Chapter 4

Recurrent Neural Networks (RNNs): Architectures, Training Tricks, and Introduction to Influential Research

Susmita Das, Amara Tariq, Thiago Santos, Sai Sandeep Kantareddy, and Imon Banerjee

Abstract

Recurrent neural networks (RNNs) are neural network architectures with hidden state and which use feedback loops to process a sequence of data that ultimately informs the final output. Therefore, RNN models can recognize sequential characteristics in the data and help to predict the next likely data point in the data sequence. Leveraging the power of sequential data processing, RNN use cases tend to be connected to either language models or time-series data analysis. However, multiple popular RNN architectures have been introduced in the field, starting from SimpleRNN and LSTM to deep RNN, and applied in different experimental settings. In this chapter, we will present six distinct RNN architectures and will highlight the pros and cons of each model. Afterward, we will discuss real-life tips and tricks for training the RNN models. Finally, we will present four popular language modeling applications of the RNN models –text classification, summarization, machine translation, and image-to-text translation– thereby demonstrating influential research in the field.

Key words Recurrent neural network (RNN), LSTM, GRU, Bidirectional RNN (BRNN), Deep RNN, Language modeling

1 Introduction

Recurrent neural network (RNN) is a specialized neural network with feedback connection for processing sequential data or time-series data in which the output obtained is fed back into it as input along with the new input at every time step. The feedback connection allows the neural network to remember the past data when processing the next output. Such processing can be defined as a recurring process, and hence the architecture is also known as recurring neural network.

RNN concept was first proposed by Rumelhart et al. [1] in a letter published by Nature in 1986 to describe a new learning procedure with a self-organizing neural network. Another important historical moment for RNNs is the (re-)discovery of Hopfield

Olivier Colliot (ed.), *Machine Learning for Brain Disorders*, Neuromethods, vol. 197, https://doi.org/10.1007/978-1-0716-3195-9_4,

networks which is a special kind of RNN with symmetric connections where the weight from one node to another and from the latter to the former are the same (symmetric). The Hopfield network [2] is fully connected, so every neuron's output is an input to all the other neurons, and updating of nodes happens in a binary way (0/1). These types of networks were specifically designed to simulate the human memory.

The other types of RNNs are input-output mapping networks, which are used for classification and prediction of sequential data. In 1993, Schmidhuber et al. [3] demonstrated credit assignment across the equivalent of 1,200 layers in an unfolded RNN and revolutionized sequential modeling. In 1997, one of the most popular RNN architectures, the long short-term memory (LSTM) network which can process long sequences, was proposed.

In this chapter, we summarize the six most popular contemporary RNN architectures and their variations and highlight the pros and cons of each. We also discuss real-life tips and tricks for training the RNN models, including various skip connections and gradient clipping. Finally, we highlight four popular language modeling applications of the RNN models –text classification, summarization, machine translation, and image-to-text translation– thereby demonstrating influential research in each area.

2 Popular RNN Architectures

In addition to the SimpleRNN architecture, many variations were proposed to address different use cases. In this section, we will unwrap some of the popular RNN architectures like LSTM, GRU, bidirectional RNN, deep RNN, and attention models and discuss their pros and cons.

2.1 SimpleRNN

SimpleRNN architecture, which is also known as SimpleRNN, contains a simple neural network with a feedback connection. It has the capability to process sequential data of variable length due to the parameter sharing which generalizes the model to process sequences of variable length. Unlike feedforward neural networks which have separate weights for each input feature, RNN shares the same weights across several time steps. In RNN, the output of a present time step depends on the previous time steps and is obtained by the same update rule which is used to obtain the previous outputs. As we will see, the RNN can be unfolded into a deep computational graph in which the weights are shared across time steps.

The RNN operating on an input sequence $\boldsymbol{x}^{(t)}$ with a time step index t ranging from 1 to τ is illustrated in Fig. 1. The time step index t may not necessarily refer to the passage of time in the real world; it can refer to the position in the sequence. The cycles in the

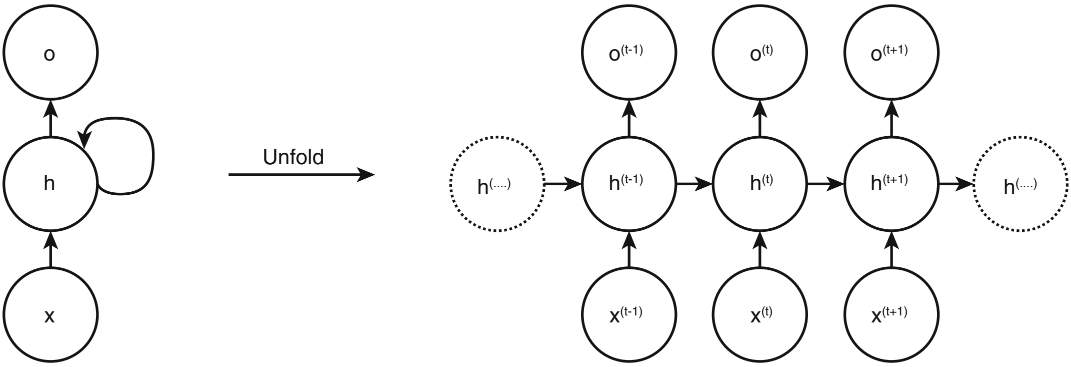

Fig. 1 (Left) Circuit diagram for SimpleRNN with input **x** being incorporated into hidden state **h** with a feedback connection and an output **o**. (Right) The same SimpleRNN network shown as an unfolded computational graph with nodes at every time step

computational graph represent the impact of the past value of a variable on the present time step. The computational graph has a repetitive structure that unfolds the recursive computation of the RNN which corresponds to a chain of events. It shows the flow of the information, forward in the time of computing the outputs and losses and backward when computing the gradients. The unfolded computational graph is shown in Fig. 1. The equation corresponding to the computational graph is $\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{W})$, where $\boldsymbol{h}$ is the hidden state of the network, $\boldsymbol{x}$ is the input, t is the time step, and $\boldsymbol{W}$ denotes the weights of the network connections comprising of input-to-hidden, hidden-to-hidden, and hidden-to-output connection weights.

2.1.1 Training Fundamentals

Training is performed by gradient computation of the loss function with respect to the parameters involved in forward propagation from left to right of the unrolled graph followed by back-propagation moving from right to left through the graph. Such gradient computation is an expensive operation as the runtime cannot be reduced by parallelism because the forward propagation is sequential in nature. The states computed in the forward pass are stored until they are reused in the back-propagation. The back-propagation algorithm applied to RNN is known as **back-propagation through time** (BPTT) [4].

The following computational operations are performed in RNN during the forward propagation to calculate the output and the loss.

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= b + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= c + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \sigma(\boldsymbol{o}^{(t)})
\end{aligned}
$$

where b and c are the biases and $\boldsymbol{U}$, $\boldsymbol{V}$, and $\boldsymbol{W}$ are the weight matrix for input-to-hidden connections, hidden-to-output connection, and hidden-to-hidden connections respectively, and σ is a sigmoid function. The total loss for a sequence of $\boldsymbol{x}$ values and its corresponding $\boldsymbol{y}$ values is obtained by summing up the losses over all time steps.

$$\sum_{t=1}^{\tau} L^{(t)} = L\Big(\big(\boldsymbol{x}^{(1)}, \ldots\ldots, \boldsymbol{x}^{(\tau)}\big), \big(\boldsymbol{y}^{(1)}, \ldots\ldots, \boldsymbol{y}^{(\tau)}\big)\Big)$$

To minimize the loss, the gradient of the loss function is calculated with respect to the parameters associated with it. The parameters associated with the nodes of the computational graph are $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$, b, c, $\boldsymbol{x}^{(t)}$, $\boldsymbol{h}^{(t)}$, $\boldsymbol{o}^{(t)}$, and $L^{(t)}$. The output $\boldsymbol{o}^{(t)}$ is the argument to the softmax to obtain the vector $\hat{\boldsymbol{y}}$ of probabilities over the output. During back-propagation, the gradient for each node is calculated recursively starting with the nodes preceding the final loss. It is then iterated backward in time to back-propagate gradients through time. *tanh* is a popular choice for activation function as it tends to avoid vanishing gradient problem by retaining non-zero value longer through the back-propagation process.

