Comprehensive Deep Dive: Recurrent Neural Networks, LSTM, and GRU

ankit kumar

February 4, 2026

Contents

1 Recurrent Neural Networks (RNN) Fundamentals

1.1 Core Concepts and Motivation

- **Problem**: Traditional feedforward networks cannot handle sequential data with temporal dependencies
- **Solution**: Introduce recurrent connections to create "memory"
- **Mathematical Formulation**:

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

- **Output Computation**:

$$y_t = g(W_{hy}h_t + b_y)$$

1.2 RNN Architecture: Detailed Analysis

1.2.1 Unfolded Computational Graph

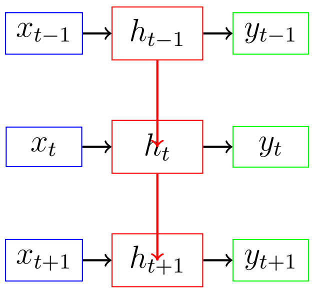

Figure 1: Detailed unfolded architecture of RNN across time steps

1.3 Mathematical Derivation

1.3.1 Forward Propagation

$$h_t = \tanh(W_h h_{t-1} + W_x x_t + b_h) \tag{1}$$

$$y_t = \mathrm{softmax}(W_y h_t + b_y) \tag{2}$$

1.3.2 Parameter Dimension Analysis

- $x_t \in \mathbb{R}^{d_x}$: Input vector dimension
- $h_t \in \mathbb{R}^{d_h}$: Hidden state dimension
- $W_h \in \mathbb{R}^{d_h \times d_h}$: Hidden layer weight matrix
- $W_x \in \mathbb{R}^{d_h \times d_x}$: Input weight matrix
- $W_y \in \mathbb{R}^{d_y \times d_h}$: Output weight matrix
- Total parameters: $d_h \times (d_h + d_x + d_y) + d_h + d_y$

2 Complete Training Process and Gradient Computation

2.1 Loss Function Detailed Derivation

2.1.1 Binary Cross-Entropy Loss (Sentiment Analysis Example)

Given training data: $\{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$, where:

- $x^{(i)} = [x_1^{(i)}, x_2^{(i)}, \ldots, x_T^{(i)}]$: Input sequence
- $y^{(i)} \in \{0, 1\}$: True label (0: negative, 1: positive)

Loss function:

$$L = -\frac{1}{N} \sum_{i=1}^{N} \left[y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right]$$

where $\hat{y}^{(i)}$ is the predicted probability.

2.1.2 Multi-Class Cross-Entropy Loss (Sequence Labeling)

For sequence labeling tasks (e.g., POS tagging):

$$L = -\frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T_i} \sum_{c=1}^{C} y_{t,c}^{(i)} \log(\hat{y}_{t,c}^{(i)})$$

where C is number of classes.

2.2 Backpropagation Through Time (BPTT) - Complete Derivation

2.2.1 Single Time Step Gradient

At time step t, for parameter θ:

$$\frac{\partial L}{\partial \theta} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial \theta}$$

The key chain rule component:

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^{t} W_h^T \text{diag}(1 - h_j^2)$$

Here $\text{diag}(1 - h_j^2)$ comes from the derivative of tanh activation.

2.2.2 Specific Parameter Gradients

$$\frac{\partial L}{\partial W_h} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial L_t}{\partial h_t} \left(\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_k}{\partial W_h} \tag{3}$$

$$\frac{\partial L}{\partial W_x} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial L_t}{\partial h_t} \left(\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_k}{\partial W_x} \tag{4}$$

2.3 Vanishing and Exploding Gradient Problem - Mathematical Analysis

2.3.1 Vanishing Gradient Problem

Consider gradient propagation factor:

$$\left\|\frac{\partial h_t}{\partial h_k}\right\| = \left\|\prod_{j=k+1}^{t} W_h^T \text{diag}(1-h_j^2)\right\|$$

Since $\|\text{diag}(1-h_j^2)\| \leq 1$, and $\tanh'(z) = 1 - \tanh^2(z) \in (0, 1]$, when $t-k$ is large:

$$\left\|\prod_{j=k+1}^{t} W_h^T \text{diag}(1-h_j^2)\right\| \approx \|W_h\|^{t-k}$$

If $\|W_h\| < 1$, gradients vanish exponentially.

