MIDDLE EAST TECHNICAL UNIVERSITY

SEMESTER I EXAMINATION 2024-2025

CENG 403 – Deep Learning - CNN Architectures & RNN Introduction (University Sources) - ANSWERED

January 2025 TIME ALLOWED: 3 HOURS

INSTRUCTIONS TO CANDIDATES

1. This examination paper contains **SEVEN (7)** questions and comprises **TEN (10)** 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 all mathematical derivations clearly with proper notation.
6. For architectural diagrams, draw clear and labeled components.
7. Calculate all requested parameters and show intermediate steps.
8. Explain computational complexity where requested.

Question 1. CNN Architectural Fundamentals and Calculations (25 marks)
Based on D2L.ai and university CNN course materials covering computational aspects.

(a) For a convolutional layer with the following specifications, calculate the output dimensions and number of parameters: (12 marks)

- Input: 224×224×3 RGB image
- 64 filters of size 7×7
- Stride: 2
- Padding: 3
- Bias terms included

Show all calculations including:

- Output height and width
- Total number of parameters
- Memory requirements for storing activations

Answer: Output dimensions: 112×112×64; Parameters: 9,472; Memory: 3.1 MB

Step-by-Step Calculations:

1. Output Dimensions: Using the formula: Output size $= \frac{\text{Input size}+2\times\text{Padding}-\text{Filter size}}{\text{Stride}}+1$

Width calculation:

$$W_{out} = \frac{224 + 2 \times 3 - 7}{2}+1 = \frac{224 + 6 - 7}{2}+1 = \frac{223}{2}+1 = 111.5+1 = 112$$

Height calculation:

$$H_{out} = \frac{224 + 2 \times 3 - 7}{2} + 1 = 112 \text{ (same as width)}$$

Output channels: 64 (number of filters)

Final output dimensions: $112 \times 112 \times 64$

2. Parameter Count:

- Weight parameters per filter: $7 \times 7 \times 3 = 147$ (filter size $\times$ input channels)
- Total weight parameters: $147 \times 64 = 9,408$ (per filter $\times$ number of filters)
- Bias parameters: 64 (one per filter)
- Total parameters: $9,408 + 64 = 9,472$

3. Memory Requirements: Assuming 32-bit floating point (4 bytes per value):

- Activation memory: $112 \times 112 \times 64 \times 4 = 3,211,264$ bytes ≈ 3.1 MB
- Parameter memory: $9,472 \times 4 = 37,888$ bytes ≈ 37 KB
- Total memory for this layer: ≈ 3.1 MB (dominated by activations)

(b) Explain the difference between "Valid Padding" and "Same Padding" in CNNs. For a 12×12 input with a 3×3 filter and stride 1: (8 marks)

- Calculate output size with valid padding
- Calculate padding needed for same padding
- Discuss trade-offs between the two approaches

Answer: Valid padding: 10×10 output; Same padding: requires 1-pixel padding for 12×12 output

Valid Padding (No Padding):

- Padding = 0
- Output size: $\frac{12+2(0)-3}{1} + 1 = \frac{9}{1} + 1 = 10$
- Output dimensions: 10×10
- Filter only applied where it completely fits within input

Same Padding:

- Goal: Output size = Input size when stride = 1
- Required output: 12×12
- To achieve this: $12 = \frac{12+2p-3}{1} + 1$

- Solving: $11 = 12 + 2p - 3$, so $2p = 2$, thus $p = 1$
- Padding needed: 1 pixel on each side
- Output dimensions: 12×12

Trade-offs:

Valid Padding Advantages:

- No artificial padding values introduced
- Faster computation (smaller output)
- Clear semantic meaning (only real data processed)

Valid Padding Disadvantages:

- Output shrinks with each layer
- Edge information lost progressively
- Limited network depth before spatial dimensions become too small

Same Padding Advantages:

- Preserves spatial dimensions
- Enables deeper networks
- Better preservation of edge information
- More flexible architecture design

Same Padding Disadvantages:

- Introduces artificial zero values
- Slightly more computation
- May learn biases related to padding

(c) Compare parameter sharing in CNNs versus fully connected networks. For an image of size 256×256×3, calculate the number of parameters needed for: (5 marks)

- First layer as fully connected (to 512 units)
- First layer as convolutional (64 filters, 5×5)

- Explain the computational advantage

Answer: Fully connected: 100M+ parameters; Convolutional: 4,864 parameters - dramatic reduction through parameter sharing

Fully Connected Layer:

- Input neurons: $256 \times 256 \times 3 = 196,608$
- Output neurons: 512
- Weight parameters: $196,608 \times 512 = 100,663,296$
- Bias parameters: 512
- Total parameters: $100,663,808 \approx 100.7$ million

Convolutional Layer:

- Filter size: $5 \times 5 \times 3 = 75$ weights per filter
- Number of filters: 64
- Weight parameters: $75 \times 64 = 4,800$
- Bias parameters: 64 (one per filter)
- Total parameters: $4,864$

Parameter Reduction:

$$\text{Reduction factor} = \frac{100,663,808}{4,864} \approx 20,690$$

Computational Advantages of CNNs:

1. Parameter Efficiency:

- Same filter applied across entire spatial dimension
- Dramatic reduction in parameters ($\sim 20,000\times$ in this example)
- Scalable to larger images without parameter explosion

2. Translation Invariance:

- Features detected regardless of spatial location
- Natural for image processing tasks

- Reduces overfitting through shared representations

3. Memory and Computational Benefits:

- Fewer parameters to store and update
- More efficient gradient computations
- Better cache locality in hardware implementations
- Enables deployment on resource-constrained devices

Question 2. ResNet Architecture and Skip Connections (30 marks)

Based on university deep learning courses and D2L.ai educational content.

(a) Explain the mathematical foundation of residual learning. Given a target function $H(x)$, derive why learning the residual mapping $F(x) = H(x) - x$ is easier than learning $H(x)$ directly. (10 marks)

Include discussion of:

- Identity function learning difficulty
- Gradient flow advantages
- Why zero functions are easier to learn

Answer: Learning residual $F(x) = H(x) - x$ is easier because when optimal mapping is close to identity, $F(x) \approx 0$, which is easier to learn than directly approximating identity through multiple nonlinear layers.

