Lecture 13

Recurrent Neural Networks for Time-Dependent Chemistry

Contents

In the previous lectures, we mainly focused on one type of learning task: given features X and labels y, we aim to build a model that connects X and y, i.e.,

$$\text{Label Prediction:} \quad X \xrightarrow{\text{Predict}} y. \tag{1}$$

These prediction tasks are suitable for **time-independent** problems, such as predicting chemical properties, reaction yields, or electronic structure outputs.

However, many chemical problems involve time evolution, including reaction kinetics, time-resolved spectroscopy, molecular dynamics, and electrochemistry. These tasks require a **time-dependent** approach: predicting the values of a variable as it evolves over time.

In **time series prediction**, instead of a single input-output pair, we are given the values of a variable x at different *time steps* $\{x(t_0), x(t_1), \cdots, x(T)\}$. The goal is to use the past values to predict what comes next, i.e.,

$$\text{Time Series Prediction:} \quad x(t_0) \xrightarrow{\text{Predict}} x(t_1) \xrightarrow{\text{Predict}} \cdots \xrightarrow{\text{Predict}} x(T) \tag{2}$$

A very naive approach is to assume that $x(t_i)$ depends only on $x(t_{i-1})$, i.e., a first-order Markov assumption. In many cases, this is too simplistic.

To see why, consider language as an example. A sentence can be seen as a time series. If you hear the word "an," many words could follow, such as "idea", "item" or "apple." But if you hear "eat an," you already have a strong hint that the next word should be a food-related term, e.g., "apple." The additional context matters.

We call the values of $x(t)$ before $x(t_i)$ the **memory**. Just like human memory, recent events generally have more influence than older ones. Therefore, instead of storing the entire history, it is

often sufficient, and more practical, to track the recent past over several time steps.

As a result, a model for time series prediction should be able to do two things:

1. Predict the next value of x.
2. Store and update information about the memory (recent past).

Recurrent Neural Networks (RNNs) are a class of artificial neural networks designed to process sequential or time-series data. In the following, we introduce specific RNN architectures that can fulfill the two tasks outlined above.

1 A Vanilla Implementation of RNN

We first introduce a vanilla realization of RNN. The key is to introduce a hidden layer $\mathbf{h}_t$ to store the memory.

The memory at the t-th step, $\mathbf{h}_t$, depends on the previous memory $\mathbf{h}_{t-1}$ and the *current* value $\mathbf{x}_t$:

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t). \tag{3}$$

Here, we do not know the function form of f, and the simplest solution is to use a simple single-layer mapping:

$$\mathbf{h}_t = \phi(W_x \mathbf{x}_t + W_h \mathbf{h}_{t-1} + b), \tag{4}$$

where W_x and W_h are weights, b is the bias, and $\phi(\cdot)$ is an activation function.

Once we have the hidden layer $\mathbf{h}_t$ (the memory), we can predict the next x value:

$$\hat{x}_{t+1} = W_o \mathbf{h}_t + c, \tag{5}$$

which is a simple affine function with weight W_x and bias c. We use the hat symbol to denote predicted value.

1.1 Implementation Details

1.1.1 Initial State

We assume that $\mathbf{h}_0 = \mathbf{0}$ or a learnable vector.

1.1.2 Using the Same Weights and Biases

One simple design of the vanilla RNN is that the weights and biases in Eqs. (4) and (5) remains the same for every time step.

