

# Recurrent Neural Networks and Long Short-Term Memory Networks

Deep Learning (ENGG\*6600\*07)

School of Engineering,  
University of Guelph, ON, Canada

Course Instructor: Benjamin Ghogh  
Fall 2023

## Introduction

# Introduction

- Before the era of transformers in deep learning, regular neural networks could not process sequences, such as sentences (sequence of words) or speech (sequence of phonemes), properly without any recurrence.
- **Recurrent Neural Network (RNN)**, proposed in [1], is a dynamical system which considers recurrence. In recurrence, the output of a model is fed as input to the model again in the next time step.
- One of the main training algorithms for RNN is **Backpropagation Through Time (BPTT)**, developed by several works [2, 3, 4, 5, 6], which is similar to the backpropagation algorithm [1] but has also chain rule through time.
- There were some problems with **gradient vanishing or explosion for long-term dependencies** in RNN [7, 8]. Several solutions were proposed for this issue, some of which are **close-to-identity weight matrix** [9], **long delays** [10], **leaky units** [11, 12], and **echo state networks** [13, 14].

# Introduction

- Sequence modeling requires both **short-term** and **long-term** dependencies.
- For example, consider the sentence “The police is chasing the thief”. In this sentence, the words “police” and “thief” are related to each other with **short-term dependency** because they are close to one another in the sequence of words.
- Another example is the sentence “I was born in France. My father was working there for many years during my childhood. My family had a great time there while my father was making money in his business there. That is why I know how to speak French”. In this second example, the words “France” and “French” are related to each other with **long-term dependency** because they are far away from one another in the sequence of words.
- That inspired researchers to propose the **Long Short-Term Memory (LSTM)** network to handle both short-term and long-term dependencies [15, 16].
- Later, **Gated Recurrent Unit (GRU)** was proposed [17] which simplified LSTM to reduce its unnecessary complexity.

# Introduction

- RNN and LSTM networks are **causal models** which condition every sequence element on the **previous** elements in the sequence.
- Later researches showed that processing the sequence in **both directions** can perform better for the sequences which can be processed offline; e.g., **if the chunks of sequence can be saved and processed** and the sequence elements should not be processed as a stream [18, 19].
- Therefore, **bidirectional RNN** [20, 21] and **bidirectional LSTM** [18, 19] were proposed to process sequences in both directions.
- The **Embeddings from Language Model (ELMo)** network [22] is a language model which makes use of the bidirectional LSTM.

## **Recurrent Neural Network**

# Dynamical System

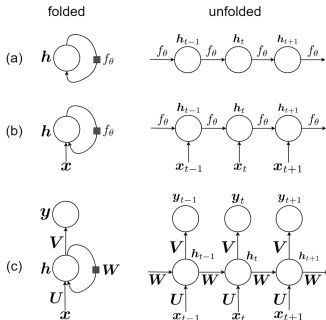
- A dynamical system is recursive and its classical form is as follows:

$$\mathbf{h}_t = f_{\theta}(\mathbf{h}_{t-1}), \quad (1)$$

where  $t$  denotes the time step,  $\mathbf{h}_t$  is the state at time  $t$ , and  $f_{\theta}(\cdot)$  is a function fixed between the states of all time steps. Dynamical systems are widely used in chaos theory [23].

- We can have a dynamical system with external input signal where  $\mathbf{x}_t$  denotes the input signal at time  $t$ . This system is modeled as:

$$\mathbf{h}_t = f_{\theta}(\mathbf{h}_{t-1}, \mathbf{x}_t). \quad (2)$$



# Parameter Sharing

- The state  $\mathbf{h}_t$  can be considered as a summary of the past sequence of inputs and states.
- If a different function  $f_\theta$  is defined for each possible sequence length, the model will not have generalization.
- If the same parameters are used for any sequence length, the model will have generalization properties.
- Therefore, the **parameters are shared** for all lengths and between all states. Such a dynamical system with parameter sharing can be implemented as a neural network with weights. Such a network is called a **Recurrent Neural Network (RNN)**, which was proposed in [1] (1986).



# Parameter Sharing

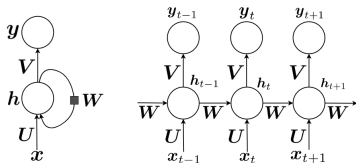
- In RNN, the same weight matrices are used for all time slots.
- RNN gets a sequence as input and outputs a sequence as a decision for a task such as regression or classification.
- Suppose the input, output, and state at time slot  $t$  are denoted by  $\mathbf{x}_t \in \mathbb{R}^d$ ,  $\mathbf{y}_t \in \mathbb{R}^q$ , and  $\mathbf{h}_t \in \mathbb{R}^p$ , respectively. Let  $\mathbf{W} \in \mathbb{R}^{p \times p}$  be the weight matrix between states,  $\mathbf{U} \in \mathbb{R}^{p \times d}$  be the weight matrix between the inputs and the states, and  $\mathbf{V} \in \mathbb{R}^{q \times p}$  denote the weight matrix between the states and outputs. The bias weights for the state and the output are denoted by  $\mathbf{b}_i \in \mathbb{R}^p$  and  $\mathbf{b}_y \in \mathbb{R}^q$ , respectively. We have:

$$\mathbb{R}^p \ni \mathbf{i}_t = \mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}_i, \quad (3)$$

$$[-1, 1]^p \ni \mathbf{h}_t = \tanh(\mathbf{i}_t) = \tanh(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}_i), \quad (4)$$

$$\mathbb{R}^q \ni \mathbf{y}_t = \mathbf{V}\mathbf{h}_t + \mathbf{b}_y, \quad (5)$$

where  $\tanh(\cdot) \in (-1, 1)$  denotes the hyperbolic tangent function, which is used as an element-wise activation function for the states.

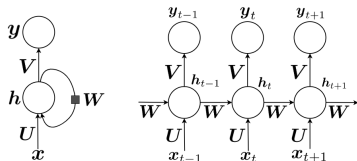


# Parameter Sharing

- If there is an activation function, such as softmax, at the output layer, we denote the output of activation function by:

$$\mathbb{R}^q \ni \hat{\mathbf{y}}_t = \text{softmax}(\mathbf{y}_t) = \frac{\exp(y_{t,1})}{\sum_{j=1}^q \exp(y_{t,j})}, \quad (6)$$

where  $y_{t,j}$  denotes the  $j$ -th component of  $\mathbf{y}_t$ .



## Backpropagation Through Time (BPTT)

# Backpropagation Through Time (BPTT)

- One of the methods for training RNN is **Backpropagation Through Time (BPTT)**, which is very similar to the backpropagation algorithm [1] because it is based on gradient descent and chain rule [24], but it has also chain rule through time.
- BPTT was developed by several works [2, 3, 4, 5, 6].
- This algorithm is very solid in theory; however, it does not show the best performance in practice.
- In BPTT, the loss is considered as a summation of loss functions at the previous time steps until now.
- As it is impractical to consider all time steps from the start of training (especially after a long time of training), we only consider the  $T$  previous time steps. In other words, we assume that RNN has  $T$ -order Markov property [25]. Therefore, the loss function is:

$$\mathbb{R} \ni \mathcal{L} = \sum_{t=1}^T \mathcal{L}_t, \quad (7)$$

where  $\mathcal{L}_1$  is the loss function at the current time slot and  $\mathcal{L}_t$  denotes the loss function at the previous  $(t - 1)$  time slot.

- This loss functions needs to be optimized using gradient descent and chain rule. Therefore, we calculate its gradient with respect to the parameters of RNN. These parameters are  $\mathbf{y}_t$ ,  $\mathbf{h}_t$ ,  $\mathbf{V}$ ,  $\mathbf{W}$ ,  $\mathbf{U}$ ,  $\mathbf{b}_i$ , and  $\mathbf{b}_y$ , based on Eqs. (3), (4), and (5) and the figure of RNN.

# Gradient With Respect to the Output

- We had:

$$\mathbb{R} \ni \mathcal{L} = \sum_{t=1}^T \mathcal{L}_t.$$

- If there is no activation function at the last layer, the gradient of the loss function of RNN with respect to the output at time  $t$  is:

$$\mathbb{R}^q \ni \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t} \stackrel{(a)}{=} \frac{\partial \mathcal{L}}{\partial \mathcal{L}_t} \times \frac{\partial \mathcal{L}_t}{\partial \mathbf{y}_t} \stackrel{(7)}{=} \frac{\partial \mathcal{L}_t}{\partial \mathbf{y}_t}, \quad (8)$$

where (a) is because of the chain rule.

- The gradient of the loss function at time  $t$  with respect to the output at time  $t$ , i.e.,  $\partial \mathcal{L}_t / \partial \mathbf{y}_t$ , is calculated based on the formula of the loss function. The loss function can be any loss function for classification, regression, or other tasks.
- If there is an activation function at the last layer (see Eq. (6)), the gradient is:

$$\mathbb{R}^q \ni \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t} \stackrel{(a)}{=} \frac{\partial \mathcal{L}}{\partial \mathcal{L}_t} \times \frac{\partial \mathcal{L}_t}{\partial \hat{\mathbf{y}}_t} \times \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{y}_t} \stackrel{(7)}{=} \frac{\partial \mathcal{L}_t}{\partial \hat{\mathbf{y}}_t} \times \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{y}_t}, \quad (9)$$

where (a) is because of the chain rule. The derivative  $\partial \hat{\mathbf{y}}_t / \partial \mathbf{y}_t$  is calculated based on the formula of the activation function. The other derivative,  $\partial \mathcal{L}_t / \partial \hat{\mathbf{y}}_t$ , is calculated based on the formula of the loss as a function of the output of the activation function.

