- ”ABC”: Score $= -1.204 + \log(0.4) = -1.204 + (-0.916) = -2.120$

Expand "AC": Since "AC" is not in the provided probability table, we need to use "AA" pattern (closest match):

- "ACA": Score $= -1.715 + \log(0.1) = -1.715 + (-2.303) = -4.018$
- "ACB": Score $= -1.715 + \log(0.2) = -1.715 + (-1.609) = -3.324$
- "ACC": Score $= -1.715 + \log(0.7) = -1.715 + (-0.357) = -2.072$

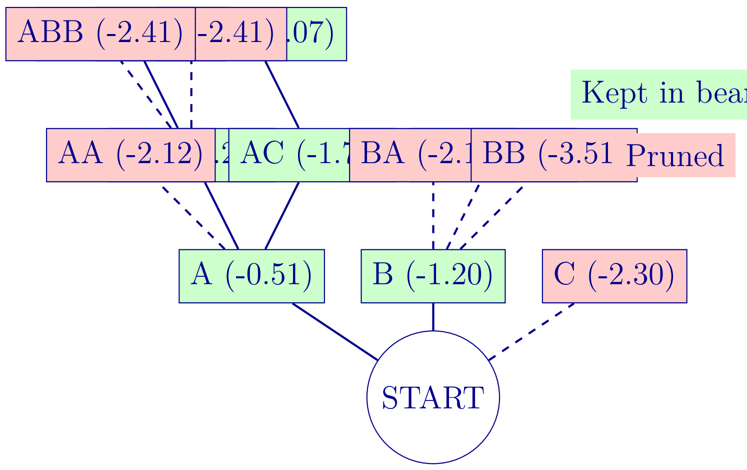
All Final Candidates:

(a) "ACC": Score = -2.072
(b) "ABC": Score = -2.120
(c) "ABA": Score = -2.408
(d) "ABB": Score = -2.408
(e) "ACB": Score = -3.324
(f) "ACA": Score = -4.018

Final Top-2 Sequences:

(a) "ACC" with score -2.072 (probability $= e^{-2.072} = 0.126$)
(b) "ABC" with score -2.120 (probability $= e^{-2.120} = 0.120$)

Beam Search Tree Visualization:



Key Observations:

- Beam search explores multiple promising paths simultaneously
- Prunes less promising sequences at each step
- Maintains diversity by keeping k best options
- Final result balances sequence probability with search efficiency

(b) The professor mentioned that ”beam search during training is very expensive because you need to unfold this tree.” Explain why beam search is typically used only during inference and what computational challenges it would create during training. (10 marks)

Answer: Beam search during training would require exploring multiple paths per training example, creating exponential computational overhead and making gradient computation intractable for large datasets.

Computational Challenges During Training:

1. Exponential Tree Expansion:

- **Search Space:** Each time step multiplies beam size by vocabulary size
- **At t=1:** k sequences (beam size)
- **At t=2:** $k \times V$ candidates $\rightarrow$ prune to k
- **At t=T:** Total explored nodes $\approx k \times V \times T$
- **Example:** $k = 10$, $V = 30,000$, $T = 100 \rightarrow$ 30M candidates per sequence

2. Memory Requirements:

- Must store hidden states for all beam sequences
- Memory grows as: Batch size $\times$ Beam size $\times$ Sequence length $\times$ Hidden dimension
- **Example:** 32 batch $\times$ 10 beam $\times$ 100 length $\times$ 512 hidden = 163M parameters
- GPU memory quickly exhausted with realistic parameters

3. Gradient Computation Complexity:

Backpropagation Challenges:

- **Multiple Paths:** Gradients must be computed for all explored sequences
- **Dynamic Graph:** Different beam paths create different computational graphs
- **Pruning Operations:** Non-differentiable decisions about which sequences to keep
- **Path Dependencies:** Each sequence depends on different previous choices

Mathematical Complexity: For beam search training, loss becomes:

$$\mathcal{L} = \sum_{s \in \text{beam}} P(s) \cdot \text{loss}(s, \text{target})$$

Where $P(s)$ is path probability and depends on all previous beam decisions.