2.3.2 Exploding Gradient Problem

If $\|W_h\| > 1$, gradients explode exponentially, causing numerical instability.

3 LSTM (Long Short-Term Memory) - Complete Detailed Analysis

3.1 LSTM Mathematical Formulation - Full Details

3.1.1 Gate Definitions

$$\text{Forget gate: } f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{5}$$

$$\text{Input gate: } i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{6}$$

$$\text{Candidate memory: } \tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{7}$$

$$\text{Output gate: } o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{8}$$

3.1.2 State Update Equations

$$\text{Cell state update: } C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{9}$$

$$\text{Hidden state output: } h_t = o_t \odot \tanh(C_t) \tag{10}$$

where $\odot$ denotes element-wise multiplication.

3.2 LSTM Gradient Flow Analysis

3.2.1 Cell State Gradient

Cell state gradient avoids multiplicative chains:

$$\frac{\partial C_t}{\partial C_{t-1}} = f_t + \text{diag}(C_{t-1})\frac{\partial f_t}{\partial C_{t-1}} + \text{diag}(\tilde{C}_t)\frac{\partial i_t}{\partial C_{t-1}}$$

3.2.2 Parameter Count Analysis

For LSTM with hidden size d_h and input size d_x:

- $W_f, W_i, W_C, W_o \in \mathbb{R}^{d_h \times (d_h + d_x)}$
- $b_f, b_i, b_C, b_o \in \mathbb{R}^{d_h}$
- Total parameters: $4 \times [d_h \times (d_h + d_x) + d_h]$

3.3 LSTM Implementation - Complete Code

Listing 1: Complete LSTM Implementation

```python
import numpy as np

class LSTMCell:
    def __init__(self, input_size, hidden_size):
        # Parameter initialization
        self.input_size = input_size
        self.hidden_size = hidden_size

        # Weights for gates (combined for efficiency)
        self.W = np.random.randn(4 * hidden_size, hidden_size + input_size)
        self.b = np.zeros((4 * hidden_size, 1))

    def forward(self, x, h_prev, C_prev):
        """
        Forward pass for single LSTM cell
        x: input vector (input_size, 1)
        h_prev: previous hidden state (hidden_size, 1)
        C_prev: previous cell state (hidden_size, 1)
        """
        # Concatenate h_prev and x
        concat = np.vstack((h_prev, x))  # (hidden_size + input_size, 1)

        # Compute all gates in one matrix multiplication
        gates = np.dot(self.W, concat) + self.b

        # Split into individual gates
        f_t = self._sigmoid(gates[0:self.hidden_size])
        i_t = self._sigmoid(gates[self.hidden_size:2*self.hidden_size])
        C_tilde = np.tanh(gates[2*self.hidden_size:3*self.hidden_size])
        o_t = self._sigmoid(gates[3*self.hidden_size:4*self.hidden_size])

        # Update cell state
        C_t = f_t * C_prev + i_t * C_tilde

        # Compute hidden state
        h_t = o_t * np.tanh(C_t)
```

```python
        # Cache for backward pass
        self.cache = (x, h_prev, C_prev, f_t, i_t, C_tilde, o_t, C_t)

        return h_t, C_t

    def backward(self, dh_next, dC_next):
        """
        Backward pass for single LSTM cell
        """
        # Unpack cache
        x, h_prev, C_prev, f_t, i_t, C_tilde, o_t, C_t = self.cache

        # Gradients from next time step
        dC_t = dC_next + dh_next * o_t * (1 - np.tanh(C_t)**2)

        # Gate gradients
        df = dC_t * C_prev * f_t * (1 - f_t)
        di = dC_t * C_tilde * i_t * (1 - i_t)
        dC_tilde = dC_t * i_t * (1 - C_tilde**2)
        do = dh_next * np.tanh(C_t) * o_t * (1 - o_t)

        # Concatenate gate gradients
        dgates = np.vstack((df, di, dC_tilde, do))

        # Gradient for previous cell state
        dC_prev = dC_t * f_t

        # Gradient for concatenated vector
        concat = np.vstack((h_prev, x))
        dW = np.dot(dgates, concat.T)
        db = np.sum(dgates, axis=1, keepdims=True)

        # Gradient for previous hidden state and input
        dconcat = np.dot(self.W.T, dgates)
        dh_prev = dconcat[:self.hidden_size]
        dx = dconcat[self.hidden_size:]

        return dx, dh_prev, dC_prev, dW, db

    def _sigmoid(self, x):
        return 1 / (1 + np.exp(-x))
```

