Recurrent Neural Networks (RNN): Logic and Mathematics

Hıncal Topçuoğlu Study Notes

February 23, 2026

1 Introduction to RNNs

Standard Feedforward Neural Networks are limited by the assumption that inputs are independent of one another. For sequential data—where the order of information matters (e.g., time series, natural language)—we require a model that can maintain state. Recurrent Neural Networks (RNNs) address this by introducing loops, allowing information to persist.

2 The Core Logic: Memory

The fundamental innovation of an RNN is the **hidden state** (h). This acts as the network's internal memory. At each time step t, the network takes the current input x_t and the memory from the previous time step h_{t-1} to update its current memory h_t.

3 Mathematical Formulation

The behavior of a vanilla RNN is defined by two primary equations for each time step t:

3.1 The Hidden State (Update Equation)

The hidden state is computed as a non-linear combination of the current input and the previous hidden state:

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

Where:

- h_t: Current hidden state (memory)
- h_{t-1}: Previous hidden state

- x_t: Input at time t
- W_{xh}: Input-to-hidden weight matrix
- W_{hh}: Hidden-to-hidden (recurrent) weight matrix
- b_h: Bias for the hidden state
- f: Activation function (typically tanh)

3.2 The Output Equation

The prediction or output at the current time step is derived from the current memory:

$$y_t = W_{hy}h_t + b_y \tag{2}$$

Where:

- y_t: Output (logits)
- W_{hy}: Hidden-to-output weight matrix
- b_y: Bias for the output

4 The Role of Weight Matrices

The matrices W_{xh}, W_{hh}, and W_{hy} represent the learned parameters of the network:

- W_{xh}: Filters the current input to find features relevant to the memory.
- W_{hh}: Decides which parts of the past memory remain relevant for the current state. This matrix is shared across all time steps.
- W_{hy}: Translates the abstract internal memory into a concrete output/prediction.

5 Visualization and Flow

5.1 Unrolling in Time

An RNN can be visualized as a sequence of identical layers. At each step t, the network receives input x_t and passes the hidden state h_t to the next step:

$$x_1 \rightarrow [RNN] \xrightarrow{h_1} [RNN] \xrightarrow{h_2} [RNN] \cdots \rightarrow y_t$$

5.2 Internal Data Flow

The data follows a specific sequence of operations:

1. **Input Transformation:** x_t is projected into the hidden space: $W_{xh} \cdot x_t$.
2. **Recurrent Transformation:** The previous memory h_{t-1} is processed: $W_{hh} \cdot h_{t-1}$.
3. **Integration:** The results are summed with a bias b_h.
4. **Activation:** The tanh function squashes the values into the $[-1, 1]$ range.
5. **Output Projection:** The new state h_t is mapped to the output space via W_{hy}.

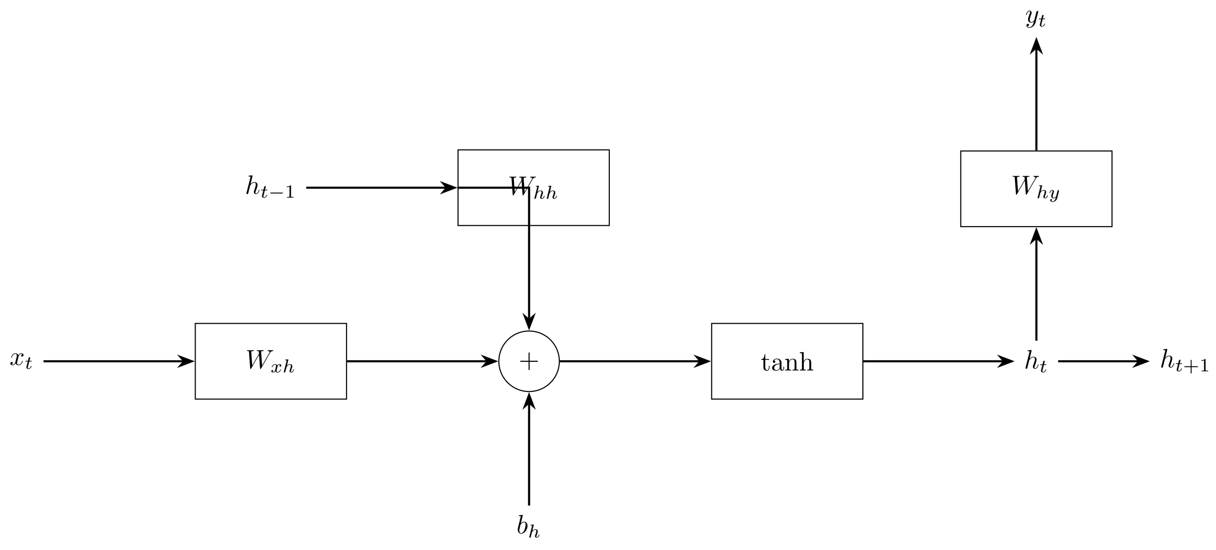

Figure 1: RNN Cell Internal Architecture

6 Dimensionality Analysis

Understanding matrix dimensions is critical for implementation. Let:

- d: Dimension of the input vector x_t.
- h: Number of hidden units (size of h_t).
- q: Dimension of the output vector y_t.

