TECHNEEDS

WEEK-4

NOTES

Topic Covered:

- **Why do we need RNN?**
- **Types of RNN**
- **Architecture of RNN**
- **Backpropagation**

Why do we need RNN?

Recurrent neural networks (RNNs) are the state of the art algorithm for sequential data and are used by Apple's Siri and Google's voice search. It is the first algorithm that remembers its input, due to an internal memory, which makes it perfectly suited for machine learning problems that involves sequential data. It is one of the algorithms behind the scenes of the amazing achievements seen in deep learning over the past few years**.**

Recurrent V/S Feed Forward neural Network

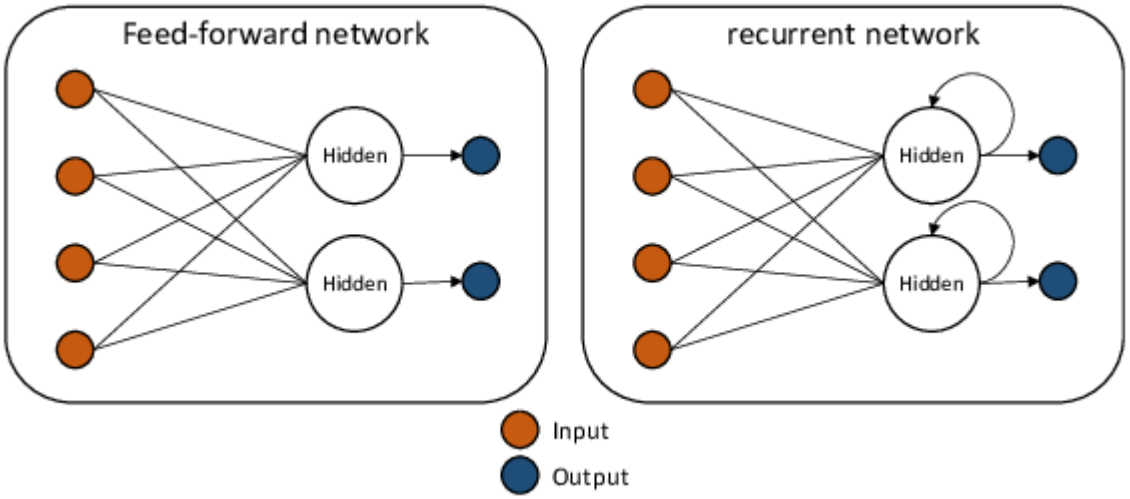
RNNs and feed-forward neural networks get their names from the way they channel information.

In a feed-forward neural network, the information only moves in one direction from the input layer, through the hidden layers, to the output layer. The information moves straight through the network.

Feed-forward neural networks have no memory of the input they receive and are bad at predicting what's coming next. Because a feed-forward network only considers the current input, it has no notion of order in time. It simply can't remember anything about what happened in the past except its training.

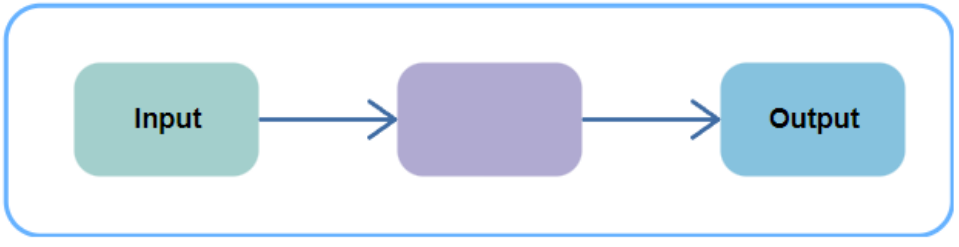
In an RNN, the information cycles through a loop. When it makes a decision, it considers the current input and also what it has learned from the inputs it received previously.

The two images below illustrate the difference in information flow between an RNN and a feed-forward neural network.