Mathematical Foundation:

Problem with Direct Learning: Given target function $H(x)$, traditional networks must learn:

$$y = H(x)$$

When optimal function is close to identity ($H(x) \approx x$), networks struggle because:

- Identity function is hard to approximate through stacked nonlinearities
- Multiple ReLU + linear layers cannot easily represent $f(x) = x$
- Small deviations from identity are difficult to learn precisely

Residual Learning Approach: Instead of learning $H(x)$ directly, learn residual:

$$F(x) = H(x) - x$$

Then the output becomes:

$$y = F(x) + x = H(x) - x + x = H(x)$$

Why This is Easier:

1. Zero Function Learning:

- When $H(x) \approx x$, then $F(x) \approx 0$
- Zero function is much easier to learn than identity
- Network can achieve good performance with $F(x) = 0$ (weights near zero)
- Any improvement requires learning only the *deviation* from identity

2. Gradient Flow Advantages: Gradient computation:

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x} + \frac{\partial x}{\partial x} = \frac{\partial F(x)}{\partial x} + 1$$

- The "+1" term ensures gradient never vanishes
- Even if $\frac{\partial F(x)}{\partial x} \to 0$, gradient magnitude remains ≥ 1
- Provides highway for gradient flow in deep networks

3. Optimization Landscape:

- Identity mapping provides a strong baseline
- Network can only improve from this baseline
- Smoother loss surface around identity manifold
- Better conditioning for optimization algorithms

(b) Design and draw a complete ResNet basic block showing: (12 marks)

- Two 3×3 convolutional layers
- Skip connection implementation
- Activation function placement
- Dimension matching considerations

Compare this with a bottleneck block design (1×1, 3×3, 1×1 structure).

- Gradient computation through skip connections

Comparison with traditional deep networks

Why identity mappings preserve gradient magnitude

Answer: ResNet avoids vanishing gradients because skip connections provide identity paths where $\frac{\partial y}{\partial x} = 1 + \frac{\partial F(x)}{\partial x}$, ensuring gradient magnitude never falls below 1, unlike vanilla networks where gradients multiply through all layers.

Gradient Flow in Vanilla Networks:

For a 50-layer network: $y = f_{50}(f_{49}(\ldots f_2(f_1(x))\ldots))$

Gradient computation using chain rule:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \prod_{i=1}^{50} \frac{\partial f_i}{\partial x_i}$$

Problems:

- Each $\frac{\partial f_i}{\partial x_i}$ typically < 1 for saturating activations
- Product of 50 terms < 1 leads to exponential decay
- For sigmoid: $\max(\sigma'(x)) = 0.25$, so gradient can shrink by factor $(0.25)^{50} \approx 10^{-30}$

Gradient Flow in ResNet:

For ResNet block: $y = F(x) + x$

Gradient computation:

$$\frac{\partial y}{\partial x} = \frac{\partial F(x)}{\partial x} + \frac{\partial x}{\partial x} = \frac{\partial F(x)}{\partial x} + 1$$

Through entire network:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \prod_{i=1}^{N} \left(\frac{\partial F_i(x_i)}{\partial x_i} + 1 \right)$$

Key Advantages:

1. Gradient Lower Bound:

- Even if $\frac{\partial F(x)}{\partial x} \to 0$, we have $\frac{\partial y}{\partial x} = 1$
- Gradient magnitude is always ≥ 1 for each block

- No exponential decay through depth

2. Multiple Gradient Paths:

- Direct path through skip connections (always magnitude 1)
- Processed paths through weight layers
- Gradients can flow through whichever path is most effective

3. Mathematical Proof of Non-Vanishing: For single ResNet block:

$$\left|\frac{\partial y}{\partial x}\right| = \left|\frac{\partial F(x)}{\partial x} + 1\right| \geq \left|1 - \left|\frac{\partial F(x)}{\partial x}\right|\right|$$

As long as $\left|\frac{\partial F(x)}{\partial x}\right| < 1$, we have $\left|\frac{\partial y}{\partial x}\right| > 0$

Comparison Summary:

- **Vanilla:** Gradient magnitude exponentially decreases with depth
- **ResNet:** Gradient magnitude maintained through identity paths
- **Result:** ResNet enables training of networks with 1000+ layers

Question 3. DenseNet and Advanced CNN Architectures (22 marks)

Based on modern CNN architecture research and educational materials.

(a) Compare DenseNet with ResNet architectures. Explain the key difference: (8 marks)

$$\text{ResNet: } x_l = H_l(x_{l-1}) + x_{l-1}$$

$$\text{DenseNet: } x_l = H_l([x_0, x_1, \ldots, x_{l-1}])$$

Discuss advantages and disadvantages of each approach.

Answer: ResNet uses element-wise addition of skip connections while DenseNet concatenates feature maps from all previous layers, creating denser connectivity but higher memory requirements.

Architectural Differences:

ResNet Approach:

- $x_l = H_l(x_{l-1}) + x_{l-1}$ (element-wise addition)
- Each layer connects to previous layer and skips one layer
- Fixed number of channels throughout residual blocks
- Additive information combination

DenseNet Approach:

- $x_l = H_l([x_0, x_1, \ldots, x_{l-1}])$ (concatenation)
- Each layer connects to ALL previous layers
- Growing number of input channels: $k_0 + l \times k$ channels for layer l
- Concatenative information combination

Advantages and Disadvantages:

	ResNet	**DenseNet**
Advantages	• Lower memory usage • Faster training/inference • Simpler implementation • Good gradient flow • Scalable to very deep networks	• Better feature reuse • Stronger gradient flow • More parameter efficient • Better performance on small datasets • Implicit regularization
Disadvantages	• Information loss through addition • Less feature reuse • May require more parameters	• High memory requirements • Slower due to concatenations • Complex implementation • Quadratic growth in connections

When to Use Each:

- **ResNet:** Large-scale problems, limited memory, need for speed
- **DenseNet:** Small datasets, parameter efficiency priority, research settings

(b) For a DenseNet block with 4 layers, each producing 12 feature maps (growth rate k=12), and input of 64 channels: (10 marks)

- Calculate the number of input channels for each layer
- Compute total memory requirements for concatenations
- Explain how transition layers reduce dimensionality
- Calculate parameters for 1×1 conv in transition layer

Answer: Layer inputs: 64, 76, 88, 100 channels; Memory grows quadratically; Transition layers use 1×1 conv for dimensionality reduction

Channel Calculation for Each Layer:

Given: Initial channels = 64, Growth rate $k = 12$

- **Layer 1:** Input channels = 64 (initial)
- **Layer 2:** Input channels = $64 + 12 = 76$ (initial + layer 1 output)
- **Layer 3:** Input channels = $64 + 12 + 12 = 88$ (initial + layer 1 + layer 2)
- **Layer 4:** Input channels = $64 + 12 + 12 + 12 = 100$ (all previous layers)

General Formula: Layer l has $k_0 + (l-1) \times k$ input channels

Memory Requirements for Concatenations:

Assuming feature maps of size $H \times W$ and 32-bit floats:

