Recurrent Neural Networks

Ruwen Qin
Stony Brook University

November 24, 2025

Data used in this module:

- The Time Machine

Python notebook used in this module

- Recurrent_Neural_Networks.ipynb

1 Introduction

So far, we have focused primarily on fixed-length data. We have seldom assumed any particular structure over the sequence of observations. Actually, many learning tasks require dealing with sequential data. Forecasting the deterioration of structural components over time, the tide at each hour of a day, and the evolution of traffic flows all require that models produce outputs consisting of sequences. Unlike regular neural networks that look at each input separately, Recurrent Neural Networks (RNNs) are designed to remember what happened before. They can connect past information to what is happening now, making them particularly suitable for understanding patterns that unfold over time. In this module, we will learn how RNNs work, why the "memory" of RNNs matters, and how they can be used in tasks like text prediction and time-series forecasting. Contents of this learning module is primarily adopted from the reference [1].

2 Models for Sequences

Before introducing CNNs and other models designed to handle sequentially structured data, let's take a look at some actual sequence data and build up some basic intuitions and statistical tools.

Considering time series data $\{\mathbf{x}_t | t = 1, 2, \ldots, T\}$, where each data point is a vector $\mathbf{x}_t \in \mathbb{R}^d$, indexed by a time step $t \in \mathbb{Z}^+$, which consists of a fixed number of components $x_1, \ldots, x_d$.

2.1 Autoregression

Autoregression models regress the value of a signal on the previous values of that same signal, in the form of probabilistic estimation:

$$P(\mathbf{x}_t | \mathbf{x}_{t-\tau}, \ldots, \mathbf{x}_{t-1}) \tag{1}$$

where τ is the duration for the model to look back in predicting the value at t.

The value prediction:

$$\widehat{\mathbf{x}}_t = f(\mathbf{x}_{t-\tau}, \ldots, \mathbf{x}_{t-1}) = \mathbb{E}_{\mathbf{x}_t \sim P}(\mathbf{x}_t | \mathbf{x}_{t-\tau}, \ldots, \mathbf{x}_{t-1}), \tag{2}$$

We might develop a latent autogression model, as illustrated in Figure 1 that maintains some summary $\mathbf{h}_t$ of the past observations and at the same time update it:

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

in addition to the prediction $\widehat{\mathbf{x}}_t$:

$$\widehat{\mathbf{x}}_t = \mathbb{E}_{\mathbf{x}_t \sim P}(\mathbf{x}_t | \mathbf{h}_t). \tag{4}$$

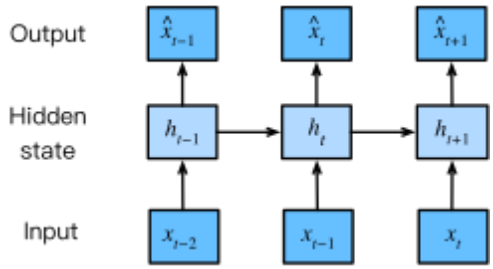

Figure 1: A latent autoregressive model (source: [1])

2.2 Sequence Models

Sometimes, especially when working with language, we wish to estimate the joint probability of an entire sequence. This is a common task when working with sequences composed of discrete *tokens*, such as characters or words. Generally, these estimated functions are called *sequence models*.

$$P(\mathbf{x}_1, \ldots, \mathbf{x}_T) = P(\mathbf{x}_1)P(\mathbf{x}_2|\mathbf{x}_1)\ldots P(\mathbf{x}_T|\mathbf{x}_{T-1}, \ldots, \mathbf{x}_1) = P(\mathbf{x}_1)\prod_{t=2}^{T} P(\mathbf{x}_t \mid \mathbf{x}_{t-1}, \ldots, \mathbf{x}_1). \tag{5}$$

If $\mathbf{x}_t$ conditions only on the τ previous time steps, i.e., $\mathbf{x}_{t-\tau}, \ldots, \mathbf{x}_{t-1}$, rather than the entire sequence history $\mathbf{x}_1, \ldots, \mathbf{x}_{T-1}$, the sequence satisfies a *Markov condition*, i.e., that the future is conditionally independent of the past, given the recent history. We say that the data is characterized by a τ^{th}-order Markov model:

$$P(\mathbf{x}_1, \ldots, \mathbf{x}_T) = P(\mathbf{x}_1, \ldots \mathbf{x}_\tau)\prod_{t=\tau+1}^{T} P(\mathbf{x}_t \mid \mathbf{x}_{t-1}, \ldots, \mathbf{x}_{t-\tau}). \tag{6}$$

2.3 k-step-ahead Prediction

Equation (2) is one-step-ahead prediction. Sometimes, we wish to predict further ahead. We can generalize the prediction model to become *k-step-ahead prediction*:

$$P(\mathbf{x}_{t+k}|\mathbf{x}_{t-\tau}, \ldots, \mathbf{x}_{t-1}). \tag{7}$$

3 Recurrent Neural Networks

3.1 Single Layer RNN

Assume that we have a minibatch of inputs $\mathbf{X}_t \in \mathbb{R}^{m\times d}$ at time step t. In other words, for a minibatch of m sequence examples, each row of $\mathbf{X}_t$ corresponds to one example at time step t from the sequence.

Hidden Layer

Next, denote by $\mathbf{H}_t \in \mathbb{R}^{m\times h}$ the hidden layer output of time step t. Here we save the hidden layer output $\mathbf{H}_{t-1}$ from the previous time step and introduce a new weight parameter $\mathbf{W}_{\text{hh}} \in \mathbb{R}^{h\times h}$ to describe how to use the hidden layer output of the previous time step in the current time step. Specifically, the calculation of the hidden layer output of the current time step is determined by the input of the current time step together with the hidden layer output of the previous time step:

$$\mathbf{H}_t = \phi(\mathbf{X}_t\mathbf{W}_{\text{xh}} + \mathbf{H}_{t-1}\mathbf{W}_{\text{hh}} + \mathbf{b}_{\text{h}}). \tag{8}$$

Equation (8) shows that the output of a hidden layer, $\mathbf{H}_t$, captures and retains the sequence's historical information up to their current time step, just like the state or memory of the neural network's current time

step. Therefore, such a hidden layer output is called a hidden state. Since the hidden state uses the same definition of the previous time step in the current time step, the computation of state is recurrent. Hence, as we said, neural networks with hidden states based on recurrent computation are named *recurrent neural networks* (RNNs). Layers that perform the computation of in RNNs are called recurrent layers.

Figure 2 illustrates the computational logic of an RNN at three adjacent time steps. At any time step t, the computation of the hidden state can be treated as:

i. concatenating the input $\mathbf{X}_t$ at the current time step t and the hidden state $\mathbf{H}_{t-1}$ at the previous time step $t-1$;

ii. feeding the concatenation result into a fully connected layer with the activation function ϕ.

