Lecture 13: Recurrent Neural Networks

ENEE 691 – Machine Learning and Photonics @ UMBC

Ergun Simsek, Ph.D. and Masoud Soroush, Ph.D.

April 19, 2023

1 Recurrent Neural Networks

In today's lecture[1], we introduce an integral part of the deep learning architecture that deals with the sequential data. This component of deep learning is referred to as the **recurrent neural networks** or RNN for short. We will introduce the basics of simple RNN networks and explore some of their applications.

1.1 Background

As we saw in the previous lecture, the convolutional neural networks are most suitable when the data are equipped with certain *spatial* structure. In contrast, the recurrent neural networks are designed when we work with *sequential* or *temporal* data. Also, unlike traditional feedforward neural networks, which process a fixed set of input data and produce a fixed output, RNNs can take a variable-length sequence of input data and produce a corresponding sequence of output data. This makes them well-suited for a wide range of regression and classification tasks, including natural language processing, speech recognition, and time series prediction.

At the heart of an RNN is a recurrent layer, which consists of a set of neurons that are connected to each other in a loop. This loop allows the network to maintain a memory of previous inputs, which is essential for processing sequences of data. Each neuron in the recurrent layer receives two inputs: the current input value and the output value from the previous time step. The neuron processes these inputs and produces an output value, which is then passed on to the next time step.

Brief History: RNNs were first introduced in the 1980s as a way to process sequences of data, but it wasn't until the mid-1990s that they gained popularity in the research community.

The early RNN models suffered from several issues, such as vanishing gradients (when the network is trained using backpropagation, the gradients used to update the weights can become very small as they are propagated back through time and this can make it difficult for the network to learn long-term dependencies between inputs, which is a common problem in sequence processing tasks). To address this issue, several variants of RNNs have been developed. For example in 1997, the Long Short-Term Memory (LSTM) architecture was introduced as a solution to these problems.[a] LSTMs use a more complex gating mechanism to allow the network to selectively remember or forget information from previous time steps, making them much better suited to processing long sequences of data.

In the years following the introduction of LSTMs, a number of improvements were made to the archi-

[1] This product is a property of UMBC, and no distribution is allowed. These notes are solely for your own use, as a registered student in this class.

tecture. For example, the Gated Recurrent Unit (GRU) was introduced in 2014 as a simpler alternative to LSTMs that achieved comparable performance on many tasks.[b]

One of the major breakthroughs in the use of RNNs came in 2014, with the introduction of the Sequence to Sequence (Seq2Seq) model.[c] The Seq2Seq model uses an encoder-decoder architecture to translate sequences of data from one form to another, such as translating natural language text from one language to another.

More recently, attention mechanisms have been introduced to further improve the performance of RNNs on sequence processing tasks.[d] Attention mechanisms allow the network to selectively focus on certain parts of the input sequence when making predictions, improving both the accuracy and efficiency of the model.

[a]https://dl.acm.org/doi/10.1162/neco.1997.9.8.1735
[b]https://arxiv.org/abs/1412.3555
[c]https://arxiv.org/abs/1409.3215
[d]https://papers.neurips.cc/paper/7181-attention-is-all-you-need.pdf

Since recurrent neural networks deal with sequential data, they are widely used as an important tool for the analysis of time series. For instance, recurrent models are widely used in analyzing the stock market data, as datasets in the context of finance typically come with a temporal structure. Another area where recurrent models are often used heavily is the natural language processing (NLP), where textual data are analyzed. Each sentence of a text consists of a number of words. However, the order in which the words appear is crucial. If the very same words of a given sentence are put in a different order, then the sentence will convey a different meaning if any! Clearly, the order in which individual words of a sentence appear plays a crucial role, and any modeling on this kind of data (textual data) must recognize this order. As we will see in the next section, the input of a recurrent neural network is a sequence (rather than a single vector). In the context of textual data, each sentence will be considered as an instance of the textual dataset. However, each sentence (*i.e.* each sample of the textual dataset) itself is formed by a sequence of words. Of course, one may say that sentences are not all of the same length! So how does a recurrent model handle this issue? The answer is that this problem is minor and can be fixed in a straightforward manner through introducing an *embedding* algorithm. However, what is crucial for recurrent models is the *sequential nature* of the input dataset.

