MA477: Data Science
Lesson 28 - 30 Outline — 26 March / 08 April 2026
United States Military Academy, West Point
Instructor: MAJ Patrick Kuiper

1 Administrative

- Calendar review
- Student review
- RNN Discussion
- Coding Exercise

2 RNN Lesson Objectives

- Be able to explain the structure of recurrent neural networks, including how they process sequential data and the role of hidden states.
- Be able to build and train RNN models using Python and deep learning frameworks
- Be able to evaluate RNN performance using appropriate metrics, compare RNNs with alternative models

3 Recurrent Neural Networks: Guided Lesson via Questions

4 Recurrent Neural Networks (RNNs): Intuition, Notation, and Motivation

Intuition and Sequential Structure

Traditional feedforward neural networks assume that inputs are independent. However, many real-world datasets are *sequential*, meaning that the current input depends on previous inputs. Examples include time series, text, and speech.

A **Recurrent Neural Network (RNN)** addresses this by maintaining a *hidden state* that carries information forward through time. At each time step t, the model takes as input:

- The current input vector $\mathbf{x}^{(t)}$
- The previous hidden state $\mathbf{h}^{(t-1)}$

It then produces:

- A new hidden state $\mathbf{h}^{(t)}$
- Optionally, an output $\mathbf{y}^{(t)}$

Unrolled Computational Graph (Visual)

An RNN can be visualized by "unrolling" it across time steps:

$$\begin{array}{ccccccc} \mathbf{x}^{(1)} & \rightarrow & \mathbf{x}^{(2)} & \rightarrow & \cdots & \rightarrow & \mathbf{x}^{(T)} \\ \downarrow & & \downarrow & & & & \downarrow \\ \mathbf{h}^{(1)} & \rightarrow & \mathbf{h}^{(2)} & \rightarrow & \cdots & \rightarrow & \mathbf{h}^{(T)} \\ \downarrow & & \downarrow & & & & \downarrow \\ \mathbf{y}^{(1)} & & \mathbf{y}^{(2)} & & & & \mathbf{y}^{(T)} \end{array}$$

Each block uses the *same parameters*, which is a key property of RNNs called **weight sharing**.

Mathematical Formulation

Using notation consistent with *Hands-On Machine Learning*, the hidden state update is:

$$\mathbf{h}^{(t)} = \tanh\left(\mathbf{W}_x\mathbf{x}^{(t)} + \mathbf{W}_h\mathbf{h}^{(t-1)} + \mathbf{b}\right)$$

where:

- $\mathbf{x}^{(t)} \in \mathbb{R}^{n_x}$ is the input at time t
- $\mathbf{h}^{(t)} \in \mathbb{R}^{n_h}$ is the hidden state
- $\mathbf{W}_x \in \mathbb{R}^{n_h \times n_x}$ maps input to hidden
- $\mathbf{W}_h \in \mathbb{R}^{n_h \times n_h}$ maps hidden to hidden
- $\mathbf{b} \in \mathbb{R}^{n_h}$ is a bias term

If outputs are produced at each step:

$$\mathbf{y}^{(t)} = \mathbf{W}_y\mathbf{h}^{(t)} + \mathbf{c}$$

Training via Backpropagation Through Time (BPTT)

To train the RNN, we define a loss across all time steps:

$$\mathcal{L} = \sum_{t=1}^{T} \ell\left(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)}\right)$$

The network is then trained using **Backpropagation Through Time (BPTT)**, which consists of:

- Unrolling the network across time
- Applying standard backpropagation
- Accumulating gradients through all time steps

This introduces challenges such as **vanishing and exploding gradients**, particularly for long sequences.

Motivating Questions

1. **Why do we need a hidden state $\mathbf{h}^{(t)}$ instead of processing each $\mathbf{x}^{(t)}$ independently?**

   Explanation: Sequential data contains temporal dependencies. The hidden state acts as a memory that summarizes past inputs, allowing the model to condition predictions on previous context.

2. **Why are the same weight matrices $\mathbf{W}_x$ and $\mathbf{W}_h$ reused at every time step?**

   Explanation: This enforces consistency across time and allows the model to generalize to sequences of varying length. It also drastically reduces the number of parameters compared to using separate weights at each step.