Output Layer

The output of such a fully connected layer is the hidden state $\mathbf{H}_t$ of the current time step t. In this case, the model parameters are the concatenation of $\mathbf{W}_{\text{xh}}$ and $\mathbf{W}_{\text{hh}}$, and a bias of $\mathbf{b}_{\text{h}}$, all from Equation (8). The hidden state of the current time step t, $\mathbf{H}_t$, will participate in computing the hidden state $\mathbf{H}_{t+1}$ of the next time step $t+1$. What is more, $\mathbf{H}_t$ will also be fed into the fully connected output layer to compute the output $\mathbf{O}_t$ of the current time step t:

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{\text{hq}} + \mathbf{b}_{\text{q}}. \tag{9}$$

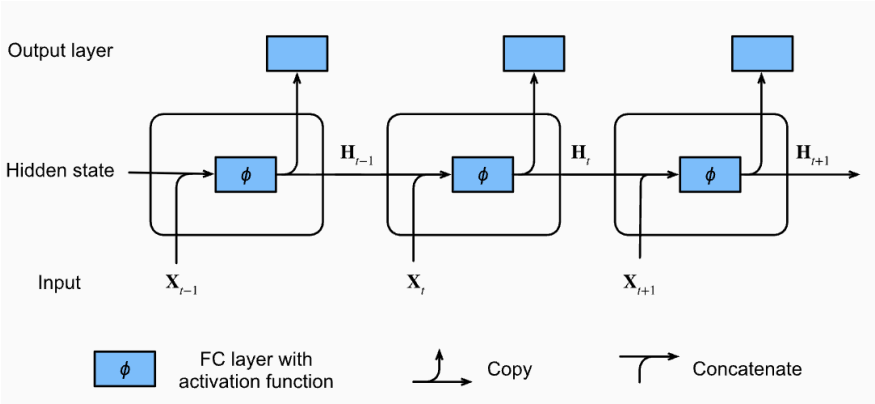

Figure 2: An RNN with a hidden state (source: [1])

Model Parameters

Parameters of the RNN include

- for the recurrent layer:
  $\mathbf{W}_{\text{xh}} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{\text{hh}} \in \mathbb{R}^{h \times h}$, $\mathbf{b}_{\text{h}} \in \mathbb{R}^{1 \times h}$ of the hidden layer,
- for output layer:
  $\mathbf{W}_{\text{hq}} \in \mathbb{R}^{h \times q}$, $\mathbf{b}_{\text{q}} \in \mathbb{R}^{1 \times q}$

It is worth mentioning that even at different time steps, RNNs always use these model parameters. Therefore, the parametrization cost of an RNN does not grow as the number of time steps increases.

Example: A Single-Layer RNN Computation We illustrate the computation of a single-layer RNN at one time step with minibatch size $m = 2$, input dimension $d = 2$, and hidden dimension $h = 2$. Let the activation function be $\phi(z) = \tanh(z)$.

Inputs and Previous Hidden State. The input matrix at time step t is

$$\mathbf{X}_t = \begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix},$$

and the previous hidden state is

$$\mathbf{H}_{t-1} = \begin{bmatrix} 0.1 & -0.2 \\ 0.0 & 0.3 \end{bmatrix}.$$

Model Parameters.

$$\mathbf{W}_{\text{xh}} = \begin{bmatrix} 0.5 & -1.0 \\ 1.2 & 0.3 \end{bmatrix}, \qquad \mathbf{W}_{\text{hh}} = \begin{bmatrix} 0.7 & -0.4 \\ 0.2 & 0.5 \end{bmatrix}, \qquad \mathbf{b}_{\text{h}} = \begin{bmatrix} 0.1 & -0.1 \end{bmatrix}.$$

Step 1: Input-to-Hidden Transformation.

$$\mathbf{X}_t \mathbf{W}_{\text{xh}} = \begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} 0.5 & -1.0 \\ 1.2 & 0.3 \end{bmatrix} = \begin{bmatrix} 2.9 & -0.4 \\ 2.7 & -1.7 \end{bmatrix}.$$

Step 2: Hidden-to-Hidden Transformation.

$$\mathbf{H}_{t-1} \mathbf{W}_{\text{hh}} = \begin{bmatrix} 0.1 & -0.2 \\ 0.0 & 0.3 \end{bmatrix} \begin{bmatrix} 0.7 & -0.4 \\ 0.2 & 0.5 \end{bmatrix} = \begin{bmatrix} 0.03 & -0.14 \\ 0.06 & 0.15 \end{bmatrix}.$$

Step 3: Add Bias. We broadcast the bias vector across the minibatch:

$$\mathbf{b}_{\text{h}} = \begin{bmatrix} 0.1 & -0.1 \end{bmatrix}.$$

Step 4: Pre-activation Hidden State.

$$\mathbf{A}_t = \mathbf{X}_t \mathbf{W}_{\text{xh}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{hh}} + \mathbf{b}_{\text{h}} = \begin{bmatrix} 3.03 & -0.64 \\ 2.86 & -1.65 \end{bmatrix}.$$

Step 5: Apply Activation.

$$\mathbf{H}_t = \tanh(\mathbf{A}_t) \approx \begin{bmatrix} 0.995 & -0.565 \\ 0.993 & -0.929 \end{bmatrix}.$$

Thus, $\mathbf{H}_t$ becomes the new hidden state at time step t and will be used to compute both the output $\mathbf{O}_t$ and the next hidden state $\mathbf{H}_{t+1}$.

3.2 Multi-layer (Deep) Recurrent Neural Networks

The one-layer RNN introduced above processes each time step using a single recurrent transformation. To increase the modeling capacity and capture more complex temporal patterns, we can stack multiple recurrent layers. The resulting architecture is called a *L-layer RNN*, illustrated in Figure 3.

Hidden States Across Layers

For a L-layer RNN, we denote the hidden state at time step t and layer ℓ by

$$\mathbf{H}_t^{(\ell)} \in \mathbb{R}^{m \times h_\ell},$$

where h_ℓ is the number of hidden units in layer ℓ and m is the minibatch size. The first layer receives the external input $\mathbf{X}_t$, while each higher layer receives the hidden state from the previous layer at the same time step.