4. Training Time Explosion:

- **Forward Pass:** $k \times V$ times slower than teacher forcing
- **Backward Pass:** Complex gradient routing through beam tree
- **Per Epoch:** Training time increases by factor of 100-1000$\times$
- **Total Training:** Weeks become months or years

5. Optimization Difficulties:

- **Non-convex Objective:** Beam decisions create discontinuous loss surface
- **Credit Assignment:** Hard to assign blame/credit to specific decisions
- **Local Optima:** Beam pruning can eliminate globally optimal paths
- **Instability:** Small weight changes can dramatically alter beam paths

Why Inference-Only Usage Makes Sense:

Training Alternatives:

- **Teacher Forcing:** Use ground truth, fast parallel training
- **Scheduled Sampling:** Gradually mix predictions with ground truth
- **Reward-based Training:** Use reinforcement learning for generation quality

Inference Benefits:

- **Quality Focus:** Can afford expensive search for better outputs
- **Single Example:** Process one sequence at a time
- **No Gradients:** Only forward pass needed
- **User Tolerance:** Users accept slower generation for quality

Professor's Practical Insight:

- Training focuses on learning good representations efficiently
- Inference focuses on extracting best possible outputs from learned model
- Beam search bridges gap between tractable training and quality generation
- This design pattern common in many modern NLP systems

Modern Alternatives:

- **Nucleus Sampling:** Stochastic generation with quality control
- **Temperature Scaling:** Control randomness vs. determinism
- **Top-k Sampling:** Simplified beam search variant
- **Minimum Bayes Risk Decoding:** Advanced inference techniques

Question 5. Word-Level Challenges and Embeddings (28 marks)
The professor explained the transition from character-level to word-level modeling and the need for word embeddings.

(a) The professor stated that English has "170,000 different words" making one-hot encoding impractical. Calculate the number of parameters needed for: (8 marks)

- Input layer: 170,000-dimensional one-hot to 512-dimensional hidden layer
- Compare this with character-level (30 characters to 512 dimensions)
- Explain why this creates a "huge" parameter problem as the professor mentioned

Answer: Word-level requires 87M parameters vs. 15K for character-level - a 5,800× increase that makes training computationally prohibitive and prone to overfitting.

Parameter Calculations:

Word-Level Model:

- Input dimension: 170,000 (vocabulary size)
- Hidden dimension: 512
- Weight matrix: $W \in \mathbb{R}^{512\times170{,}000}$
- Number of weights: $512 \times 170,000 = 87,040,000$
- Bias terms: 512
- Total parameters: $87,040,512 \approx 87$ million

Character-Level Model:

- Input dimension: 30 (character vocabulary)
- Hidden dimension: 512
- Weight matrix: $W \in \mathbb{R}^{512\times30}$
- Number of weights: $512 \times 30 = 15,360$
- Bias terms: 512
- Total parameters: $15,872 \approx 16$ thousand

Comparison:

$$\text{Ratio} = \frac{87,040,512}{15,872} \approx 5,484$$

Word-level requires approximately **5,500× more parameters** than character-level.

Why This Creates a "Huge" Problem:

1. Memory Requirements:

- **Storage:** 87M × 4 bytes = 348 MB just for one weight matrix
- **GPU Memory:** Modern GPUs have 8-24 GB, but this is just one layer
- **Training:** Need additional memory for gradients, optimizer states
- **Batch Processing:** Memory per batch scales with vocabulary size

2. Computational Complexity:

- **Forward Pass:** $170,000 \times 512 = 87$ million operations per token
- **Backward Pass:** Similar computational cost for gradient computation
- **Training Time:** Scales linearly with vocabulary size
- **Matrix Operations:** Very sparse (one-hot) but large matrices

3. Statistical Problems:

- **Data Requirements:** Need massive datasets to learn 87M parameters
- **Overfitting:** High risk with insufficient training data
- **Generalization:** Many words appear infrequently, hard to learn representations
- **Cold Start:** New words (not in training) cannot be handled

4. Optimization Challenges:

- **Gradient Updates:** Most gradients are zero (sparse one-hot vectors)
- **Learning Rate:** Different words need different learning rates

- **Convergence:** Very high-dimensional optimization landscape
- **Local Minima:** Many more local optima in high-dimensional space