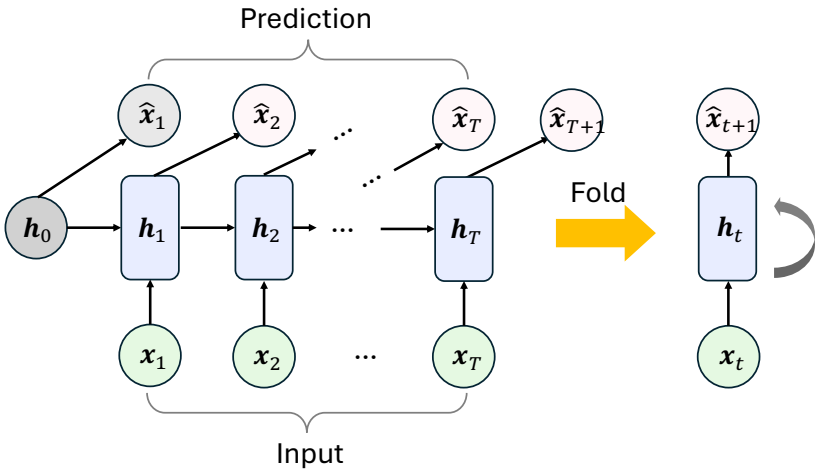

Figure 1: RNN architecture and the folded (rolled-up) version.

Therefore, there are only the following parameters in the above RNN architecture

$$W_x, W_h, W_o, b, c. \tag{6}$$

This means we believe that each update step $\mathbf{x}_t \rightarrow \mathbf{x}_{t-1}$ is based on the same update rule.

Because we are using the same weights and biases each step, we can simplify the RNN illustration as an folded (rolled-up) version. The unrolled and rolled-up versions of the RNN is illustrated in Fig. 1.

One time step is often called a **cell**.

1.1.3 Input and Output

Typically, the input to an RNN is defined as the **first** $(T-1)$ states of a sequence, while the target consists of the subsequent $(T-1)$ states:

$$\begin{aligned} \text{Input:} \quad & (\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_{T-1}) \\ \text{Target:} \quad & (\mathbf{x}_2, \mathbf{x}_3, \cdots, \mathbf{x}_T) \end{aligned} \tag{7}$$

From the input sequence, the RNN produces the following predictions:

$$\text{Predictions (outputs):} \quad (\hat{\mathbf{x}}_2, \hat{\mathbf{x}}_3, \cdots, \hat{\mathbf{x}}_T). \tag{8}$$

The predictions can then be directly compared with the reference targets to evaluate the loss.

In some cases, a special initial value $\mathbf{x}_0$ is defined, from which $\hat{\mathbf{x}}_1$ is predicted. This allows all

time steps to be used in the loss evaluation. An example of this setup is shown in the molecular generation task in Section 3.2.

1.1.4 Loss Function

To train the RNN, i.e., to update the weights and biases, we minimize the following loss function:

$$\mathcal{L} = \sum_t (\mathbf{x}_t - \hat{\mathbf{x}}_t)^2 , \tag{9}$$

where $\mathbf{x}_t$ denotes the reference value at time step t, and $\hat{\mathbf{x}}_t$ is the corresponding prediction produced by the RNN.

1.1.5 Inference

Once the RNN is trained, i.e., the parameters W_x, W_h, W_o, b, and c have been optimized, it can be used to generate a time series. Starting from an initial input $\mathbf{x}_1$,

$$\mathbf{x}_1 \to \mathbf{h}_1 \to \hat{\mathbf{x}}_2 \to \mathbf{h}_2 \to \cdots \to \hat{\mathbf{x}}_{T'}. \tag{10}$$

Because the parameters are shared across all time steps, the RNN can generate sequences of arbitrary length.

1.1.6 Deep RNN

A single-layer RNN as in Eq. (4) can sometimes be too simple for modeling complex temporal patterns. One way to increase expressiveness is to stack multiple recurrent layers.

For a two-layer RNN, the hidden states are computed as

$$\begin{aligned} \mathbf{h}_t^{(1)} &= \phi\big(W_x^{(1)}\mathbf{x}_t + W_h^{(1)}\mathbf{h}_{t-1}^{(1)} + b^{(1)}\big), \\ \mathbf{h}_t^{(2)} &= \phi\big(W_x^{(2)}\mathbf{h}_t^{(1)} + W_h^{(2)}\mathbf{h}_{t-1}^{(2)} + b^{(2)}\big), \end{aligned} \tag{11}$$

where the superscript denotes the layer index. Each layer has its own set of weights and biases, allowing the network to learn different levels of temporal abstraction. The first layer processes the raw input sequence, while higher layers process transformed representations from the layer below.

For pedagogical purposes, in this lecture we will focus on a single-layer RNN and its hidden states. Multi-layer (deep) RNNs are more expressive but follow the same principles, with each layer maintaining its own hidden state.