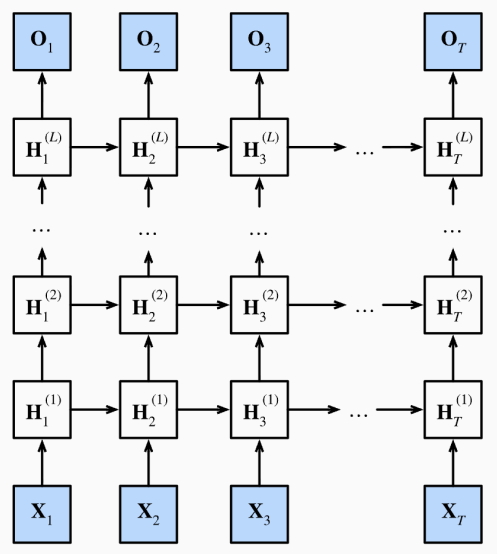

Figure 3: Multiple Layer RNN (source: [1]

Computation in Each Layer

For the first recurrent layer ($\ell = 1$), the hidden state extends naturally from the one-layer case:

$$\mathbf{H}_t^{(1)} = \phi\Big(\mathbf{X}_t \mathbf{W}_{\text{xh}}^{(1)} + \mathbf{H}_{t-1}^{(1)} \mathbf{W}_{\text{hh}}^{(1)} + \mathbf{b}_{\text{h}}^{(1)}\Big). \tag{10}$$

For any upper recurrent layer $\ell = 2, \ldots, K$, the hidden state is computed by using the hidden state from the previous layer at the same time step:

$$\mathbf{H}_t^{(\ell)} = \phi\Big(\mathbf{H}_t^{(\ell-1)} \mathbf{W}_{\text{xh}}^{(\ell)} + \mathbf{H}_{t-1}^{(\ell)} \mathbf{W}_{\text{hh}}^{(\ell)} + \mathbf{b}_{\text{h}}^{(\ell)}\Big). \tag{11}$$

Thus, each recurrent layer maintains its own hidden state sequence, while receiving a full sequence of hidden states from the layer below.

Output Layer

The output at time step t is computed from the top-layer hidden state:

$$\mathbf{O}_t = \mathbf{H}_t^{(L)} \mathbf{W}_{\text{hq}} + \mathbf{b}_{\text{q}}.$$

Model Parameters The model parameters now include:

- for each recurrent layer $\ell = 1, \ldots, L$:

$$\mathbf{W}_{\text{xh}}^{(\ell)}, \quad \mathbf{W}_{\text{hh}}^{(\ell)}, \quad \mathbf{b}_{\text{h}}^{(\ell)},$$

- the output-layer parameters:

$$\mathbf{W}_{\text{hq}}, \quad \mathbf{b}_{\text{q}}.$$

Importantly, even though hidden states evolve over time, the model reuses the same parameters for every time step within each layer. Thus, the total number of parameters grows with the number of layers L but does not grow with the sequence length.

Interpretation

A K-layer RNN performs recurrent computation both *over time* and *over depth*. Lower layers extract lower-level temporal features, while upper layers extract increasingly abstract and long-range patterns. This hierarchical representation often leads to substantially improved performance on complex sequence modeling tasks.

Example: A Fully Explicit Two-Layer RNN Computation

We illustrate the computation of a two-layer RNN at a single time step with a minibatch size $m = 2$, input dimension $d = 2$, and hidden dimension $h_1 = h_2 = 2$. Let the activation function be $\phi(z) = \tanh(z)$.

Inputs and Previous Hidden States: Let the input at time step t be

$$\mathbf{X}_t = \begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix},$$

and the previous hidden states be

$$\mathbf{H}_{t-1}^{(1)} = \begin{bmatrix} 0.1 & -0.2 \\ 0.0 & 0.3 \end{bmatrix}, \qquad \mathbf{H}_{t-1}^{(2)} = \begin{bmatrix} -0.1 & 0.4 \\ 0.2 & -0.3 \end{bmatrix}.$$

Layer 1 Parameters:

$$\mathbf{W}_{\text{xh}}^{(1)} = \begin{bmatrix} 0.5 & -1.0 \\ 1.2 & 0.3 \end{bmatrix}, \qquad \mathbf{W}_{\text{hh}}^{(1)} = \begin{bmatrix} 0.7 & -0.4 \\ 0.2 & 0.5 \end{bmatrix}, \qquad \mathbf{b}_{\text{h}}^{(1)} = \begin{bmatrix} 0.1 & -0.1 \end{bmatrix}.$$

Layer 1 Pre-activation Computation:

$$\begin{aligned} \mathbf{A}_t^{(1)} &= \mathbf{X}_t \mathbf{W}_{\text{xh}}^{(1)} + \mathbf{H}_{t-1}^{(1)} \mathbf{W}_{\text{hh}}^{(1)} + \mathbf{b}_{\text{h}}^{(1)} \\ &= \underbrace{\begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} 0.5 & -1.0 \\ 1.2 & 0.3 \end{bmatrix}}_{\mathbf{X}_t \mathbf{W}_{\text{xh}}^{(1)} = \begin{bmatrix} 2.9 & -0.4 \\ 2.7 & -1.7 \end{bmatrix}} + \underbrace{\begin{bmatrix} 0.1 & -0.2 \\ 0.0 & 0.3 \end{bmatrix} \begin{bmatrix} 0.7 & -0.4 \\ 0.2 & 0.5 \end{bmatrix}}_{\mathbf{H}_{t-1}^{(1)} \mathbf{W}_{\text{hh}}^{(1)} = \begin{bmatrix} 0.03 & -0.14 \\ 0.06 & 0.15 \end{bmatrix}} + \begin{bmatrix} 0.1 & -0.1 \end{bmatrix}. \end{aligned}$$

Thus,

$$\mathbf{A}_t^{(1)} = \begin{bmatrix} 2.9 & -0.4 \\ 2.7 & -1.7 \end{bmatrix} + \begin{bmatrix} 0.03 & -0.14 \\ 0.06 & 0.15 \end{bmatrix} + \begin{bmatrix} 0.1 & -0.1 \end{bmatrix} = \begin{bmatrix} 3.03 & -0.64 \\ 2.86 & -1.65 \end{bmatrix}.$$

Layer 1 Activation:

$$\mathbf{H}_t^{(1)} = \tanh(\mathbf{A}_t^{(1)}) \approx \begin{bmatrix} 0.995 & -0.565 \\ 0.993 & -0.929 \end{bmatrix}.$$

Layer 2 Parameters:

$$\mathbf{W}_{\text{xh}}^{(2)} = \begin{bmatrix} 0.4 & -0.6 \\ 0.9 & 0.2 \end{bmatrix}, \qquad \mathbf{W}_{\text{hh}}^{(2)} = \begin{bmatrix} -0.3 & 0.7 \\ 0.5 & -0.1 \end{bmatrix}, \qquad \mathbf{b}_{\text{h}}^{(2)} = \begin{bmatrix} 0.0 & 0.2 \end{bmatrix}.$$