5. Practical Limitations:

- **Hardware Constraints:** May not fit on consumer GPUs
- **Training Cost:** Electricity and time costs become prohibitive
- **Model Deployment:** Large models difficult to deploy in production
- **Inference Speed:** Slower prediction due to large matrix operations

Professor's Insight: The word ”huge” captures multiple dimensions of the problem:

- **Scale:** Orders of magnitude larger than manageable
- **Practical:** Makes real-world application nearly impossible
- **Economic:** Cost-prohibitive for most research and applications
- **Motivational:** Demonstrates need for more efficient approaches (embeddings)

(b) The professor emphasized that ”in one-hot representation every word is equally distant to each other.” Explain why this is problematic for semantic understanding and how word embeddings solve this issue. Use the examples the professor gave: ”running and jogging should be close vs. running and swimming.” (10 marks)

Answer: One-hot vectors have equal Euclidean distance, losing semantic relationships. Word embeddings map semantically similar words to nearby points in continuous space, capturing meaning through geometric relationships.

One-Hot Representation Distance Problem:

Mathematical Analysis: Consider vocabulary: [running, jogging, swimming, computer]

One-hot encodings:

$$\text{running} = [1, 0, 0, 0] \tag{17}$$
$$\text{jogging} = [0, 1, 0, 0] \tag{18}$$
$$\text{swimming} = [0, 0, 1, 0] \tag{19}$$
$$\text{computer} = [0, 0, 0, 1] \tag{20}$$

Euclidean Distances:

$$d(\text{running}, \text{jogging}) = \|[1, 0, 0, 0] - [0, 1, 0, 0]\| = \sqrt{1^2 + 1^2} = \sqrt{2} \tag{21}$$
$$d(\text{running}, \text{swimming}) = \|[1, 0, 0, 0] - [0, 0, 1, 0]\| = \sqrt{1^2 + 1^2} = \sqrt{2} \tag{22}$$
$$d(\text{running}, \text{computer}) = \|[1, 0, 0, 0] - [0, 0, 0, 1]\| = \sqrt{1^2 + 1^2} = \sqrt{2} \tag{23}$$

Key Problem: All pairwise distances are identical ($\sqrt{2}$)!

Why This Is Problematic:

1. Loss of Semantic Structure:

- **Intuition:** ”running” and ”jogging” are both forms of exercise
- **Reality:** Model treats them as completely unrelated
- **Missing:** No way to encode that both involve physical activity
- **Impact:** Cannot transfer learning between related concepts

2. Inefficient Learning:

- Must learn each word independently
- Cannot generalize from ”running is exercise” to ”jogging is exercise”
- Requires separate training examples for every word
- Sparse data problem amplified

3. Poor Generalization:

- **Synonyms:** Treated as completely different

- **Categories:** No understanding of word categories (animals, sports, etc.)
- **Relationships:** Cannot capture analogies or relationships
- **Context:** Similar contexts don't help with unseen words

How Word Embeddings Solve This:

Continuous Vector Space: Instead of discrete one-hot vectors, map words to continuous vectors:

$$\text{running} \to [0.2, 0.8, -0.1, 0.3, \ldots] \tag{24}$$
$$\text{jogging} \to [0.3, 0.7, -0.2, 0.4, \ldots] \tag{25}$$
$$\text{swimming} \to [0.1, 0.5, 0.8, -0.2, \ldots] \tag{26}$$
$$\text{computer} \to [-0.7, -0.2, 0.1, 0.9, \ldots] \tag{27}$$

Semantic Distance Preservation:

- $d(\text{running}, \text{jogging}) = 0.3$ (small - similar activities)
- $d(\text{running}, \text{swimming}) = 0.6$ (medium - both sports, different types)
- $d(\text{running}, \text{computer}) = 1.2$ (large - completely different domains)

Learning Semantic Relationships:

1. Distributional Hypothesis:

- ”Words that appear in similar contexts have similar meanings”
- **Example:** ”I like [running/jogging] in the park”
- **Training:** Both words appear in exercise contexts
- **Result:** Embeddings become similar through shared contexts