It turns out that there are a number of different types of recurrent neural networks. In this lecture, we delve into the details of the simplest type of a recurrent network, and then we will briefly mention how more advanced recurrent models are constructed without providing much detail. The aim of today's lecture is to provide an introduction to the subject, and our treatment of recurrent models is by no means comprehensive. Interested readers should aim for more advanced courses on deep learning to explore recurrent models in a greater detail.

1.2 Single Layer RNN Architecture

In here, we delve into the details of the architecture of a single layer recurrent neural network. First we need to discuss the structure of the input features and the target of the model. As illustrated earlier, the input of recurrent models comes in the format of a *sequence with a chronological order*. Mathematically speaking, the features of a recurrent model have the tensor structure (B, T, d), where B refers to batch/sample number, T is the length of the sequence, and d is the dimension of each vector element of the sequence (*i.e.* the number of continuous features). In other words, the input of the RNN model takes the following structure

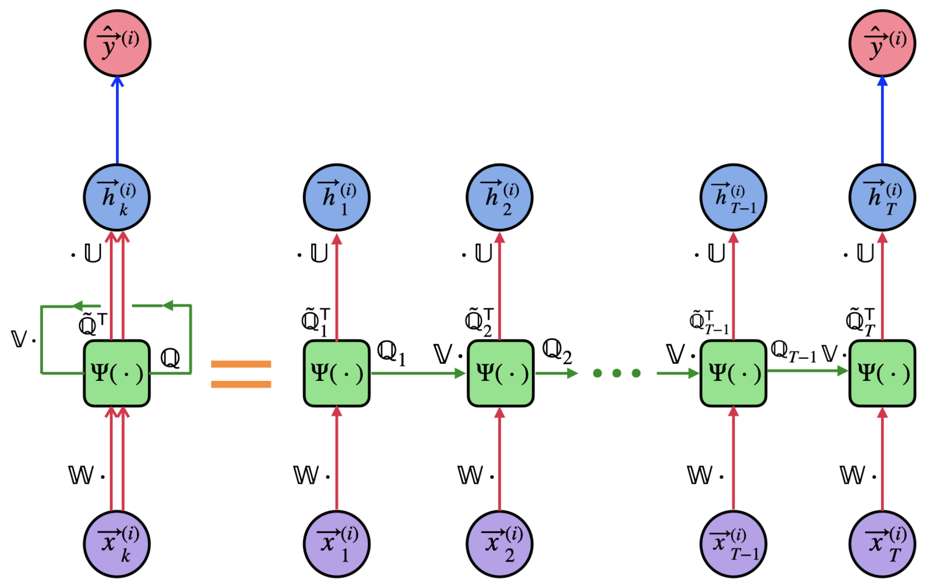
$$\left(\vec{x}_1^{(i)}, \vec{x}_2^{(i)}, \cdots, \vec{x}_T^{(i)}\right), \qquad i = 1, 2, \cdots, n, \qquad \text{and} \qquad \vec{x}_j^{(i)} \in \mathbb{R}^d, \begin{cases} \forall i \in \{1, 2, \cdots, n\}, \\ \forall j \in \{1, 2, \cdots, T\}, \end{cases} \tag{1}$$

where n denotes the number of samples in the dataset. As is clear from (1), the features of every sample of the dataset come in the format of a sequence (of length T) of vectors in $\mathbb{R}^d$. In most of the previous models we covered in this course, the values of features for every sample of the dataset were captured by one single $\mathbb{R}^d$-vector. However, in the context of RNN models, each sample of the dataset is represented by a **sequence of $\mathbb{R}^d$-vectors**. Recall that in the regression lecture, we introduced the $n \times (d+1)$ dimensional matrix $\mathbb{X}$ which was encoding the whole dataset. In the context of RNN models, since associated to each sample, there is a sequence of vectors in $\mathbb{R}^d$, we have to create a sequence $(\mathbb{X}_1, \mathbb{X}_1, \cdots, \mathbb{X}_T)$ of the corresponding $n \times (d+1)$ matrices $\mathbb{X}_k$ which are defined in a similar manner as follows

$$\mathbb{X}_k = \begin{pmatrix} 1 & x_{k,1}^{(1)} & x_{k,2}^{(1)} & \cdots & x_{k,d}^{(1)} \\ 1 & x_{k,1}^{(2)} & x_{k,2}^{(2)} & \cdots & x_{k,d}^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{k,1}^{(n)} & x_{k,2}^{(n)} & \cdots & x_{k,d}^{(n)} \end{pmatrix}, \tag{2}$$

where subscript k refers to the k-th element of the input sequence. Now that we have a clear understanding about the structure of the input of an RNN model, we discuss the target of the model. As mentioned in the previous section, RNN models can be employed for both regression and classification tasks. For concreteness, we will assume that the target variable is a continuous vector $\vec{y}^{(i)} \in \mathbb{R}^m$. In the case of classification with m classes, as we saw in the last two lectures, we first make predictions for the m scores of the corresponding m classes, and finally convert scores into a discrete probability distribution through a $\mathsf{Softmax}$ layer at the final stage of the network.