Layer 2 Pre-activation Computation:

$$\begin{aligned} \mathbf{A}_t^{(2)} &= \mathbf{H}_t^{(1)} \mathbf{W}_{\text{xh}}^{(2)} + \mathbf{H}_{t-1}^{(2)} \mathbf{W}_{\text{hh}}^{(2)} + \mathbf{b}_{\text{h}}^{(2)} \\ &= \underbrace{\begin{bmatrix} 0.995 & -0.565 \\ 0.993 & -0.929 \end{bmatrix} \begin{bmatrix} 0.4 & -0.6 \\ 0.9 & 0.2 \end{bmatrix}}_{\mathbf{H}_t^{(1)} \mathbf{W}_{\text{xh}}^{(2)} = \begin{bmatrix} -0.024 & -0.702 \\ -0.315 & -0.829 \end{bmatrix}} + \underbrace{\begin{bmatrix} -0.1 & 0.4 \\ 0.2 & -0.3 \end{bmatrix} \begin{bmatrix} -0.3 & 0.7 \\ 0.5 & -0.1 \end{bmatrix}}_{\mathbf{H}_{t-1}^{(2)} \mathbf{W}_{\text{hh}}^{(2)} = \begin{bmatrix} 0.23 & -0.11 \\ -0.21 & 0.17 \end{bmatrix}} + \begin{bmatrix} 0.0 & 0.2 \end{bmatrix}. \end{aligned}$$

Thus,

$$\mathbf{A}_t^{(2)} = \begin{bmatrix} -0.024 & -0.702 \\ -0.315 & -0.829 \end{bmatrix} + \begin{bmatrix} 0.23 & -0.11 \\ -0.21 & 0.17 \end{bmatrix} + \begin{bmatrix} 0.0 & 0.2 \end{bmatrix} = \begin{bmatrix} 0.206 & -0.612 \\ -0.525 & -0.459 \end{bmatrix}.$$

Layer 2 Activation:

$$\mathbf{H}_t^{(2)} = \tanh(\mathbf{A}_t^{(2)}) \approx \begin{bmatrix} 0.203 & -0.546 \\ -0.482 & -0.430 \end{bmatrix}.$$

This example explicitly shows how each layer performs linear transformation of inputs and previous hidden states, followed by elementwise nonlinear activation, producing a progressively more abstract hidden representation through the stacked recurrent layers.

4 Sequential Data Pre-processing

To train an RNN model effectively, we must represent sequential inputs in a form that the network can interpret and process over time. Raw text, sensor streams, or other temporal signals typically come in unstructured formats that cannot be fed directly into an RNN. Therefore, before model training begins, we apply a series of pre-processing steps to convert the raw sequence data into numerical sequences with consistent structure, vocabulary, and length. This prepares the data for efficient mini-batch computation and enables the RNN to learn temporal patterns across time steps.

In this learning module, we use text data as an illustrative example, which are represented as sequences of words, characters, or phrases. Before we apply RNNs to text data, we will need some basic tools for converting raw text into sequences of the appropriate form. Typical pre-processing pipelines execute the following steps:

1. Load text as strings into memory.
2. Split the strings into tokens (e.g., words or characters).
3. Build a vocabulary dictionary to associate each vocabulary element with a numerical index.
4. Convert the text into sequences of numerical indices.

We skip the detail of data pre-processing. Instead, we briefly summarize the implementation in Jupyter Notebook. Interested students can review the code in Section 1 " Sequential Text Data Pre-processing" in this learning module's Jupyter Notebook.

We define functions and classes that will be used for pre-processing text data. These include:

- Data Load Function (load_text): for loading the raw text data
- Data Clean Function (clean_text): for data cleaning
- Tokenization Function (tokenize): for split the cleaned text data into a sequence of tokens
- Vocabulary Class (Vocab): for assigning each unique token with a numerical index in the built vocabulary dictionary
- Build Corpus and Vocabulary Function(build_time_machine): representing the sequence of tokens as a sequence of token IDs by integrating the four functions and classes above.

5 Language Model

With data pre-processing, a text sequence is split into tokens, which can be viewed as a sequence of discrete observations such as words or characters. Assume that the tokens in a text sequence of length T are in turn $x_1, x_2, \ldots, x_T$. The goal of *language models* is to estimate the joint probability of the whole sequence.

$$P(x_1, x_2, \ldots, x_T). \tag{12}$$

5.1 Learning Language Model

The obvious question is how we should model a document, or even a sequence of tokens. Suppose that we tokenize text data at the word level. Let's start by applying basic probability rules:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1}). \tag{13}$$

The general n-gram language model approximates the joint probability of a sequence by assuming that each token depends only on the preceding $n-1$ tokens. Formally, the n-gram approximation of the joint distribution $P(x_1, \ldots, x_T)$ is:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}), \tag{14}$$

Under this model, the conditional probability $P(x_t \mid x_{t-n+1}, \ldots, x_{t-1})$ serves as a language model parameter that must be estimated from data, typically by counting the relative frequencies of n-length token sequences. The special cases $n = 1, 2$, and 3 correspond to the well-known unigram, bigram, and trigram language models, respectively.

Table 1: Unigram, Bigram, Trigram, and General n-gram Language Models

Model	Order	Probability Approximation
Unigram	$n = 1$	$P(x_1, x_2, \ldots, x_T) \approx \prod_{t=1}^{T} P(x_t)$
Bigram	$n = 2$	$P(x_1, x_2, \ldots, x_T) \approx \prod_{t=1}^{T} P(x_t \mid x_{t-1})$
Trigram	$n = 3$	$P(x_1, x_2, \ldots, x_T) \approx \prod_{t=1}^{T} P(x_t \mid x_{t-2}, x_{t-1})$
General n-gram	arbitrary n	$P(x_1, x_2, \ldots, x_T) \approx \prod_{t=1}^{T} P(x_t \mid x_{t-n+1}, \ldots, x_{t-1})$

For example, the probability of a text sequence containing four words would be given as:

$$\begin{aligned} &P(\text{deep}, \text{learning}, \text{is}, \text{fun}) \\ =&P(\text{deep})P(\text{learning} \mid \text{deep})P(\text{is} \mid \text{deep}, \text{learning})P(\text{fun} \mid \text{deep}, \text{learning}, \text{is}). \end{aligned}$$

In order to compute the language model, we need to calculate the probability of words and the conditional probability of a word given the previous few words. Note that such probabilities are language model parameters.

$$\begin{aligned} P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2)P(x_3)P(x_4), \\ P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_2)P(x_4 \mid x_3), \\ P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_1, x_2)P(x_4 \mid x_2, x_3). \end{aligned} \tag{15}$$

The probability formulae that involve one, two, and three variables are typically referred to as unigram, bigram, and trigram models, respectively. In order to compute the language model, we need to calculate the probability of words and the conditional probability of a word given the previous few words. Note that such probabilities are language model parameters.