# Gradient With Respect to the State

- We had:

$$\begin{aligned}\mathbb{R}^p \ni \mathbf{i}_t &= \mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}_i, \\ [-1, 1]^p \ni \mathbf{h}_t &= \tanh(\mathbf{i}_t) = \tanh(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}_i), \\ \mathbb{R}^q \ni \mathbf{y}_t &= \mathbf{V}\mathbf{h}_t + \mathbf{b}_y.\end{aligned}$$

- The gradient of the loss function of RNN with respect to the state at time  $t$  is:

$$\begin{aligned}\mathbb{R}^p \ni \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} &\stackrel{(a)}{=} \left( \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t} \times \frac{\partial \mathbf{y}_t}{\partial \mathbf{h}_t} \right) + \left( \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} \times \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t} \right) \\ &\stackrel{(5)}{=} \left( \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t} \times \mathbf{V} \right) + \left( \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} \times \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t} \right),\end{aligned}\tag{10}$$

where (a) is because changing  $\mathbf{h}_t$  affects both  $\mathbf{y}_t$  and  $\mathbf{h}_{t+1}$ . We denote  $\delta_t := \partial \mathcal{L} / \partial \mathbf{h}_t$  so Eq. (10) becomes:

$$\delta_t = \left( \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t} \times \mathbf{V} \right) + \left( \delta_{t+1} \times \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t} \right).\tag{11}$$

# Gradient With Respect to the State

- We had:

$$\mathbb{R}^p \ni \mathbf{i}_t = \mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}_i,$$

$$[-1, 1]^p \ni \mathbf{h}_t = \tanh(\mathbf{i}_t) = \tanh(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}_i),$$

$$\mathbb{R}^q \ni \mathbf{y}_t = \mathbf{V}\mathbf{h}_t + \mathbf{b}_y.$$

- According to Eqs. (3) and (4), we have:

$$\mathbf{i}_{t+1} = \mathbf{W}\mathbf{h}_t + \mathbf{U}\mathbf{x}_{t+1} + \mathbf{b}_i, \quad (12)$$

$$\mathbf{h}_{t+1} = \tanh(\mathbf{i}_{t+1}) = \tanh(\mathbf{W}\mathbf{h}_t + \mathbf{U}\mathbf{x}_{t+1} + \mathbf{b}_i). \quad (13)$$

Therefore:

$$\mathbb{R}^{p \times p} \ni \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t} \stackrel{(a)}{=} \left( \frac{\partial \mathbf{i}_{t+1}}{\partial \mathbf{h}_t} \right)^\top \times \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{i}_{t+1}} \stackrel{(b)}{=} \mathbf{W}(1 - \mathbf{h}_{t+1}^\top \mathbf{h}_{t+1}) \mathbf{I}_{p \times p},$$

where  $\mathbf{I}_{p \times p}$  is the identity matrix of size  $(p \times p)$ , (a) is because of the chain rule, and (b) is because  $\mathbb{R}^{p \times p} \ni \partial \mathbf{i}_{t+1} / \partial \mathbf{h}_t = \mathbf{W}^\top$  for Eq. (12), and we have:

$$\mathbb{R}^{p \times p} \ni \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{i}_{t+1}} = (1 - \mathbf{h}_{t+1}^\top \mathbf{h}_{t+1}) \mathbf{I}_{p \times p}, \quad (14)$$

based on Eq. (13) and the formula for derivative of the hyperbolic tangent function, noticing that the state is multidimensional and not a scalar.

# Gradient With Respect to the State

- Eqs. (5) and (8) were:

$$\begin{aligned}\mathbb{R}^q \ni \mathbf{y}_t &= \mathbf{V}\mathbf{h}_t + \mathbf{b}_y, \\ \mathbb{R}^q \ni \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t} &\stackrel{(a)}{=} \frac{\partial \mathcal{L}}{\partial \mathcal{L}_t} \times \frac{\partial \mathcal{L}_t}{\partial \mathbf{y}_t} \stackrel{(7)}{=} \frac{\partial \mathcal{L}_t}{\partial \mathbf{y}_t}.\end{aligned}$$

- We had:

$$\delta_t := \partial \mathcal{L} / \partial \mathbf{h}_t.$$

- For the time slot  $t = T$ , the derivative  $\partial \mathcal{L} / \partial \mathbf{h}_T$  is much simpler:

$$\mathbb{R}^p \ni \delta_T = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_T} \stackrel{(a)}{=} \frac{\partial \mathcal{L}}{\partial \mathbf{y}_T} \times \frac{\partial \mathbf{y}_T}{\partial \mathbf{h}_T} \stackrel{(5)}{=} \frac{\partial \mathcal{L}}{\partial \mathbf{y}_T} \times \mathbf{V} \stackrel{(8)}{=} \frac{\partial \mathcal{L}_T}{\partial \mathbf{y}_T} \times \mathbf{V}, \quad (15)$$

where (a) is because of the chain rule and the derivative  $\partial \mathcal{L}_T / \partial \mathbf{y}_T$  is computed based on the formula of loss function.



# Gradient With Respect to $\mathbf{V}$

- The gradient of the loss function of RNN with respect to the weight matrix  $\mathbf{V}$  is:

$$\mathbb{R}^{q \times p} \ni \frac{\partial \mathcal{L}}{\partial \mathbf{V}} \stackrel{(a)}{=} \sum_{t=1}^T \left( \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t} \times \frac{\partial \mathbf{y}_t}{\partial \mathbf{V}} \right) \stackrel{(b)}{=} \sum_{t=1}^T \left( \frac{\partial \mathcal{L}_t}{\partial \mathbf{y}_t} \times \mathbf{h}_t^\top \right), \quad (16)$$

where (a) is because  $\mathbf{V}$  exists in all time slots and changing  $\mathbf{V}$  affects the loss  $\mathcal{L}$  in all time slots. The equation (b) is because of Eqs. (8) and (5):

$$\begin{aligned} \mathbb{R}^q \ni \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t} &\stackrel{(a)}{=} \frac{\partial \mathcal{L}}{\partial \mathcal{L}_t} \times \frac{\partial \mathcal{L}_t}{\partial \mathbf{y}_t} \stackrel{(7)}{=} \frac{\partial \mathcal{L}_t}{\partial \mathbf{y}_t}, \\ \mathbb{R}^q \ni \mathbf{y}_t &= \mathbf{V} \mathbf{h}_t + \mathbf{b}_y. \end{aligned}$$

- The derivative  $\partial \mathcal{L}_t / \partial \mathbf{y}_t \in \mathbb{R}^q$  is calculated based on the formula of the loss function.

# Gradient With Respect to $\mathbf{W}$

- The gradient of the loss function of RNN with respect to the weight matrix  $\mathbf{W}$  is:

$$\mathbb{R}^{p \times p} \ni \frac{\partial \mathcal{L}}{\partial \mathbf{W}} \stackrel{(a)}{=} \sum_{t=1}^T \text{vec}_{p \times p}^{-1} \left( \left( \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}} \right)^\top \times \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \right), \quad (17)$$

where  $\text{vec}_{p \times p}^{-1}(\cdot)$  de-vectorizes the vector of length  $p^2$  to a matrix of size  $(p \times p)$  and (a) is because  $\mathbf{W}$  exists in all time slots and changing  $\mathbf{W}$  affects the loss  $\mathcal{L}$  in all time slots.

- The derivative  $\partial \mathcal{L} / \partial \mathbf{h}_t \in \mathbb{R}^p$  in Eq. (17) was computed before.
- The derivative  $\partial \mathbf{h}_t / \partial \mathbf{W}$  in Eq. (17) is:

$$\mathbb{R}^{p \times p^2} \ni \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{i}_t} \times \frac{\partial \mathbf{i}_t}{\partial \mathbf{W}},$$

because of the chain rule. The first term is:

$$\mathbb{R}^{p \times p} \ni \frac{\partial \mathbf{h}_t}{\partial \mathbf{i}_t} = (1 - \mathbf{h}_t^\top \mathbf{h}_t) \mathbf{I}_{p \times p}, \quad (18)$$

according to Eq. (14). Based on the Magnus-Neudecker convention [24], the second term is calculated as:

$$\mathbb{R}^{p \times p^2} \ni \frac{\partial \mathbf{i}_t}{\partial \mathbf{W}} = \mathbf{h}_{t-1}^\top \otimes \mathbf{I}_{p \times p},$$

where  $\otimes$  denotes the Kronecker product (see our tutorial [24] for more information on the Magnus-Neudecker convention).

# Gradient With Respect to $\mathbf{U}$

- The gradient of the loss function of RNN with respect to the weight matrix  $\mathbf{U}$  is:

$$\mathbb{R}^{p \times d} \ni \frac{\partial \mathcal{L}}{\partial \mathbf{U}} \stackrel{(a)}{=} \sum_{t=1}^T \text{vec}_{p \times d}^{-1} \left( \left( \frac{\partial \mathbf{h}_t}{\partial \mathbf{U}} \right)^\top \times \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \right), \quad (19)$$

where (a) is because  $\mathbf{U}$  exists in all time slots and changing  $\mathbf{U}$  affects the loss  $\mathcal{L}$  in all time slots.

- The derivative  $\partial \mathcal{L} / \partial \mathbf{h}_t \in \mathbb{R}^p$  in Eq. (17) was computed before.
- The derivative  $\partial \mathbf{h}_t / \partial \mathbf{U}$  in Eq. (17) is:

$$\mathbb{R}^{p \times (pd)} \ni \frac{\partial \mathbf{h}_t}{\partial \mathbf{U}} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{i}_t} \times \frac{\partial \mathbf{i}_t}{\partial \mathbf{U}},$$

because of the chain rule.