We are now ready to discuss the structure of recurrent neural networks. In here, we delve into the structure of the simplest RNN model, namely the single hidden layer recurrent neural network. As mentioned earlier, in the context of recurrent models, the basic idea is to *exploit the order in which the elements of the input sequence appear*. The recurrent models are constructed in a way that they process one element of the input sequence at a time but, as they process individual elements of the sequence, they keep the history of the effects of earlier elements of the input sequence on the target. Building the history of the effects of earlier terms of the sequence is done through a **recursive** process. This implies that the history of the effects of the earlier terms is updated every time a new element of the input sequence is processed. In other words, in order to train the model, two terms are simultaneously fed into the model: The current term $\vec{x}_k^{(i)}$ of the input sequence for the i-th training sample, and the history of the effects of all previous terms in the input sequence $\vec{q}_{k-1}^{(i)}$. The final prediction of the model $\hat{\vec{y}}^{(i)}$ is then made only when all elements of the input sequence are seen by the model. The following picture depicts the internal structure of a single layer RNN model.



As is seen from the above picture, initially there is no history of previous terms when the first element of the sequence, $\vec{x}_1^{(i)}$, is fed into the model. Once the model processes the first element and an output $\vec{q}_1^{i}$ is generated, the history of the model is updated by this output. In the second step, $\vec{x}_2^{i}$ and $\vec{q}_1^{(i)}$ are simultaneously fed into the model. An outcome $\vec{q}_2^{(i)}$ is generated and the history is updated accordingly. The process continues until all elements of the input sequence are processed by the model. To make the final prediction, the complete history of the model, $\vec{q}_T^{(i)}$, is used to construct the predicted target $\hat{\vec{y}}^{(i)}$. Now, we need to explain the structure of the weights of the above RNN model. The weights of the above RNN model are captured by three matrices $\mathbb{W}$, $\mathbb{V}$, and $\mathbb{U}$. Matrix $\mathbb{W}$ is an $N \times (d+1)$, where N denotes the number of neurons (*i.e.* this is a hyperparameter of the model) of the RNN layer. Matrix $\mathbb{V}$ is a square $N \times N$ matrix, and $\mathbb{U}$ is an $(N+1) \times m$ matrix from which the final outputs of the model $\vec{h}_k^{(i)}$ are generated. One should note an important feature of the RNN model from the above picture. As you noticed, at each stage when an element of the sequence is fed into the model, the model uses the *same weights* (At each stage of the time sequence, the same matrices $\mathbb{W}$, $\mathbb{V}$, and $\mathbb{U}$ are used). This important characteristic of RNN models is referred to as **weight sharing**. In other words, all elements of the time sequence of the input matrix share the same weights, as they are fed into the model in a sequential manner. Now the question is why all elements of the sequence share the same weights.

Question: Wouldn't it be more accurate to assign different weight matrices $(\mathbb{W}, \mathbb{V}, \mathbb{U})$ at each stage of the time sequence? In other words, wouldn't it be more appropriate to consider T triplets of weight matrices $(\mathbb{W}, \mathbb{V}, \mathbb{U})$ rather than just one?

Answer: There are several good reasons for considering only one triplet of weight matrices. First, it should be noted that even if we consider T triplet of weight matrices (one associated with each element of the time sequence), the difference between the accuracies of the model with one and T triplets is negligible. The interested reader can construct neural networks with different weights for each element of the time sequence, and compare the difference in results. There is yet another important reason to prefer weight sharing over separate weights for individual elements of the time sequence. Note that the triplet $(\mathbb{W}, \mathbb{V}, \mathbb{U})$ already introduces a large number of parameters for the model. Considering T separate triplets $(\mathbb{W}, \mathbb{V}, \mathbb{U})$ multiplies the total number of weights by T. This means that for $T = 10$ (which is a relatively low number for the length of the time sequence), the RNN model will have 10 times more weights than the case in which weight sharing is exercised! The huge number of weights, when weight sharing is not exercised, makes the training process a lot more time consuming, and more importantly, it introduces a large variance into the model. Therefore, for both conceptual and practical reasons, weight sharing must be considered as part of the structure of recurrent neural networks.