4 GRU (Gated Recurrent Unit) - Complete Analysis

4.1 GRU Mathematical Formulation

4.1.1 Gate Definitions

$$\text{Reset gate: } r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r) \tag{11}$$

$$\text{Update gate: } z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z) \tag{12}$$

$$\text{Candidate activation: } \tilde{h}_t = \tanh(W \cdot [r_t \odot h_{t-1}, x_t] + b) \tag{13}$$

4.1.2 State Update Equation

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

4.2 GRU vs LSTM - Detailed Comparison

Aspect	**LSTM**	**GRU**
Gates	3 gates: Forget, Input, Output	2 gates: Reset, Update
Parameters	$4(d_h \times (d_h + d_x) + d_h)$	$3(d_h \times (d_h + d_x) + d_h)$
Memory Mechanism	Separate cell state C_t and hidden state h_t	Combined hidden state h_t
Gradient Flow	Direct path through cell state	Adaptive gating for gradient flow
Training Speed	Slower due to more parameters	Faster with fewer parameters
Performance	Better for very long sequences	Comparable for moderate sequences

Table 1: LSTM vs GRU detailed comparison

4.3 GRU Implementation - Complete Code

Listing 2: Complete GRU Implementation

```
class GRUCell:
    def __init__(self, input_size, hidden_size):
        self.input_size = input_size
        self.hidden_size = hidden_size

        # Initialize weights
        self.W_z = np.random.randn(hidden_size, hidden_size + input_size) *
        self.W_r = np.random.randn(hidden_size, hidden_size + input_size) *
        self.W_h = np.random.randn(hidden_size, hidden_size + input_size) *

        # Initialize biases
        self.b_z = np.zeros((hidden_size, 1))
```

```
        self.b_r = np.zeros((hidden_size, 1))
        self.b_h = np.zeros((hidden_size, 1))

    def forward(self, x, h_prev):
        """
        Forward pass for GRU cell
        """
        # Concatenate h_prev and x
        concat = np.vstack((h_prev, x))

        # Reset gate
        r = self._sigmoid(np.dot(self.W_r, concat) + self.b_r)

        # Update gate
        z = self._sigmoid(np.dot(self.W_z, concat) + self.b_z)

        # Concatenate reset gate output with input
        concat_reset = np.vstack((r * h_prev, x))

        # Candidate hidden state
        h_tilde = np.tanh(np.dot(self.W_h, concat_reset) + self.b_h)

        # Final hidden state
        h_t = (1 - z) * h_prev + z * h_tilde

        # Cache for backward pass
        self.cache = (x, h_prev, r, z, h_tilde, concat, concat_reset)

        return h_t

    def backward(self, dh_next):
        """
        Backward pass for GRU cell
        """
        # Unpack cache
        x, h_prev, r, z, h_tilde, concat, concat_reset = self.cache

        # Gradients
        dh_prev = dh_next * (1 - z)
        dz = dh_next * (h_tilde - h_prev) * z * (1 - z)
        dh_tilde = dh_next * z * (1 - h_tilde**2)

        # Gradients for reset gate
        dr = np.dot(self.W_h[:, :self.hidden_size].T, dh_tilde) * h_prev *

        # Gradients for weights
        dW_z = np.dot(dz, concat.T)
        dW_r = np.dot(dr, concat.T)
```

```
        dW_h = np.dot(dh_tilde, concat_reset.T)

        # Gradient for input
        dx_part = (
            np.dot(self.W_z[:, self.hidden_size:].T, dz) +
            np.dot(self.W_r[:, self.hidden_size:].T, dr) +
            np.dot(self.W_h[:, self.hidden_size:].T, dh_tilde)
        )

        return dx_part, dh_prev, dW_z, dW_r, dW_h, dz, dr, dh_tilde

    def _sigmoid(self, x):
        return 1 / (1 + np.exp(-x))
```

5 From PDF Content: Practical Examples and Data Processing

5.1 Sentiment Analysis Example from PDF

From the PDF, we see repeated patterns like:

```
1 → [you are good] 0
2 → [you are bad] 0
3 → [you are good] 0
4 → [you are bad] 0
...
```