1.2 Limitations of Vanilla RNN

The limitation of a plain RNN is well known: it does not handle long-term dependencies effectively. Although the hidden state $\mathbf{h}_t$ depends on $\mathbf{h}_{t-1}$ and, indirectly, on $\mathbf{h}_{t-2}$ and earlier states, information from the distant past is repeatedly transformed through the same single-layer mapping. As a result, long-term memory becomes increasingly diluted and difficult to preserve.

Therefore, an explicit mechanism for retaining and managing the history of a time series is needed, which motivates the following two sections.

2 Long short-term memory (LSTM)

LSTM follows the same basic logic as the vanilla RNN, but introduces an additional mechanism that allows it to handle long-term memory more effectively.

Consider your computer as an analogy. For short-term memory, information is stored in RAM, while information that needs to be preserved for a longer time is written to disk. In other words, *short-term and long-term memories are stored separately*. LSTM adopts a similar strategy. At each time step t, we define two states: a **hidden state** $\mathbf{h}_t$, which stores short-term memory, and a **cell state** $\mathbf{c}_t$, which stores long-term memory. Both states are represented as vectors.

- Cell state $\mathbf{c}_t$: long-term memory, capable of carrying information across many time steps.
- Hidden state $\mathbf{h}_t$: short-term memory, used to produce the output for the next step $\mathbf{x}_{t+1}$.

Going from $\mathbf{x}_t$ to $\mathbf{x}_{t+1}$ involves the following three steps:

1. Carefully update the cell state. The cell state is updated using the current variable $\mathbf{x}_t$, the previous hidden state $\mathbf{h}_{t-1}$, and the previous cell state $\mathbf{c}_{t-1}$.
2. Update the hidden state $\mathbf{h}_t$ based on the updated cell state $\mathbf{c}_t$.
3. Predict $\mathbf{x}_{t+1}$ based on the hidden state $\mathbf{h}_t$.

The overall flow is

$$\begin{pmatrix} \mathbf{x}_t \\ \mathbf{h}_{t-1} \\ \mathbf{c}_{t-1} \end{pmatrix} \rightarrow \mathbf{c}_t \rightarrow \mathbf{h}_t \rightarrow \mathbf{x}_{t+1} \tag{12}$$

In the following, we describe each step in detail.

2.1 Updating the Cell State

Updating long-term memory consists of two parts, which are implemented using two **gates**:

- Forget or fade unnecessary memory: the **forget gate** $\mathbf{f}_t$.
- Add new information: the **input gate** $\mathbf{i}_t$ together with the candidate memory $\tilde{\mathbf{c}}_t$.

Although they are called gates, $\mathbf{f}_t$ and $\mathbf{i}_t$ are vectors with the same dimension as $\mathbf{c}_t$. The candidate memory represents new content that can be added to the old memory after being filtered by the input gate $\mathbf{i}_t$.

§ **Forget gate** The forget gate (a vector) $\mathbf{f}_t$ is computed using a single-layer neural network:

$$\mathbf{f}_t = \text{Sigmoid}(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1} + b_f), \tag{13}$$

where W_f and U_f are weight matrices and b_f is a bias vector. The sigmoid activation ensures that each element of $\mathbf{f}_t$ lies between 0 and 1.

Each element of $\mathbf{f}_t$ acts as a gate on the **old cell state**: a value of 0 means the corresponding memory component is completely forgotten, while a value of 1 means it is fully retained. Values between 0 and 1 correspond to partial retention.

§ **Input gate** The input gate (vector) $\mathbf{i}_t$ is defined in a similar way:

$$\mathbf{i}_t = \text{Sigmoid}(W_i \mathbf{x}_t + U_i \mathbf{h}_{t-1} + b_i), \tag{14}$$

Each element of $\mathbf{i}_t$ also lies between 0 and 1 and acts as a gate on the **candidate memory**. A value of 0 blocks new information, while a value of 1 allows it to pass through completely.