The shapes of the tensors are as follows:

- $x_t \in \mathbb{R}^{d \times 1}$
- $h_t, b_h \in \mathbb{R}^{h \times 1}$
- $y_t, b_y \in \mathbb{R}^{q \times 1}$
- $W_{xh} \in \mathbb{R}^{h \times d}$
- $W_{hh} \in \mathbb{R}^{h \times h}$
- $W_{hy} \in \mathbb{R}^{q \times h}$

6.1 Parameter Calculation

To calculate the total number of trainable parameters in a vanilla RNN, we sum the number of elements in each weight matrix and bias vector:

1. **Input to Hidden:** W_{xh} has $h \times d$ parameters, b_h has h parameters.
2. **Hidden to Hidden:** W_{hh} has $h \times h$ parameters.
3. **Hidden to Output:** W_{hy} has $q \times h$ parameters, b_y has q parameters.

The total number of parameters is:

$$\text{Total Parameters} = (h \cdot d + h) + (h \cdot h) + (q \cdot h + q) = h(d + h + q + 1) + q \quad (3)$$

Note that the number of parameters does *not* depend on the sequence length T, as the same weights are reused at every time step.

Example Calculation

Consider an RNN designed for character prediction with the following dimensions:

- Input size ($d = 26$): (e.g., 26 letters of the alphabet)
- Hidden size ($h = 100$): (Number of neurons in the memory layer)
- Output size ($q = 26$): (Probability distribution over 26 letters)

The parameter count would be:

- $W_{xh} : 100 \times 26 = 2,600$
- $b_h : 100$
- $W_{hh} : 100 \times 100 = 10,000$
- $W_{hy} : 26 \times 100 = 2,600$
- $b_y : 26$

Total Parameters: $2,600 + 100 + 10,000 + 2,600 + 26 = 15,326$ Using the formula: $100(26 + 100 + 26 + 1) + 26 = 100(153) + 26 = 15,326$.

7 Activation Functions and Why Tanh?

In the hidden state update equation (1), the activation function f is almost exclusively the hyperbolic tangent (tanh) in vanilla RNNs.

7.1 Range Control and Stability

The tanh function squashes any real-valued input into the range $(-1, 1)$. Since an RNN involves repeated matrix multiplications across sequence steps, a function like ReLU (which can output values ≥ 1) could cause the hidden state values to explode exponentially over time. tanh ensures the "memory" remains bounded and numerically stable.

7.2 Zero-Centered Output

Unlike the Sigmoid function (range $(0, 1)$), tanh is zero-centered. This property helps the optimization process by ensuring that the gradients during backpropagation have more balanced updates, leading to faster convergence.

7.3 Computational Efficiency of Gradients

The derivative of tanh is mathematically simple and computationally efficient to calculate:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x) \tag{4}$$

Since we already store the output of the activation function (tanh) during the forward pass, computing the derivative during the backward pass requires only one subtraction and one multiplication.

8 The Loss Function

To train the network, we must quantify its performance. For tasks like next-character prediction, we use the **Cross-Entropy Loss**.

8.1 Sequence Loss

The total loss L for a sequence of length T is the sum of the individual losses at each time step:

$$L = \sum_{t=1}^{T} L_t \tag{5}$$

8.2 Time-step Loss

At each time step t, if y_t is the predicted probability distribution and p_t is the target (one-hot ground truth), the loss is:

$$L_t = -\sum_i p_{t,i} \log(y_{t,i}) \tag{6}$$

9 Backpropagation Through Time (BPTT)

BPTT is the algorithm used to calculate the gradients of the loss with respect to the weights in an RNN.