5.1.1 Data Preprocessing

Listing 3: Data preprocessing for sentiment analysis

```
import numpy as np

class SentimentDataProcessor:
    def __init__(self, vocab_size=1000, max_length=10):
        self.vocab_size = vocab_size
        self.max_length = max_length
        self.word_to_idx = {}
        self.idx_to_word = {}

    def preprocess(self, sentences, labels):
        """
        Convert sentences to one-hot encoded sequences
        """
        # Build vocabulary
        vocab = set()
        for sentence in sentences:
            for word in sentence.split():
```

```
            vocab.add(word)

    # Create word mappings
    self.word_to_idx = {word: i for i, word in enumerate(vocab)}
    self.idx_to_word = {i: word for word, i in self.word_to_idx.items()

    # Convert sentences to one-hot
    X = []
    for sentence in sentences:
        words = sentence.split()
        seq = []
        for word in words:
            if word in self.word_to_idx:
                one_hot = np.zeros(self.vocab_size)
                one_hot[self.word_to_idx[word]] = 1
                seq.append(one_hot)

        # Pad or truncate to max_length
        if len(seq) < self.max_length:
            padding = [np.zeros(self.vocab_size)] * (self.max_length -
            seq = seq + padding
        else:
            seq = seq[:self.max_length]

        X.append(np.array(seq))

    return np.array(X), np.array(labels)
```

5.2 Loss Function Implementation

Listing 4: Complete loss function implementation

```
def binary_cross_entropy_loss(y_pred, y_true):
    """
    Binary cross-entropy loss implementation
    y_pred: predicted probabilities [batch_size, 1]
    y_true: true labels [batch_size, 1]
    """
    # Clip predictions to avoid log(0)
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)

    # Calculate loss
    loss = -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_p

    # Gradient
    dy_pred = (y_pred - y_true) / (y_pred * (1 - y_pred) + epsilon)
```

```python
    return loss, dy_pred

def categorical_cross_entropy_loss(y_pred, y_true):
    """
    Categorical cross-entropy loss
    y_pred: predicted probabilities [batch_size, num_classes]
    y_true: one-hot encoded labels [batch_size, num_classes]
    """
    # Clip predictions
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)

    # Calculate loss
    loss = -np.sum(y_true * np.log(y_pred)) / y_pred.shape[0]

    # Gradient
    dy_pred = (y_pred - y_true) / y_pred.shape[0]

    return loss, dy_pred
```

6 Optimization Algorithms

6.1 Gradient Descent Implementation

Listing 5: Gradient descent optimizer with momentum

```python
class GradientDescentOptimizer:
    def __init__(self, learning_rate=0.01, momentum=0.9):
        self.learning_rate = learning_rate
        self.momentum = momentum
        self.velocity = {}

    def update(self, parameters, gradients):
        """
        Update parameters using gradient descent with momentum
        """
        for param_name in parameters:
            if param_name not in self.velocity:
                self.velocity[param_name] = np.zeros_like(gradients[param_n

            # Update velocity
            self.velocity[param_name] = (
                self.momentum * self.velocity[param_name] -
                self.learning_rate * gradients[param_name]
            )

            # Update parameters
            parameters[param_name] += self.velocity[param_name]
```

```
        return parameters

class AdamOptimizer:
    def __init__(self, learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon
        self.learning_rate = learning_rate
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.m = {}
        self.v = {}
        self.t = 0

    def update(self, parameters, gradients):
        self.t += 1

        for param_name in parameters:
            if param_name not in self.m:
                self.m[param_name] = np.zeros_like(gradients[param_name])
                self.v[param_name] = np.zeros_like(gradients[param_name])

            # Update biased first moment estimate
            self.m[param_name] = self.beta1 * self.m[param_name] + (1 - sel

            # Update biased second raw moment estimate
            self.v[param_name] = self.beta2 * self.v[param_name] + (1 - sel

            # Compute bias-corrected first moment estimate
            m_hat = self.m[param_name] / (1 - self.beta1 ** self.t)

            # Compute bias-corrected second raw moment estimate
            v_hat = self.v[param_name] / (1 - self.beta2 ** self.t)

            # Update parameters
            parameters[param_name] -= self.learning_rate * m_hat / (np.sqrt

        return parameters
```