- **After Layer 1:** $(64 + 12) \times H \times W \times 4 = 76HW \times 4$ bytes
- **After Layer 2:** $(64 + 12 + 12) \times H \times W \times 4 = 88HW \times 4$ bytes
- **After Layer 3:** $(64 + 12 + 12 + 12) \times H \times W \times 4 = 100HW \times 4$ bytes
- **After Layer 4:** $(64+12+12+12+12) \times H \times W \times 4 = 112HW \times 4$ bytes

Total Memory: $(76 + 88 + 100 + 112) \times HW \times 4 = 376HW \times 4$ bytes

Transition Layers:

Purpose: Reduce the number of feature maps to control model complexity

Structure:

- **1×1 Convolution:** Reduces number of channels (typically by factor of 2)

- **Average Pooling:** Reduces spatial dimensions (typically 2×2 with stride 2)

Example Transition Layer:

- Input: 112 channels from dense block
- 1×1 Conv: 112 → 56 channels (compression factor = 0.5)
- 2×2 Average Pooling: $(H, W) \rightarrow (H/2, W/2)$

Parameter Calculation for 1×1 Transition Conv:

- Input channels: 112
- Output channels: 56 (with compression factor 0.5)
- Filter size: 1×1
- Weight parameters: $1 \times 1 \times 112 \times 56 = 6,272$
- Bias parameters: 56
- Total parameters: $6,272 + 56 = 6,328$

Benefits of Transition Layers:

- Control computational complexity
- Reduce memory requirements
- Maintain model efficiency as network grows
- Enable deeper DenseNet architectures

(c) Design a Highway Network gate mechanism. Write the mathematical equations for: (4 marks)

$$y = H(x, W_H) \cdot T(x, W_T) + x \cdot C(x, W_C)$$

Explain how this differs from standard residual connections.

Answer: Highway networks use learnable gates $T(x)$ and $C(x) = 1 - T(x)$ to control information flow, unlike ResNet's fixed additive skip connections.

Highway Network Mathematical Formulation:

Complete Equations:

$$T(x, W_T) = \sigma(W_T x + b_T) \quad \text{(Transform gate)} \tag{1}$$
$$C(x, W_C) = 1 - T(x, W_T) \quad \text{(Carry gate)} \tag{2}$$
$$H(x, W_H) = \text{ReLU}(W_H x + b_H) \quad \text{(Transform function)} \tag{3}$$
$$y = H(x, W_H) \cdot T(x, W_T) + x \cdot C(x, W_C) \tag{4}$$

Where:

- σ is the sigmoid function ensuring $T(x) \in [0, 1]$
- $T(x)$ controls how much transformed information passes through
- $C(x) = 1 - T(x)$ controls how much original information passes through
- $T(x) + C(x) = 1$ ensures information conservation

Differences from Standard ResNet:

Highway Networks	**ResNet**
$y = H(x) \cdot T(x) + x \cdot C(x)$	$y = H(x) + x$
Learnable, adaptive gates	Fixed additive skip
$T(x), C(x)$ are input-dependent	Skip connection always active
Multiplicative gating	Additive skip connection
Can completely block paths	Both paths always contribute
More parameters (gate weights)	Fewer parameters
Adaptive information routing	Fixed information combination

Key Advantages of Highway Networks:

- **Adaptive Control:** Gates learn when to transform vs. preserve information
- **Flexible Routing:** Can dynamically choose optimal information paths
- **Input-Dependent Behavior:** Different inputs may use different paths

Why ResNet Became More Popular:

- **Simplicity:** Fixed addition is easier to implement and understand
- **Efficiency:** Fewer parameters and computations
- **Reliability:** Less complex optimization landscape
- **Effectiveness:** Achieved excellent results with simpler approach

Question 4. CNN Optimization and Efficiency (20 marks)
Based on practical CNN implementation and optimization techniques.

(a) Analyze binary neural networks for edge deployment. Given a standard CNN with: (10 marks)

- 10M parameters (32-bit floats)
- 50 GFLOPS for inference

Calculate:

- Memory reduction with binary weights
- Speed improvement estimates
- Accuracy trade-offs to consider
- When binary networks are appropriate

Answer: Memory reduction: 32×; Speed improvement: 10-30×; Accuracy loss: 5-15%; Suitable for edge devices and real-time applications.

Memory Reduction Calculation:

Standard CNN:

- Parameters: 10M × 32 bits = 320M bits = 40 MB
- Each parameter: 32-bit float (4 bytes)

Binary CNN:

- Parameters: 10M × 1 bit = 10M bits = 1.25 MB
- Each parameter: 1 bit (±1 values)

Memory Reduction: $\frac{40 \text{ MB}}{1.25 \text{ MB}} = 32$

Speed Improvement Analysis:

Computational Changes:

- **Standard:** Floating-point multiplications + additions
- **Binary:** XNOR operations + bit counting (popcount)

Operation Comparison:

- FP32 multiplication: 100-200 CPU cycles
- XNOR + popcount: 1-2 CPU cycles
- Theoretical speedup: 50-200×
- Practical speedup: 10-30× (due to memory access, other operations)

Estimated Performance:

- Original: 50 GFLOPS
- Binary (conservative): $50 \times 10 = 500$ GFLOPS equivalent
- Binary (optimistic): $50 \times 30 = 1500$ GFLOPS equivalent

Accuracy Trade-offs:

Typical Accuracy Loss:

- **CIFAR-10:** 2-5% accuracy drop
- **ImageNet:** 10-15% accuracy drop
- **Simple tasks:** Minimal impact
- **Complex tasks:** Significant degradation

Factors Affecting Accuracy:

- Task complexity
- Network architecture
- Training methodology
- Binary quantization scheme

When Binary Networks are Appropriate:

Suitable Applications:

- **Edge Computing:** IoT devices, mobile phones
- **Real-time Processing:** Video analysis, autonomous systems
- **Energy-Constrained:** Battery-powered devices

- **Simple Tasks:** Object detection, basic classification
- **Privacy-Critical:** On-device processing requirements

Unsuitable Applications:

- **High-Accuracy Required:** Medical diagnosis, scientific computing
- **Complex Tasks:** Fine-grained classification, segmentation
- **Research Applications:** Where accuracy is paramount
- **Cloud Computing:** Where resources are abundant

(b) Compare different normalization strategies in deep CNNs: (10 marks)

- Batch Normalization: benefits and limitations
- Why BatchNorm helps in very deep networks (10,000+ layers)
- Relationship between BatchNorm and gradient stability
- Alternative normalization methods

Answer: BatchNorm normalizes layer inputs to stabilize training, enabling very deep networks by maintaining gradient flow and reducing internal covariate shift, but has limitations with small batches and inference.