2.1.2 SimpleRNN Architecture Variations Based on Parameter Sharing

Variations of SimpleRNN can be designed depending upon the style of graph unrolling and parameter sharing [5]:

- *Connection between hidden units.* The RNN produces outputs at every time step, and the parameters are passed between hidden-to-hidden units (Fig. 2a). This corresponds to the standard SimpleRNN presented above and is widely used.
- *Connection between outputs to hidden units.* The RNN produces outputs at every time step, and the parameters are passed from an output at a particular time step to the hidden unit at the next time step (Fig. 2b).
- *Sequential input to single output.* The RNN produces a single output at the end after reading the entire sequence and has connections between the hidden units at every time step (Fig. 2c).

2.1.3 SimpleRNN Architecture Variations Based on Inputs and Outputs

Different variations also exist depending on the number of inputs and outputs:

- *One-to-one:* The traditional RNN has one-to-one input to output mapping at each time step t as shown in Fig. 3a.
- *One-to-many:* One-to-many RNN has one input at a time step for which it generates a sequence of outputs at consecutive time steps as shown in Fig. 3b. This type of RNN architecture is often used for image captioning.

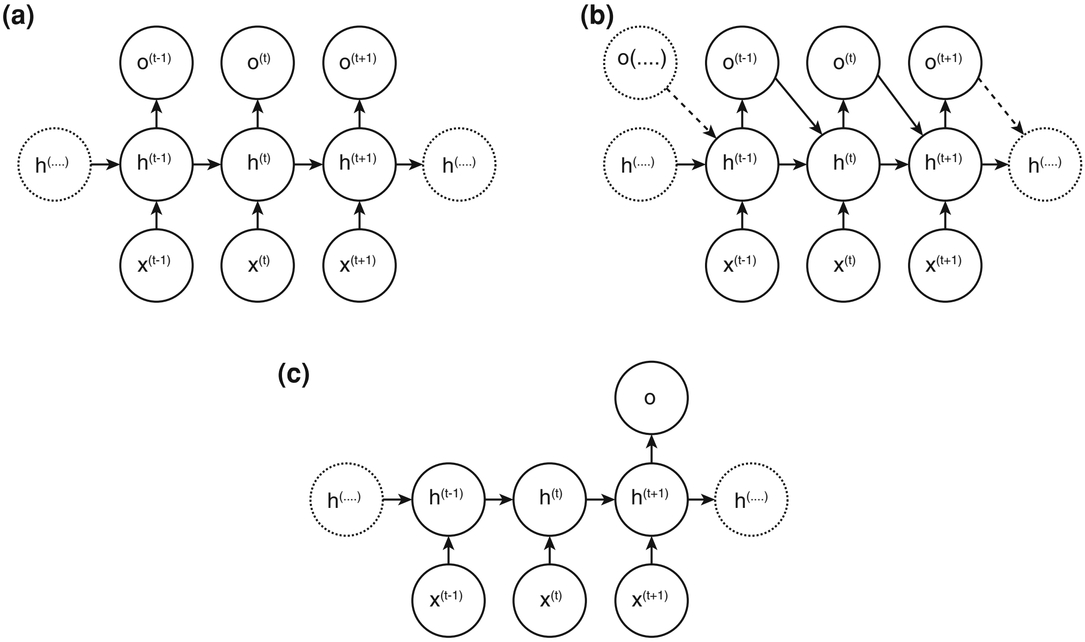

Fig. 2 Types of SimpleRNN architectures based on parameter sharing: (**a**) SimpleRNN with connections between hidden units, (**b**) SimpleRNN with connections from output to hidden units, and (**c**) SimpleRNN with connections between hidden units that read the entire sequence and produce a single output

- *Many-to-one:* Many-to-one RNN has many inputs and one output, at each time step as shown in Fig. 3c. This type of RNN architecture is used for text classification.
- *Many-to-many:* Many-to-many RNN architecture can be designed in two ways. First, the input is taken by the RNN and the corresponding output is given at the same time step as illustrated in Fig. 3d. This type of RNN is used for named entity recognition. Second, the input is taken by the RNN at each time step and the output is given by the RNN at the next time step depending upon all the input sequence as illustrated in Fig. 3e. Popular uses of this type of RNN architecture are in machine translation.

2.1.4 Challenges of Long-Term Dependencies in SimpleRNN

SimpleRNN works well with the short-term dependencies, but when it comes to long-term dependencies, it fails to remember the long-term information. This problem arises due to the vanishing gradient or exploding gradient [6]. When the gradients are propagated over many stages, it tends to vanish most of the times or sometimes explodes. The difficulty arises due to the exponentially smaller weight assigned to the long-term interactions compared to the short-term interactions. It takes very long time to learn the long-term dependencies as the signals from these dependencies tend to be hidden by the small fluctuations arising from the short-term dependencies.

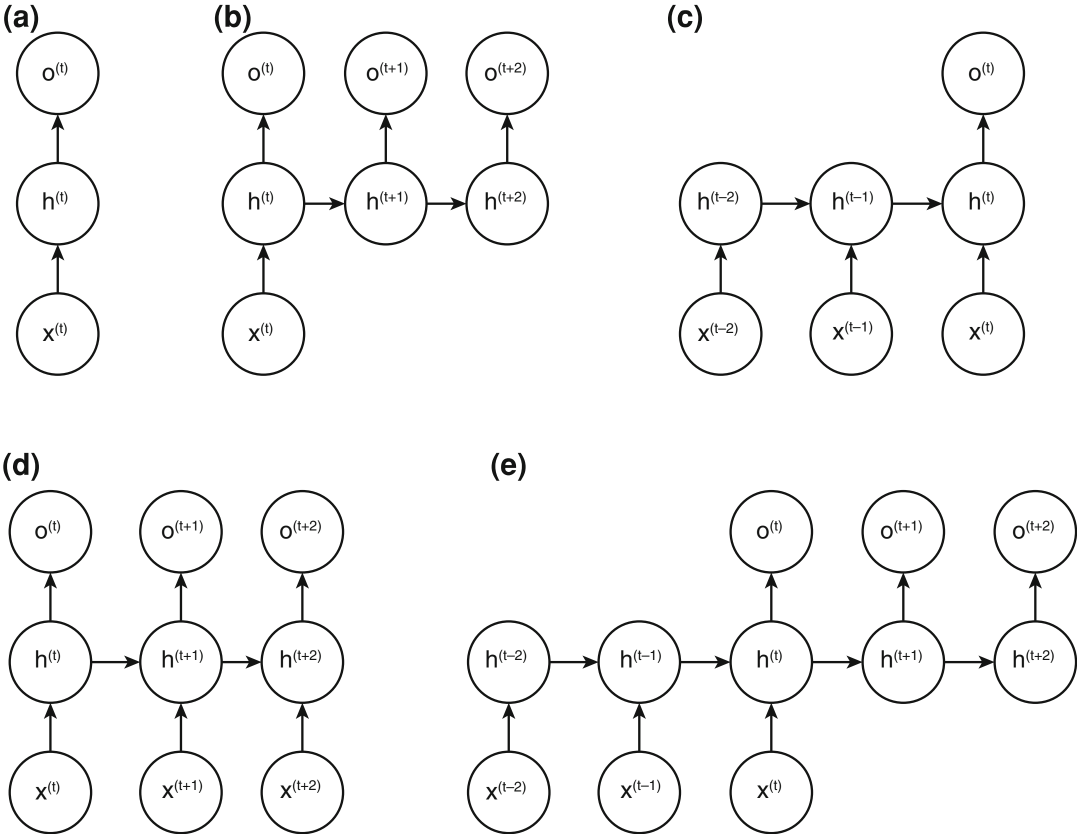

Fig. 3 (**a**) One-to-one RNN. (**b**) One-to-many RNN. (**c**) Many-to-one RNN. (**d**) Many-to-many RNN. (**e**) Many-to-many RNN. **x** represents the input and **o** represents the output