Now, let us discuss how the mechanic of the above RNN model works precisely. Note that in order to calculate the history of the model $\vec{q}_k^{(i)}$ for the i-th sample at step k (of the time sequence), we can proceed in a systematic manner. We collect the history of the model at each step for *all samples* of the dataset and represent by the matrix

$$\mathbb{Q}_k = \begin{pmatrix} q_{k,1}^{(1)} & q_{k,1}^{(2)} & \cdots & q_{k,1}^{(n)} \\ q_{k,2}^{(1)} & q_{k,2}^{(2)} & \cdots & q_{k,2}^{(n)} \\ \vdots & \vdots & \ddots & \vdots \\ q_{k,N}^{(1)} & q_{k,N}^{(2)} & \cdots & q_{k,N}^{(n)} \end{pmatrix} . \tag{3}$$

Matrix $\mathbb{Q}$ in (3) is an $N \times n$ matrix, where N and n denote the number of RNN neurons and the number of samples of the dataset, respectively. Matrix $\mathbb{Q}_k$ which captures the history of the model after seeing k elements of the time sequence can be calculated in a recursive manner through the following equation

$$\boxed{\mathbb{Q}_k = \Psi\Big(\mathbb{W} \cdot \mathbb{X}_k^{\mathsf{T}} + \mathbb{V} \cdot \mathbb{Q}_{k-1}\Big) , \qquad \text{for} \quad k = 1, 2, \cdots , T ,} \tag{4}$$

where $\mathbb{Q}_0$ is the zero matrix ($\mathbb{Q}_0 = 0$), and Ψ applies a chosen activation function ψ to all elements of its input matrix (*i.e.* Ψ acts on its input matrix in an elementwise manner). Equation (4) is a recursive equation which states that $\mathbb{Q}_k$ (*i.e.* the history of the model after seeing k elements of the time sequence) cannot be calculated unless all the previous histories are calculated. In other words, to take the advantage of (4), we first set $k = 1$. Then equation (4) states that $\mathbb{Q}_1 = \Psi(\mathbb{W} \cdot \mathbb{X}_1^{\mathsf{T}})$. Now that the history $\mathbb{Q}_1$, after seeing one element of the time sequence, has been calculated, we can move to the second element of the sequence $\mathbb{X}_2$. Using the same equation (4) (this time k is $k = 2$), we find

$$\mathbb{Q}_2 = \Psi\big(\mathbb{W} \cdot \mathbb{X}_2^{\mathsf{T}} + \mathbb{V} \cdot \mathbb{Q}_1\big) = \Psi\Big(\mathbb{W} \cdot \mathbb{X}_2^{\mathsf{T}} + \mathbb{V} \cdot \Psi(\mathbb{W} \cdot \mathbb{X}_1^{\mathsf{T}})\Big) . \tag{5}$$

This process continues until the final history of the RNN model $\mathbb{Q}_T$ is calculated, and all elements of the time sequence are seen by the model. The final prediction of the model is made when the full history matrix $\mathbb{Q}_T$ is calculated. In order to be able to predict the target variable with the correct dimension, we need to multiply the history matrices $\mathbb{Q}_k$ by an appropriate matrix. First in order to accommodate bias terms in linear combinations of rows of the history matrices, we add a row of ones to each history matrix $\mathbb{Q}_k$ as follows

$$\tilde{\mathbb{Q}}_k = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ q_{k,1}^{(1)} & q_{k,1}^{(2)} & \cdots & q_{k,1}^{(n)} \\ q_{k,2}^{(1)} & q_{k,2}^{(2)} & \cdots & q_{k,2}^{(n)} \\ \vdots & \vdots & \ddots & \vdots \\ q_{k,N}^{(1)} & q_{k,N}^{(2)} & \cdots & q_{k,N}^{(n)} \end{pmatrix} . \tag{6}$$