Batch Normalization Fundamentals:

Mathematical Formulation: For mini-batch $\mathcal{B} = \{x_1, x_2, \ldots, x_m\}$:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{5}$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \tag{6}$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{7}$$

$$y_i = \gamma \hat{x}_i + \beta \tag{8}$$

Benefits of Batch Normalization:

1. Gradient Stability:

- Prevents gradient vanishing/exploding
- Normalizes gradients across layers
- Enables higher learning rates

2. Reduced Internal Covariate Shift:

- Stabilizes input distributions to each layer
- Reduces dependence on careful weight initialization
- Accelerates training convergence

3. Regularization Effect:

- Adds noise through batch statistics
- Reduces overfitting
- Can reduce need for Dropout

Why BatchNorm Enables Very Deep Networks (10,000+ layers):

1. Gradient Flow Preservation:

- Maintains unit variance across layers
- Prevents gradient magnitude from shrinking/growing exponentially
- Each layer receives well-conditioned gradients

2. Activation Distribution Control:

- Keeps activations in linear region of activation functions
- Prevents saturation that causes gradient vanishing
- Maintains expressiveness throughout the network

3. Decoupled Layer Dependencies:

- Reduces interdependence between layer parameters
- Enables more independent layer-wise learning
- Stabilizes very deep optimization landscapes

Relationship Between BatchNorm and Gradient Stability:

Gradient Flow Analysis: Without BatchNorm: $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot W$ With BatchNorm: $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot \frac{\gamma}{\sqrt{\sigma^2+\epsilon}}$

Key Improvements:

- Gradient magnitude controlled by γ (learnable)
- Variance normalization prevents explosion/vanishing
- More stable gradient flow enables deeper architectures

Limitations of Batch Normalization:

1. Batch Size Dependency:

- Poor performance with small batches
- Batch statistics become unreliable
- Different behavior during training vs. inference

2. Sequential Data Issues:

- Difficulty with RNNs and variable-length sequences
- Temporal dependencies can be disrupted

3. Inference Complications:

- Requires running statistics from training
- Domain shift can affect normalization statistics

Alternative Normalization Methods:

1. Layer Normalization:

- Normalizes across feature dimension instead of batch
- Better for RNNs and small batches
- $\mu_l = \frac{1}{H}\sum_{i=1}^{H} x_i$, $\sigma_l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(x_i - \mu_l)^2}$

2. Instance Normalization:

- Normalizes each sample independently
- Useful for style transfer and GANs

- Removes instance-specific contrast information

3. Group Normalization:

- Divides channels into groups for normalization
- Batch-size independent
- Good compromise between Layer and Instance normalization

4. Weight Normalization:

- Normalizes weight vectors instead of activations
- Decouples weight magnitude from direction
- Computational benefits for certain architectures

Question 5. RNN Fundamentals and Unfolding (28 marks)
Based on sequence modeling and RNN theory from university courses.

(a) Classify the following problems and suggest appropriate architectures: (8 marks)

- Image captioning
- Spam email detection
- Machine translation
- Real-time speech recognition

For each, specify: one-to-one, one-to-many, many-to-one, or many-to-many architecture.

Answer: Image captioning: One-to-many; Spam detection: Many-to-one; Translation: Many-to-many; Speech recognition: Many-to-many

1. Image Captioning: One-to-Many

- **Input:** Single image (one)
- **Output:** Sequence of words (many)
- **Architecture:** CNN encoder + RNN decoder
- **Example:** Image → "A dog playing in the park"
- **Implementation:** CNN extracts image features, RNN generates caption word by word

2. Spam Email Detection: Many-to-One

- **Input:** Sequence of words/tokens (many)
- **Output:** Single classification (spam/not spam) (one)
- **Architecture:** RNN encoder with final classification layer
- **Example:** "Free money click now" → Spam
- **Implementation:** Process email text sequentially, final hidden state feeds classifier

3. Machine Translation: Many-to-Many (Sequence-to-Sequence)

- **Input:** Sequence in source language (many)
- **Output:** Sequence in target language (many)
- **Architecture:** Encoder-Decoder with attention
- **Example:** "How are you?" → "Comment allez-vous?"
- **Implementation:** Encoder RNN processes source, decoder RNN generates target

4. Real-time Speech Recognition: Many-to-Many (Synchronous)

- **Input:** Stream of audio features (many)
- **Output:** Stream of text/phonemes (many)
- **Architecture:** Streaming RNN with CTC or attention
- **Example:** Audio waveform → Real-time transcription
- **Implementation:** Process audio frames sequentially, output text in real-time

Architecture Design Guidelines:

- **One-to-Many:** Use RNN decoder with initial state from encoded input
- **Many-to-One:** Use RNN encoder, take final hidden state for classification
- **Many-to-Many:** Use encoder-decoder or synchronous RNN depending on alignment

(b) Explain RNN unfolding process. For the recurrent equation: (12 marks)

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

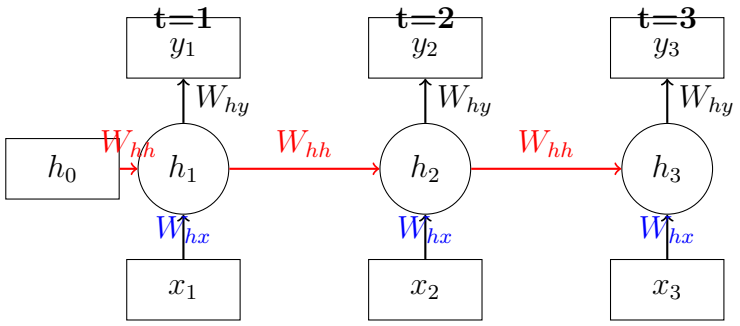
$$y_t = W_{hy}h_t + b_y$$

Draw the unfolded network for T=3 time steps showing:

- Weight sharing across time
- Hidden state connections

- How this becomes a feedforward network
- Why sequences of different lengths can be handled

Unfolded RNN as Feedforward Network



Same weights W_{hh}, W_{hx}, W_{hy} shared across time

RNN Unfolding Process:

Original Recurrent Formulation:

$$h_t = \tanh(W_{hx}x_t + W_{hh}h_{t-1} + b_h) \tag{9}$$
$$y_t = W_{hy}h_t + b_y \tag{10}$$

Unfolded Equations for T=3:

$$h_1 = \tanh(W_{hx}x_1 + W_{hh}h_0 + b_h) \tag{11}$$
$$h_2 = \tanh(W_{hx}x_2 + W_{hh}h_1 + b_h) \tag{12}$$
$$h_3 = \tanh(W_{hx}x_3 + W_{hh}h_2 + b_h) \tag{13}$$
$$y_1 = W_{hy}h_1 + b_y \tag{14}$$
$$y_2 = W_{hy}h_2 + b_y \tag{15}$$
$$y_3 = W_{hy}h_3 + b_y \tag{16}$$