Example: Unigram, Bigram, and Trigram Probability Computation Consider a small vocabulary

$$\mathcal{V} = \{\text{“the”}, \text{“time”}, \text{“traveler”}, \text{“arrived”}\}.$$

Suppose the following counts are collected from a corpus:

Unigram counts:

$$C(\text{the}) = 50,\ C(\text{time}) = 20,\ C(\text{traveler}) = 10,\ C(\text{arrived}) = 5.$$

Bigram counts:

$$C(\text{the}, \text{time}) = 15,\ C(\text{time}, \text{traveler}) = 8,\ C(\text{traveler}, \text{arrived}) = 3.$$

Trigram counts:

$$C(\text{the}, \text{time}, \text{traveler}) = 5,\ C(\text{time}, \text{traveler}, \text{arrived}) = 2.$$

The total number of tokens is

$$N = 50 + 20 + 10 + 5 = 85.$$

We evaluate the probability of the sequence:

$$\text{the} \ \rightarrow \ \text{time} \ \rightarrow \ \text{traveler} \ \rightarrow \ \text{arrived}.$$

Unigram Model. Under the unigram model,

$$P_{\text{uni}}(x_1, x_2, x_3, x_4) = P(x_1)\, P(x_2)\, P(x_3)\, P(x_4).$$

The probabilities are

$$P(\text{the}) = \frac{50}{85}, \quad P(\text{time}) = \frac{20}{85}, \quad P(\text{traveler}) = \frac{10}{85}, \quad P(\text{arrived}) = \frac{5}{85}.$$

Thus,

$$P_{\text{uni}} = \frac{50}{85} \cdot \frac{20}{85} \cdot \frac{10}{85} \cdot \frac{5}{85} = 9.58 \times 10^{-5}.$$

Bigram Model. The bigram model uses the approximation

$$P_{\text{bi}}(x_1, x_2, x_3, x_4) = P(x_1)\, P(x_2 \mid x_1)\, P(x_3 \mid x_2)\, P(x_4 \mid x_3).$$

Conditional probabilities:

$$P(\text{time} \mid \text{the}) = \frac{C(\text{the}, \text{time})}{C(\text{the})} = \frac{15}{50},$$

$$P(\text{traveler} \mid \text{time}) = \frac{C(\text{time}, \text{traveler})}{C(\text{time})} = \frac{8}{20},$$

$$P(\text{arrived} \mid \text{traveler}) = \frac{C(\text{traveler}, \text{arrived})}{C(\text{traveler})} = \frac{3}{10}.$$

Thus,

$$P_{\text{bi}} = \frac{50}{85} \cdot \frac{15}{50} \cdot \frac{8}{20} \cdot \frac{3}{10} = 0.0212.$$

Trigram Model. The trigram approximation is

$$P_{\text{tri}}(x_1, x_2, x_3, x_4) = P(x_1)\, P(x_2 \mid x_1)\, P(x_3 \mid x_1, x_2)\, P(x_4 \mid x_2, x_3).$$

Higher-order conditional probabilities:

$$P(\text{traveler} \mid \text{the}, \text{time}) = \frac{C(\text{the}, \text{time}, \text{traveler})}{C(\text{the}, \text{time})} = \frac{5}{15},$$

$$P(\text{arrived} \mid \text{time}, \text{traveler}) = \frac{C(\text{time}, \text{traveler}, \text{arrived})}{C(\text{time}, \text{traveler})} = \frac{2}{8}.$$

Therefore,

$$P_{\text{tri}} = \frac{50}{85} \cdot \frac{15}{50} \cdot \frac{5}{15} \cdot \frac{2}{8} = 0.0145.$$

5.2 Perplexity

A standard evaluation metric for language models is *perplexity*, defined as

$$\text{PP}(x_1, \ldots, x_T) = P(x_1, \ldots, x_T)^{-1/T}. \tag{16}$$

Equivalently, using conditional probabilities,

$$\text{PP}(x_1, \ldots, x_T) = \left(\prod_{t=1}^{T} P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) \right)^{-1/T}, \tag{17}$$

which is the reciprocal of the geometric mean of the number of real choices that we have when deciding which token to pick next.

Intuitively, perplexity is the exponential of the average negative log-likelihood:

$$\text{PP} = \exp\left(-\frac{1}{T} \sum_{t=1}^{T} \log P(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) \right). \tag{18}$$

A lower perplexity indicates a better predictive model.

5.3 Partitioning Sequences

We will design language models using neural networks and use perplexity to evaluate how good the model is at predicting the next token given the current set of tokens in text sequences. Let's assume that it processes a minibatch of sequences with predefined length at a time. Now the question is how to read minibatches of input sequences and target sequences at random.

Suppose that the dataset takes the form of a sequence of T token indices in corpus. We will partition it into subsequences, where each subsequence has n tokens. To iterate over (almost) all the tokens of the entire dataset for each epoch and obtain all possible length-n subsequences, we can introduce randomness. at the beginning of each epoch, discard the first d tokens, where $d \in [0, n)$ is uniformly sampled at random. The rest of the sequence is then partitioned into $m = \lfloor (T - d - 1)/n \rfloor$ subsequences. Denote by $\mathbf{x}_t = [x_t, \ldots, x_{t+n-1}]$ the length-n subsequence starting from token x_t at time step t. The resulting m partitioned subsequences are $\mathbf{x}_d, \mathbf{x}_{d+n}, \ldots, \mathbf{x}_{d+n(m-1)}$. Each subsequence will be used as an input sequence into the language model.

For language modeling, the goal is to predict the next token based on the tokens we have seen so far; hence the targets (labels) are the original sequence, shifted by one token. The target sequence for any input sequence $\mathbf{x}_t$ is $\mathbf{x}_{t+1}$ with length n.

Figure 4 shows an example of obtaining five pairs of input sequences and target sequences with $n = 5$ and $d = 2$.

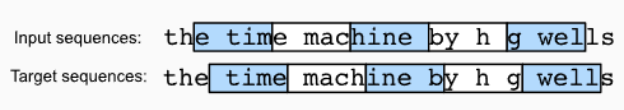

Figure 4: Five pairs of input sequences and target sequences from partitioned length-5 subsequences.

5.4 Modeling Training

Training a recurrent neural network relies on minimizing a loss function that measures how well the model predicts the next token in a sequence. In practice, the most widely used training objective is the *cross-entropy loss*, computed between the predicted probability distribution over the vocabulary and the true next token at each time step. For a sequence of length T, the training loss is typically the average cross-entropy across all time steps, which corresponds to the negative log-likelihood of the observed sequence under the model.