2.2 Long Short-Term Memory (LSTM)

To address this long-term dependency problem, gated RNNs were proposed. Long short-term memory (LSTM) is a type of gated RNN which was proposed in 1997 [7]. Due to the property of remembering the long-term dependencies, LSTM has been a successful model in many applications like speech recognition, machine translation, image captioning, etc. LSTM has an inner self loop in addition to the outer recurrence of the RNN. The gradients in the inner loop can flow for longer duration and are conditioned on the context rather than being fixed. In each cell, the input and output is the same as that of ordinary RNN but has a system of gating units to control the flow of information. Figure 4 shows the flow of the information in LSTM with its gating units.

There are three gates in the LSTM—the **external input gate**, the **forget gate**, and the **output gate**. The **forget gate** at time t and state s_i ($f_i^{(t)}$) decides which information should be removed from the cell state. The gate controls the self loop by setting the weight between 0 and 1 via a sigmoid function σ. When the value is near to 1, the information of the past is retained, and if the value is near to

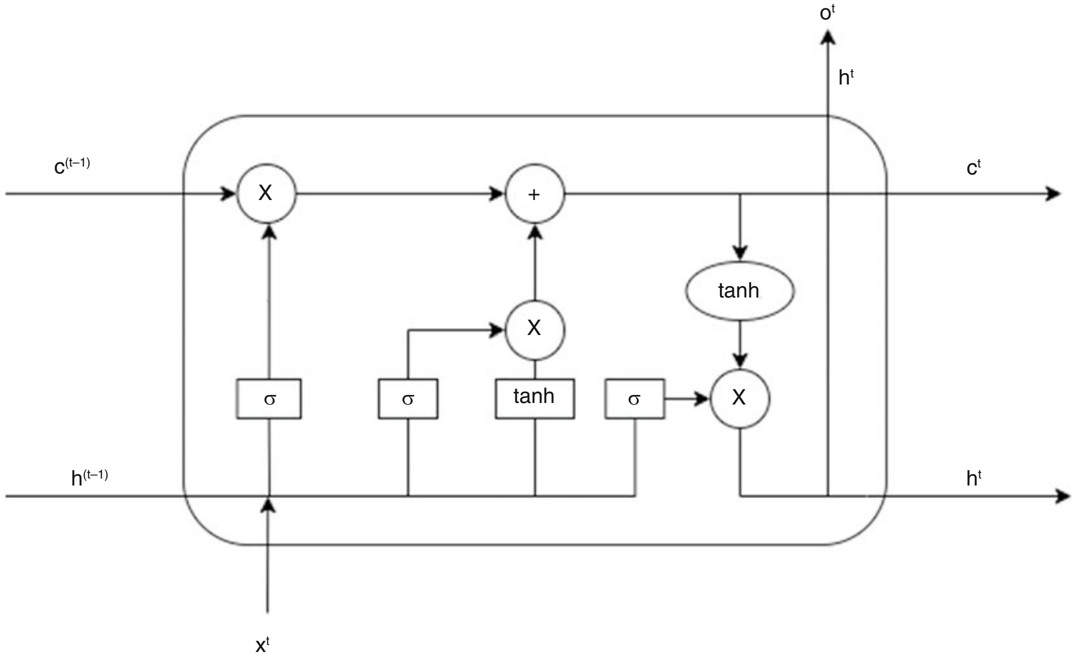

Fig. 4 Long short-term memory with cell state $\boldsymbol{c}^t$, hidden state $\boldsymbol{h}^t$, input $\boldsymbol{x}^t$, and output $\boldsymbol{o}^t$

0, the information is discarded. After the **forget gate**, the internal state $s_i^{(t)}$ is updated. Computation for **external input gate** (g_i^t) is similar to that of **forget gate** with a sigmoid function to obtain a value between 0 and 1 but with its own parameters. The **output gate** of the LSTM also has a sigmoid unit which determines whether to output the value or to shut off the value $\boldsymbol{h}_i^t$ via the **output gate** q_i^t.

$$f_i^{(t)} = \sigma\left(\sum_j \boldsymbol{U}^f_{i,j}\boldsymbol{x}_j^t + \sum_j \boldsymbol{W}^f_{i,j}\boldsymbol{h}_j^{(t-1)} + b_i^f\right)$$

$$\boldsymbol{s}_i^{(t)} = f_i^t \boldsymbol{s}_i^{(t-1)} + g_i^t \sigma\left(b_i + \sum_j \boldsymbol{U}_{i,j}\boldsymbol{x}_j^t + \sum_j \boldsymbol{W}_{i,j}\boldsymbol{h}_j^{(t-1)}\right)$$

$$g_i^t = \sigma\left(b_i^g + \sum_j \boldsymbol{U}^g_{i,j}\boldsymbol{x}_j^t + \sum_j \boldsymbol{W}^g_{i,j}\boldsymbol{h}_j^{(t-1)}\right)$$

$$\boldsymbol{h}_i^t = tanh(\boldsymbol{s}_i^t)q_i^t$$

$$q_i^t = \sigma\left(b_i^o + \sum_j \boldsymbol{U}^o_{i,j}\boldsymbol{x}_i^t + \sum \boldsymbol{W}^o_{i,j}\boldsymbol{h}_j^{(t-1)}\right)$$

$\boldsymbol{x}^t$ is the input vector at time t, $\boldsymbol{h}^{(t)}$ is the hidden layer vector, b_i denote the biases, and $\boldsymbol{U}_i$ and $\boldsymbol{W}_i$ represent the input weights and the recurrent weights, respectively.

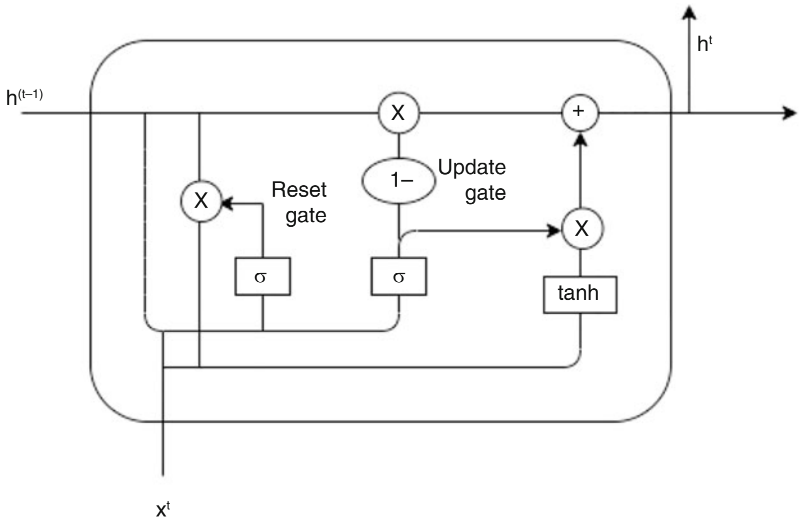

Fig. 5 Gated recurrent neural network (GRU) with input x^t and hidden unit h^t

2.3 Gated Recurrent Unit (GRU)

In LSTM, the computation time is large as there are a lot of parameters involved during back-propagation. To reduce the computation time, gated recurrent unit (GRU) was proposed in the year 2014 by Cho et al. with less gates than in LSTM [8]. The functionality of the GRU is similar to that of LSTM but with a modified architecture. The representation diagram for GRU can be found in Fig. 5. Like LSTM, GRU also solves the vanishing and exploding gradient problem by capturing the long-term dependencies with the help of gating units. There are two gates in GRU, the **reset gate** and the **update gate**. The **reset gate** determines how much of the past information it needs to forget, and the **update gate** determines how much of the past information it needs to carry forward.