7 Complete RNN Model Implementation

Listing 6: Complete RNN model with training loop

```
class RNNModel:
    def __init__(self, input_size, hidden_size, output_size, cell_type='LST
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.cell_type = cell_type
```

```
        # Initialize RNN cell
        if cell_type == 'LSTM':
            self.rnn_cell = LSTMCell(input_size, hidden_size)
        elif cell_type == 'GRU':
            self.rnn_cell = GRUCell(input_size, hidden_size)
        else:  # Simple RNN
            self.W_h = np.random.randn(hidden_size, hidden_size) * 0.01
            self.W_x = np.random.randn(hidden_size, input_size) * 0.01
            self.b_h = np.zeros((hidden_size, 1))

        # Output layer
        self.W_y = np.random.randn(output_size, hidden_size) * 0.01
        self.b_y = np.zeros((output_size, 1))

    def forward(self, X):
        """
        Forward pass through entire sequence
        X: input sequence [seq_length, batch_size, input_size]
        """
        batch_size = X.shape[1]
        seq_length = X.shape[0]

        # Initialize hidden state
        if self.cell_type == 'LSTM':
            h = np.zeros((self.hidden_size, batch_size))
            C = np.zeros((self.hidden_size, batch_size))
            hidden_states = []
            cell_states = []
        else:
            h = np.zeros((self.hidden_size, batch_size))
            hidden_states = []

        outputs = []

        for t in range(seq_length):
            if self.cell_type == 'LSTM':
                h, C = self.rnn_cell.forward(X[t], h, C)
                hidden_states.append(h)
                cell_states.append(C)
            elif self.cell_type == 'GRU':
                h = self.rnn_cell.forward(X[t], h)
                hidden_states.append(h)
            else:  # Simple RNN
                h = np.tanh(np.dot(self.W_h, h) + np.dot(self.W_x, X[t]) +
                hidden_states.append(h)

            # Output layer
```

```
            y_t = np.dot(self.W_y, h) + self.b_y
            if self.output_size == 1:  # Binary classification
                y_t = self._sigmoid(y_t)
            else:  # Multi-class classification
                y_t = self._softmax(y_t)

            outputs.append(y_t)

        # Cache for backward pass
        self.cache = {
            'X': X,
            'hidden_states': hidden_states,
            'outputs': outputs
        }

        if self.cell_type == 'LSTM':
            self.cache['cell_states'] = cell_states

        return outputs

    def backward(self, dY):
        """
        Backward pass through entire sequence
        dY: gradient of loss w.r.t. outputs
        """
        X = self.cache['X']
        hidden_states = self.cache['hidden_states']
        outputs = self.cache['outputs']

        batch_size = X.shape[1]
        seq_length = X.shape[0]

        # Initialize gradients
        if self.cell_type == 'LSTM':
            dW, db = self.rnn_cell.backward.initialize_gradients()
        elif self.cell_type == 'GRU':
            dW_z, dW_r, dW_h = np.zeros_like(self.rnn_cell.W_z), np.zeros_l
            db_z, db_r, db_h = np.zeros_like(self.rnn_cell.b_z), np.zeros_l

        dW_y = np.zeros_like(self.W_y)
        db_y = np.zeros_like(self.b_y)

        # Initialize gradient for next hidden state
        dh_next = np.zeros((self.hidden_size, batch_size))
        if self.cell_type == 'LSTM':
            dC_next = np.zeros((self.hidden_size, batch_size))

        # Backward through time
```

```
    for t in reversed(range(seq_length)):
        # Gradient from output layer
        if self.output_size == 1:
            dy_t = outputs[t] * (1 - outputs[t]) * dY[t]
        else:
            dy_t = dY[t]

        # Output layer gradients
        dW_y += np.dot(dy_t, hidden_states[t].T)
        db_y += np.sum(dy_t, axis=1, keepdims=True)

        # Gradient w.r.t. hidden state
        dh = np.dot(self.W_y.T, dy_t) + dh_next

        if self.cell_type == 'LSTM':
            dx, dh_prev, dC_prev, dW_t, db_t = self.rnn_cell.backward(c
            dW += dW_t
            db += db_t
            dh_next = dh_prev
            dC_next = dC_prev
        elif self.cell_type == 'GRU':
            dx, dh_prev, dW_z_t, dW_r_t, dW_h_t, db_z_t, db_r_t, db_h_t
            dW_z += dW_z_t
            dW_r += dW_r_t
            dW_h += dW_h_t
            db_z += db_z_t
            db_r += db_r_t
            db_h += db_h_t
            dh_next = dh_prev

    gradients = {
        'W_y': dW_y / batch_size,
        'b_y': db_y / batch_size
    }

    if self.cell_type == 'LSTM':
        gradients.update(self.rnn_cell.get_gradients())
    elif self.cell_type == 'GRU':
        gradients.update({
            'W_z': dW_z / batch_size,
            'W_r': dW_r / batch_size,
            'W_h': dW_h / batch_size,
            'b_z': db_z / batch_size,
            'b_r': db_r / batch_size,
            'b_h': db_h / batch_size
        })

    return gradients
```

```python
    def train(self, X_train, y_train, epochs=100, learning_rate=0.01):
        """
        Complete training loop
        """
        losses = []

        for epoch in range(epochs):
            # Forward pass
            outputs = self.forward(X_train)

            # Calculate loss
            loss = self._calculate_loss(outputs[-1], y_train)
            losses.append(loss)

            # Backward pass
            gradients = self.backward(self._loss_gradient(outputs[-1], y_tr

            # Update parameters
            self._update_parameters(gradients, learning_rate)

            if epoch % 10 == 0:
                print(f"Epoch {epoch}, Loss: {loss:.4f}")

        return losses

    def _sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def _softmax(self, x):
        exp_x = np.exp(x - np.max(x, axis=0, keepdims=True))
        return exp_x / np.sum(exp_x, axis=0, keepdims=True)
```