- The first term is already calculated in Eq. (18).
- Based on the Magnus-Neudecker convention [24], the second term is calculated as:

$$\mathbb{R}^{p \times (pd)} \ni \frac{\partial \mathbf{i}_t}{\partial \mathbf{U}} = \mathbf{x}_t^\top \otimes \mathbf{I}_{p \times p}.$$

## Gradient With Respect to $\mathbf{b}_i$

- The gradient of the loss function of RNN with respect to the bias  $\mathbf{b}_i$  is:

$$\mathbb{R}^p \ni \frac{\partial \mathcal{L}}{\partial \mathbf{b}_i} \stackrel{(a)}{=} \sum_{t=1}^T \left( \left( \frac{\partial \mathbf{h}_t}{\partial \mathbf{b}_i} \right)^\top \times \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \right), \quad (20)$$

where (a) is because  $\mathbf{b}_i$  exists in all time slots and changing  $\mathbf{b}$  affects the loss  $\mathcal{L}$  in all time slots.

- The derivative  $\partial \mathcal{L} / \partial \mathbf{h}_t$  was already calculated before.
- The derivative  $\partial \mathbf{h}_t / \partial \mathbf{b}_i$  is calculated as:

$$\mathbb{R}^{p \times p} \ni \frac{\partial \mathbf{h}_t}{\partial \mathbf{b}_i} \stackrel{(a)}{=} \frac{\partial \mathbf{h}_t}{\partial \mathbf{i}_t} \times \frac{\partial \mathbf{i}_t}{\partial \mathbf{b}_i} \stackrel{(3)}{=} \frac{\partial \mathbf{h}_t}{\partial \mathbf{i}_t},$$

where (a) is because of the chain rule and the derivative  $\partial \mathbf{h}_t / \partial \mathbf{i}_t$  was already calculated in Eq. (18).

# Gradient With Respect to $\mathbf{b}_y$

- Eqs. (5) and (8) were:

$$\begin{aligned}\mathbb{R}^q \ni \mathbf{y}_t &= \mathbf{V}\mathbf{h}_t + \mathbf{b}_y, \\ \mathbb{R}^q \ni \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t} &\stackrel{(a)}{=} \frac{\partial \mathcal{L}}{\partial \mathcal{L}_t} \times \frac{\partial \mathcal{L}_t}{\partial \mathbf{y}_t} \stackrel{(7)}{=} \frac{\partial \mathcal{L}_t}{\partial \mathbf{y}_t}.\end{aligned}$$

- The gradient of the loss function of RNN with respect to the bias  $\mathbf{b}_y$  is:

$$\mathbb{R}^q \ni \frac{\partial \mathcal{L}}{\partial \mathbf{b}_y} \stackrel{(a)}{=} \sum_{t=1}^T \left( \frac{\partial \mathcal{L}}{\partial \mathbf{y}_t} \times \frac{\partial \mathbf{y}_t}{\partial \mathbf{b}_y} \right) \stackrel{(b)}{=} \sum_{t=1}^T \frac{\partial \mathcal{L}_t}{\partial \mathbf{y}_t}, \quad (21)$$

where (a) is because  $\mathbf{b}_y$  exists in all time slots and changing  $\mathbf{b}_y$  affects the loss  $\mathcal{L}$  in all time slots. The equation (b) is because of Eqs. (8) and (5).

- The derivative  $\partial \mathcal{L}_t / \partial \mathbf{y}_t \in \mathbb{R}^q$  is calculated based on the formula of the loss function.

# Updates by Gradient Descent

- BPPT updates the parameters of RNN by gradient descent [24] using the calculated gradients:

$$\mathbf{h}_t := \mathbf{h}_t - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}, \quad \forall t \in \{1, \dots, T\},$$

$$\mathbf{V} := \mathbf{V} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{V}},$$

$$\mathbf{W} := \mathbf{W} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}},$$

$$\mathbf{U} := \mathbf{U} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{U}},$$

$$\mathbf{b}_i := \mathbf{b}_i - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{b}_i},$$

$$\mathbf{b}_y := \mathbf{b}_y - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{b}_y},$$

where  $\eta > 0$  is the learning rate and the gradients are calculated by Eqs. (10), (16), (17), (19), (20), and (21).

**Gradient Vanishing or  
Explosion in Long-term  
Dependencies**

# Gradient Vanishing or Explosion in Long-term Dependencies

- In recurrent neural networks, so as in deep neural networks, the final output is the composition of a large number of non-linear transformations. This results in the problem of either **vanishing or exploding gradients** in recurrent neural networks, especially for capturing long-term dependencies in sequence processing [7, 8]. This problem is explained in the following.
- Recall Eq. (2) for a dynamical system:

$$\mathbf{h}_t = f_{\theta}(\mathbf{h}_{t-1}, \mathbf{x}_t).$$

By induction, the hidden state at time  $t$ , i.e.,  $\mathbf{h}_t$ , can be written as the previous  $T$  time steps. If the subscript  $t$  denotes the previous  $t$  time steps, we have by induction [26, 27]:

$$\mathbf{h}_1 = f_{\theta}\left(f_{\theta}\left(\dots f_{\theta}(\mathbf{h}_T, \mathbf{x}_{T+1}) \dots, \mathbf{x}_2\right), \mathbf{x}_1\right).$$

- Then, by the chain rule in derivatives, the derivative loss at time  $T$ , i.e.,  $\mathcal{L}_T$ , is:

$$\frac{\partial \mathcal{L}_T}{\partial \theta} = \sum_{t \leq T} \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \theta} \stackrel{(a)}{=} \sum_{t \leq T} \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_T} \frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \theta} \stackrel{(2)}{=} \sum_{t \leq T} \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_T} \frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_t} \frac{\partial f_{\theta}(\mathbf{h}_{t-1}, \mathbf{x}_t)}{\partial \theta}, \quad (22)$$

where (a) is because of the chain rule. In this expression, there is the derivative of  $\mathbf{h}_T$  with respect to  $\mathbf{h}_t$  which itself can be calculated by the chain rule:

$$\frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_t} = \frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_{T-1}} \times \frac{\partial \mathbf{h}_{T-1}}{\partial \mathbf{h}_{T-2}} \times \dots \times \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}. \quad (23)$$



# Gradient Vanishing or Explosion in Long-term Dependencies

- We found:

$$\frac{\partial \mathcal{L}_T}{\partial \theta} = \sum_{t \leq T} \frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_T} \frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_t} \frac{\partial f_\theta(\mathbf{h}_{t-1}, \mathbf{x}_t)}{\partial \theta},$$
$$\frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_t} = \frac{\partial \mathbf{h}_T}{\partial \mathbf{h}_{T-1}} \times \frac{\partial \mathbf{h}_{T-1}}{\partial \mathbf{h}_{T-2}} \times \dots \times \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}.$$

- For capturing **long-term dependencies** in the sequence,  $T$  should be **large**. This means that in Eq. (23), and hence in Eq. (22), the number of multiplicand terms becomes huge.
- On the one hand, if each derivative is **slightly smaller than one**, the entire derivative in the chain rule becomes very small for multiplication of many terms smaller than one. This problem is referred to as **gradient vanishing**.
- On the other hand, if every derivative is **slightly larger than one**, the entire derivative in the chain rule explodes, resulting in the problem of **exploding gradients**.
- Note that gradient vanishing is more common than gradient explosion in recurrent networks.
- There exist various attempts for resolving the problem of gradient vanishing or explosion [26, 28]. In the following, some of these attempts are introduced.

# Close-to-identity Weight Matrix

- As Eq. (4):

$$[-1, 1]^p \ni \mathbf{h}_t = \tanh(\mathbf{i}_t) = \tanh(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}_i),$$

shows, the state is multiplied by a weight matrix  $\mathbf{W}$  at every time step and if there is long-term dependency, many of these  $\mathbf{W}$  matrices are multiplied.

- Suppose the eigenvalue decomposition [29] of the matrix  $\mathbf{W}$  is  $\mathbf{W} = \mathbf{A}\mathbf{\Lambda}\mathbf{A}^\top$  where  $\mathbf{A} \in \mathbb{R}^{p \times p}$  and  $\mathbf{\Lambda} := \text{diag}([\lambda_1, \dots, \lambda_p]^\top)$  contain the eigenvectors and eigenvalues of  $\mathbf{W}$ , respectively. Eq. (4) is restated as:

$$\mathbf{h}_t = \tanh(\mathbf{A}\mathbf{\Lambda}\mathbf{A}^\top \mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}_i). \quad (24)$$

- If a change  $\varepsilon$  in some element of the state  $\mathbf{h}_{t-1}$  is aligned with an eigenvector of the weight matrix  $\mathbf{W}$ , then the effect of this change in  $\mathbf{h}_t$  will be  $(\lambda^t \varepsilon)$  after  $t$  time steps, according to Eq. (24).
- Two cases may happen:
  - ▶ If the largest eigenvalue is less than one, i.e.,  $\lambda < 1$ , then the change  $(\lambda^t \varepsilon)$  is contrastive because  $\lambda^t \ll 1$  for long-term dependencies. In this case, gradient vanishing occurs in long-term dependency and the network forgets very long time ago.
  - ▶ If the largest eigenvalue is less than one, i.e.,  $\lambda > 1$ , then the change  $(\lambda^t \varepsilon)$  is diverging because  $\lambda^t \gg 1$  for long-term dependencies. In this case, the gradient network forgets very long time ago. In this case, gradient explosion occurs in long-term dependency and remembering very long time ago dominates the short-term memories.