As is clear from (6), matrix $\tilde{\mathbb{Q}}_k$ is an $(N+1) \times n$ matrix. We collect all output vectors $\vec{h}_k^{(i)}$ - sometimes called the **hidden states** - for all samples of a dataset in the matrix $\mathbb{H}_k$

$$\mathbb{H}_k = \begin{pmatrix} h^{(1)}_{k,1} & h^{(1)}_{k,2} & \cdots & h^{(1)}_{k,m} \\ h^{(2)}_{k,1} & h^{(2)}_{k,2} & \cdots & h^{(2)}_{k,m} \\ \vdots & \vdots & \ddots & \vdots \\ h^{(n)}_{k,1} & h^{(n)}_{k,2} & \cdots & h^{(n)}_{k,m} \end{pmatrix} , \tag{7}$$

which is $n \times m$. To get from $\tilde{\mathbb{Q}}_k$ to $\mathbb{H}_k$, we multiply the transpose of $\tilde{\mathbb{Q}}_k$ by $\mathbb{U}$ from the right as follows

$$\mathbb{H}_k = \tilde{\mathbb{Q}}_k^{\mathsf{T}} \cdot \mathbb{U} , \qquad \text{for} \quad k = 1, 2, \cdots , T . \tag{8}$$

In (8), matrix $\mathbb{U}$ is an $(N+1) \times m$ matrix that includes $m(N+1)$ weights. Now the matrix $\mathbb{H}$ constructed above has the correct dimensions as the target matrix $\mathbb{Y}$. As mentioned earlier, the final prediction $\hat{\mathbb{Y}}$ of the RNN model for the target variable is made when the model has seen all elements of the input sequence. This means that we can declare the last $\mathbb{H}$-matrix as the final prediction of the model for the target

$$\hat{\mathbb{Y}} = \mathbb{H}_T . \tag{9}$$

It would now be interesting to find the total number of parameters of the RNN model we established above. As we discussed earlier, the RNN model enjoys the weight sharing property, and all the weights of the model are captured by the triplet $(\mathbb{W}, \mathbb{V}, \mathbb{U})$. Hence, by counting the number of entries of these matrices, we can find the total number of weights of the model as follows

$$\begin{aligned} \#\ \text{parameters of a single layer RNN model} &= \overbrace{N(d+1)}^{\#\text{ of elements of } \mathbb{W}} + \overbrace{N \times N}^{\#\text{ of elements of } \mathbb{V}} + \overbrace{(N+1) \times m}^{\#\text{ of elements of } \mathbb{U}} \\ &= N^2 + N(d+m+1) + m . \end{aligned} \tag{10}$$

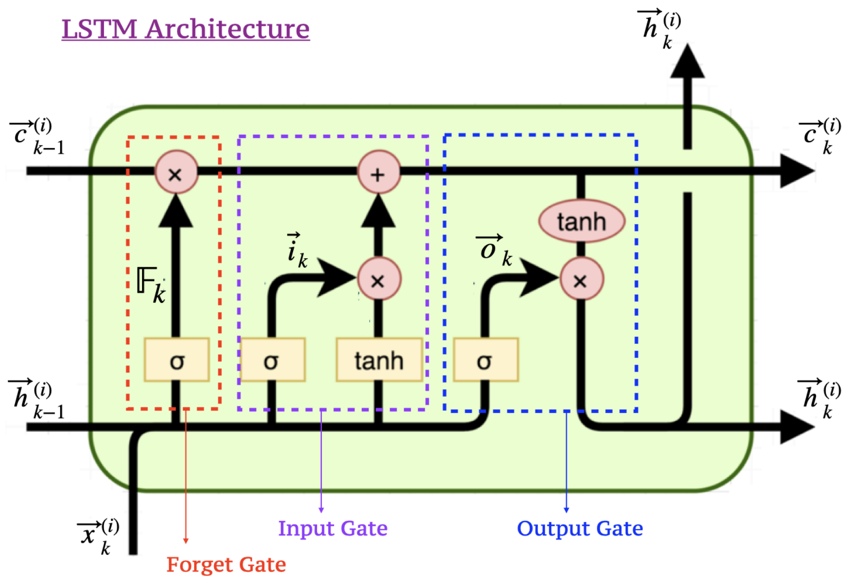
1.3 More Sophisticated Recurrent Models

The single layer RNN model we discussed in the previous section is, in fact, the simplest recurrent model that can be constructed in the context of deep learning. Many more advanced recurrent deep models have been invented during the past two decades. We do not intend to present a detailed treatment of other types of recurrent deep models in here, as they are beyond the scope of this course (The interested reader should pursue the more advanced recurrent models in a deep learning course). We rather provide a brief discussion on how more advanced recurrent models can be developed.