3. **Why does training require unrolling the network and using Backpropagation Through Time?**

   Explanation: The hidden state at each time depends recursively on all previous states. To compute gradients correctly, we must account for how earlier inputs influence later outputs, which requires propagating gradients through all time steps.

4. **Why do vanishing and exploding gradients occur in RNNs?**

   Explanation: During BPTT, gradients are repeatedly multiplied by $\mathbf{W}_h$. If its eigenvalues are less than one, gradients shrink (vanish); if greater than one, they grow exponentially (explode). This makes learning long-term dependencies difficult.

4.1 Backpropagation Through Time: Numerical Example with $\tanh$

The $\tanh$ Activation and Its Derivative

The hyperbolic tangent function is commonly used in basic RNNs:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

It maps inputs to the range $(-1, 1)$, which helps stabilize the hidden state.
Its derivative has a particularly convenient form:

$$\frac{d}{dz}\tanh(z) = 1 - \tanh^2(z)$$

This means that if we already know $h = \tanh(z)$, then:

$$\frac{d}{dz}\tanh(z) = 1 - h^2$$

Setup of a Simple RNN

We consider a minimal RNN with:

- Sequence length $T = 2$
- Scalar inputs, hidden states, and outputs

The model is:

$$h^{(t)} = \tanh\left(W_x x^{(t)} + W_h h^{(t-1)}\right)$$
$$y^{(t)} = W_y h^{(t)}$$

Loss at each step:

$$L^{(t)} = \left(y^{(t)} - \hat{y}^{(t)}\right)^2$$
$$L = L^{(1)} + L^{(2)}$$

Given Values

- $W_x = 1, \quad W_h = 2, \quad W_y = 1$
- $x^{(1)} = 1, \quad x^{(2)} = 2$
- $h^{(0)} = 0$
- $\hat{y}^{(1)} = 0, \quad \hat{y}^{(2)} = 1$

Forward Pass

Time Step $t = 1$

$$z^{(1)} = 1 \cdot 1 + 2 \cdot 0 = 1$$
$$h^{(1)} = \tanh(1) \approx 0.761$$
$$y^{(1)} = 0.761$$
$$L^{(1)} = (0.761 - 0)^2 \approx 0.579$$

Time Step $t = 2$

$$z^{(2)} = 1 \cdot 2 + 2 \cdot 0.761 = 3.522$$
$$h^{(2)} = \tanh(3.522) \approx 0.998$$
$$y^{(2)} = 0.998$$
$$L^{(2)} = (0.998 - 1)^2 \approx 0.000004$$

$$L \approx 0.579$$

Backward Pass (Backpropagation Through Time)

We compute the gradient $\frac{\partial L}{\partial W_h}$.

Step 1: Gradients at $t = 2$

$$\frac{\partial L}{\partial y^{(2)}} = 2(0.998 - 1) \approx -0.004$$
$$\frac{\partial L}{\partial h^{(2)}} = -0.004$$

Using $\tanh'(z) = 1 - h^2$:

$$1 - (0.998)^2 \approx 0.004$$

$$\delta^{(2)} = -0.004 \cdot 0.004 \approx -0.000016$$

$$\frac{\partial L}{\partial W_h}\Big|_{t=2} = \delta^{(2)} \cdot h^{(1)} \approx -0.000012$$

Step 2: Backpropagate to $t = 1$ Contribution from future time step:

$$\frac{\partial L}{\partial h^{(1)}}\Big|_{\text{from } t=2} = \delta^{(2)} \cdot W_h \approx -0.000032$$

Direct contribution from $t = 1$:

$$\frac{\partial L}{\partial y^{(1)}} = 2(0.761 - 0) = 1.522$$
$$\frac{\partial L}{\partial h^{(1)}}\Big|_{\text{from } t=1} = 1.522$$

Total:

$$\frac{\partial L}{\partial h^{(1)}} \approx 1.522 - 0.000032 \approx 1.522$$

Derivative of $\tanh$:

$$1 - (0.761)^2 \approx 0.42$$

$$\delta^{(1)} \approx 1.522 \cdot 0.42 \approx 0.64$$

$$\frac{\partial L}{\partial W_h}\Big|_{t=1} = \delta^{(1)} \cdot h^{(0)} = 0$$

Final Gradient

$$\frac{\partial L}{\partial W_h} \approx -0.000012$$

Interpretation

- Gradients accumulate across time steps
- Each step contributes both directly and through future dependencies
- The small derivative of $\tanh$ at $h^{(2)} \approx 0.998$ causes the gradient to shrink significantly

Repeated multiplication by small derivatives leads to the **vanishing gradient problem**, which makes learning long-term dependencies difficult in basic RNNs.