10 Gradient Computation Recipes

For implementation, the gradients at time step t are calculated as follows (assuming Cross-Entropy loss and Softmax output):

10.1 Output Gradients

The error at the output layer:

$$dy_t = y_t - p_t \tag{7}$$

Gradients for the output weights and bias:

$$dW_{hy} += dy_t \cdot h_t^T, \quad db_y += dy_t \tag{8}$$

10.2 Hidden State Gradients

The gradient flowing into the hidden state h_t comes from both the current output and the next time step:

$$dh_t = (W_{hy}^T \cdot dy_t) + dh_{next} \tag{9}$$

Backpropagating through the tanh activation:

$$dz_t = dh_t \cdot (1 - h_t^2) \tag{10}$$

10.3 Recurrent and Input Gradients

Gradients for the internal weights and bias:

$$dW_{xh} += dz_t \cdot x_t^T, \quad dW_{hh} += dz_t \cdot h_{t-1}^T, \quad db_h += dz_t \tag{11}$$

The gradient to be passed to the previous time step:

$$dh_{prev} = W_{hh}^T \cdot dz_t \tag{12}$$

10.4 Summation of Gradients

Because the same weight matrices (parameters) are reused at every time step, the gradient of the total loss with respect to a weight matrix W is the sum of the gradients from each time step:

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W} \tag{13}$$

10.5 The Temporal Dependency

The fundamental challenge of BPTT is that the hidden state h_t at time t depends on h_{t-1}, which in turn depends on h_{t-2}, and so on.

10.5.1 Logic of the Chain Rule

The **Chain Rule** is a fundamental rule in calculus used to find the derivative of a composite function (a function within a function). In the context of RNNs, we use it to calculate how a small change in our weights (W) at the very beginning of a long sequence affects the final loss (L) at the very end.

The aim is to measure the **contribution of early states to the final error**. Because the state at step T was built step-by-step using the state at $T-1$, $T-2$, and so on, the error at the last state is essentially "inherited" from all previous states. The Chain Rule allows us to "unroll" this dependency and see how much the very first state's calculation affected the final result.

10.5.2 Mathematical Chain

Applying the chain rule to calculate $\frac{\partial L_t}{\partial W_{hh}}$ involves propagating the gradient backwards through the entire history of hidden states:

$$\frac{\partial L_t}{\partial W_{hh}} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial h_t} \cdot \underbrace{\left(\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right)}_{\text{Chain from } k \text{ to } t} \cdot \frac{\partial h_k}{\partial W_{hh}} \tag{14}$$

This recursive multiplication is what leads to the **Vanishing or Exploding Gradient** problem. If the gradients in the chain are mostly small (< 1), the product shrinks to zero, and the network "forgets" long-term dependencies. If they are large (> 1), the product grows to infinity (exploding gradients).

11 Gradient Clipping

To mitigate the problem of **Exploding Gradients**, where the gradient norm grows so large that it causes weight updates to overshoot and the network to become unstable (resulting in NaN values), we apply gradient clipping.

If the norm of the total gradient $\|\mathbf{g}\|$ exceeds a predefined threshold τ, we rescale the gradient:

$$\mathbf{g} = \mathbf{g} \cdot \frac{\tau}{\max(\tau, \|\mathbf{g}\|)} \tag{15}$$

This operation preserves the direction of the gradient vector but limits its magnitude to τ.

12 Softmax and Numerical Stability

For classification tasks, the raw output vector y_t (logits) is converted into a probability distribution using the **Softmax** function.

12.1 Standard Softmax

The probability for class i is calculated as:

$$P(y_{t,i}) = \frac{e^{y_{t,i}}}{\sum_j e^{y_{t,j}}} \tag{16}$$

12.2 Numerical Stability Trick

Directly calculating e^x for large x can lead to numerical overflow. To prevent this, we subtract the maximum value of the input vector before exponentiation:

$$\text{StableSoftmax}(y_t) = \text{Softmax}(y_t - \max(y_t)) \tag{17}$$

This mathematically equivalent operation ensures that the largest exponent is $e^0 = 1$, preventing overflow while maintaining the correct probability ratios.