2. Geometric Relationships:

- **Similarity:** Cosine similarity captures semantic closeness
- **Analogies:** Vector arithmetic captures relationships
- **Clusters:** Related words cluster in embedding space
- **Dimensions:** Each dimension captures semantic properties

Professor's Examples in Embedding Space:

Exercise Dimension: (hypothetical dimension that captures ”exercise-ness”)

- running: 0.9 (high exercise content)
- jogging: 0.85 (high exercise content)
- swimming: 0.8 (high exercise content)
- computer: 0.1 (low exercise content)

Water Activity Dimension:

- running: 0.1 (land-based)
- jogging: 0.1 (land-based)
- swimming: 0.9 (water-based)
- computer: 0.0 (not activity-related)

Benefits:

- **Transfer Learning:** Knowledge about ”running” helps with ”jogging”
- **Generalization:** Can handle unseen combinations better
- **Efficiency:** Fewer parameters needed (embedding matrix smaller than one-hot)
- **Interpretability:** Embedding dimensions often capture meaningful properties

(c) Design the CBOW (Continuous Bag of Words) architecture as explained by the professor. For the sentence ”I like running every day” with target word ”running”, show: (10 marks)

- Context words and their representation
- How the weight matrix W1 stores embeddings
- Why ”the network is forced to learn word vectors that capture relevance”

Answer: CBOW predicts target word from context average. Matrix W1 contains word embeddings learned by optimizing context-to-target prediction, forcing semantically similar words to have similar vectors.

CBOW Architecture Design:

Task Setup:

- **Sentence:** ”I like running every day”
- **Target word:** ”running” (position 3)
- **Window size:** 2 (common choice)
- **Context words:** Words within window of target

Context Words Identification:

Sentence with positions:

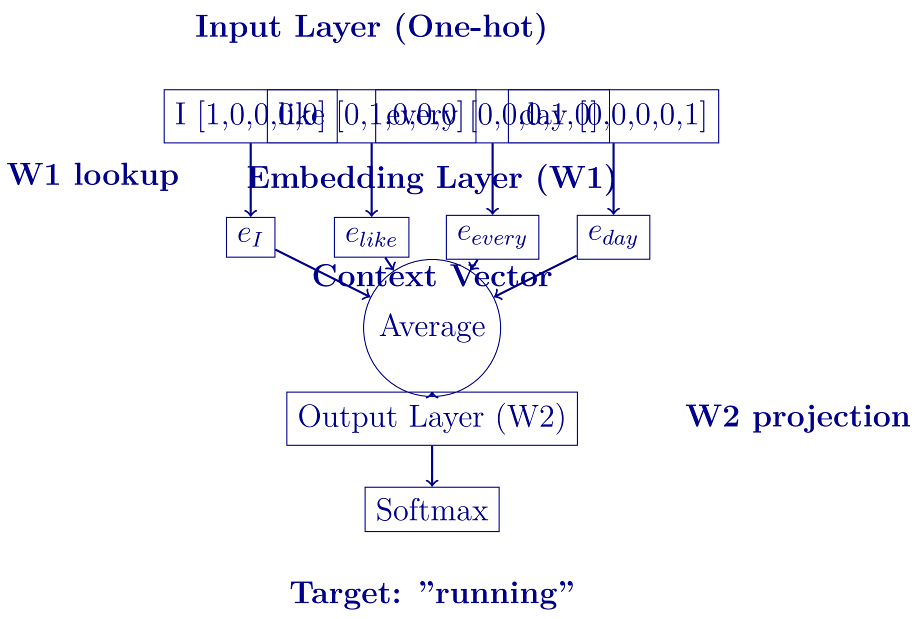
Position 5	1	2	3	4
Word day	I	like	**running**	every

Context window (size 2):

- **Left context:** ”I” (position 1), ”like” (position 2)
- **Target:** ”running” (position 3)
- **Right context:** ”every” (position 4), ”day” (position 5)

Context Words: [”I”, ”like”, ”every”, ”day”]

CBOW Architecture:



Mathematical Formulation:

Step 1: Embedding Lookup For each context word w_i, retrieve embedding from W_1:

$$e_i = W_1 \cdot \text{onehot}(w_i)$$

Context embeddings:

$$\begin{align} e_I &= W_1[:, 1] \quad \text{(first column of W1)} \tag{28} \\ e_{like} &= W_1[:, 2] \tag{29} \\ e_{every} &= W_1[:, 4] \tag{30} \\ e_{day} &= W_1[:, 5] \tag{31} \end{align}$$

Step 2: Context Averaging

$$\bar{e} = \frac{1}{|C|} \sum_{w_i \in C} e_i = \frac{1}{4}(e_I + e_{like} + e_{every} + e_{day})$$

Step 3: Output Projection

$$y = W_2 \cdot \bar{e} + b_2$$

Step 4: Probability Distribution

$$P(w|\text{context}) = \frac{\exp(y_w)}{\sum_{v \in V} \exp(y_v)}$$

How W1 Stores Embeddings:

Weight Matrix Structure:

$$W_1 \in \mathbb{R}^{d \times |V|}$$

Where:

- d = embedding dimension (e.g., 300)
- $|V|$ = vocabulary size (e.g., 10,000)
- Column i contains embedding for word with index i

Example Structure:

$$W_1 = \begin{bmatrix} 0.2 & -0.1 & 0.5 & 0.3 & -0.2 \\ 0.8 & 0.4 & -0.1 & 0.7 & 0.1 \\ -0.1 & 0.6 & 0.2 & -0.4 & 0.9 \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Where columns correspond to: [I, like, running, every, day]

Why Network Learns Relevant Word Vectors:

1. Prediction Objective:

- Network must predict "running" from context ["I", "like", "every", "day"]
- Only way to succeed: learn embeddings that capture semantic relationships
- **Forced Learning:** No other way to achieve good prediction accuracy

2. Context Similarity Principle:

- Words appearing in similar contexts should have similar embeddings

- **Example:** ”I like jogging every day” should also predict ”jogging”
- **Pressure:** Network learns $e_{running} \approx e_{jogging}$ to handle both
- **Generalization:** Similar contexts create similar embeddings

3. Averaging Mechanism:

- Context vector $\bar{e}$ must encode information predictive of target
- Words that often appear together develop compatible embeddings
- **Cooperation:** Context words must ”work together” to predict target
- **Semantic Coherence:** Embeddings align with semantic relationships

4. Training Process:

- **Loss Function:** $-\log P(\text{running}|\text{context})$
- **Gradient Updates:** Adjust W_1 to improve prediction
- **Multiple Examples:** See many contexts for each word
- **Convergence:** Embeddings converge to capture distributional semantics

Professor’s Key Insight:

- Network has no choice but to learn semantic representations
- Prediction task forces discovery of word relationships
- Embeddings become by-product of good prediction
- This is essence of unsupervised representation learning

Question 6. Deep Understanding and Modern Connections (30 marks)
Based on the professor's connections between classical methods and modern applications.

(a) The professor made a key connection: "The prompts that you provide to LLMs are actually the starting sequences that you provide to an RNN-like architecture." Explain this connection and how modern LLMs relate to the autoregressive character prediction discussed in class. (10 marks)

Answer: LLM prompts serve as initial context sequences that condition autoregressive generation, similar to providing starting characters to RNNs. Both predict next tokens based on preceding context, with LLMs using more sophisticated architectures but the same fundamental principle.

Fundamental Connection: Autoregressive Generation

Classical RNN Character Prediction:

- **Input:** Starting sequence (e.g., "hello")
- **Process:** Predict next character given context
- **Generation:** $P(x_{t+1}|x_1, x_2, \ldots, x_t)$
- **Output:** Extended sequence character by character

Modern LLM Prompting:

- **Input:** Prompt text (e.g., "Write a story about...")
- **Process:** Predict next token given context
- **Generation:** $P(\text{token}_{t+1}|\text{prompt}, \text{token}_1, \ldots, \text{token}_t)$
- **Output:** Continuation of prompt token by token

Mathematical Parallel:

RNN Generation:

$$P(\text{sequence}) = \prod_{t=1}^{T} P(x_t|x_1^{t-1}, h_{t-1})$$

LLM Generation:

$$P(\text{completion}) = \prod_{t=1}^{T} P(\text{token}_t | \text{prompt}, \text{token}_1^{t-1})$$