# Close-to-identity Weight Matrix

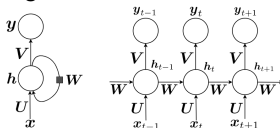
- As remembering short-term memories is usually more important than remembering very past time in different tasks, it is recommended to use the weight matrix  $\mathbf{W}$  whose largest eigenvalue is less than one; this makes the RNN have the Markovian property because it forgets very past after some point.
- However, if the largest eigenvalue of  $\mathbf{W}$  is much less than one, i.e.,  $\lambda \ll 1$ , gradient vanishing happens very sooner than expected.
- Therefore, it is recommended to use the weight matrix  $\mathbf{W}$  whose largest eigenvalue is **slightly** less than one, i.e.,  $\lambda \lesssim 1$ . This makes the RNN **slightly contrastive**.
- One way to have the weight matrix  $\mathbf{W}$  whose largest eigenvalue is slightly less than one is to make this matrix close to the identity matrix [9].
- There exist some other ways to determine the weight matrix  $\mathbf{W}$ . For example, the weight matrix can be set to be an orthogonal matrix [30].
- Another approach is to copy the previous state exactly to the current state. In this approach, the Eq. (4) is modified to [31]:

$$\begin{aligned} \mathbf{h}_t &= \tanh(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}_i) \\ &= \tanh((\mathbf{W} + \mathbf{I})\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}_i), \end{aligned} \tag{25}$$

where  $\mathbf{I}$  is the identity matrix. This prevents gradient vanishing because it brings a copy of the previous step to the current state. This can also be interpreted as strengthening the diagonal of the weight matrix  $\mathbf{W}$ ; hence, increasing the largest eigenvalue of  $\mathbf{W}$  for preventing gradient vanishing.

# Long Delays

- As Eq. (4) and the figure of RNN show, in the regular RNN, every state  $\mathbf{h}_t$  is fed by its previous state  $\mathbf{h}_{t-1}$  through the weight matrix  $\mathbf{W}$ .



- As discussed before, in the regular RNN, the effect of the change  $\varepsilon$  in a state results in  $(\lambda^t \varepsilon)$  after  $t$  time steps, where  $\lambda$  is the largest eigenvalue of  $\mathbf{W}$ .
- As shown in the figure of RNN, the regular RNN has one-step connections or delays between the states. It is possible to have **longer delays** between the states in addition to the one-step delays [10].
- In other words, it is possible to have **higher levels of Markov property** in the network.
- Let  $\mathbf{W}_k$  denote the weight matrix for  $k$ -step delays between the states. Then, Eq. (4) can be modified to:

$$\mathbf{h}_t = \tanh \left( \sum_k \mathbf{W}_k \mathbf{h}_{t-k} + \mathbf{U} \mathbf{x}_t + \mathbf{b}_i \right), \quad (26)$$

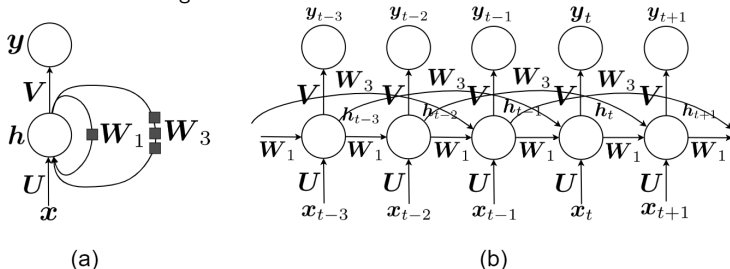
where the summation is over the  $k$  values for the existing delays in the RNN structure.

# Long Delays

- An example for an RNN network with one-step and three-step delays is:

$$h_t = \tanh(W_1 h_{t-1} + W_3 h_{t-3} + U x_t + b_i),$$

which is illustrated in this figure.



- Having long delays in RNN is one of the attempts for **preventing gradient vanishing** [10]. This is justified because **every state is having impact not only from the previous state but also from the more previous states**. Therefore, in backpropagation through time, there is **some skip in gradient flow from a state to more previous states** without the need to go through the middle states in the chain rule.

# Leaky Units

- Another way to resolve the problem of gradient vanishing is **leaky units** [11, 12].
- Let  $h_{t,j}$  denote the  $j$ -th element of the state  $\mathbf{h}_t \in [-1, 1]^p$ . In leaky units, Eq. (4) is modified to the following element-wise equation:

$$h_{t,j} = (1 - \frac{1}{\tau_j}) h_{t-1,j} + \frac{1}{\tau_j} \tanh(\mathbf{W}_j \mathbf{h}_{t-1} + \mathbf{U}_j \mathbf{x}_t + b_{i,j}), \quad (27)$$

where  $1 \leq \tau_j < \infty$  and  $\mathbf{W}_j$  is the  $j$ -th row of  $\mathbf{W}$  and  $\mathbf{U}_j$  is the  $j$ -th row of  $\mathbf{U}$  and  $b_{i,j}$  is the  $j$ -th element of  $\mathbf{b}_i$ .

- When  $\tau_j = 1$ , then Eq. (27) becomes:

$$h_{t,j} = \frac{1}{\tau_j} \tanh(\mathbf{W}_j \mathbf{h}_{t-1} + \mathbf{U}_j \mathbf{x}_t + b_{i,j}),$$

which gives back Eq. (4) in the **regular RNN**.

- However, when  $\tau_j \gg 1$ , then Eq. (27) becomes:

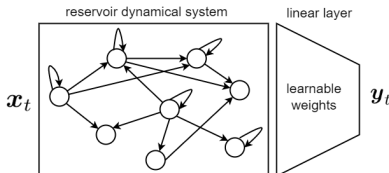
$$h_{t,j} = h_{t-1,j},$$

which means that **the previous state is copied to the current state**.

- The larger the  $\tau_i$ , the easier the gradient propagates for  $h_{t,i}$ . Therefore, by tuning  $\tau_i$ , it is possible to **control how much of the past should be directly copied and how much should be passed through the weight matrix**. This can control the **amount of gradient vanishing**.
- Note that leaky units use different  $\tau_i$ 's because there may be a need to keep some of the directions of states (with  $\tau_i = 1$ ) or forget some of the directions (with  $\tau_i \gg 1$ ). In other words, it **decides about the  $p$  directions of states separately**.

# Echo State Networks

- One of the approaches to handle the problem of gradient vanishing in RNN is to use **echo state networks** [13, 14].
- These networks consider the recurrent neural network as a **black box having hidden units with nonlinear activation functions and connections between them**.
- This black box of recurrent connections is called the **reservoir dynamical system** which models the internal structure of a computer or brain.
- The connections in the reservoir system are usually **sparse** and the weights of these connections are considered to be **fixed**.
- The output of the reservoir system is connected to an additional **linear output layer** whose weights are **learnable**.
- The echo state network minimizes the **mean squared error** in the output layer; hence, it performs **linear regression** in the last layer [13].
- Because of **not learning the recurrent weights in the reservoir system** and sufficing to learn the weights of the output layer, the echo state network does not face the **gradient vanishing problem**.
- A tutorial on this topic is [32].



## **Long Short-Term Memory Network**

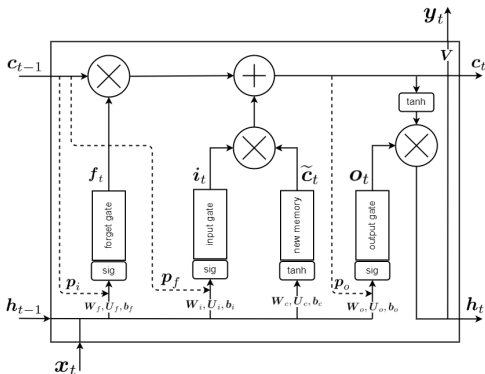
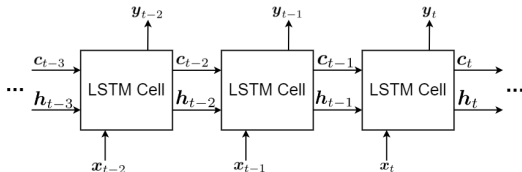


# Long Short-Term Memory Network

- As the examples in introduction showed, we need **short-term relations** in some cases and **long-term relations** in some other cases.
- RNN learns the sequence based on one or several previous states, depending on its structure and the level of its Markov property (see the figure for long delays). Therefore, we need to **decide on the structure of RNN to be able to handle short-term or long-term dependencies** in the sequence.
- Instead of **manual design of the RNN structure** or **deciding manually when to clear the state**, we can **let the neural network learn by itself** when to clear the state based on its input sequence.
- **Long Short-Term Memory (LSTM)**, initially proposed in [15, 16] (1995-1997), is able to do this; it learns from its input sequence when to use short-term dependency (i.e., when to clear the state) and when to use the long-term memory (i.e., when not to clear the state).

# Long Short-Term Memory Network

- LSTM consists of several cells, each of which corresponds to a time slot. Every LSTM cell contains several gates for learning different aspects of the input time series.