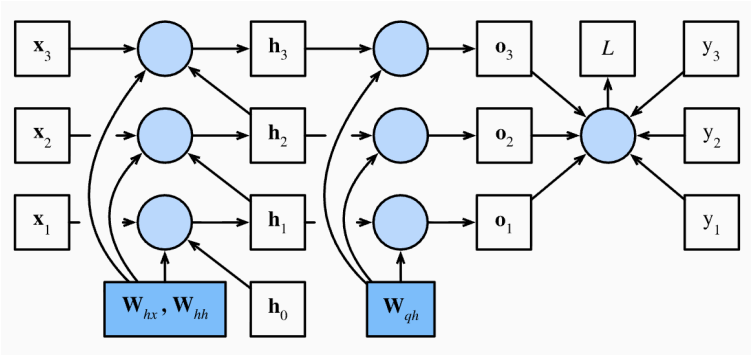

Figure 5: Computational graph showing dependencies for an RNN model with three time steps. Boxes represent variables (not shaded) or parameters (shaded) and circles represent operators. (source: [1]

It is important to distinguish between cross-entropy and perplexity. Perplexity is not used directly for optimization; rather, it serves as an evaluation metric that summarizes model performance. Perplexity is defined as the exponential of the average cross-entropy loss, and therefore decreases whenever the cross-entropy loss is reduced. While cross-entropy provides the gradient signals needed for parameter updates, perplexity offers an interpretable measure of how well the model predicts sequences.

During training, the model performs forward propagation through the sequence to compute the hidden states and output distributions, computes the cross-entropy loss, and then applies backpropagation through time (BPTT) to obtain gradients with respect to all parameters. Because unrolling the RNN through long sequences is both memory-intensive and prone to numerical instability, truncated BPTT is typically employed, limiting gradient propagation to a fixed number of time steps. Parameter updates are then carried out using stochastic gradient descent or other adaptive optimization algorithms. Techniques such as gradient clipping, dropout, and learning-rate scheduling are often used to promote stable training and good generalization performance.

We skip the detailed mathematical formula of BBTT. Interested students can review Section 9.7 in [1].

Example: Cross-Entropy Loss and Perplexity. Consider a vocabulary of size $V = 5$ and a single time step t where the true next token is indexed by $y_t = 3$. Suppose the RNN predicts the probability distribution $\mathbf{p}_t$ over the vocabulary as

$$\mathbf{p}_t = \begin{bmatrix} 0.10, & 0.05, & 0.70, & 0.10, & 0.05 \end{bmatrix}. \tag{19}$$

The cross-entropy loss at time step t is

$$l_t = -y_t \log p_t = -\log(0.70) \approx 0.357. \tag{20}$$

For a sequence of length T, the total loss is the average over time:

$$L = \frac{1}{T}\sum_{t=1}^{T} l_t. \tag{21}$$

Perplexity measures how "surprised" the model is by the true sequence, and is defined as the exponential of the average cross-entropy:

$$\text{PP} = \exp(L). \tag{22}$$

In this example, if the sequence consists of only this one time step, then

$$\text{PP} = \exp(0.357) \approx 1.43. \tag{23}$$

A lower perplexity indicates that the model assigns higher probability to the correct tokens.

6 Long Short-Term Memory (LSTM)

Early RNNs had trouble learning long-term patterns because their gradients either exploded or faded away. Gradient clipping helped control exploding gradients, but vanishing gradients required a new solution. This led to the creation of LSTMs in 1997. Instead of using a simple recurrent node, an LSTM uses a special memory cell that can hold information for a long time through a stable self-connection. This allows LSTMs to remember important information across many time steps and makes training more reliable.

Figure 6 shows that LSTMs have three types of gates: input gates, forget gates, and output gates that control the flow of information. The hidden layer output of LSTM includes the hidden state and the memory cell internal state. Only the hidden state is passed into the output layer while the memory cell internal state remains entirely internal. LSTMs can alleviate vanishing and exploding gradients.

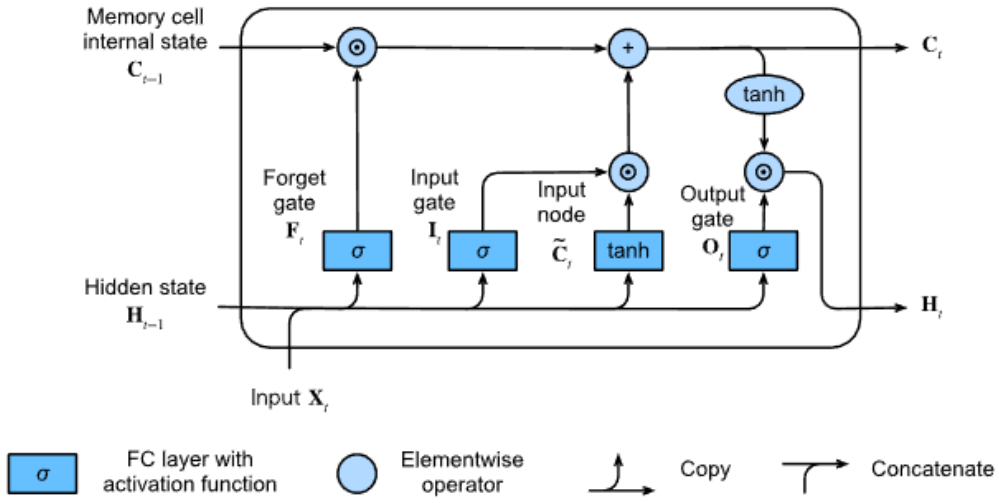

Figure 6: LSTM model. (source:[1])

The key distinction between vanilla RNNs and LSTMs is that the latter supports **gating of the hidden state**. This means that we have dedicated mechanisms for when a hidden state should be updated and also for when it should be reset. These mechanisms are learned and they address the concerns listed above.

Input, Forget, and Output Gates

An LSTM, shown in Figure 6, replaces the standard RNN hidden unit with a *memory cell* that maintains an internal state $\mathbf{C}_t$. Three multiplicative gates control information flow:

- Input gate $\mathbf{I}_t$: how much new information enters the memory cell.
- Forget gate $\mathbf{F}_t$: how much of the previous memory $\mathbf{C}_{t-1}$ is kept.
- Output gate $\mathbf{O}_t$: how much of the memory cell contributes to the hidden output.

These gates enable selective remembering and forgetting across long sequences.