The computation at the **reset gate** (r_i^t) and the **update gate** (u_i^t), as well as hidden state (h_i^t) and the at time t, can be represented by the following:

$$r_i^{(t)} = \sigma\Big(b_i^r + \sum_j \boldsymbol{U}_{i,j}^r \boldsymbol{x}_j^{(t)} + \sum_j \boldsymbol{W}_{i,j}^r \boldsymbol{h}_j^{(t)}\Big)$$

$$u_i^{(t)} = \sigma\Big(b_i^u + \sum_j \mathbf{U}_{i,j}^u \boldsymbol{x}_j^{(t)} + \sum_j \boldsymbol{W}_{i,j}^u \boldsymbol{h}_j^{(t)}\Big)$$

$$\begin{aligned} \boldsymbol{h}_i^{(t)} &= u_i^{(t-1)} \boldsymbol{h}_i^{(t-1)} + (1 - u_i) \\ &\quad \times \ \sigma\Big(b_i + \sum_j \boldsymbol{U}_{i,j} \boldsymbol{x}_j^{(t-1)} + \sum_j \boldsymbol{W}_{i,j} r_j^{(t-1)} \boldsymbol{h}_j^{(t-1)}\Big) \end{aligned}$$

where b_i denotes biases and $\boldsymbol{U}_i$ and $\boldsymbol{W}_i$ denote initial and recurrent weights, respectively.

When the **reset gate** value is close to 0, the previous hidden state value is discarded and reset with the present value. This enables the hidden state to forget the past information that is irrelevant for future. The **update gate** determines how much of the relevant past information to carry forward for future.

The property of the **update gate** to carry forward the past information allows it to remember the long-term dependencies. For short-term dependencies, the **reset gate** will be frequently active to reset with current values and remove the previous ones, while, for long-term dependencies, the **update gate** will be often active for carrying forward the previous information.

2.3.1 Advantage of LSTM and GRU over SimpleRNN

The LSTM and GRU can handle the vanishing gradient issue of SimpleRNN with the help of gating units. The LSTM and GRU have the additive feature that they retain the past information by adding the relevant past information to the present state. This additive property makes it possible to remember a specific feature in the input for longer time. In SimpleRNN, the past information loses its relevance when new input is seen. In LSTM and GRU, any important feature is not overwritten by new information. Instead, it is added along with the new information.

2.3.2 Differences Between LSTM and GRU

There are a few differences between LSTM and GRU in terms of gating mechanism which in turn result in differences observed in the content generated. In LSTM unit, the amount of the memory content to be used by other units of the network is regulated by the **output gate**, whereas in GRU, the full content that is generated is exposed to other units. Another difference is that the LSTM computes the new memory content without controlling the amount of previous state information flowing. Instead, it controls the new memory content that is to be added to the network. On the other hand, the GRU controls the flow of the past information when computing the new candidate without controlling the candidate activation.

2.4 Bidirectional RNN (BRNN)

In SimpleRNN, the output of a state at time t only depends on the information of the past $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t-1)}$ and the present input $\boldsymbol{x}^{(t)}$. However, for many sequence-to-sequence applications, the present state output depends on the whole sequence information. For example, in language translation, the correct interpretation of the current word depends on the past words as well as the next words. To overcome this limitation of SimpleRNN, bidirectional RNN (BRNN) was proposed by Schuster and Paliwal in the year 1997 [9].

Bidirectional RNNs combine an RNN which moves forward with time, beginning from the start of the sequence, with another RNN that moves backward through time, beginning from the end of the sequence. Figure 6 illustrates a bidirectional RNN with $\boldsymbol{h}^{(t)}$

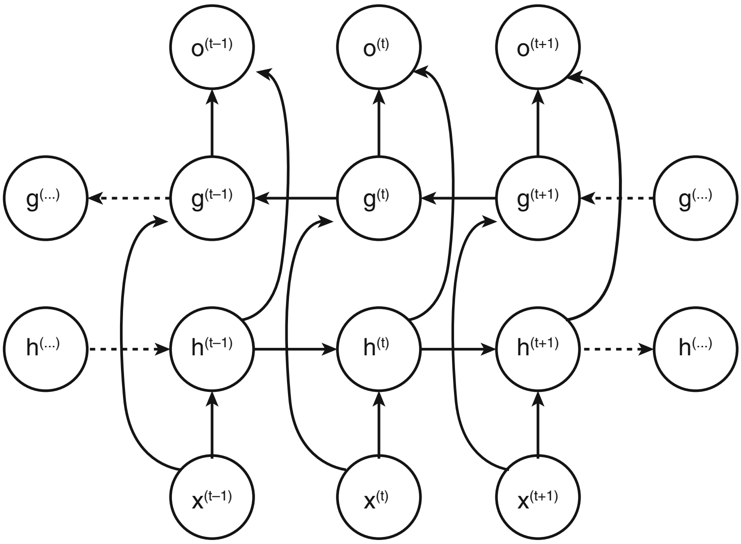

Fig. 6 Bidirectional RNN with forward sub-RNN having h^t hidden state and backward sub-RNN having g^t hidden state

the state of the sub-RNN that moves forward through time and $g^{(t)}$ the state of the sub-RNN that moves backward with time. The output of the sub-RNN that moves forward is not connected to the inputs of sub-RNN that moves backward and vice versa. The output $o^{(t)}$ depends on both past and future sequence data but is sensitive to the input values around t.

2.5 Deep RNN

Deep models are more efficient than their shallow counterparts, and, with the same hypothesis, deep RNN was proposed by Pascanu et al. in 2014 [10]. In "shallow" RNN, there are generally three blocks for computation of parameters: the input state, the hidden state, and the output state. These blocks are associated with a single weight matrix corresponding to a shallow transformation which can be represented by a single-layer multilayer perceptron (MLP). In deep RNN, the state of the RNN can be decomposed into multiple layers. Figure 7 shows in general a deep RNN with multiple deep MLPs. However, different types of depth in an RNN can be considered separately like input-to-hidden, hidden-to-hidden, and hidden-to-output layer. The lower layer in the hierarchy can transform the input into an appropriate representation for higher levels of hidden state. In hidden-to-hidden state, it can be constructed with a previous hidden state and a new input. This introduces additional non-linearity in the architecture which becomes easier to quickly adapt changing modes of the input. By introducing deep MLP in hidden-to-output state makes the layer compact which helps in summarizing the previous inputs and helps in predicting the output easily. Due to the deep MLP in the RNN architecture, the learning becomes slow and optimization is difficult.