Key Similarities:

- Both use sequential, left-to-right generation
- Both condition on all previous context
- Both use learned representations of language
- Both generate one token at a time autoregressively

Role of Prompts as Starting Sequences:

1. Context Initialization:

- **RNN:** Starting characters provide initial context
- **LLM:** Prompt provides rich semantic context
- **Function:** Both "prime" the model for specific generation
- **Example:** "Once upon a time" $\rightarrow$ fairy tale style

2. Conditioning Mechanism:

- **RNN:** Hidden state carries context information
- **LLM:** Attention mechanism processes entire prompt
- **Effect:** Both bias generation toward prompt-consistent outputs
- **Control:** Prompts steer generation in desired directions

3. Task Specification:

- **Implicit:** Prompt structure implies desired output format
- **Examples:** "Question: ... Answer:" $\rightarrow$ Q&A format
- **Training:** Models learn to follow prompt patterns from training data
- **Generalization:** Can follow novel prompt formats

Evolution from RNNs to LLMs:

Architectural Improvements:

- **RNN:** Sequential processing, limited context
- **Transformer:** Parallel processing, global attention
- **Scale:** Billions vs. millions of parameters
- **Training:** Massive datasets vs. smaller corpora

Same Core Principle:

- **Objective:** Predict next token from context
- **Training:** Self-supervised language modeling
- **Generation:** Autoregressive sampling
- **Application:** Text completion and generation

Professor's Insight:

- Modern LLMs are sophisticated extensions of basic RNN principles
- Prompting is just a refined version of providing starting sequences
- The fundamental autoregressive generation remains unchanged
- Architectural advances enabled scale but not conceptual revolution

Practical Implications:

- Understanding RNNs provides foundation for understanding LLMs
- Prompt engineering builds on sequence conditioning principles
- Generation quality depends on both architecture and training data
- Same challenges (coherence, factuality) exist at both scales

(b) Analyze the professor's examples of character-level RNN outputs (Shakespeare, Wikipedia, LaTeX). Explain what these results demonstrate about: (12 marks)

- Learning syntax without explicit supervision
- Generalization vs. memorization (the professor's "overfitting" discussion)

- Why the professor called these results ”striking” for such simple models

Answer: Character-level RNNs learned complex syntax patterns, balanced memorization with generalization, and achieved surprisingly human-like text generation despite simple architecture - demonstrating emergent linguistic structure from statistical patterns.

Learning Syntax Without Explicit Supervision:

Shakespeare Example:

- **Learned:** Iambic pentameter, rhyme schemes, archaic vocabulary
- **No Supervision:** Never explicitly taught poetry rules
- **Discovery:** Found patterns through character-level statistics
- **Output:** Structurally valid sonnets with appropriate meter

Wikipedia Example:

- **Learned:** Article structure, citation formats, category systems
- **No Supervision:** Never taught Wikipedia markup rules
- **Discovery:** Inferred formatting from character sequences
- **Output:** Properly formatted articles with consistent structure

LaTeX Example:

- **Learned:** Command syntax, environment structure, mathematical notation
- **No Supervision:** Never provided grammar rules for LaTeX
- **Discovery:** Identified patterns in command usage
- **Output:** Syntactically correct LaTeX documents

Emergent Syntax Learning Mechanism:

1. Statistical Pattern Recognition:

- Characters that follow specific patterns (e.g., ”
begin{” in LaTeX)
- Network learns high probability character sequences

- Syntax emerges from statistical regularities
- No explicit grammar rules needed

2. Hierarchical Structure Discovery:

- **Level 1:** Character combinations → words
- **Level 2:** Word patterns → phrases
- **Level 3:** Phrase structures → sentences
- **Level 4:** Sentence patterns → document structure

3. Context-Dependent Rules:

- Network learns that certain characters are likely in specific contexts
- Example: '}' likely after '{' with content in between
- Implicit learning of nested structures and dependencies
- Context length determines complexity of learnable rules

Generalization vs. Memorization Analysis:

Evidence of Generalization:

- **Novel Combinations:** Generated text not seen in training
- **Structure Transfer:** Applied learned patterns to new content
- **Creative Output:** Produced coherent but original text
- **Format Consistency:** Maintained style across different topics