§ **Candidate memory** The candidate memory, constructed from the current input $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$, is given by

$$\tilde{\mathbf{c}}_t = \tanh(W_c \mathbf{x}_t + U_c \mathbf{h}_{t-1} + b_c) \tag{15}$$

Each element of $\tilde{\mathbf{c}}_t$ lies between -1 and 1, representing potential positive or negative contributions to the memory.

§ **Putting everything together** The cell state is updated as

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t, \tag{16}$$

where $\odot$ denotes elementwise multiplication. For example,

$$[1, 2] \odot [3, 4] = [1 \times 3, 2 \times 4] = [3, 8].$$

2.2 Updating the Hidden State

To update the hidden state, we introduce an additional gate, called the **output gate**, which determines which parts of the long-term memory are exposed:

$$\mathbf{o}_t = \text{Sigmoid}(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} + b_o), \tag{17}$$

which has the same functional form as the forget and input gates.

The new hidden state is then computed as

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t). \tag{18}$$

Since the elements of $\mathbf{o}_t$ lie in $(0, 1)$ and the elements of $\tanh(\mathbf{c}_t)$ lie in $[-1, 1]$, the components of the hidden state $\mathbf{h}_t$ are also bounded within $[-1, 1]$.

2.3 Updating x

The update rule for $\mathbf{x}$ is the same as in the vanilla RNN:

$$\hat{x}_{t+1} = W_x \mathbf{h}_t + c, \tag{19}$$

The LSTM network is trained using the same loss function:

$$\mathcal{L} = \sum_t (\mathbf{x}_t - \hat{\mathbf{x}}_t)^2 . \tag{20}$$

2.3.1 Dimensions of the vectors

LSTM involves several vectors: the input variable $\mathbf{x}_t$, the hidden state $\mathbf{h}_t$, and the cell state $\mathbf{c}_t$. We assume that $\mathbf{x}_t \in \mathbb{R}^n$, while both the hidden state $\mathbf{h}_t$ and the cell state $\mathbf{c}_t$ have hidden dimension d. As a consequence, all gates in the LSTM, the forget gate, input gate, and output gate, are also vectors of dimension d.

2.4 Time-invariant parameters

At each time step, the LSTM applies the following single-layer mappings:

$$\begin{aligned}
\mathbf{f}_t &= \text{Sigmoid}(W_f \mathbf{x}_t + U_f \mathbf{h}_{t-1} + b_f), \\
\mathbf{i}_t &= \text{Sigmoid}(W_i \mathbf{x}_t + U_i \mathbf{h}_{t-1} + b_i), \\
\mathbf{o}_t &= \text{Sigmoid}(W_o \mathbf{x}_t + U_o \mathbf{h}_{t-1} + b_o), \\
\tilde{\mathbf{c}}_t &= \tanh(W_c \mathbf{x}_t + U_c \mathbf{h}_{t-1} + b_c), \\
\hat{x}_{t+1} &= W_x \mathbf{h}_t + c.
\end{aligned} \tag{21}$$

We assume that all weights and biases appearing above are **shared across time**, i.e., they take the same values at every time step. In this sense, an LSTM is similar to a physical time-evolution rule: the state of the system changes with time, but the underlying governing equations remain the same.

One LSTM cell is often illustrated as in Fig. 2.

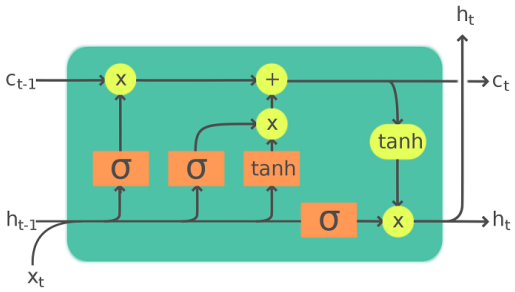

Figure 2: An LSTM cell. (Source: quantconnect.com, orginally from Wikipedia.)

2.5 Gated Recurrent Units (GRUs)

Gated Recurrent Units (GRUs) are a simplified alternative to LSTMs. LSTMs can be computationally heavy due to multiple states and gates per time step. GRUs reduce this complexity with two key simplifications:

- Merge the hidden state (short-term memory) and cell state (long-term memory) into a single hidden state $\mathbf{h}_t$.
- Use only two gates instead of three: an *update gate* $\mathbf{z}_t$ and a *reset gate* $\mathbf{r}_t$.