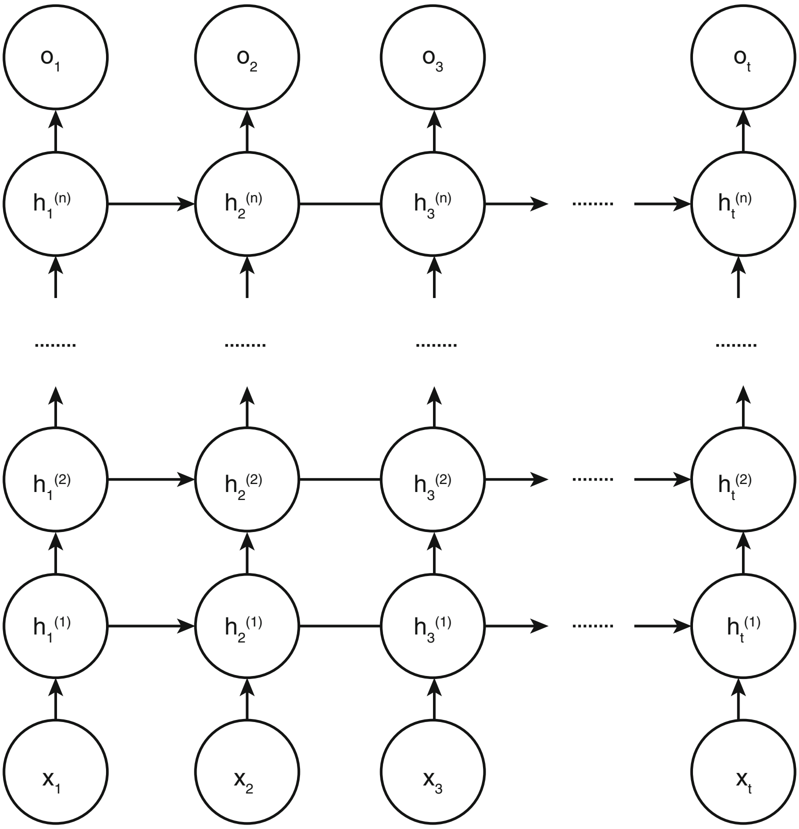

Fig. 7 Deep recurrent neural network

2.6 Encoder–Decoder

Encoder–decoder architecture was proposed by Cho et al. (2014) [8] to map a variable length input sequence to a variable length output sequence. Therefore, it is also known as sequence-to-sequence architecture. Before encoder–decoder was introduced, there were RNN models which were used for sequence-to-sequence applications, but they had limitations as the input and output sequences had to have the same length. Encoder–decoder was used for addressing variable length sequence-to-sequence problems such as machine translation or speech recognition where the input sequence and output sequence lengths may not be the same in most of the cases. Encoder and decoder are both RNNs where the encoder RNN encodes the whole input $\boldsymbol{X} = \boldsymbol{x}^{(1)}, \ldots.., \boldsymbol{x}^{(n_x)}$ into a context vector $\boldsymbol{c}$ and outputs the context vector $\boldsymbol{c}$ which is fed as an input to the decoder RNN. The decoder RNN generates an output sequence $\boldsymbol{Y} = \boldsymbol{y}^{(1)}, \ldots.., \boldsymbol{y}^{(n_y)}$. In the encoder–decoder model, the input length $\boldsymbol{x}^{(n_x)}$ and the output length $\boldsymbol{y}^{(n_y)}$ can be different unlike the previous RNN models. The number of hidden layers in encoder and decoder are not necessarily be the same. The limitation of this architecture is that it fails to properly summarize a long sequence if the context vector is too small. This problem was solved by Bahdanau et al. (2015) [11] by making the context vector a variable length sequence with added attention mechanism.

2.7 Attention Models (Transformers)

Due to the sequential learning mechanism, the context vector generated by the encoder (*see* Subheading 2.6) is more focused on the later part of the sequence than on the earlier part. An extension to the encoder–decoder model was proposed by Bahdanau et al. [11] for machine translation where the model generates each word based on the most relevant information in the source sentence and previously generated words. Unlike the previous encoder–decoder model where the whole input sequence is encoded into a single context vector, this extended encoder–decoder model learns to give attention to the relevant words present in the source sequence regardless of the position in the sequence by encoding the input sequence into sequences of vectors and chooses selectively while decoding each word. This mechanism of paying attention to the relevant information that are related to each word is known as attention mechanism.

Although this model solves the problem for fixed-length context vectors, the sequential decoding problem still persists. To decode the sequence in less time by introducing parallelism, self-attention was proposed by Google Brain team, Ashish Vaswani et al. [12]. They invented the Transformer model which is based on self-attention mechanism and was designed to reduce the computation time. It computes the representation of a sequence that relates to different positions of the same sequence. The self-attention mechanism was embedded in the Transformer model. The Transformer model has a stack of six identical layers each for encoding the sequence and decoding the sequence as illustrated in Fig. 8. Each layer of the encoder and decoder has sub-layers comprising multi-head self-attention mechanisms and position-wise fully connected layers. There is a residual connection around the two sub-layers followed by normalization. In addition to the two sub-layers, there is a third layer in the decoder that performs multi-head attention over the output of the encoder stack. In the decoder, the multi-head attention is masked to prevent the position from attending the later part of the sequence. This ensures that the prediction for a position p depends only on the positions less than p in the sequence. The attention function can be described as mapping a query and key-value pairs to an output. All the parameters involved in the computation are all vectors. To calculate the output, scalar dot product operation is performed on the query and all keys, and divide each key by $\sqrt{d_k}$ (where d_k is the dimension on the keys). Finally, the softmax is applied to it to obtain the weights on the values. The computation of attention function can be represented by the following equation:

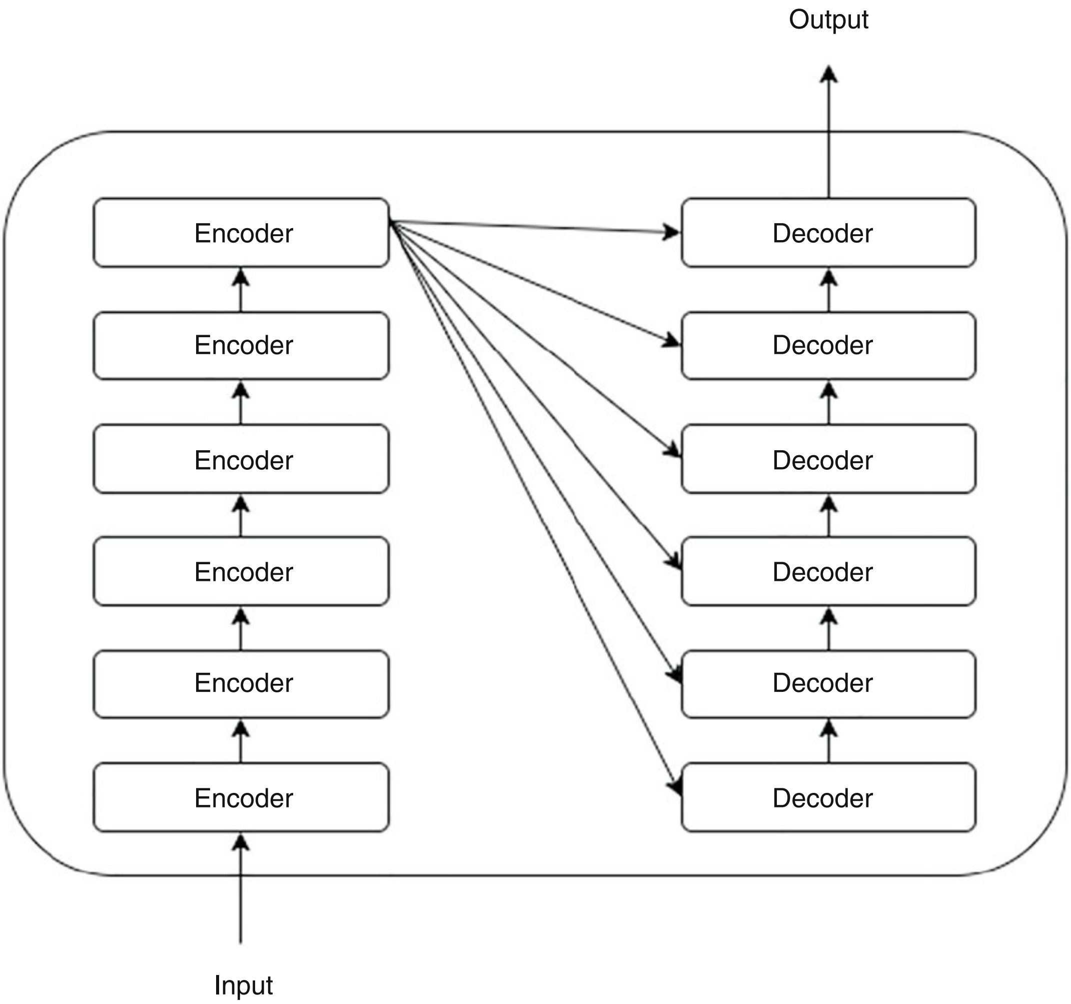

Fig. 8 Transformer with six layers of encoders and six layers of decoders

$Attention(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)\boldsymbol{V}$, where $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$are all matrices corresponding to query, keys, and values, respectively. A more in-depth coverage of Transformers is provided in Chap. 6.