- **Multilayer RNN Models:** An obvious direction for generalizing the simple RNN model discussed in the previous section is to add more RNN layers. In constructing a multilayer RNN model, the output vectors $\vec{h}^{(i)}$ of the first RNN layer will be the inputs of the next RNN layer. In other words, each RNN layer would come with its own weight matrices $(\mathbb{W}^{(I)}, \mathbb{V}^{(I)}, \mathbb{U}^{(I)})$, where index I runs over different RNN layers of the model. In this construction, each RNN layer enjoys its own weight sharing among different elements of the time sequence.

- **Bidirectional RNN Models:** Another variation of RNN models is introduced when we allow the input sequence to be analyzed from both ends. In our treatment in the previous section, the input sequence $(\vec{x}^{(i)}_1, \vec{x}^{(i)}_2, \cdots , \vec{x}^{(i)}_T)$ is analyzed from left to right. This means that the RNN model sees $\vec{x}^{(i)}_1$ as the first element, $\vec{x}^{(i)}_2$ as the next element, and so on. However, in certain cases, it may be beneficial to feed $\vec{x}^{(i)}_T$ as the first element into the model, then $\vec{x}^{(i)}_{T-1}$ as the next element until we get to $\vec{x}^{(i)}_1$ as the last element to be fed into the model. In bidirectional RNN models, there are secretly two copies of the (one directional) RNN model. One analyzes the input sequence from left to right, and the other analyzes the input sequence from right to left. The two copies of the (one directional) RNN model are not decorrelated in the bidirectional case, and there are interactions between the two (They use the outputs of each other in a precise manner).

- **Gated RNN Models:** A number of more powerful variations of the recurrent models take the advantage of the *gating mechanism*. Roughly speaking, a gating mechanism produces a number $p \in [0,1]$ and allows only $100p$ percent of the incoming information to flow through the gate and be delivered to the next step (For instance, if $p = 0.25$, then only 25% of the incoming information to the gate would pass through the gate and will be delivered as the output of the gate to the next step). Two of the prominent gated RNN models that are commonly used in deep learning are the **GRU** (Gated Recurrent Unit), and the **LSTM** (Long Short Term Memory) models. For instance in the LSTM architecture, in addition to the hidden states $\vec{h}_k^{(i)}$, the model creates another set of states $\vec{c}_k^{(i)}$, called **context states**. In this architecture, the hidden states $\vec{h}_k^{(i)}$ learn from the input sequence in the usual way as is the case in the simple RNN model we discussed in the previous section. However, the context vectors $\vec{c}_k^{(i)}$ only keeps the useful data from the history to be used at later stages in the time sequence. In this manner, $\vec{h}_k^{(i)}$ states store the short term history whereas $\vec{c}_k^{(i)}$ store the useful long term history to be used at later stages. The following picture depicts the internal structure of the LSTM design.



As seen from the above picture, the architecture of the LSTM model consists of three gates: the forget gate, the input gate and the output gate. As it stands from the name, the forget gate forgets the unuseful part of the context $\vec{c}_{k-1}^{(i)}$. The input gate determines what to be added to $\vec{c}_{k-1}^{(i)}$, and the output gate decides what parts of $\vec{c}_k^{(i)}$ to be included in $\vec{h}_k^{(i)}$. In here, we simply offered a very brief summary of the LSTM model but the details and the math behind the model should be pursued in a deep learning course.

1.4 Pros and Cons of Recurrent Neural Networks

Advantages:

- RNNs are designed to process sequences of data points, making them particularly effective for applications such as natural language processing, speech recognition, and time series prediction. They can be used for both classification and regression tasks.

- RNNs can learn to maintain a memory of past inputs, which is useful for predicting future outputs based on past information. Also, RNNs can be updated incrementally, making them well-suited for online learning scenarios where data arrives in a stream.

Disadvantages

- Recurrent neural networks may not offer an effective handle if the length of the input sequence is too large (*e.g.* sequential data in genomics problems typically involve extremely lengthy sequences). In that case, training a recurrent model is very slow. A direct corollary of this is that RNN models are very time-consuming and expensive in terms of the necessary computational power.
- While RNNs can theoretically capture long-term dependencies, in practice they can struggle with sequences that are very long or have complex dependencies between distant time steps.
- Under some circumstances, training recurrent neural networks in a stable manner is notoriously difficult!