At time step t, the LSTM receives the input $\mathbf{X}_t \in \mathbb{R}^{m\times d}$ and previous hidden state $\mathbf{H}_{t-1} \in \mathbb{R}^{m\times h}$. Three fully connected layers with sigmoid activation compute the gates:

$$\mathbf{I}_t = \sigma(\mathbf{X}_t\mathbf{W}_{\text{xi}} + \mathbf{H}_{t-1}\mathbf{W}_{\text{hi}} + \mathbf{b}_{\text{i}}) \tag{24}$$

$$\mathbf{F}_t = \sigma(\mathbf{X}_t\mathbf{W}_{\text{xf}} + \mathbf{H}_{t-1}\mathbf{W}_{\text{hf}} + \mathbf{b}_{\text{f}}) \tag{25}$$

$$\mathbf{O}_t = \sigma(\mathbf{X}_t\mathbf{W}_{\text{xo}} + \mathbf{H}_{t-1}\mathbf{W}_{\text{ho}} + \mathbf{b}_{\text{o}}) \tag{26}$$

where $\mathbf{W}_{\text{xi}}, \mathbf{W}_{\text{xf}}, \mathbf{W}_{\text{xo}} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{\text{hi}}, \mathbf{W}_{\text{hf}}, \mathbf{W}_{\text{ho}} \in \mathbb{R}^{h\times h}$ are weight parameters and $\mathbf{b}_{\text{i}}, \mathbf{b}_{\text{f}}, \mathbf{b}_{\text{o}} \in \mathbb{R}^{1\times h}$ are bias parameters. Note that broadcasting is triggered during the summation. We use sigmoid functions to map the input values to the interval $(0, 1)$.

Input Nodes

A separate *input node*, illustrated in Figure 6. produces candidate new information via tanh:

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{\text{xc}} + \mathbf{H}_{t-1}\mathbf{W}_{\text{hc}} + \mathbf{b}_{\text{c}}) \tag{27}$$

where $\mathbf{W}_{\text{xc}} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{\text{hc}} \in \mathbb{R}^{h\times h}$ are weight parameters and $\mathbf{b}_{\text{c}} \in \mathbb{R}^{1\times h}$ is a bias parameter.

Memory Cell Internal State

The memory cell, illustrated in Figure 6, combines old and new information using element-wise multiplication ($\odot$):

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t \tag{28}$$

where the input gate $\mathbf{I}_t$ governs how much we take new data into account via $\tilde{\mathbf{C}}_t$ and the forget gate $\mathbf{F}_t$ addresses how much of the old cell internal state $\mathbf{C}_{t-1} \in \mathbb{R}^{n\times h}$ we retain. This update rule allows long-term storage and mitigates the vanishing gradient problem.

Hidden State

Last, we need to define how to compute the output of the memory cell, i.e., the hidden state $\mathbf{H}_t \in \mathbb{R}^{m\times h}$, as seen by other layers. The hidden state is

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t) \tag{29}$$

alway in the range of $(-1, 1)$. The memory cell can store information silently (when $\mathbf{O}_t \approx 0$) and reveal it only when needed.

7 Gated Recurrent Unit (GRU)

Some researchers began to experiment with simplified architectures in hopes of retaining the key idea of incorporating an internal state and multiplicative gating mechanisms but with the aim of speeding up computation. The *gated recurrent unit* (GRU) offers a streamlined version of the LSTM memory cell that often achieves comparable performance but with the advantage of being faster to compute. Figure 7 illustrates the design of GRU.

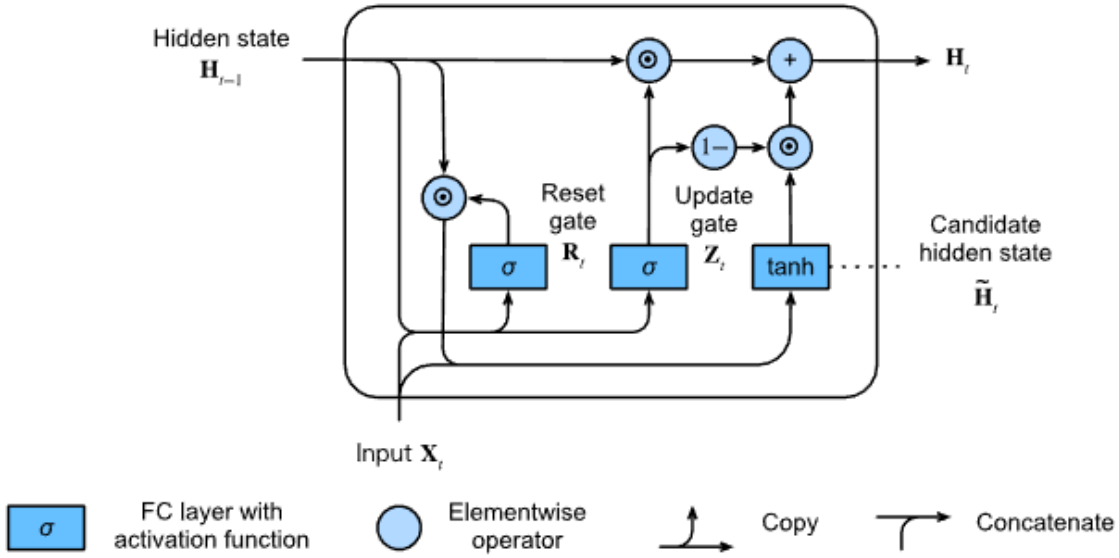

Figure 7: GRU Model (source: [1])

Reset Gate and Update Gate

A GRU simplifies the LSTM by using only two gates: a *reset gate* and an *update gate*, both using sigmoid activations so their values lie in $(0, 1)$.

- Reset Gate $\mathbf{R}_t$: how much of the previous hidden state is used when computing the new candidate state
- Update Gate $\mathbf{Z}_t$: how much of the hidden state should be carried forward unchanged

Mathematically, for a given time step t, suppose that the input is a minibatch $\mathbf{X}_t \in \mathbb{R}^{m\times d}$ (number of examples $= n$; number of inputs $= d$) and the hidden state of the previous time step is $\mathbf{H}_{t-1} \in \mathbb{R}^{m\times h}$ (number of hidden units $= h$). Then the reset gate $\mathbf{R}_t \in \mathbb{R}^{m\times h}$ and update gate $\mathbf{Z}_t \in \mathbb{R}^{m\times h}$ are computed as follows:

$$\mathbf{R}_t = \sigma(\mathbf{X}_t\mathbf{W}_{\text{xr}} + \mathbf{H}_{t-1}\mathbf{W}_{\text{hr}} + \mathbf{b}_{\text{r}}), \tag{30}$$

$$\mathbf{Z}_t = \sigma(\mathbf{X}_t\mathbf{W}_{\text{xz}} + \mathbf{H}_{t-1}\mathbf{W}_{\text{hz}} + \mathbf{b}_{\text{z}}), \tag{31}$$

where $\mathbf{W}_{\text{xr}}, \mathbf{W}_{\text{xz}} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{\text{hr}}, \mathbf{W}_{\text{hz}} \in \mathbb{R}^{h\times h}$ are weight parameters and $\mathbf{b}_{\text{r}}, \mathbf{b}_{\text{z}} \in \mathbb{R}^{1\times h}$ are bias parameters. This streamlined design makes the GRU easier to train while still capturing long-term dependencies.