5 RNN Input–Output Structures and Training Implications

Recurrent Neural Networks (RNNs) are designed to model sequential data by maintaining a hidden state that evolves over time. Different tasks require different mappings between input and output structures. The three most common configurations are sequence-to-sequence, sequence-to-vector, and vector-to-sequence.

1. Sequence-to-Vector (Many-to-One)

A sequence-to-vector model takes an input sequence and produces a single output:

$$\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(T)}\} \;\rightarrow\; \mathbf{y}$$

Examples could include predicting sentiment score of a movie review.

Typically, the final hidden state is used:

$$\mathbf{y} = g\big(\mathbf{W}_y \mathbf{h}^{(T)} + \mathbf{c}\big)$$

Implications:

- The loss is computed only at the final time step:

$$\mathcal{L} = \ell(\mathbf{y}, \hat{\mathbf{y}})$$

- Gradients must propagate through all time steps to update early hidden states.
- Susceptible to vanishing gradients for long sequences.

2. Sequence-to-Sequence (Many-to-Many)

A sequence-to-sequence model maps an input sequence to an output sequence:

$$\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)}\} \;\rightarrow\; \{\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(T)}\}$$

Examples could include a question answer model.

At each time step:

$$\mathbf{y}^{(t)} = g\big(\mathbf{W}_y \mathbf{h}^{(t)} + \mathbf{c}\big)$$

Implications:

- Loss is computed at every time step:

$$\mathcal{L} = \sum_{t=1}^{T} \ell(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)})$$

- Gradients are distributed across time steps, improving learning signal.
- Still requires Backpropagation Through Time (BPTT) across the full sequence.

3. Vector-to-Sequence (One-to-Many)

A vector-to-sequence model generates a sequence from a single input:

$$\mathbf{x} \rightarrow \{\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(T)}\}$$

Examples could include providing a caption for a image.
The initial hidden state is typically initialized from the input:

$$\mathbf{h}^{(0)} = f(\mathbf{W}_x \mathbf{x} + \mathbf{b})$$

Then the RNN generates outputs over time.
Implications:

- Loss is computed across all generated outputs:

$$\mathcal{L} = \sum_{t=1}^{T} \ell(\mathbf{y}^{(t)}, \hat{\mathbf{y}}^{(t)})$$

- Errors at later time steps influence earlier hidden states through BPTT.

Backpropagation Through Time (BPTT)

Training an RNN requires unrolling it across time and applying backpropagation:

- The same parameters are reused at each time step.
- Gradients are computed using the chain rule across time.
- Each parameter accumulates contributions from all time steps.

For example, for recurrent weights $\mathbf{W}_h$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_h} = \sum_{t=1}^{T} \delta^{(t)} \, \mathbf{h}^{(t-1)^\top}$$

where $\delta^{(t)}$ is the error signal at time t.

Partial Gradients and Temporal Dependencies

A key property of RNN training is that gradients are *partial* with respect to each time step:

- The loss at time t depends on all previous hidden states.
- Therefore, gradients propagate backward through multiple time steps.
- Earlier time steps receive gradients through repeated multiplication by $\mathbf{W}_h$ and activation derivatives.

This leads to:

- **Vanishing gradients**: repeated multiplication by values less than one shrinks gradients.
- **Exploding gradients**: repeated multiplication by values greater than one amplifies gradients.

Key Intuition

- Sequence-to-vector models concentrate learning at the final step.
- Sequence-to-sequence models distribute learning across time.
- Vector-to-sequence models generate outputs step-by-step from an initial state.
- BPTT links all time steps through shared parameters and accumulated gradients.

RNN structure determines where loss is applied, which in turn determines how gradients flow through time. This directly affects the model's ability to learn short- and long-term dependencies.

Summary

RNNs extend neural networks to sequential data by introducing a hidden state that evolves over time. Their structure is defined by shared parameters and recursive updates, and they are trained using Backpropagation Through Time. While powerful, they require careful handling due to optimization challenges.