3 Tips and Tricks for RNN Training

As previously stated, the vanishing gradient and exploding gradient problems are well-known concerns when it comes to properly training RNN models [13, 14]. The fundamental challenge arises from the fact that RNNs can be naturally unfolded, allowing their recurrent connections to perform feedforward calculations, which result in an RNN with the same number of layers as the number of elements in the sequence. Two major issues arise as a result:

- *Gradient vanishing problem*. It becomes difficult to effectively learn long-term dependencies in sequences due to the gradient vanishing problem [6]. As a result, a prospective model prediction will be essentially unaffected by earlier layers.
- *Exploding gradient problem*. Adding more layers to the network amplifies the effect of large gradients, increasing the risk of a learning derailment since significant changes to the network weights can be performed at each step, potentially causing the gradients to blow out exponentially. In fact, weights that are closer to the input layer will obtain larger updates than weights that are closer to the output layer, and the network may become unable to learn correlations between temporally distant events.

To overcome these limitations, we need to create solutions so that the RNN model can work on various time scales, with some sections operating on fine-grained time scales and handling small details and others operating on coarse time scales and efficiently transferring information from the distant past to the present. In this section, we discuss several popular strategies to tackle these issues.

3.1 Skip Connection

The practice of skipping layers effectively simplifies the network by using fewer direct connected layers in the initial training stages. This speeds learning by reducing the impact of vanishing gradients, as there are fewer layers to propagate through. As the network learns the feature space during the training phase, it gradually restores the skipped layers. Lin et al. [15] proposed the use of such skip connections, which follows from the idea of incorporating delays in feedforward neural networks from Lang et al. [16]. Conceptually, skip connections are a standard module in deep architectures and are commonly referred to as residual networks, as described by He et al. [17]. They are responsible to skip layers in the neural network and feeding the output of one layer as the input to the next layers. This technique is used to allow gradients to flow through a network directly, without passing through non-linear activation functions, and it has been empirically proven that these additional steps are often beneficial for the model convergence [17]. Skip connections can be used through the non-sequential layer in two fundamental ways in neural networks:

- **Additive Skip Connections.** In this type of design, the data from early layers is transported to deeper layers via matrix addition, causing back-propagation to be done via addition (Fig. 9b). This procedure does not require any additional parameters because the output from the previous layer is added to the layer ahead. One of the most common techniques used in this type of architecture is to stack the skip residual blocks together and use an identity function to preserve the gradient [18]. The core concept is to use a vector addition to back-

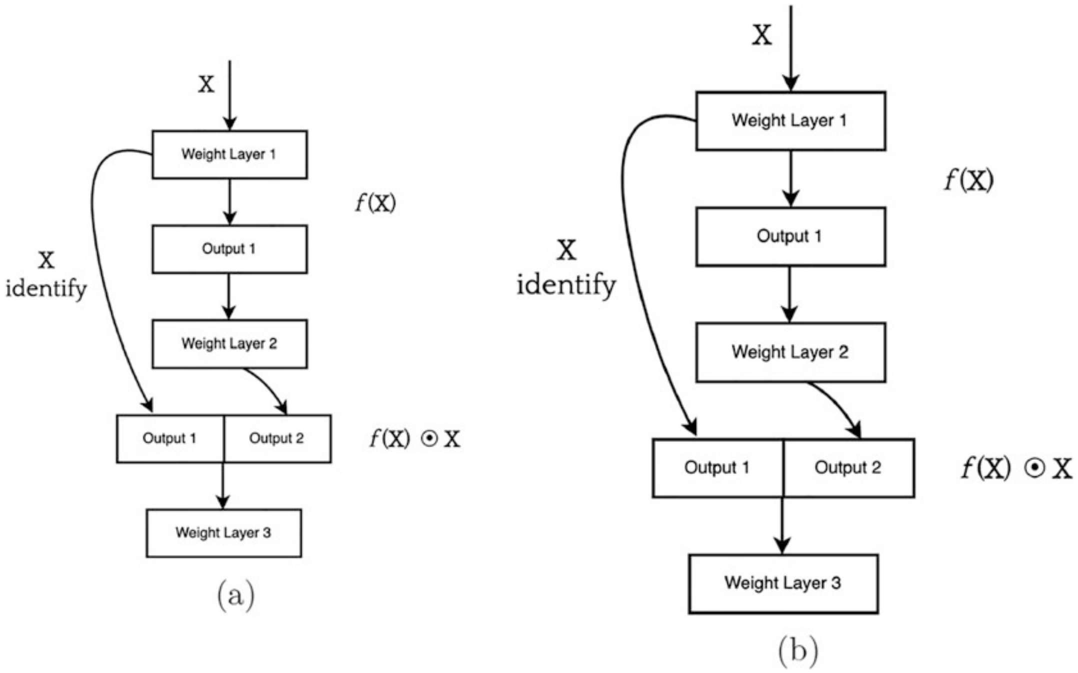

Fig. 9 Skip connection residual architectures: (**a**) concatenate output of previous layer and skip connection; (**b**) sum of the output of previous layer and skip connection

propagate through the identity function. The gradient is then simply multiplied by one, and its value is preserved in the earlier layers.

- **Concatenative Skip Connections.** Another way for establishing skip connections is to concatenate previous feature maps. The aim of concatenation is to leverage characteristics acquired in prior layers to deeper layers. In addition, concatenating skip connections provides an alternate strategy for assuring feature reusability of the same dimensionality from prior layers without the need to learn duplicate maps. Figure 9(a) illustrates a diagram example of how the architecture looks like. The primary concept of the architecture is to allow subsequent layers to reuse intermediary representations, allowing them to maintain more information and enhance long-term dependency performance.

3.2 Leaky Units

One of the major challenges when training RNNs is capturing long-term dependencies and efficiently transferring information from distant past to present. An effective method to obtain coarse time scales is to employ leaky units [19], which are hidden units with linear self-connections and a weight on the connections that is close to one. In a leaky RNN, hidden units are able to access values from prior states and can be utilized to obtain temporal representations. Formula $h_t = \alpha * h_{t-1} + (1 - \alpha) * h_t$ expresses the state update rule of a leaky unit, where $\alpha \in (0, 1)$ is an example of a linear self-connection from $\boldsymbol{h}_{t-1}$ to $\boldsymbol{h}_t$, and it is a parameter to be learned during the training stage. Essentially, α controls the information flow in the state. When α is near one, the state is almost unchanged, and information about the past is retained for a long time, and when α is close to zero, the information about the past is rapidly discarded, and the state is largely replaced by a new state $\boldsymbol{h}_t$.

3.3 Clipping Gradients

Gradient clipping is a technique that tries to overcome the exploding gradient problem in RNN training, by constraining gradient norms (element-wise) to a predetermined minimum or maximum threshold value since the exploding gradients are clipped and the optimization begins to converge to the minimum point. Gradient clipping can be used in two fundamental ways:

- **Clipping-by-value.** Using this technique, we define a minimum clip value and a maximum clip value. If a gradient exceeds the threshold value, we clip the gradient to the maximum threshold. If the gradient is less than the lower limit of the threshold, we clip the gradient to the minimum threshold.
- **Clipping-by-norm.** The idea behind this technique is very similar to clipping-by-value. The key difference is that we clip the gradients by multiplying the unit vector of the gradients with the threshold. Gradient descent will be able to behave properly even if the loss landscape of the model is irregular since the weight updates will also be rescaled. This significantly reduces the likelihood of an overflow or underflow of the model.