Candidate Hidden State

Next, we integrate the reset gate $\mathbf{R}_t$ with the regular updating mechanism, leading to the following *candidate hidden state* $\tilde{\mathbf{H}}_t \in \mathbb{R}^{m\times h}$ at time step t:

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{\text{xh}} + (\mathbf{R}_t \odot \mathbf{H}_{t-1})\,\mathbf{W}_{\text{hh}} + \mathbf{b}_{\text{h}}), \tag{32}$$

where $\mathbf{W}_{\text{xh}} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{\text{hh}} \in \mathbb{R}^{h \times h}$ are weight parameters, $\mathbf{b}_{\text{h}} \in \mathbb{R}^{1 \times h}$ is the bias, and the symbol $\odot$ is the elementwise product operator. Here we use a tanh activation function. The result is a candidate, since we still need to incorporate the action of the update gate.

Hidden State

Finally, we incorporate the effect of the update gate $\mathbf{Z}_t$. This determines the extent to which the new hidden state $\mathbf{H}_t \in \mathbb{R}^{m \times h}$ matches the old state $\mathbf{H}_{t-1}$ compared with how much it resembles the new candidate state $\tilde{\mathbf{H}}_t$. The update gate $\mathbf{Z}_t$ can be used for this purpose, simply by taking elementwise convex combinations of $\mathbf{H}_{t-1}$ and $\tilde{\mathbf{H}}_t$. This leads to the final update equation for the GRU:

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t. \tag{33}$$

In summary, GRUs have the following two distinguishing features:

- Reset gates help capture short-term dependencies in sequences.
- Update gates help capture long-term dependencies in sequences.

8 Bidirectional Recurrent Neural Networks

There are many other sequence learning tasks contexts where it is perfectly fine to condition the prediction at every time step on both the leftward and the rightward context. For instance, part of speech detection. Why not taking the context in both directions into account when assessing the part of speech associated with a given word? Another common task, useful as a pretraining exercise prior to fine-tuning a model on an actual task of interest, is to mask out random tokens in a text document and then to train a sequence model to predict the values of the missing tokens.

A simple technique transforms any unidirectional RNN into a bidirectional RNN, as illustrated in Figure 8. We simply implement two unidirectional RNN layers chained together in opposite directions and acting on the same input. For the first RNN layer, the first input is $\mathbf{x}_1$ and the last input is $\mathbf{x}_T$, but for the second RNN layer, the first input is $\mathbf{x}_T$ and the last input is $\mathbf{x}_1$. To produce the output of this bidirectional RNN layer, we simply concatenate together the corresponding outputs of the two underlying unidirectional RNN layers.

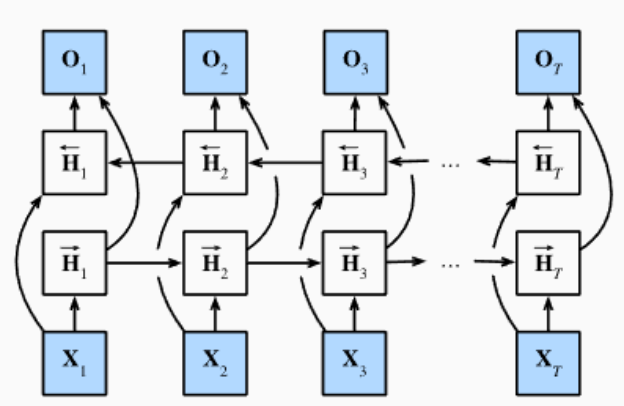

Figure 8: Bidirectional RNN (source:[1])

At each time step t, we consider a minibatch input $\mathbf{X}_t \in \mathbb{R}^{m \times d}$, where m is the batch size and d is the number of input features. Let ϕ denote the activation function. A bidirectional RNN maintains both a *forward* hidden state $\overrightarrow{\mathbf{H}}_t \in \mathbb{R}^{m \times h}$ and a *backward* hidden state $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{m \times h}$.

The forward and backward recurrences are:

$$\overrightarrow{\mathbf{H}}_t = \phi\Big(\mathbf{X}_t \mathbf{W}_{\text{xh}}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{\text{hh}}^{(f)} + \mathbf{b}_{\text{h}}^{(f)}\Big), \tag{34}$$

$$\overleftarrow{\mathbf{H}}_t = \phi\Big(\mathbf{X}_t \mathbf{W}_{\text{xh}}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{\text{hh}}^{(b)} + \mathbf{b}_{\text{h}}^{(b)}\Big). \tag{35}$$

where the weights $\mathbf{W}_{\text{xh}}^{(f)} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{\text{hh}}^{(f)} \in \mathbb{R}^{h \times h}$, $\mathbf{W}_{\text{xh}}^{(b)} \in \mathbb{R}^{d \times h}$, and $\mathbf{W}_{\text{hh}}^{(b)} \in \mathbb{R}^{h \times h}$, and the biases $\mathbf{b}_{\text{h}}^{(f)} \in \mathbb{R}^{1 \times h}$ and $\mathbf{b}_{\text{h}}^{(b)} \in \mathbb{R}^{1 \times h}$ are all the model parameters.

The forward and backward hidden states are concatenated to form the combined hidden state used by the output layer:

$$\mathbf{H}_t = \big[\, \overrightarrow{\mathbf{H}}_t \;\; \overleftarrow{\mathbf{H}}_t \,\big] \in \mathbb{R}^{m \times 2h}. \tag{36}$$

For an output dimension q, the output at time step t is:

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{\text{hq}} + \mathbf{b}_{\text{q}}, \tag{37}$$

where $\mathbf{W}_{\text{hq}} \in \mathbb{R}^{2h \times q}$ and $\mathbf{b}_{\text{q}} \in \mathbb{R}^{1 \times q}$ are output-layer parameters.

Although it is possible to use different hidden dimensions for the forward and backward directions, in practice both directions typically use the same number of hidden units h.

References

[1] Aston Zhang, Zachary C Lipton, Mu Li, and Alexander J Smola. *Dive into deep learning*. Cambridge University Press, 2023.