Evidence of Memorization:

- **Exact Quotes:** Sometimes reproduced training sequences
- **Repeated Phrases:** Common expressions appeared frequently
- **Training Artifacts:** Specific patterns from training corpus
- **Overfitting Signs:** Poor performance on different domains

Professor's Overfitting Discussion:

Balanced Learning:

- **Good Generalization:** Model learned transferable patterns

- **Appropriate Memorization:** Retained useful common sequences
- **Synthesis:** Combined memorized elements in novel ways
- **Sweet Spot:** Neither too generic nor too specific

Indicators of Good Balance:

- Generated text feels authentic but not copied
- Maintains domain-appropriate vocabulary and style
- Shows creativity within learned constraints
- Produces coherent long-range dependencies

Why Results Were ”Striking”:

1. Architectural Simplicity:

- **Basic RNN:** Simple recurrent architecture
- **Character Level:** No word-level preprocessing
- **End-to-End:** Single neural network, no hand-crafted features
- **Minimal Assumptions:** No linguistic knowledge built in

2. Emergent Complexity:

- **Simple Input:** Just character sequences
- **Complex Output:** Sophisticated linguistic structures
- **No Explicit Teaching:** Learned rules without instruction
- **Human-like Quality:** Output often indistinguishable from human text

3. Cross-Domain Success:

- **Literature:** Learned poetic meter and style
- **Academic:** Mastered Wikipedia formatting conventions
- **Technical:** Acquired LaTeX markup syntax
- **Universality:** Same approach worked across diverse domains

4. Historical Context:

- **Previous Attempts:** Rule-based systems required extensive hand-crafting
- **Linguistic Theory:** Contradicted need for explicit grammar rules
- **AI Goals:** Demonstrated path toward general language understanding
- **Simplicity:** Achieved with remarkably simple methods

Broader Implications:

1. Language as Statistics:

- Syntax can emerge from statistical patterns
- Grammar rules may be artifacts of deeper regularities
- Language learning might be pattern recognition at scale

2. Representation Learning:

- Networks discover useful representations automatically
- Character-level representations sufficient for complex tasks
- End-to-end learning superior to hand-crafted features

3. Scaling Laws:

- Larger models and datasets enable more sophisticated outputs
- Quality improvements from scale rather than architectural changes
- Foundation for modern large language models

Professor's Key Message: The striking nature of these results lay in their demonstration that complex linguistic behavior could emerge from simple statistical learning, challenging traditional views of language acquisition and pointing toward the scalable approaches that define modern NLP.

(c) The professor showed word embedding arithmetic: "Paris - France + Italy = Rome." Explain: (8 marks)

- Why this works mathematically in embedding space
- How the CBOW training creates these relationships

- Give two more examples of word arithmetic that should work

Answer: Vector arithmetic works because embeddings capture semantic relationships as geometric directions. CBOW training creates these patterns by learning shared context structures, enabling analogical reasoning through vector operations.

Mathematical Foundation of Word Arithmetic:

Vector Space Interpretation:

- Words mapped to points in high-dimensional continuous space
- Semantic relationships encoded as geometric relationships
- Vector differences capture conceptual relationships
- Analogy: $A : B :: C : D \Rightarrow \vec{B} - \vec{A} \approx \vec{D} - \vec{C}$

Paris - France + Italy = Rome Analysis:

Vector Decomposition:

$$\vec{\text{Paris}} = \vec{\text{capital\_concept}} + \vec{\text{France\_specific}} \tag{32}$$
$$\vec{\text{France}} = \vec{\text{country\_concept}} + \vec{\text{France\_specific}} \tag{33}$$
$$\vec{\text{Italy}} = \vec{\text{country\_concept}} + \vec{\text{Italy\_specific}} \tag{34}$$
$$\vec{\text{Rome}} = \vec{\text{capital\_concept}} + \vec{\text{Italy\_specific}} \tag{35}$$