13 Python Implementation

The following implementation uses NumPy to build a character-level RNN. It includes forward and backward passes, gradient clipping to prevent exploding gradients, and a sampling method for inference.

```python
import numpy as np

class SimpleRNN:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Weight Initialization (Scaled random)
        self.Wxh = np.random.randn(hidden_size, input_size) * 0.01
        self.Whh = np.random.randn(hidden_size, hidden_size) * 0.01
        self.Why = np.random.randn(output_size, hidden_size) * 0.01

        # Biases
        self.bh = np.zeros((hidden_size, 1))
        self.by = np.zeros((output_size, 1))

    def forward(self, inputs, h_prev):
        xs, hs, ps = {}, {}, {}, {}
        hs[-1] = np.copy(h_prev)

        for t in range(len(inputs)):
            # One-hot encoding concept
            xs[t] = np.zeros((self.input_size, 1))
            xs[t][inputs[t]] = 1

            # Hidden state update
            hs[t] = np.tanh(np.dot(self.Wxh, xs[t]) + np.dot(self.
    Whh, hs[t-1]) + self.bh)

            # Output (unnormalized log probabilities)
            ys[t] = np.dot(self.Why, hs[t]) + self.by

            # Softmax (numerically stable)
            exp_y = np.exp(ys[t] - np.max(ys[t]))
            ps[t] = exp_y / np.sum(exp_y)

        return xs, hs, ps

    def backward(self, xs, hs, ps, targets):
        dWxh = np.zeros_like(self.Wxh)
        dWhh = np.zeros_like(self.Whh)
        dWhy = np.zeros_like(self.Why)
        dbh = np.zeros_like(self.bh)
        dby = np.zeros_like(self.by)
        dh_next = np.zeros_like(hs[0])

        for t in reversed(range(len(xs))):
            dy = np.copy(ps[t])
            dy[targets[t]] -= 1  # Softmax + Cross Entropy gradient

            dWhy += np.dot(dy, hs[t].T)
            dby += dy

            dh = np.dot(self.Why.T, dy) + dh_next
            dh_raw = (1 - hs[t] * hs[t]) * dh  # tanh derivative

            dbh += dh_raw
            dWxh += np.dot(dh_raw, xs[t].T)
            dWhh += np.dot(dh_raw, hs[t-1].T)

            dh_next = np.dot(self.Whh.T, dh_raw)

        # Gradient Clipping (Global)
        for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
            np.clip(dparam, -5, 5, out=dparam)

        return dWxh, dWhh, dWhy, dbh, dby

    def update_weights(self, grads, lr=0.1):
        dWxh, dWhh, dWhy, dbh, dby = grads
        self.Wxh -= lr * dWxh
        self.Whh -= lr * dWhh
        self.Why -= lr * dWhy
        self.bh -= lr * dbh
        self.by -= lr * dby

    def sample(self, h, start_index, n):
        x = np.zeros((self.input_size, 1))
        x[start_index] = 1
        indices = []
        for t in range(n):
            h = np.tanh(np.dot(self.Wxh, x) + np.dot(self.Whh, h) + self.bh)
            y = np.dot(self.Why, h) + self.by
            exp_y = np.exp(y - np.max(y))
            p = exp_y / np.sum(exp_y)

            idx = np.random.choice(range(self.input_size), p=p.ravel())
            x = np.zeros((self.input_size, 1))
            x[idx] = 1
            indices.append(idx)
        return indices

# Define Data
chars = sorted(list(set("hello")))
char_to_ix = {ch: i for i, ch in enumerate(chars)}
ix_to_char = {i: ch for i, ch in enumerate(chars)}
data = [char_to_ix[ch] for ch in "hello"]

# Input/Target for "hello" -> "ello "
inputs = data[:-1]
targets = data[1:]

# Instantiate Model
hidden_size = 100
rnn = SimpleRNN(len(chars), hidden_size, len(chars))
h_prev = np.zeros((hidden_size, 1))

# Training Loop
for epoch in range(100):
    xs, hs, ps = rnn.forward(inputs, h_prev)

    loss = 0
    for t in range(len(targets)):
        loss += -np.log(ps[t][targets[t], 0])

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

    grads = rnn.backward(xs, hs, ps, targets)
    rnn.update_weights(grads)

# Prediction / Inference
print("\nPrediction after training:")
h = np.zeros((hidden_size, 1))
start_char = "h"
```

```python
print(start_char, end="")
result_indices = rnn.sample(h, char_to_ix[start_char], 4)
for idx in result_indices:
    print(ix_to_char[idx], end="")
print()
```

References

[1] Karpathy, Andrej (2015). *The Unreasonable Effectiveness of Recurrent Neural Networks.* Andrej Karpathy Blog. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`

[2] Karpathy, Andrej. *Minimal character-level language model with a Vanilla RNN-cell (min-char-rnn.py).* GitHub Gist. `https://gist.github.com/karpathy/d42664d5d53a23362a71`

[3] Bengio, Y., Simard, P., and Frasconi, P. (1994). *Learning long-term dependencies with gradient descent is difficult.* IEEE Transactions on Neural Networks, 5(2), 157-166.

[4] Hochreiter, S., and Schmidhuber, J. (1997). *Long Short-Term Memory.* Neural Computation, 9(8), 1735-1780.

[5] Graves, A. (2013). *Generating Sequences With Recurrent Neural Networks.* arXiv preprint arXiv:1308.0850.