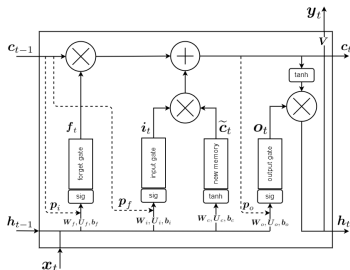
# The Input Gate

- One of the gates in the LSTM cell is the **input gate**, first proposed in [15, 16].
- This gate takes the input at the current time slot,  $\mathbf{x}_t \in \mathbb{R}^d$ , and the hidden state of the last time slot,  $\mathbf{h}_{t-1} \in [-1, 1]^p$ , and outputs the signal  $\mathbf{i}_t \in [0, 1]^p$ :

$$\mathbf{i}_t = \text{sig}(\mathbf{W}_i \mathbf{h}_{t-1} + \mathbf{U}_i \mathbf{x}_t + (\mathbf{p}_i \odot \mathbf{c}_{t-1}) + \mathbf{b}_i), \quad (28)$$

where  $\mathbf{W}_i \in \mathbb{R}^{p \times p}$ ,  $\mathbf{U}_i \in \mathbb{R}^{p \times d}$ , and the bias  $\mathbf{b}_i \in \mathbb{R}^p$  are the learnable weights for the input gate,  $\odot$  denotes the Hadamard (element-wise) product,  $\mathbf{c}_{t-1} \in \mathbb{R}^p$  is the final memory of the last time slot (which will be explained later), and  $\mathbf{p}_i \in \mathbb{R}^p$  is the learnable **peephole weight** [33] letting a possible leak of information from the previous final memory. The function  $\text{sig}(\cdot) \in (0, 1)$  is the sigmoid function which is applied element-wise:

$$\text{sig}(x) = \frac{1}{1 + \exp(-x)}. \quad (29)$$



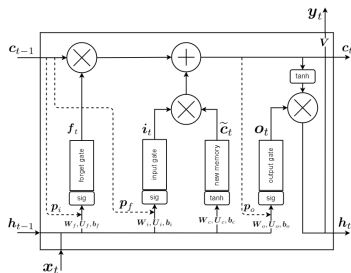
# The Input Gate

- As Eq. (28):

$$i_t = \text{sig}(W_i h_{t-1} + U_i x_t + (p_i \odot c_{t-1}) + b_i),$$

demonstrates, the input gate considers the **effect of the input and the previous hidden state**.

- It may also use a **leak of information from the previous memory** through the peephole.
- This gate carries the **importance of the information of the input at the current time slot**.



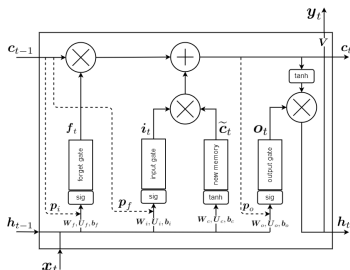
# The Forget Gate

- Another gate in the LSTM cell is the **forget gate**, first proposed in [34].
- This gate also takes the input at the current time slot,  $\mathbf{x}_t \in \mathbb{R}^d$ , and the hidden state of the last time slot,  $\mathbf{h}_{t-1} \in [-1, 1]^p$ , and outputs the signal  $\mathbf{f}_t \in [0, 1]^p$ :

$$\mathbf{f}_t = \text{sig}(\mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{U}_f \mathbf{x}_t + (\mathbf{p}_f \odot \mathbf{c}_{t-1}) + \mathbf{b}_f), \quad (30)$$

where  $\mathbf{W}_f \in \mathbb{R}^{p \times p}$ ,  $\mathbf{U}_f \in \mathbb{R}^{p \times d}$ , and the bias  $\mathbf{b}_f \in \mathbb{R}^p$  are the learnable weights for the forget gate, and  $\mathbf{p}_f \in \mathbb{R}^p$  is the learnable **peephole weight** [33] letting a possible leak of information from the previous final memory.

- As Eq. (30) shows, the forget gate considers the **effect of the input and the previous hidden state**, and **perhaps a leak of information from the previous memory**.
- This gate controls the **amount of forgetting the previous information with respect to the new-coming information**.



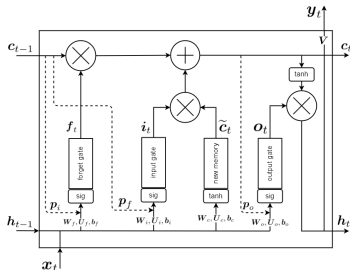
# The Output Gate

- The next gate in the LSTM cell is the **output gate** first proposed in [15, 16]. This gate also takes the input at the current time slot,  $\mathbf{x}_t \in \mathbb{R}^d$ , and the hidden state of the last time slot,  $\mathbf{h}_{t-1} \in [-1, 1]^p$ , and outputs the signal  $\mathbf{o}_t \in [0, 1]^p$ :

$$\mathbf{o}_t = \text{sig}(\mathbf{W}_o \mathbf{h}_{t-1} + \mathbf{U}_o \mathbf{x}_t + (\mathbf{p}_o \odot \mathbf{c}_t) + \mathbf{b}_o), \quad (31)$$

where  $\mathbf{W}_o \in \mathbb{R}^{p \times p}$ ,  $\mathbf{U}_o \in \mathbb{R}^{p \times d}$ , and the bias  $\mathbf{b}_o \in \mathbb{R}^p$  are the learnable weights for the output gate, and  $\mathbf{p}_o \in \mathbb{R}^p$  is the learnable **peephole weight** [33] letting a possible leak of information from the current final memory.

- As shown in Eq. (31), the output gate considers the **effect of the input and the previous hidden state**, and a **possible information leak from the current memory**.



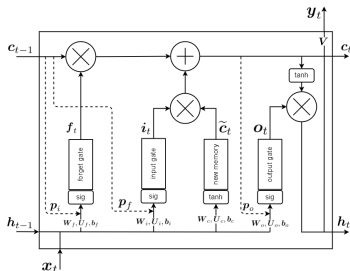
# The New Memory Cell (Block Input)

- The LSTM cell includes a gate named the **new memory cell**. This gate takes the input at the current time slot,  $\mathbf{x}_t \in \mathbb{R}^d$ , and the hidden state of the last time slot,  $\mathbf{h}_{t-1} \in [-1, 1]^p$ , and outputs the signal  $\tilde{\mathbf{c}}_t \in [-1, 1]^p$ .
- This gate considers **the effect of the input and the previous hidden state to represent the new information of current input**. It is formulated as:

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c \mathbf{h}_{t-1} + \mathbf{U}_c \mathbf{x}_t + \mathbf{b}_c), \quad (32)$$

where  $\mathbf{W}_c \in \mathbb{R}^{p \times p}$ ,  $\mathbf{U}_c \in \mathbb{R}^{p \times d}$ , and the bias  $\mathbf{b}_c$  are the learnable weights for the new memory cell.

- The new memory cell is also referred to as the **block input** in the literature [35]. The signal  $\tilde{\mathbf{c}}_t$  is sometimes denoted by  $\mathbf{z}_t$  in the literature.



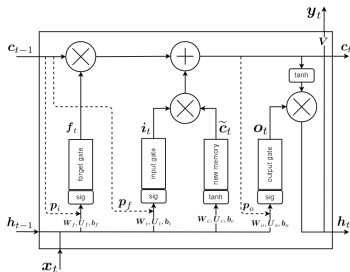
# The Final Memory Calculation

- After computation of the outputs of the input gate  $i_t$ , the forget gate  $f_t$ , and the new memory cell  $\tilde{c}_t$ , we calculate the **final memory**  $c_t \in \mathbb{R}^p$ :

$$c_t = (f_t \odot c_{t-1}) + (i_t \odot \tilde{c}_t), \quad (33)$$

where  $c_{t-1} \in \mathbb{R}^p$  is the final memory of the previous time slot.

- As Eq. (33) demonstrates, the final memory considers the **effect of the forget gate, the previous memory, the input, and the new memory**.
- In the first term, i.e.,  $f_t \odot c_{t-1}$ , the forget gate  $f_t \in [0, 1]^p$  controls **how much of the previous memory  $c_{t-1}$  should be forgotten**. The closer the  $f_t$  is to zero, the more the network forgets the previous memory  $c_{t-1}$ .
- In the second term, i.e.,  $i_t \odot \tilde{c}_t$ , the input gate  $i_t \in [0, 1]^p$  and the new memory cell  $\tilde{c}_t \in [-1, 1]^p$  both control **how much of the new input information should be used**. The closer the input gate  $i_t$  is to one and the closer the new memory cell  $\tilde{c}_t$  is to  $\pm 1$ , the more the input information is used.



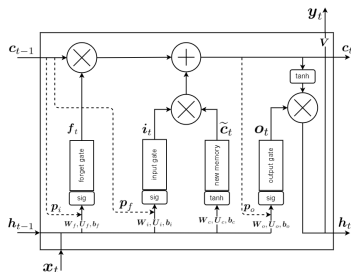


# The Final Memory Calculation

- We had:

$$\mathbf{c}_t = (\mathbf{f}_t \odot \mathbf{c}_{t-1}) + (\mathbf{i}_t \odot \tilde{\mathbf{c}}_t).$$

- In other words, the first and second terms in Eq. (33) determine the **trade-off of usage of old versus new information** in the sequence.
- The weights of these gates are trained in a way that they **pass or block the input/previous information based on the input sequence and the time step in the sequence**.

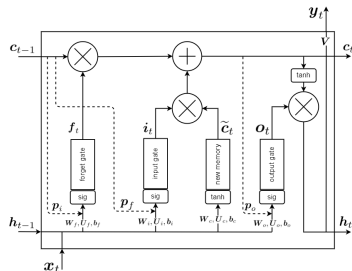


# The Hidden State (Block Output)

- After computation of the output of the output gate  $\mathbf{o}_t$  and the final memory  $\mathbf{c}_t$ , we calculate the **hidden state**  $\mathbf{h}_t \in [-1, 1]^p$ :

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t). \quad (34)$$

- This hidden state is also considered as the **block output** of the LSTM cell.