Key Properties of Unfolding:

1. Weight Sharing Across Time:

- Same W_{hx} processes all inputs x_1, x_2, x_3
- Same W_{hh} connects all hidden states

- Same W_{hy} produces all outputs
- This sharing enables generalization across sequence positions

2. Hidden State Connections:

- Each h_t depends on previous h_{t-1} and current x_t
- Information flows left-to-right through hidden states
- Creates memory mechanism across time steps
- h_3 contains information from x_1, x_2, x_3 through recurrent connections

3. Feedforward Network Equivalence:

- Unfolded RNN becomes a feedforward network with shared weights
- Can apply standard backpropagation algorithm
- Each time step becomes a layer in feedforward network
- Enables gradient computation through time (BPTT)

4. Variable Length Sequence Handling:

- **Shorter sequences:** Simply stop unfolding earlier (T=2, T=1)
- **Longer sequences:** Continue unfolding for more time steps (T=4, T=5, ...)
- **Same weights:** W_{hx}, W_{hh}, W_{hy} work for any sequence length
- **Dynamic computation:** Network adapts to input sequence length automatically

Practical Implications:

- Training uses BPTT on unfolded network
- Gradient flows backward through time from h_3 to h_1 to h_0
- Memory requirements scale with sequence length
- Longer sequences may cause vanishing gradient problems

(c) Discuss the Turing completeness of RNNs. Explain: (8 marks)

- What it means for RNNs to be Turing complete

- The role of recurrent connections in providing memory
- Difference between theoretical capacity and practical training
- Comparison with multilayer perceptrons as universal approximators

Answer: RNNs are Turing complete, meaning they can theoretically compute any computable function given sufficient time and precision, but practical limitations prevent achieving this theoretical capacity.

Turing Completeness Definition:

What it means:

- RNNs can simulate any Turing machine given sufficient resources
- Can compute any algorithmically computable function
- Equivalent computational power to general-purpose computers
- Can perform any computation that can be described algorithmically

Theoretical Foundation:

- Proven by Siegelmann & Sontag (1995) for rational-weighted RNNs
- Even simple RNNs with proper weights can simulate Turing machines
- Requires only finite precision arithmetic (rational numbers)

Role of Recurrent Connections in Memory:

Memory Mechanism:

- Hidden state h_t serves as external memory tape
- Recurrent weights W_{hh} implement read/write operations
- Information can be stored, retrieved, and modified over time
- Unbounded computation time allows unlimited memory access

Computational Model:

- **Input:** Sequence of symbols (like Turing machine tape)

- **Processing:** Hidden state transformations (like state transitions)
- **Memory:** Recurrent connections maintain information (like memory cells)
- **Output:** Generated sequences (like Turing machine output)

Difference Between Theoretical and Practical:

Theoretical Capacity:

- Infinite precision arithmetic
- Unlimited sequence length
- Perfect weight optimization
- No numerical stability issues

Practical Limitations:

- **Finite Precision:** Floating-point arithmetic introduces errors
- **Vanishing Gradients:** Long sequences cause training difficulties
- **Limited Memory:** Hidden state size constrains memory capacity
- **Training Challenges:** Finding correct weights is computationally intractable
- **Sequence Length:** Practical sequences are finite and bounded

Real-World Implications:

- RNNs can learn complex algorithms but often fail to do so in practice
- Simple tasks like counting or copying can be challenging to learn
- Need architectural innovations (LSTM, attention) for practical success

Comparison with MLPs as Universal Approximators:

RNNs (Turing Complete)	**MLPs (Universal Approximators)**
Can compute any algorithm	Can approximate any continuous function
Sequential, temporal processing	Static, feedforward processing
Unbounded computation time	Fixed computation time
Memory through recurrence	No memory mechanism
Dynamic input/output	Fixed input/output size
Can handle algorithms	Cannot handle algorithms requiring memory

Key Distinctions:

- **MLPs:** Can approximate functions $f : \mathbb{R}^n \to \mathbb{R}^m$ (static mapping)
- **RNNs:** Can compute algorithms $f : \Sigma^* \to \Sigma^*$ (dynamic computation)
- **MLPs:** Limited to pattern recognition and regression
- **RNNs:** Can perform reasoning, counting, and algorithmic computation

Practical Significance:

- Turing completeness provides theoretical foundation for sequence modeling
- Explains why RNNs are suitable for algorithmic tasks
- Motivates research into making RNNs more practically capable
- Highlights importance of memory and sequential processing in AI

Question 6. RNN Training and Gradient Issues (25 marks)
Based on RNN training theory and backpropagation through time.

(a) Explain why hyperbolic tangent is preferred over ReLU in vanilla RNNs. Discuss: (8 marks)

- Need for bounded hidden state values
- Consistency of state representation across time
- Problems with unbounded activations in recurrent connections
- Trade-offs with gradient flow

Answer: Hyperbolic tangent is preferred in vanilla RNNs because it provides bounded outputs $[-1, 1]$, preventing explosive growth in recurrent connections, though it contributes to vanishing gradients.

Need for Bounded Hidden State Values:

Mathematical Analysis: RNN hidden state update: $h_t = \tanh(W_{hx}x_t + W_{hh}h_{t-1} + b_h)$

With Tanh:

- Output range: $h_t \in [-1, 1]$
- Bounded regardless of input magnitude
- Prevents hidden state explosion
- Provides numerical stability

With ReLU:

- Output range: $h_t \in [0, \infty)$
- Unbounded positive growth possible
- Can lead to explosive hidden states
- Numerical instability over time

Consistency of State Representation:

Tanh Advantages:

- Symmetric around zero: $\tanh(-x) = -\tanh(x)$

- Consistent scale across all time steps
- Hidden states remain in predictable range
- Weights can be initialized to work with bounded inputs

State Evolution Example:

- $h_0 \in [-1, 1]$ (initial state)
- $h_1 = \tanh(\cdot) \in [-1, 1]$ (bounded)
- $h_2 = \tanh(\cdot) \in [-1, 1]$ (still bounded)
- Consistent representation maintained over time

Problems with Unbounded Activations:

ReLU in Recurrent Connections: Consider: $h_t = \text{ReLU}(W_{hh}h_{t-1} + W_{hx}x_t + b)$

Exponential Growth Scenario:

- If $W_{hh} > 1$ and inputs are positive
- $h_1 = \text{ReLU}(W_{hh}h_0 + \cdots) \geq W_{hh}h_0$
- $h_2 \geq W_{hh}h_1 \geq W_{hh}^2 h_0$
- $h_t \geq W_{hh}^t h_0$ (exponential growth)