Both gates are computed with a single-layer neural network:

$$\begin{aligned}
\mathbf{z}_t &= \text{Sigmoid}(W_z \mathbf{x}_t + U_z \mathbf{h}_{t-1} + b_z), \\
\mathbf{r}_t &= \text{Sigmoid}(W_r \mathbf{x}_t + U_r \mathbf{h}_{t-1} + b_r).
\end{aligned} \tag{22}$$

The hidden state $\mathbf{h}_t$ is updated by combining the previous hidden state (long-term memory) with

a candidate hidden state $\tilde{\mathbf{h}}_t$ (new information):

$$\mathbf{h}_t = \underbrace{(1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1}}_{\text{long-term memory}} + \underbrace{\mathbf{z}_t \odot \tilde{\mathbf{h}}_t}_{\text{new information}}, \tag{23}$$

where the candidate hidden state is computed as

$$\tilde{\mathbf{h}}_t = \tanh(W_h \mathbf{x}_t + U_h(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + b_h). \tag{24}$$

Here, the reset gate $\mathbf{r}_t$ controls how much of the previous hidden state contributes to the candidate memory, while the update gate $\mathbf{z}_t$ balances the contribution of new versus old information.

3 Applications

3.1 Reaction Kinetics Prediction

Chemical reactions are characterized not only by *what* transforms into *what*, but also by *how fast* these transformations occur and how chemical species evolve over time. The study of time-dependent reaction behavior is known as **reaction kinetics**.

From an experimental perspective, reaction kinetics naturally produces **time-resolved data**: concentrations of chemical species are measured at successive time points, and the goal is to understand or predict their temporal evolution.

Traditionally, reaction kinetics is modeled using rate equations derived from chemical mechanisms, as illustrated below.

Example: Rate Equations for Reaction Kinetics

Consider a reaction

$$\mathrm{A} \rightarrow \mathrm{B}.$$

Starting from an initial amount of A, the concentration of A decreases with time while the concentration of B increases. A typical experiment records

$$t_1, t_2, \ldots, t_T \quad \longrightarrow \quad [A](t_i), [B](t_i),$$

where $[A](t)$ denotes the concentration of A at time t. These measurements form a **time series** rather than a single input–output pair.

In classical kinetics, the reaction is described using differential equations derived from

assumed mechanisms. For example, a first-order reaction follows

$$\frac{d[A]}{dt} = -k[A],$$

where k is the reaction rate constant.
For reaction rates of common chemical reactions, you can find them from the NIST Chemical Kinetics Database. `https://kinetics.nist.gov/kinetics/`.

While rate equations are effective for simple reactions with well-understood mechanisms, they rely heavily on prior chemical knowledge. For complex systems, such as reactions with unknown mechanisms, large reaction networks, or noisy experimental measurements, this approach can become inaccurate or impractical.

3.1.1 Data-Driven Modeling

Since concentration measurements over time naturally form a time series, reaction kinetics can be modeled as a sequence prediction problem. This motivates the use of **recurrent neural networks (RNNs)**.

Data-driven modeling does not require explicit knowledge of reaction mechanisms or analytical solutions. Instead, the model learns the temporal evolution of the system directly from experimental data, making it suitable for complex or poorly understood reactions.

Reaction kinetics exhibits strong **temporal dependence**. Reaction rates depend on accumulated history, such as reactant depletion, intermediate buildup, or long-lived environmental effects. As a result, the system cannot be fully described by a single snapshot in time. Properly handling **memory** is therefore essential for accurate kinetics prediction.

In an RNN-based formulation, the modeling task is defined as:

- **Input:** reaction state (e.g., concentrations of chemical species) at time t.
- **Output:** reaction state at time $t + 1$.

The primary objective is to learn the time evolution of the system from data, rather than to explicitly infer reaction mechanisms. Nevertheless, the learned concentration trajectories can later be analyzed or fitted to extract mechanistic insights if desired.