4 RNN Applications in Language Modeling

Language modeling is the process of learning meaningful vector representations for language or text using sequence information and is generally trained to predict the next token or word given the input sequence of tokens or words. Bengio et al. [20] proposed a framework for neural network-based language modeling. RNN architecture is particularly suited to processing free-flowing natural language due to its sequential nature. As described by Mikolov et al. [21], RNNs can learn to compress a whole sequence as opposed to feedforward neural networks that compress only a single input item. Language modeling can be an independent task or be part of a language processing pipeline with downstream prediction or classification task. In this section, we will discuss applications of RNN for various language processing tasks.

4.1 Text Classification

Many interesting real-world applications concerning language data can be modeled as text classification. Examples include sentiment classification, topic or author identification, and spam detection with applications ranging from marketing to query-answering [22, 23]. In general, models for text classification include some RNN layers to process sequential input text [22, 23]. The embedding of the input learnt by these layers is later processed through varying classification layers to predict the final class label. Many-to-one RNN architectures are often employed for text classification.

As a recent technical innovation, RNNs have been combined with convolutional neural networks (CNNs), thus combining the strengths of two architectures, to process textual data for classification tasks. LSTMs are popular RNN architecture for processing textual data because of their ability to track patterns over long sequences, while CNNs have the ability to learn spatial patterns from data with two or more dimensions. Convolutional LSTM (C-LSTM) combines these two architectures to form a powerful architecture that can learn local phrase-level patterns as well as global sentence-level patterns [24]. While CNN can learn local and position-invariant features and RNN is good at learning global patterns, another variation of RNN has been proposed to introduce position-invariant local feature learning into RNN. This variation is called disconnected RNN (DRNN) [25]. Information flow between tokens/words at the hidden layer is limited by a hyperparameter called *window size*, allowing the developer to choose the *width* of the *context* to be considered while processing text. This architecture has shown better performance than both RNN and CNN on several text classification tasks [25].

4.2 Text Summarization

Text summarization approaches can be broadly categorized into (1) extractive and (2) abstractive summarization. The first approach relies on selection or extraction of sentences that will be part of the summary, while the latter generates new text to build a summary. RNN architectures have been used for both types of summarization techniques.

4.2.1 Extractive Text Summarization

Extractive summarization frameworks use many-to-one RNN as a classifier to distinguish sentences that should be part of the summary. For example, a two-layer RNN architecture is presented in [26] where one layer processes words in one sentence and the other layer processes many sentences as a sequence. The model generates sentence-level labels indicating whether the sentence should be part of the summary or not, thus producing an extractive summary of the input document. Xu et al. have presented a more sophisticated extractive summarization model that not only extracts sentences to be part of the summary but also proposes possible syntactic compressions for those sentences [27]. Their proposed architecture is a combination of CNN and bidirectional LSTM, while a neural classifier evaluates possible syntactic compressions in the context of the sentence as well as the broader context of the document.

4.2.2 Abstractive Text Summarization

Abstractive summarization frameworks expect the RNN to process input text and generate a new sequence of text that is the summary of input text, effectively using many-to-many RNN as a text generation model. While it is relatively straightforward for extractive summarizers to achieve basic grammatical correctness as correct sentences are picked from the document to generate a summary, it has been a major challenge for abstractive summarizers. Grammatical correctness depends on the quality of the text generation module. Grammatical correctness of abstractive text summarizers has improved recently due to developments in contextual text processing, language modeling, as well as availability of computational power to process large amounts of text.

Handling of rare tokens/words is a major concern for modern abstractive summarizers. For example, proper nouns such as specific names of people and places occur less frequently in the text; however, generated summaries are incomplete and incomprehensible if such tokens are ignored. Nallapati et al. proposed a novel solution composed of GRU-RNN layers with attention mechanism by including switching decoder in their abstractive summarizer architecture [28] where the text generator module has a switch which can enable the module to choose between two options: (1) generate a word from the vocabulary and (2) point to one of the words in the input text. Their model is capable of handling rare tokens by pointing to their position in the original text. They also employed *large vocabulary trick* which limits the vocabulary of the generator module to tokens of the source text only and then adds frequent tokens to the vocabulary set until its size reaches a certain threshold. This trick is useful in limiting the size of the network.

Summaries have latent structural information, i.e., they convey information following certain linguistic structures such as "What-Happended" or "Who-Action-What." Li et al. presented a recurrent generative decoder based on variational auto-encoder (VAE) [29]. VAE is a generative model that takes into account latent variables, but is not inherently sequential in nature. With the historical dependencies in latent space, it can be transformed into a sequential model where generative output is taking into account history of latent variables, hence producing a summary following latent structures.

4.3 Machine Translation

Neural machine translation (NMT) models are trained to process input sequence of text and generate an output sequence which is the translation of the input sequence in another language. As mentioned in Subheading 2.6, machine translation is a classic example of conversion of one sequence to another using encoder–

decoder architecture where lengths of both sequences may be different. In 2014, many-to-many RNN-based encoder–decoder architecture was proposed where one RNN encodes the input sequence of text to a fixed-length vector representation, while another RNN decodes the fixed-length vector to the target translated sequence [30]. Both RNNs are jointly trained to maximize the conditional probability of the target sequence given the input sequence. Later, attention-based modeling was added to vanilla encoder–decoder architecture for machine translation. Luong et al. discussed two types of attention mechanism in their work on NMT: (i) global and (ii) local attention [31]. In global attention, a global context vector is estimated by learning variable length alignment and attention scores for all source words. In local attention, the model predicts a single aligned position for the current target word and then computes a local context vector from attention predicted for source words within a small window of the aligned position. Their experiments show significant improvement in translation performance over models without attention. Local attention mechanism has the advantage of being computationally less expensive than global attention mechanism.

4.4 Image-to-Text Translation

Image-to-text translation models are expected to convert visual data (i.e., images) into textual data (i.e., words). In general, the image input is passed through some convolutional layers to generate a dense representation of the visual data. Then, the embedded representation of the visual data is fed to an RNN to generate a sequence of text. Many-to-one RNN architectures are popular for this task.

In 2015, Karpathy et al. [32] presented their influential work on training region convolutional neural network (RCNN) to generate representation vectors for image regions and bidirectional RNN to generate representation vectors for corresponding caption in semantic alignment with each other. They also proposed novel multi-modal RNN to generate a caption that is semantically aligned with the input image. Image regions were selected based on the ranked output of an object detection CNN.

Xu et al. proposed an attention-based framework to generate image caption that was inspired by machine translation models [33]. They used image representations generated by lower convolutional layers from a CNN model rather than the last fully connected layer and used an LSTM to generate words based on hidden state, last generated word, and context vector. They defined the context vector as a dynamic representation of the image generated by applying an attention mechanism on image representation vectors from lower convolutional layers of CNN. Attention mechanism allowed the model to dynamically select the region to focus on while generating a word for image caption. An additional advantage of their approach was intuitive visualization of the model's focus for generation of each word. Their visualization experiments showed that their model was focused on the right part of the image while generating each important word.

Such influential works in the field of automatic image captioning were based on image representations generated by CNNs designed for object detection. Some recently proposed captioning models have sought to change this trend. Biten et al. proposed a captioning model for images used to illustrate new articles [34]. Their caption generation LSTM takes into account both CNN-generated image features and semantic embeddings to the text of corresponding new articles to generate a template of a caption. This template contains spaces for the names of entities like organizations and places. These places are filled in using attention mechanism on the text of the corresponding article.