Consequences:

- Hidden states can grow arbitrarily large
- Gradients become unstable
- Numerical overflow in practice
- Loss of information through saturation

Trade-offs with Gradient Flow:

Tanh Gradient Properties:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x) \in (0, 1]$$

Advantages:

- Non-zero gradients everywhere (no dead neurons)
- Smooth, continuous gradients
- Maximum gradient of 1 at $x = 0$

Disadvantages:

- Gradients approach 0 for large $|x|$ (vanishing gradient)
- Saturated activations stop learning
- Contributes to long-term dependency problems

ReLU Gradient Properties:

$$\frac{d}{dx}\text{ReLU}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

Advantages:

- No vanishing gradient for positive activations
- Constant gradient of 1 in active region
- Promotes sparse representations

Disadvantages in RNNs:

- Dead neurons (gradient = 0) can kill information flow
- Unbounded activations cause instability
- Asymmetric activation disrupts recurrent dynamics

Modern Solutions:

- **LSTM/GRU:** Use gating to control information flow
- **Gradient Clipping:** Bound gradient magnitudes
- **Careful Initialization:** Control initial weight magnitudes
- **Residual Connections:** Add skip connections in deep RNNs

(b) For an RNN unfolded for 100 time steps, analyze the vanishing gradient problem: (12 marks)

- Why this creates a 100-layer feedforward network
- Mathematical explanation of gradient diminishing
- Effect of squashing activation functions
- Impact on learning long-term dependencies

Include analysis of gradient computation:

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial W_{hh}}$$

Answer: Unfolding creates a 100-layer network where gradients must flow backward through all layers, experiencing multiplicative decay through tanh derivatives, making learning of early dependencies nearly impossible.

100-Layer Feedforward Network Creation:

Unfolding Process:

- RNN unfolded for 100 time steps becomes 100-layer feedforward network
- Each time step corresponds to one layer
- Weights W_{hh}, W_{hx}, W_{hy} are shared across all layers
- Information flows forward through layers: $h_1 \to h_2 \to \cdots \to h_{100}$

Network Structure:

$$h_1 = \tanh(W_{hx}x_1 + W_{hh}h_0) \tag{17}$$
$$h_2 = \tanh(W_{hx}x_2 + W_{hh}h_1) \tag{18}$$
$$\vdots \tag{19}$$
$$h_{100} = \tanh(W_{hx}x_{100} + W_{hh}h_{99}) \tag{20}$$

Mathematical Analysis of Gradient Diminishing:

Full Gradient Computation:

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{100} \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial W_{hh}}$$

Expanding $\frac{\partial h_t}{\partial W_{hh}}$ using chain rule:

$$\frac{\partial h_t}{\partial W_{hh}} = \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial W_{hh}} + \left.\frac{\partial h_t}{\partial W_{hh}}\right|_{\text{direct}}$$

For early time steps (e.g., $t = 1$):

$$\frac{\partial h_{100}}{\partial h_1} = \prod_{k=2}^{100} \frac{\partial h_k}{\partial h_{k-1}} = \prod_{k=2}^{100} W_{hh} \cdot \tanh'(u_k)$$

Where $u_k = W_{hx} x_k + W_{hh} h_{k-1} + b_h$

Gradient Magnitude Analysis:

$$\left|\frac{\partial h_{100}}{\partial h_1}\right| = \left|\prod_{k=2}^{100} W_{hh} \cdot \tanh'(u_k)\right| = |W_{hh}|^{99} \prod_{k=2}^{100} |\tanh'(u_k)|$$

Effect of Squashing Activation Functions:

Tanh Derivative Properties:

- $\tanh'(x) = 1 - \tanh^2(x) \in (0, 1]$
- Maximum value: $\tanh'(0) = 1$
- For most values: $\tanh'(x) < 1$
- Typical range in practice: $\tanh'(x) \in [0.1, 0.8]$

Cumulative Effect:

- Product of 99 terms each < 1
- Even with $\tanh'(x) = 0.5$ (optimistic): $(0.5)^{99} \approx 1.6 \times 10^{-30}$
- With typical values ≈ 0.3: $(0.3)^{99} \approx 10^{-52}$
- Gradients become negligibly small

Weight Matrix Contribution:

- If $|W_{hh}| < 1$: $|W_{hh}|^{99}$ further diminishes gradient
- If $|W_{hh}| > 1$: Can lead to exploding gradients

- Sweet spot $|W_{hh}| \approx 1$ is difficult to maintain

Impact on Learning Long-term Dependencies:

Information Loss:

- Gradient from h_{100} to h_1 is $\approx 10^{-30}$ or smaller
- Weights receive virtually no learning signal from distant time steps
- Network cannot learn correlations between x_1 and y_{100}

Practical Consequences:

- **Short-term Memory Only:** RNN learns dependencies within 10-20 time steps
- **Information Decay:** Older inputs have exponentially decreasing influence
- **Training Inefficiency:** Early layers receive weak gradients, learn slowly
- **Task Limitations:** Cannot solve problems requiring long-term memory

Examples of Affected Tasks:

- Language modeling: Cannot use context from beginning of long sentences
- Machine translation: Cannot align distant words in long sentences
- Time series: Cannot capture yearly patterns in daily data
- Algorithmic tasks: Cannot maintain counters or stack operations

Mathematical Summary: For learning dependency between x_1 and y_{100}:

$$\frac{\partial L}{\partial W_{hh}} \propto \frac{\partial L}{\partial y_{100}} \frac{\partial y_{100}}{\partial h_{100}} \frac{\partial h_{100}}{\partial h_1} \frac{\partial h_1}{\partial W_{hh}}$$

The term $\frac{\partial h_{100}}{\partial h_1} \approx 10^{-30}$ makes this gradient negligible, preventing learning.

(c) Compare solutions to RNN gradient problems: (5 marks)

- Gradient clipping for exploding gradients
- Penalty terms for vanishing gradients
- When each approach is appropriate

Answer: Gradient clipping limits maximum gradient magnitude to prevent explosions; penalty terms and architectural solutions (LSTM, residual connections) address vanishing gradients; choice depends on which problem dominates.