3.2 Molecular Generation

Molecular generation is the task of creating valid chemical structures. Historically, SMILES strings have been widely used for this task.

Unlike reaction kinetics, where the goal is to predict the time evolution of a system, molecular generation aims to *produce new sequences of tokens* that correspond to chemically valid molecules.

For example:

$$\mathrm{CCO} \rightarrow \mathtt{C}, \mathtt{C}, \mathtt{O}$$
$$x_1 = \mathtt{C}, x_2 = \mathtt{C}, x_3 = \mathtt{O}$$

Here, the tokens are **discrete** values, in contrast to the continuous concentrations in reaction kinetics modeling. An RNN is used to evaluate the probability of the next token:

$$p(x_t \mid x_1, \ldots, x_{t-1}).$$

The probability of generating a molecule with sequence $\mathbf{x}$ is then

$$p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}).$$

3.2.1 Softmax Activation

Instead of using a plain linear mapping, $x_{t+1} = W_x \mathbf{h}_t + c$, we apply a Softmax function to produce a probability distribution over possible tokens:

$$p(x_{t+1} \mid \mathbf{h}_t) = \mathrm{Softmax}(W_x \mathbf{h}_t + c).$$

The Softmax function can be used for multiple categories, defined as

$$p(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}, \tag{25}$$

where y_i corresponds to the prediction of category i.

3.2.2 Sequence Generation

To generate a new molecule, we start with a special *start-of-sequence token* (<SOS>) and feed it to the RNN. At each step, the model predicts a probability distribution over possible next tokens. A token is sampled from this distribution and fed back as input for the next step. Generation continues until a special *end-of-sequence token* (<EOS>) is produced or a maximum sequence length is reached.

An example of a generated sequence is:

$$\texttt{<SOS>} \xrightarrow{\text{predict}} \mathtt{C} \xrightarrow{\text{predict}} \mathtt{(} \xrightarrow{\text{predict}} \mathtt{O} \xrightarrow{\text{predict}} \mathtt{)} \xrightarrow{\text{predict}} \mathtt{C} \xrightarrow{\text{predict}} \texttt{<EOS>}$$

This generation procedure is **autoregressive**, meaning each token depends on all previously generated tokens.

Sometimes, we would like our input sequences to be of the same length, then we need to add the *padding token* (<PAD>).

3.2.3 Invalid Generations

A key challenge of SMILES-based molecular generation is that many generated strings may not correspond to valid molecules. An alternative is to use SELFIES [1, 2], which guarantees that any string corresponds to a valid molecular structure.

Today, approaches using molecular graphs or 3D structures for generation are increasingly common, gradually replacing 1D string-based methods.

4 Lab

Reaction Kinetics Prediction with RNNs

We assume the following reaction network

$$\begin{aligned} C_2H_5OH &\xrightarrow{k_1} CH_3CHO \\ CH_3CHO &\xrightarrow{k_2} CH_2O \\ CH_2O &\xrightarrow{k_3} CO \\ CO &\xrightarrow{k_4} CO_2 \end{aligned}$$

We will work on a `csv` file with concentrations of the above substances at different time steps.

The link to the colab is

`https://colab.research.google.com/drive/1kDLXLMR2AGWwOlRyVXdyUkAdyEx2mSjc?usp=sharing`

Molecular Generation with RNNs

We will train an LSTM to generate molecules.

Link:

`https://colab.research.google.com/drive/1ElYfXgy4k0V0umiOArYUVzSBkaZuvTKB?usp=sharing`

References

[1] Mario Krenn, Florian Häse, AkshatKumar Nigam, Pascal Friederich, and Alan Aspuru-Guzik. Self-referencing embedded strings (selfies): A 100% robust molecular string representation.

Machine Learning: Science and Technology, 1(4):045024, 2020.

[2] Mario Krenn, Qianxiang Ai, Senja Barthel, Nessa Carson, Angelo Frei, Nathan C Frey, Pascal Friederich, Théophile Gaudin, Alberto Alexander Gayle, Kevin Maik Jablonka, et al. Selfies and the future of molecular string representations. *Patterns*, 3(10), 2022.