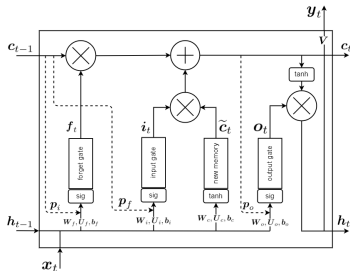
# The Output

- The output  $\mathbf{y}_t \in \mathbb{R}^q$  of the LSTM cell is as follows:

$$\mathbf{y}_t = \mathbf{V}\mathbf{h}_t + \mathbf{b}_y, \quad (35)$$

where  $\mathbf{V} \in \mathbb{R}^{q \times p}$  and the bias  $\mathbf{b}_y \in \mathbb{R}^q$  are the learnable weights for the output. It is possible to use an activation function, such as Eq. (6), after this output signal.

- Note that in the literature, the output is sometimes considered to be equal to the hidden state, i.e.,  $\mathbf{y}_t = \mathbf{h}_t$ , by setting  $\mathbf{V} = \mathbf{I}$  (the identity matrix) and  $\mathbf{b}_y = \mathbf{0}$  (the zero vector).



## History and Variants of LSTM

# History and Variants of LSTM

- LSTM has gone through various developments and improvements gradually [35].
- Some of the variants of LSTM do not have the peepholes. In this case, the Eqs. (28), (30), and (31) are simplified to:

$$\mathbf{i}_t = \text{sig}(\mathbf{W}_i \mathbf{h}_{t-1} + \mathbf{U}_i \mathbf{x}_t + \mathbf{b}_i), \quad (36)$$

$$\mathbf{f}_t = \text{sig}(\mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{U}_f \mathbf{x}_t + \mathbf{b}_f), \quad (37)$$

$$\mathbf{o}_t = \text{sig}(\mathbf{W}_o \mathbf{h}_{t-1} + \mathbf{U}_o \mathbf{x}_t + \mathbf{b}_o), \quad (38)$$

respectively.

- In the following, we review a history of variants of the LSTM networks.

# Original LSTM

- LSTM was originally proposed by Hochreiter and Schmidhuber in years 1995 to 1997 [15, 16]. We call it the **original LSTM** [16].
- The original LSTM had **only the input and output gates**, introduced before, and it did not have a forget gate.
- It also did **not contain the peepholes**; therefore, its gates were Eqs. (36) and (38).
- The original LSTM trained the network using **BPTT** (introduced before) and a mixture of real-time recurrent learning [2, 4].

# Vanilla LSTM

- Later, Gers *et. al.* [34, 33] applied some changes to the original LSTM.
- The **forget gate**, introduced before, was proposed for the first time in [34] to let the network forget its previous states either completely or partially.
- The **peephole connections**, introduced before, were first proposed in [33]. The peepholes let a possible leak of information from the previous or current final memory. This lets the memory control the gates.
- These two papers [34, 33] also incorporated the **full gate recurrence**, in which **all gates receive additional recurrent inputs from all gates at the previous time step**.
- In full gate recurrence, the Eqs. (28), (30), and (31) become:

$$\mathbf{i}_t = \text{sig}(\mathbf{W}_i \mathbf{h}_{t-1} + \mathbf{U}_i \mathbf{x}_t + (\mathbf{p}_i \odot \mathbf{c}_{t-1}) + \mathbf{b}_i + \mathbf{R}_{ii} \mathbf{i}_{t-1} + \mathbf{R}_{if} \mathbf{f}_{t-1} + \mathbf{R}_{io} \mathbf{o}_{t-1}). \quad (39)$$

$$\mathbf{f}_t = \text{sig}(\mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{U}_f \mathbf{x}_t + (\mathbf{p}_f \odot \mathbf{c}_{t-1}) + \mathbf{b}_f + \mathbf{R}_{fi} \mathbf{i}_{t-1} + \mathbf{R}_{ff} \mathbf{f}_{t-1} + \mathbf{R}_{fo} \mathbf{o}_{t-1}). \quad (40)$$

$$\mathbf{o}_t = \text{sig}(\mathbf{W}_o \mathbf{h}_{t-1} + \mathbf{U}_o \mathbf{x}_t + (\mathbf{p}_o \odot \mathbf{c}_t) + \mathbf{b}_o + \mathbf{R}_{oi} \mathbf{i}_{t-1} + \mathbf{R}_{of} \mathbf{f}_{t-1} + \mathbf{R}_{oo} \mathbf{o}_{t-1}), \quad (41)$$

where  $\mathbf{R}_{ii}, \mathbf{R}_{if}, \mathbf{R}_{io}, \mathbf{R}_{fi}, \mathbf{R}_{ff}, \mathbf{R}_{fo}, \mathbf{R}_{oi}, \mathbf{R}_{of}, \mathbf{R}_{oo} \in \mathbb{R}^{P \times P}$  are the learnable recurrent weights.

- Note that the full gate recurrence often **disappeared** in later papers on LSTM.

# Vanilla LSTM

- Later, Graves and Schmidhuber adapted the original LSTM and proposed the **vanilla LSTM** in 2005 [18], which is one of the most common LSTMs in the literature.
- The vanilla LSTM incorporated the structures of the original LSTM [16] and the papers [34, 33].
- The full BPTT, introduced before, was used for LSTM in the vanilla LSTM [18].



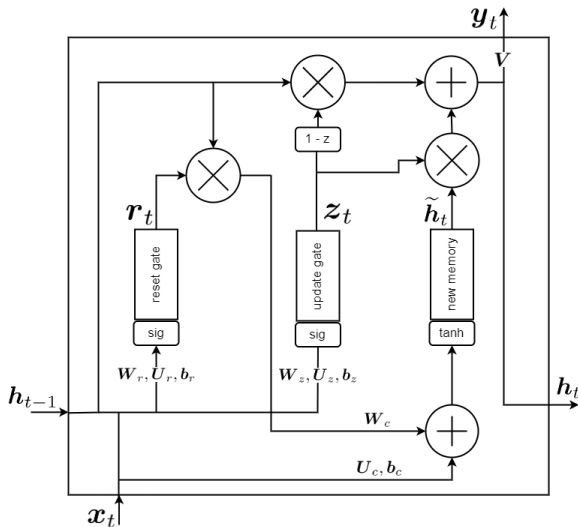
# Other LSTM Variants

- There are other variants of LSTM [35, 36]; although, the most common used LSTM is the **vanilla LSTM** [18].
- **BPTT** was used for LSTM training in [18]; however, **Kalman filtering** was used for its training [37] before that. Another training method for LSTM was **evolutionary learning** [38]. **Context-sensitive evolutionary learning** was also used for LSTM training [39].
- Finally in 2014, one the biggest improvements of LSTM was proposed, which was named the **Gated Recurrent Units (GRU)** [17].
- The philosophy of GRU was to **simplify the LSTM cell** because we may not need to have a very complicated cell to learn the sequence information. In other words, GRU raised the question of **whether we need to be that flexible like LSTM to learn the sequence**. GRU is **less flexible than LSTM** but it is **good enough** for sequence learning.
- GRU **redesigned** the LSTM cell by introducing **reset gate, update gate, and new memory cell**; therefore, the **number of gates were reduced from four to three**.
- It was empirically shown in [40] that the performance of LSTM **improves** by using GRU cells.
- Later in 2017, the GRU was further simplified by **merging the reset and update gates into a forget gate** [41].
- Nowadays, GRU is the most commonly used LSTM structure.

## Gated Recurrent Units (GRU)

# GRU: Fully Gated Unit

- The main GRU was the **fully gated unit** [17], whose gates are introduced in the following.





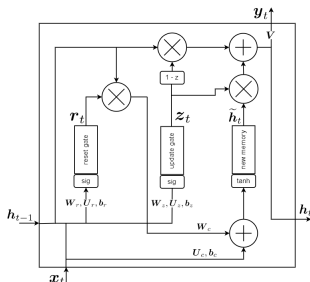
# GRU: Fully Gated Unit

- **The Update Gate:** Another gate in the GRU cell is the **update gate**. This gate also takes the input at the current time slot,  $\mathbf{x}_t \in \mathbb{R}^d$ , and the hidden state of the last time slot,  $\mathbf{h}_{t-1} \in [-1, 1]^p$ , and outputs the signal  $\mathbf{z}_t \in [0, 1]^p$ :

$$\mathbf{z}_t = \text{sig}(\mathbf{W}_z \mathbf{h}_{t-1} + \mathbf{U}_z \mathbf{x}_t + \mathbf{b}_z), \quad (43)$$

where  $\mathbf{W}_z \in \mathbb{R}^{p \times p}$ ,  $\mathbf{U}_z \in \mathbb{R}^{p \times d}$ , and the bias  $\mathbf{b}_z \in \mathbb{R}^p$  are the learnable weights for the update gate.

- The update gate considers the **effect of the input and the previous hidden state**, and it controls the **amount of using the new input data for updating the cell by the coming information of sequence**.
- Comparing Eqs. (36) and (43) shows that the **update gate in the GRU cell** is similar to the **input gate in the LSTM cell**.



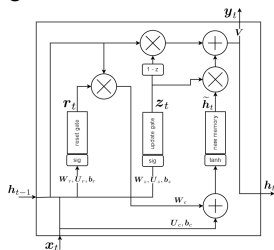
# GRU: Fully Gated Unit

- **The New Memory Cell:** The GRU cell includes a gate named the **new memory cell**. This gate takes the input at the current time slot,  $\mathbf{x}_t \in \mathbb{R}^d$ , and the hidden state of the last time slot,  $\mathbf{h}_{t-1} \in [-1, 1]^p$ , and outputs the signal  $\tilde{\mathbf{h}}_t \in [-1, 1]^p$ :

$$\tilde{\mathbf{h}}_t = \tanh\left(\mathbf{W}_c (\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{U}_c \mathbf{x}_t + \mathbf{b}_c\right), \quad (44)$$

where  $\mathbf{W}_c \in \mathbb{R}^{p \times p}$ ,  $\mathbf{U}_c \in \mathbb{R}^{p \times d}$ , and the bias  $\mathbf{b}_c$  are the learnable weights for the new memory cell.