Gradient Clipping for Exploding Gradients:

Method:

- Monitor gradient norm: $\|\nabla\| = \sqrt{\sum_i (\frac{\partial L}{\partial \theta_i})^2}$
- If $\|\nabla\| > \text{threshold}$: $\nabla \leftarrow \frac{\text{threshold}}{\|\nabla\|} \nabla$
- Preserves gradient direction, scales down magnitude

Mathematical Formulation:

$$\tilde{\nabla} = \begin{cases} \nabla & \text{if } \|\nabla\| \leq \tau \\ \frac{\tau}{\|\nabla\|} \nabla & \text{if } \|\nabla\| > \tau \end{cases}$$

Advantages:

- Simple to implement
- Effective for preventing training divergence
- Maintains gradient direction information
- Allows training of deeper RNNs

Limitations:

- Doesn't address vanishing gradients
- Threshold selection is crucial but difficult
- Can slow convergence if threshold too conservative

Penalty Terms and Regularization for Vanishing Gradients:

1. Gradient Penalty Methods:

- Add terms to loss: $L' = L + \lambda \| \frac{\partial h_t}{\partial h_{t-k}} \|^2$
- Encourage gradients to maintain magnitude
- Directly optimize for better gradient flow

2. Weight Regularization:

- Encourage recurrent weights near identity: $L' = L + \lambda \| W_{hh} - I \|^2$
- Promotes gradient preservation
- Based on theory that identity matrix preserves gradients

3. Spectral Regularization:

- Control eigenvalues of W_{hh}: $L' = L + \lambda(\rho(W_{hh}) - 1)^2$
- $\rho(W_{hh})$ is spectral radius (largest eigenvalue magnitude)
- Keeps eigenvalues near 1 for stable gradients

Architectural Solutions:

1. LSTM/GRU:

- Gating mechanisms control information flow
- Additive updates rather than multiplicative
- Dedicated memory pathways

2. Residual Connections:

- Add skip connections: $h_t = f(h_{t-1}, x_t) + h_{t-1}$
- Provides gradient highways
- Similar to ResNet for sequences

When Each Approach is Appropriate:

Gradient Clipping:

- **When:** Training becomes unstable, loss oscillates wildly
- **Signs:** NaN values, exponentially growing loss
- **Best for:** Quick fix during training, debugging
- **Use with:** Any RNN architecture as safety measure

Penalty Terms:

- **When:** Research settings, when you want to keep vanilla RNN
- **Signs:** Poor long-term memory, gradients approach zero
- **Best for:** Understanding gradient behavior, proof-of-concept
- **Limitations:** Often less effective than architectural solutions

Architectural Solutions (LSTM/GRU):

- **When:** Production applications, need reliable long-term memory
- **Signs:** Penalty methods insufficient, complex sequence tasks
- **Best for:** Most practical applications
- **Trade-off:** More complex but much more effective

Combined Approach (Recommended):

- Use LSTM/GRU architecture
- Apply gradient clipping as safety measure
- Add residual connections for very deep networks
- Monitor gradient norms during training

Question 7. LSTM Architecture and Memory Mechanisms

(30 marks)

Based on LSTM theory and gating mechanisms for sequence modeling.

(a) Design the complete LSTM architecture with mathematical equations. For input x_t, previous hidden state h_{t-1}, and previous cell state C_{t-1}, derive: (15 marks)

- Forget gate: f_t =?
- Input gate: i_t =?
- Candidate values: $\tilde{C}_t$ =?
- Cell state update: C_t =?
- Output gate: o_t =?
- Hidden state: h_t =?

Answer: Complete LSTM equations with three gates controlling information flow through dedicated cell state pathway.

Complete LSTM Mathematical Formulation:

1. Forget Gate:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Purpose: Decides what information to discard from cell state

- $f_t \in [0, 1]$ for each dimension of cell state
- $f_t = 0$: completely forget old information
- $f_t = 1$: completely retain old information
- Sigmoid ensures values in $[0, 1]$ range

2. Input Gate:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

Purpose: Decides which new information to store in cell state

- Controls which candidate values will be added

- $i_t \in [0, 1]$ acts as importance weighting
- Works in conjunction with candidate values

3. Candidate Values:

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Purpose: Creates vector of new candidate values for cell state

- $\tilde{C}_t \in [-1, 1]$ due to tanh activation
- Represents potential new information to be stored
- Modulated by input gate to determine actual contribution

4. Cell State Update:

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

Purpose: Updates cell state by forgetting old and adding new information

- $\odot$ denotes element-wise multiplication
- $f_t \odot C_{t-1}$: selectively retain old information
- $i_t \odot \tilde{C}_t$: selectively add new information
- Additive update prevents vanishing gradients

5. Output Gate:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

Purpose: Decides what parts of cell state to output as hidden state

- Controls which information is exposed to next time step
- $o_t \in [0, 1]$ acts as filtering mechanism
- Allows internal memory to differ from external output

6. Hidden State:

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

Purpose: Produces final output based on filtered cell state

- $\tanh(C_t)$ squashes cell state to $[-1, 1]$
- o_t selectively filters this information
- h_t becomes input to next time step and current output

Parameter Dimensions: For hidden size h and input size x:

- $W_f, W_i, W_C, W_o \in \mathbb{R}^{h \times (h+x)}$
- $b_f, b_i, b_C, b_o \in \mathbb{R}^h$
- Total parameters: $4h(h + x) + 4h = 4h(h + x + 1)$

Information Flow Summary:

i. Forget gate removes irrelevant information from C_{t-1}
ii. Input gate and candidates decide what new information to add
iii. Cell state combines retained and new information additively
iv. Output gate filters cell state for external use
v. Hidden state provides processed information to next time step

(b) Explain why LSTM solves the vanishing gradient problem. Focus on: (10 marks)

- Gradient flow through the cell state path
- Why $\frac{\partial C_t}{\partial C_{t-1}}$ doesn't involve squashing functions
- How this enables learning of long-term dependencies
- Mathematical comparison with vanilla RNN gradient flow

Answer: LSTM solves vanishing gradients through additive cell state updates where $\frac{\partial C_t}{\partial C_{t-1}} = f_t$, providing unimpeded gradient flow compared to vanilla RNN's multiplicative updates.

Gradient Flow Through Cell State Path:

LSTM Cell State Update:

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

Gradient Computation:

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

Key Insight: The gradient $\frac{\partial C_t}{\partial C_{t-1}} = f_t$ is controlled by the forget gate, not by activation function derivatives!