8 Performance Metrics and Evaluation

8.1 Evaluation Metrics Implementation

Listing 7: Evaluation metrics for sequence models

```python
def accuracy_score(y_pred, y_true):
    """
    Calculate accuracy for classification tasks
    """
    if y_pred.shape != y_true.shape:
        y_pred = np.argmax(y_pred, axis=0)
        y_true = np.argmax(y_true, axis=0)

    return np.mean(y_pred == y_true)
```

```
def precision_score(y_pred, y_true):
    """
    Calculate precision for binary classification
    """
    tp = np.sum((y_pred == 1) & (y_true == 1))
    fp = np.sum((y_pred == 1) & (y_true == 0))

    return tp / (tp + fp + 1e-10)

def recall_score(y_pred, y_true):
    """
    Calculate recall for binary classification
    """
    tp = np.sum((y_pred == 1) & (y_true == 1))
    fn = np.sum((y_pred == 0) & (y_true == 1))

    return tp / (tp + fn + 1e-10)

def f1_score(y_pred, y_true):
    """
    Calculate F1 score
    """
    precision = precision_score(y_pred, y_true)
    recall = recall_score(y_pred, y_true)

    return 2 * (precision * recall) / (precision + recall + 1e-10)

def perplexity_score(y_pred, y_true):
    """
    Calculate perplexity for language models
    """
    cross_entropy = -np.sum(y_true * np.log(y_pred + 1e-10))
    return np.exp(cross_entropy / y_true.shape[0])
```

9 Conclusion and Best Practices

9.1 Key Takeaways from PDF Analysis

- **Simple RNN Limitations**: Forget information quickly, cannot handle long sequences
- **Gradient Problems**: Vanishing/exploding gradients limit training
- **LSTM Solution**: Gates control information flow, solve long-term dependency problem
- **GRU Alternative**: Simpler architecture with comparable performance

9.2 Practical Recommendations

1. **Sequence Length**: Use LSTM for very long sequences (¿100 steps), GRU for moderate lengths
2. **Initialization**: Use Xavier/Glorot initialization for better convergence
3. **Gradient Clipping**: Always use gradient clipping (norm ¡= 5) for stability
4. **Dropout**: Apply dropout between RNN layers to prevent overfitting
5. **Batch Normalization**: Use layer normalization instead of batch normalization for RNNs
6. **Teacher Forcing**: Use during training for faster convergence
7. **Attention Mechanism**: Combine with RNN for better performance on long sequences

9.3 Future Directions

- **Transformers**: Have largely replaced RNNs for many NLP tasks
- **Attention is All You Need**: Self-attention mechanisms for parallel processing
- **BERT/GPT**: Pre-trained transformer models for transfer learning
- **Hybrid Models**: Combine RNNs with attention for specific applications