- This gate considers the **effect of the input and the previous hidden state to represent the new information of current input**.
- Comparing Eqs. (32) and (44) shows that the **new memory cell in the GRU cell** is similar to the **new memory cell in the LSTM cell**.
- Note that, in the LSTM cell, the hidden state (see Eq. (34)) and the new memory cell (see Eq. (32)) were **different**; however, the hidden state of the GRU cell (see Eq. (44)) **replaces** the new memory signal in the LSTM cell.

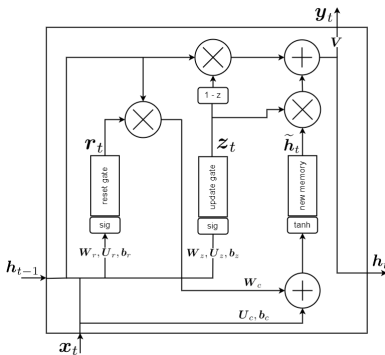


# GRU: Fully Gated Unit

- **The Final Memory (Hidden State):** After computation of the outputs of the update gate  $z_t$  and the new memory cell  $\tilde{h}_t$ , we calculate the **final memory** or the **hidden state**  $h_t \in \mathbb{R}^p$ :

$$h_t = ((1 - z_t) \odot h_{t-1}) + (z_t \odot \tilde{h}_t), \quad (45)$$

where  $h_{t-1} \in \mathbb{R}^p$  is the hidden state of the previous time slot.



# GRU: Fully Gated Unit

- We had:

$$\mathbf{h}_t = ((\mathbf{1} - \mathbf{z}_t) \odot \mathbf{h}_{t-1}) + (\mathbf{z}_t \odot \tilde{\mathbf{h}}_t),$$

where  $\mathbf{h}_{t-1} \in \mathbb{R}^p$  is the hidden state of the previous time slot.

- As Eq. (45) demonstrates, the final memory considers the **effect of the update gate, the previous memory, and the new memory**.
- In the first term, i.e.,  $(\mathbf{1} - \mathbf{z}_t) \odot \mathbf{h}_{t-1}$ , the update gate  $\mathbf{z}_t \in [0, 1]^p$  controls **how much of the previous state  $\mathbf{h}_{t-1}$  should be used based on the input data**. The closer the  $\mathbf{z}_t$  is to one (resp. zero), the more the network forgets (resp. considers) the previous state  $\mathbf{h}_{t-1}$ .
- In the second term, i.e.,  $\mathbf{z}_t \odot \tilde{\mathbf{h}}_t$ , the update gate  $\mathbf{z}_t \in [0, 1]^p$  and the new memory cell  $\tilde{\mathbf{h}}_t \in [-1, 1]^p$  both control **how much of the new input information should be used**. In other words, it controls **how much the information should be updated by the new information**. The closer the update gate  $\mathbf{z}_t$  is to one and the closer the new memory cell  $\tilde{\mathbf{h}}_t$  is to  $\pm 1$ , the more the input information is used.
- Overall, the first and second terms in Eq. (45) determine the **trade-off of usage of old versus new information** in the sequence. The weights of these gates are trained in a way that they **pass or block the input/previous information based on the input sequence and the time step in the sequence**.
- Comparing Eqs. (33) and (45) shows that the **final memory in the GRU cell** is in the form of the **final memory in the LSTM cell**; however, they have somewhat different functionality.



# GRU: Minimal Gated Unit

- **Minimal gated unit** [41] is another variant of GRU which has simplified the gate by merging the reset and update gates into a forget gate. This merging is possible because the forget gate can control both the previous and new information of the sequence.
- **The Forget Gate:** The *forget gate* takes the input at the current time slot,  $\mathbf{x}_t \in \mathbb{R}^d$ , and the hidden state of the last time slot,  $\mathbf{h}_{t-1} \in [-1, 1]^p$ , and outputs the signal  $\mathbf{r}_t \in [0, 1]^p$ :

$$\mathbf{f}_t = \text{sig}(\mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{U}_f \mathbf{x}_t + \mathbf{b}_f), \quad (46)$$

where  $\mathbf{W}_f \in \mathbb{R}^{p \times p}$ ,  $\mathbf{U}_f \in \mathbb{R}^{p \times d}$ , and the bias  $\mathbf{b}_f \in \mathbb{R}^p$  are the learnable weights for the forget gate.

- The forget gate considers the **effect of the input and the previous hidden state**, and it controls the **amount of forgetting the previous information with respect to the new-coming information**. Therefore, it controls **both forgetting or using the previous memory and using the new coming information**.

# GRU: Minimal Gated Unit

- **The New Memory Cell and the Final Memory:** Because the forget gate replaces the reset and the update gate in the minimal gate unit, Eqs. (44) and (45):

$$\tilde{\mathbf{h}}_t = \tanh\left(\mathbf{W}_c (\mathbf{r}_t \odot \mathbf{h}_{t-1})\right) + \mathbf{U}_c \mathbf{x}_t + \mathbf{b}_c,$$

$$\mathbf{h}_t = ((1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1}) + (\mathbf{z}_t \odot \tilde{\mathbf{h}}_t),$$

are changed to:

$$\tilde{\mathbf{h}}_t = \tanh\left(\mathbf{W}_c (\mathbf{f}_t \odot \mathbf{h}_{t-1})\right) + \mathbf{U}_c \mathbf{x}_t + \mathbf{b}_c, \quad (47)$$

$$\mathbf{h}_t = ((1 - \mathbf{f}_t) \odot \mathbf{h}_{t-1}) + (\mathbf{f}_t \odot \tilde{\mathbf{h}}_t), \quad (48)$$

respectively, to be the new memory cell and the final memory in the minimal gate unit.

## **Bidirectional RNN and LSTM**

# Justification of Bidirectional Processing

- A bidirectional RNN or LSTM network processes the sequence in **both directions; left to right and right to left**.
- In the first glance, **online causal tasks** such as **reading a text or listening to a speech** do not have access to the **future**. Therefore, bidirectional networks seem to **violate causality** in them.
- However, in many of these tasks, it is possible to **wait for the completion of a part of the sequence such as a sentence and then decide about it**.
- For example, it is normal to **wait for the completion of sentence in speech recognition** and then recognize it [18, 19].
- In **text processing**, the text is **usually available except in a streaming text**. Even in streaming text, it is possible to **wait for a sentence to complete**.
- Therefore, it makes sense to use bidirectional networks for processing sequences because, sometimes, the important related word comes after a word and not necessarily before it.
- An example for such a case is the sentence **“The police is chasing the thief”** where the word **“thief”** is a strongly related (opposite) word for the word **“police”**. In this sentence, both the words **“thief”** and **“police”** are related and it is worth to process the sentence in both directions.

# Bidirectional RNN

- The **bidirectional RNN** was first proposed in (1997) [20] and further exploited in (1999) [21].
- It uses **two sets of states each for one of the directions** in the sequence. Let the states for left-to-right and right-to-left processing be denoted by  $\vec{h}_t$  and  $\overleftarrow{h}_t$ , respectively. In the bidirectional RNN, Eq. (4) is replaced by two equations [42]:

$$\vec{h}_t = \tanh(\vec{W}\vec{h}_{t-1} + \vec{U}x_t + \vec{b}_i), \quad (49)$$

$$\overleftarrow{h}_t = \tanh(\overleftarrow{W}\overleftarrow{h}_{t+1} + \overleftarrow{U}x_t + \overleftarrow{b}_i), \quad (50)$$

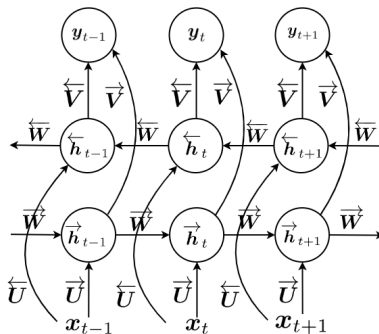
and Eq. (5) is replaced by:

$$y_t = \vec{V}\vec{h}_t + \overleftarrow{V}\overleftarrow{h}_t + b_y, \quad (51)$$

where the arrows show the parameters for each direction of processing.

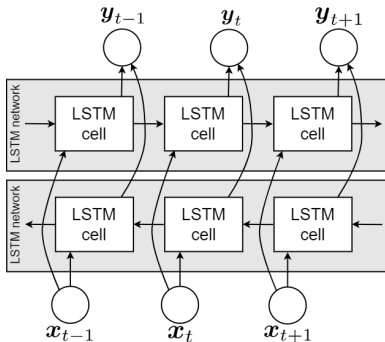
# Bidirectional RNN

- The unfolding schematic of the bidirectional RNN is illustrated in figure below. As this figure shows, the **outputs of both directions are connected to an output layer**. In some cases, this output layer may be replaced by a third multi-layer neural network.
- All weights of the bidirectional RNN are trained using **backpropagation through time** similarly to what was explained before.
- It is noteworthy that the deep variant of bidirectional RNN has been proposed in [42].