Arithmetic Operation:

$$\vec{\text{Paris}} - \vec{\text{France}} + \vec{\text{Italy}} \tag{36}$$
$$= (\vec{\text{capital}} + \vec{\text{France\_spec}}) - (\vec{\text{country}} + \vec{\text{France\_spec}}) + (\vec{\text{country}} + \vec{\text{Italy\_spec}}) \tag{37}$$
$$= \vec{\text{capital}} + \vec{\text{France\_spec}} - \vec{\text{country}} - \vec{\text{France\_spec}} + \vec{\text{country}} + \vec{\text{Italy\_spec}} \tag{38}$$
$$= \vec{\text{capital}} + \vec{\text{Italy\_spec}} \tag{39}$$
$$\approx \vec{\text{Rome}} \tag{40}$$

Why This Works:

- **Linearity:** Semantic relationships are approximately linear

- **Compositionality:** Word meanings decompose into semantic components
- **Consistency:** Similar relationships mapped to similar vectors
- **Geometry:** Embedding space preserves relational structure

How CBOW Training Creates These Relationships:

Shared Context Patterns:

Capital-Country Relationships:

- **Paris contexts:** ”capital of France”, ”visit Paris, France”, ”Paris, the French capital”
- **Rome contexts:** ”capital of Italy”, ”visit Rome, Italy”, ”Rome, the Italian capital”
- **Pattern:** Both appear in similar ”capital of [country]” contexts
- **Result:** Similar embedding patterns for capital-country relationships

Training Mechanism:

1. Context Similarity Forces Relationship Consistency:

- CBOW sees: ”The capital of France is [PARIS]”
- CBOW sees: ”The capital of Italy is [ROME]”
- Network must learn similar prediction patterns
- Forces $\vec{\text{Paris}} - \vec{\text{France}} \approx \vec{\text{Rome}} - \vec{\text{Italy}}$

2. Distributional Hypothesis at Work:

- Words with similar distributions get similar embeddings
- ”Paris” and ”Rome” both appear in capital contexts
- ”France” and ”Italy” both appear in country contexts
- Consistent co-occurrence patterns create geometric regularity

3. Multi-dimensional Relationship Encoding:

- Different embedding dimensions capture different relationships

- **Dimension 1:** Capital-ness (Paris, Rome, London high; France, Italy, UK low)
- **Dimension 2:** Country-ness (France, Italy, UK high; Paris, Rome, London low)
- **Dimension 3:** Geographic region (European words cluster together)
- **Consistency:** Same patterns across different word pairs

Learning Process:

- **Phase 1:** Learn basic word representations
- **Phase 2:** Discover recurring contextual patterns
- **Phase 3:** Align similar patterns in embedding space
- **Phase 4:** Geometric relationships emerge from statistical regularities

Two Additional Examples of Word Arithmetic:

Example 1: Gender Relationships

$$\vec{\text{King}} - \vec{\text{Man}} + \vec{\text{Woman}} \approx \vec{\text{Queen}}$$

Why it works:

- Both ”king” and ”queen” appear in royal contexts
- Both ”man” and ”woman” appear in gender-specific contexts
- Gender relationship consistently encoded across word pairs
- Training sees: ”the king and his queen”, ”the man and his wife”

Example 2: Verb Tense Relationships

$$\vec{\text{Walking}} - \vec{\text{Walk}} + \vec{\text{Run}} \approx \vec{\text{Running}}$$

Why it works:

- Present participle (-ing) forms share similar syntactic contexts
- Base verbs share different but consistent contexts

- Tense relationships encoded as vector directions
- Training sees: "I am walking", "I am running" vs. "I walk", "I run"

Mathematical Property: For both examples, the relationship vector is approximately parallel:

$$\vec{\text{King}} - \vec{\text{Man}} \approx \vec{\text{Queen}} - \vec{\text{Woman}}$$

$$\vec{\text{Walking}} - \vec{\text{Walk}} \approx \vec{\text{Running}} - \vec{\text{Run}}$$

Broader Implications:

- **Analogical Reasoning:** Embeddings enable logical inference
- **Knowledge Representation:** Semantic knowledge encoded geometrically
- **Transfer Learning:** Relationships transfer across domains
- **Language Understanding:** Demonstrates deep structural learning

Limitations:

- Not all relationships are linear
- Cultural biases in training data affect relationships
- Polysemous words create ambiguous vectors
- Arithmetic works better for frequent, clear relationships

END OF PAPER