Why No Squashing Functions in Gradient Path:

LSTM Advantage:

- Cell state update is additive: $C_t = f_t \odot C_{t-1} + (\text{new info})$
- Gradient: $\frac{\partial C_t}{\partial C_{t-1}} = f_t$ (no activation derivative)
- $f_t \in [0, 1]$ is learned, can be close to 1 when memory needed
- No forced multiplication by $\tanh'$ or similar derivatives

Vanilla RNN Problem:

- Hidden state update: $h_t = \tanh(W_{hh}h_{t-1} + W_{hx}x_t + b)$
- Gradient: $\frac{\partial h_t}{\partial h_{t-1}} = W_{hh} \odot \tanh'(\cdot)$
- Always includes $\tanh'(\cdot) \in (0, 1]$ which typically < 1
- Forces gradient reduction at every time step

Long-term Gradient Flow Analysis:

LSTM Through Multiple Time Steps:

$$\frac{\partial C_T}{\partial C_1} = \prod_{t=2}^{T} \frac{\partial C_t}{\partial C_{t-1}} = \prod_{t=2}^{T} f_t$$

Gradient Preservation:

- If $f_t \approx 1$ (forget gate learns to remember): $\frac{\partial C_T}{\partial C_1} \approx 1$
- Gradient magnitude preserved across time steps
- Network can choose when to forget ($f_t \approx 0$) vs. remember ($f_t \approx 1$)
- Learning determines optimal forget gate values

Vanilla RNN Comparison:

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

Forced Decay:

- Each $\tanh'(u_t) \leq 1$, typically much less
- Product of many terms < 1 causes exponential decay
- No mechanism to prevent this decay
- Gradient vanishing is inevitable for long sequences

Mathematical Comparison:

LSTM	**Vanilla RNN**
$\frac{\partial C_T}{\partial C_1} = \prod_{t=2}^{T} f_t$	$\frac{\partial h_T}{\partial h_1} = \prod_{t=2}^{T} W_{hh} \tanh'(u_t)$
f_t is learned (can be ≈ 1)	$\tanh'(u_t)$ is forced < 1
Additive cell state updates	Multiplicative hidden state updates
Selective gradient flow control	No gradient flow control
Can maintain gradient magnitude	Gradient magnitude always decreases

Enabling Long-term Dependencies:

1. Gradient Highway:

- Cell state provides "highway" for gradients
- Gradients can flow back unimpeded when $f_t \approx 1$
- Learning signal reaches early time steps effectively

2. Adaptive Memory:

- Network learns when to remember vs. forget
- Important information maintained over long periods
- Irrelevant information discarded to make room for new data

3. Practical Benefits:

- Can learn dependencies spanning 100+ time steps
- Successful on tasks requiring long-term memory
- More stable training than vanilla RNNs
- Better performance on sequence modeling tasks

Example Scenario: For learning dependency between x_1 and y_{100}:

- **LSTM:** If important, forget gates learn $f_t \approx 1$ for $t = 2, \ldots, 100$
- **Result:** $\frac{\partial C_{100}}{\partial C_1} \approx 1$, strong gradient flow
- **Vanilla RNN:** $\frac{\partial h_{100}}{\partial h_1} \approx 0.5^{99} \approx 10^{-30}$, no learning

(c) Compare LSTM variants: (5 marks)

- LSTM with peephole connections
- Coupled forget and input gates
- GRU vs LSTM trade-offs
- When to choose each variant

Answer: LSTM variants offer different trade-offs: peephole connections add precision, coupled gates reduce parameters, GRU provides simplicity with comparable performance.

LSTM with Peephole Connections:

Standard vs. Peephole LSTM:

Standard: Gates depend only on h_{t-1} and x_t **Peephole:** Gates also depend on cell state C_{t-1} (and C_t for output gate)

Modified Equations:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + W_{pf} \odot C_{t-1} + b_f) \tag{21}$$
$$i_t = \sigma(W_i[h_{t-1}, x_t] + W_{pi} \odot C_{t-1} + b_i) \tag{22}$$
$$o_t = \sigma(W_o[h_{t-1}, x_t] + W_{po} \odot C_t + b_o) \tag{23}$$

Advantages:

- Gates can directly observe cell state information
- Better timing control for gate activations

- Can learn more precise gating decisions
- Useful for tasks requiring precise timing

Disadvantages:

- Additional parameters (W_{pf}, W_{pi}, W_{po})
- Increased computational complexity
- Marginal performance improvement in most cases

Coupled Forget and Input Gates:

Motivation: In many cases, when we forget old information, we should add new information (and vice versa)

Modified Equations:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{24}$$
$$i_t = 1 - f_t \tag{25}$$
$$C_t = f_t \odot C_{t-1} + (1 - f_t) \odot \tilde{C}_t \tag{26}$$

Advantages:

- Reduces parameters (no separate input gate)
- Enforces complementary behavior
- Faster computation
- Often performs similarly to standard LSTM

Disadvantages:

- Less flexibility in gating decisions
- Cannot simultaneously retain old and add new information
- May be suboptimal for some tasks

GRU vs LSTM Trade-offs:

GRU Architecture:

$$r_t = \sigma(W_r[h_{t-1}, x_t]) \quad \text{(reset gate)} \tag{27}$$
$$z_t = \sigma(W_z[h_{t-1}, x_t]) \quad \text{(update gate)} \tag{28}$$
$$\tilde{h}_t = \tanh(W_h[r_t \odot h_{t-1}, x_t]) \quad \text{(candidate)} \tag{29}$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{30}$$

Comparison Table:

Aspect	LSTM	GRU
Parameters	$4h(h + x + 1)$	$3h(h + x + 1)$
Gates	3 gates + cell state	2 gates only
Memory	Separate cell state	Hidden state as memory
Complexity	Higher	Lower
Training Speed	Slower	Faster
Performance	Slightly better on complex tasks	Comparable on most tasks
Gradient Flow	Excellent (dedicated cell state)	Good (through hidden state)

When to Choose Each Variant:

Standard LSTM:

- **Use when:** Maximum performance needed, complex sequential tasks
- **Best for:** Language modeling, machine translation, complex time series
- **Advantages:** Most expressive, best long-term memory
- **Trade-offs:** Highest computational cost

Peephole LSTM:

- **Use when:** Precise timing is crucial, tasks with clear temporal patterns
- **Best for:** Speech recognition, music generation, periodic time series
- **Advantages:** Better temporal precision
- **Trade-offs:** More parameters, minimal performance gain

Coupled Gates LSTM:

- **Use when:** Want LSTM benefits with fewer parameters
- **Best for:** Resource-constrained environments, simpler tasks

- **Advantages:** Reduced parameters, enforced gate coupling
- **Trade-offs:** Less flexibility than standard LSTM

GRU:

- **Use when:** Good trade-off between performance and efficiency needed
- **Best for:** Most practical applications, real-time systems
- **Advantages:** Simpler, faster, comparable performance
- **Trade-offs:** Slightly less expressive than LSTM

Decision Guidelines:

- **Start with GRU:** Good default choice for most applications
- **Try LSTM:** If GRU performance insufficient for complex tasks
- **Consider peephole:** Only if timing precision is critical
- **Use coupled gates:** When parameter budget is tight

END OF PAPER