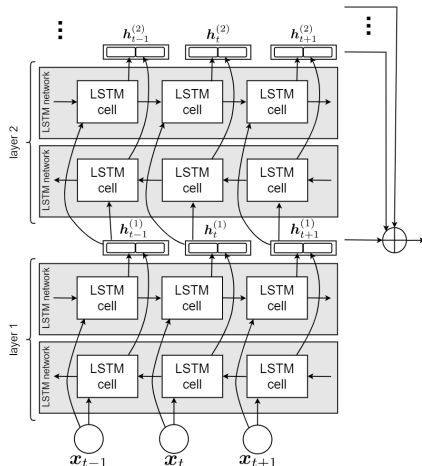
# Bidirectional LSTM

- **Bidirectional LSTM** was first proposed in (2005) [18, 19].
- As obvious from its name, the bidirectional LSTM includes **two LSTM networks each of which processes the sequence from one direction**.
- In other words, there are **two LSTM networks which are fed with the sequence in opposite orders**.
- Experiments have shown that the bidirectional LSTM outperforms the unidirectional LSTM [18, 43].



# Embeddings from Language Model (ELMo)

- The **Embeddings from Language Model (ELMo)** network, first proposed in (2018) [22], is a language model which makes use of **bidirectional LSTM networks**.
- It is one of the successful **context-aware language modeling networks**.
- ELMo contains  $L$  layers of bidirectional LSTM networks where the output of each bidirectional LSTM is fed to the next bidirectional LSTM.





# Embeddings from Language Model (ELMo)

- In the bidirectional LSTM networks of ELMo,  $\mathbf{V} = \mathbf{I}$  is set so that the output  $\mathbf{y}$  becomes equal to the hidden states  $\mathbf{h}$ .
- At time slot  $t$  and layer  $l$ , the outputs (or hidden states) of the two directions of LSTM are concatenated together to make  $\mathbf{h}_t^{(l)}$ :

$$\mathbf{h}_t^{(l)} := [\vec{\mathbf{h}}_t^{(l)\top}, \overleftarrow{\mathbf{h}}_t^{(l)\top}]^\top.$$

- Then, a linear combination of these hidden states of layers is considered to be the embedding vector of ELMo network at time  $t$  [22]:

$$\mathbf{y}_t^{\text{ELMo}} := \gamma \sum_{l=1}^L s_l \mathbf{h}_t^{(l)}, \quad (52)$$

where  $\gamma$  and  $\{s_l\}_{l=1}^L$  are the hyperparameter scalar weights which are determined according to the specific task (e.g., question answering, translation, etc) in natural language processing.

# Acknowledgment

- This slide deck is based on our tutorial paper: “Recurrent Neural Networks and Long Short-Term Memory Networks: Tutorial and Survey” [44].
- Some slides of this slide deck are inspired by teachings of Prof. Ali Ghodsi at University of Waterloo, Department of Statistics.

# References

- [1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.
- [2] A. Robinson and F. Fallside, "The utility driven dynamic error propagation network," tech. rep., Department of Engineering, University of Cambridge, 1987.
- [3] P. J. Werbos, "Generalization of backpropagation with application to a recurrent gas market model," *Neural networks*, vol. 1, no. 4, pp. 339–356, 1988.
- [4] R. J. Williams, "Complexity of exact gradient computation algorithms for recurrent neural networks," tech. rep., NU-CCS-89-27, Northeastern University, 1989.
- [5] R. J. Williams and D. Zipser, "Gradient-based learning algorithms for recurrent networks and their computational complexity," *Backpropagation: Theory, architectures, and applications*, vol. 433, p. 17, 1995.
- [6] M. C. Mozer, "A focused backpropagation algorithm for temporal pattern recognition," *Backpropagation: Theory, architectures, and applications*, vol. 137, 1995.
- [7] Y. Bengio, P. Frasconi, and P. Simard, "The problem of learning long-term dependencies in recurrent networks," in *IEEE international conference on neural networks*, pp. 1183–1188, IEEE, 1993.
- [8] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

## References (cont.)

- [9] T. Mikolov, A. Joulin, S. Chopra, M. Mathieu, and M. Ranzato, “Learning longer memory in recurrent neural networks,” *Workshop at the International Conference on Learning Representations*, 2015.
- [10] T. Lin, B. G. Horne, P. Tino, and C. L. Giles, “Learning long-term dependencies is not as difficult with narx recurrent neural networks,” *Advances in neural information processing systems*, 1995.
- [11] H. Jaeger, M. Lukoševičius, D. Popovici, and U. Siewert, “Optimization and applications of echo state networks with leaky-integrator neurons,” *Neural networks*, vol. 20, no. 3, pp. 335–352, 2007.
- [12] I. Sutskever and G. Hinton, “Temporal-kernel recurrent neural networks,” *Neural Networks*, vol. 23, no. 2, pp. 239–243, 2010.
- [13] H. Jaeger and H. Haas, “Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication,” *Science*, vol. 304, no. 5667, pp. 78–80, 2004.
- [14] H. Jaeger, “Echo state network,” *Scholarpedia*, vol. 2, no. 9, p. 2330, 2007.
- [15] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” tech. rep., FKI-207-95, Department of Fakultät für Informatik, Technical University of Munich, Munich, Germany, 1995.
- [16] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

## References (cont.)

- [17] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, “On the properties of neural machine translation: Encoder-decoder approaches,” in *Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation (SSST-8)*, 2014.
- [18] A. Graves and J. Schmidhuber, “Framewise phoneme classification with bidirectional LSTM and other neural network architectures,” *Neural networks*, vol. 18, no. 5-6, pp. 602–610, 2005.
- [19] A. Graves and J. Schmidhuber, “Framewise phoneme classification with bidirectional lstm networks,” in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 4, pp. 2047–2052, IEEE, 2005.
- [20] M. Schuster and K. K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.
- [21] P. Baldi, S. Brunak, P. Frasconi, G. Soda, and G. Pollastri, “Exploiting the past and the future in protein secondary structure prediction,” *Bioinformatics*, vol. 15, no. 11, pp. 937–946, 1999.
- [22] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, “Deep contextualized word representations,” in *North American Chapter of the Association for Computational Linguistics (NAACL)*, 2018.
- [23] H. W. Broer and F. Takens, *Dynamical systems and chaos*, vol. 172. Springer, 2011.

# References (cont.)

- [24] B. Ghojogh, A. Ghodsi, F. Karray, and M. Crowley, “KKT conditions, first-order and second-order optimization, and distributed optimization: Tutorial and survey,” *arXiv preprint arXiv:2110.01858*, 2021.
- [25] B. Ghojogh, F. Karray, and M. Crowley, “Hidden Markov model: Tutorial,” *Engineering Archive*, 2019.
- [26] S. Hochreiter, “The vanishing gradient problem during learning recurrent neural nets and problem solutions,” *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.
- [27] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, *et al.*, “Gradient flow in recurrent nets: the difficulty of learning long-term dependencies,” 2001.
- [28] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu, “Advances in optimizing recurrent networks,” in *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 8624–8628, IEEE, 2013.
- [29] B. Ghojogh, F. Karray, and M. Crowley, “Eigenvalue and generalized eigenvalue problems: Tutorial,” *arXiv preprint arXiv:1903.11240*, 2019.
- [30] M. Arjovsky, A. Shah, and Y. Bengio, “Unitary evolution recurrent neural networks,” in *International conference on machine learning*, pp. 1120–1128, PMLR, 2016.
- [31] Y. Hu, A. Huber, J. Anumula, and S.-C. Liu, “Overcoming the vanishing gradient problem in plain recurrent networks,” *arXiv preprint arXiv:1801.06105*, 2018.

# References (cont.)

- [32] H. Jaeger, “Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the “echo state network” approach,” 2002.
- [33] F. A. Gers and J. Schmidhuber, “Recurrent nets that time and count,” in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, vol. 3, pp. 189–194, IEEE, 2000.
- [34] F. A. Gers, J. Schmidhuber, and F. Cummins, “Learning to forget: Continual prediction with LSTM,” *Neural computation*, vol. 12, no. 10, pp. 2451–2471, 2000.
- [35] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, “LSTM: A search space odyssey,” *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, pp. 2222–2232, 2016.
- [36] R. Jozefowicz, W. Zaremba, and I. Sutskever, “An empirical exploration of recurrent network architectures,” in *International conference on machine learning*, pp. 2342–2350, PMLR, 2015.
- [37] F. A. Gers, J. A. Pérez-Ortiz, D. Eck, and J. Schmidhuber, “DEKF-LSTM,” in *10th European Symposium on Artificial Neural Networks (ESANN)*, 2002.
- [38] J. Schmidhuber, D. Wierstra, M. Gagliolo, and F. Gomez, “Training recurrent networks by Evolino,” *Neural computation*, vol. 19, no. 3, pp. 757–779, 2007.

# References (cont.)

- [39] J. Bayer, D. Wierstra, J. Togelius, and J. Schmidhuber, “Evolving memory cell structures for sequence learning,” in *International conference on artificial neural networks*, pp. 755–764, Springer, 2009.
- [40] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” *arXiv preprint arXiv:1412.3555*, 2014.
- [41] J. C. Heck and F. M. Salem, “Simplified minimal gated unit variations for recurrent neural networks,” in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 1593–1596, IEEE, 2017.
- [42] A. Graves, A.-r. Mohamed, and G. Hinton, “Speech recognition with deep recurrent neural networks,” in *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 6645–6649, IEEE, 2013.
- [43] A. Graves, S. Fernández, and J. Schmidhuber, “Bidirectional LSTM networks for improved phoneme classification and recognition,” in *International conference on artificial neural networks*, pp. 799–804, Springer, 2005.
- [44] B. Ghojogh and A. Ghodsi, “Recurrent neural networks and long short-term memory networks: Tutorial and survey,” *arXiv preprint arXiv:2304.11461*, 2023.