4.5 ChatBot for Mental Health and Autism Spectrum Disorder

ChatBots are automatic conversation tools that have gained vast popularity in e-commerce and as digital personal assistants like Apple's Siri and Amazon's Alexa. ChatBots represent an ideal application for RNN models as conversations with ChatBots represent sequential data. Questions and answers in a conversation should be based on past iterations of questions and answers in that conversation as well as patterns of sequences learned from other conversations in the dataset.

Recently, ChatBots have found application in screening and intervention for mental health disorders such as autism spectrum disorder (ASD). Zhong et al. designed a Chinese-language ChatBot using bidirectional LSTM in sequence-to-sequence framework which showed great potential for conversation-mediated intervention for children with ASD [35]. They used 400,000 selected sentences from chatting histories involving children in many cases. Rakib et al. developed similar sequence-to-sequence model based on Bi-LSTM to design a ChatBot to respond empathetically to mentally ill patients [36]. A detailed survey of medical ChatBots is presented in [37]. This survey includes references to ChatBots built using NLP techniques, knowledge graphs, as well as modern RNN for a variety of applications including diagnosis, searching through medical databases, dialog with patients, etc.

5 Conclusion

Due to the sequential nature of their architecture, RNNs are applied for ordinal or temporal problems, such as language translation, text summarization, and image captioning, and are incorporated into popular applications such as Siri, voice search, and Google Translate. In addition, they are also often used to analyze longitudinal data in medical applications (i.e., cases where repeated observations are available at different time points for each patient of a dataset). While research in RNN is still an evolving area and new architectures are being proposed, this chapter summarizes fundamentals of RNN including different traditional architectures, training strategies, and influential work. It may serve as a stepping stone for exploring sequential models using RNN and provides reference pointers.

References

1. Rumelhart DE, Hinton GE, Williams RJ (1986) Learning representations by back-propagating errors. Nature 323(6088): 533–536
2. Hopfield JJ (1982) Neural networks and physical systems with emergent collective computational abilities. Proc Natl Acad Sci 79(8): 2554–2558
3. Schmidhuber J (1993) Netzwerkarchitekturen, Zielfunktionen und Kettenregel (Network architectures, objective functions, and chain rule), Habilitation thesis, Institut für Informatik, Technische Universitüt München
4. Mozer MC (1995) A focused backpropagation algorithm for temporal. Backpropag Theory Architect Appl 137
5. Goodfellow I, Bengio Y, Courville A (2016) Deep learning. MIT Press, Cambridge
6. Hochreiter S (1998) The vanishing gradient problem during learning recurrent neural nets and problem solutions. Int J Uncertainty Fuzziness Knowledge Based Syst 6(02):107–116
7. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8): 1735–1780
8. Cho K, Van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, Bengio Y (2014) Learning phrase representations using RNN encoder-decoder for statistical machine translation. Preprint. arXiv:14061078
9. Schuster M, Paliwal KK (1997) Bidirectional recurrent neural networks. IEEE Trans Signal Process 45(11):2673–2681
10. Pascanu R, Gulcehre C, Cho K, Bengio Y (2013) How to construct deep recurrent neural networks. Preprint. arXiv:13126026
11. Bahdanau D, Cho K, Bengio Y (2014) Neural machine translation by jointly learning to align and translate. Preprint. arXiv:14090473
12. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. In: Advances in neural information processing systems, pp 5998–6008
13. Bengio Y, Simard P, Frasconi P (1994) Learning long-term dependencies with gradient descent is difficult. IEEE Trans Neural Netw 5(2):157–166. https://doi.org/10.1109/72.279181
14. Pascanu R, Mikolov T, Bengio Y (2013) On the difficulty of training recurrent neural networks. In: Dasgupta S, McAllester D (eds) Proceedings of the 30th international conference on machine learning, PMLR, Atlanta, vol 28, pp 1310–1318
15. Berger AL, Pietra VJD, Pietra SAD (1996) A maximum entropy approach to natural language processing. Comput Linguist 22(1): 39–71
16. Becker S, Hinton G (1992) Self-organizing neural network that discovers surfaces in random-dot stereograms. Nature 355:161–163. https://doi.org/10.1038/355161a0
17. He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: 2016 IEEE conference on computer vision and pattern recognition (CVPR), pp 770–778. https://doi.org/10.1109/CVPR.2016.90
18. Wu H, Zhang J, Zong C (2016) An empirical exploration of skip connections for sequential tagging. Preprint. arXiv:161003167
19. Jaeger H (2002) Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the echo state network approach. GMD-Forschungszentrum Informationstechnik 5
20. Bengio Y, Ducharme R, Vincent P (2001) A neural probabilistic language model. In: Advances in neural information processing systems, pp 932–938
21. Mikolov T, Karafiát M, Burget L et al (2010) Recurrent neural network based language model. In: INTERSPEECH 2010. Citeseer
22. Jain G, Sharma M, Agarwal B (2019) Optimizing semantic lstm for spam detection. Int J Inform Technol 11(2):239–250

23. Bagnall D (2015) Author identification using multi-headed recurrent neural networks. Preprint. arXiv:150604891
24. Zhou C, Sun C, Liu Z, Lau F (2015) A C-LSTM neural network for text classification. Preprint. arXiv:151108630
25. Wang B (2018) Disconnected recurrent neural networks for text categorization. In: Proceedings of the 56th annual meeting of the association for computational linguistics (volume 1: long papers), pp 2311–2320
26. Nallapati R, Zhai F, Zhou B (2017) Summarunner: a recurrent neural network based sequence model for extractive summarization of documents. In: Thirty-first AAAI conference on artificial intelligence
27. Xu J, Durrett G (2019) Neural extractive text summarization with syntactic compression. Preprint. arXiv:190200863
28. Nallapati R, Zhou B, dos Santos C, Gulcehre Ç, Xiang B (2016) Abstractive text summarization using sequence-to-sequence rnns and beyond. In: Proceedings of the 20th SIGNLL conference on computational natural language learning, pp 280–290
29. Li P, Lam W, Bing L, Wang Z (2017) Deep recurrent generative decoder for abstractive text summarization. In: Proceedings of the 2017 conference on empirical methods in natural language processing, pp 2091–2100
30. Cho K, van Merrienboer B, Gülçehre Ç, Bahdanau D, Bougares F, Schwenk H, Bengio Y (2014) Learning phrase representations using RNN encoder-decoder for statistical machine translation. In: The 2014 conference on empirical methods in natural language processing (EMNLP)
31. Luong MT, Pham H, Manning CD (2015) Effective approaches to attention-based neural machine translation. Preprint. arXiv:150804025
32. Karpathy A, Fei-Fei L (2015) Deep visual-semantic alignments for generating image descriptions. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 3128–3137
33. Xu K, Ba J, Kiros R, Cho K, Courville A, Salakhudinov R, Zemel R, Bengio Y (2015) Show, attend and tell: neural image caption generation with visual attention. In: International conference on machine learning, PMLR, pp 2048–2057
34. Biten AF, Gomez L, Rusinol M, Karatzas D (2019) Good news, everyone! context driven entity-aware captioning for news images. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pp 12466–12475
35. Zhong H, Li X, Zhang B, Zhang J (2020) A general chinese chatbot based on deep learning and its' application for children with ASD. Int J Mach Learn Comput 10:519–526. https://doi.org/10.18178/ijmlc.2020.10.4.967
36. Rakib AB, Rumky EA, Ashraf AJ, Hillas MM, Rahman MA (2021) Mental healthcare chatbot using sequence-to-sequence learning and bilstm. In: Brain informatics, springer international publishing, pp 378–387
37. Tjiptomongsoguno ARW, Chen A, Sanyoto HM, Irwansyah E, Kanigoro B (2020) Medical chatbot techniques: a review. In: Silhavy R, Silhavy P, Prokopova Z (eds) Software engineering perspectives in intelligent systems. Springer International Publishing, Cham, pp 346–356