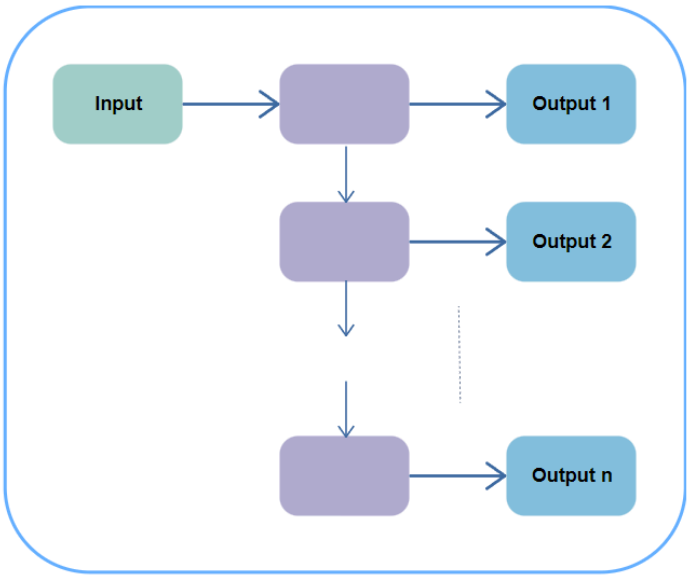
Types of RNN

- **ONE TO ONE RNN:**
  The simplest type of RNN is One-to-One, which allows a single input and a single output. It has fixed input and output sizes and acts as a traditional neural network. The One-to-One application can be found in *Image Classification*.



One-to-One

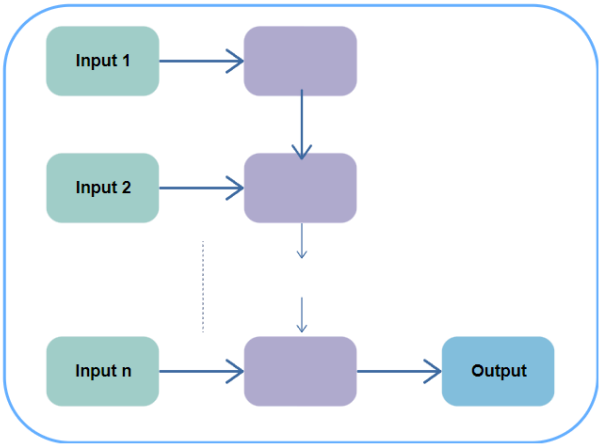
- **ONE TO MANY RNN:**
  One-to-Many is a type of RNN that gives multiple outputs when given a single input. It takes a fixed input size and gives a sequence of data outputs. Its applications can be found in *Music Generation* and *Image Captioning*.



One-to-Many

- **MANY TO ONE RNN:**

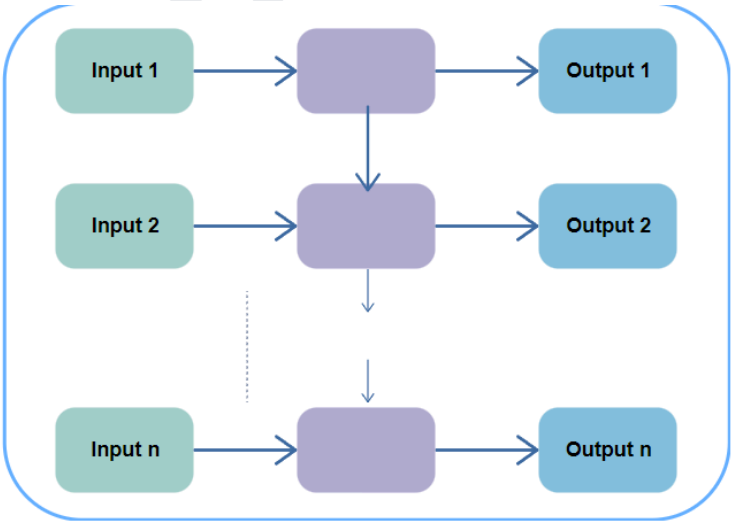
  Many-to-One is used when a single output is required from multiple input units or a sequence of them. It takes a sequence of inputs to display a fixed output. *Sentiment analysis* is a common example of this type of Recurrent Neural Network.



Many-to-One

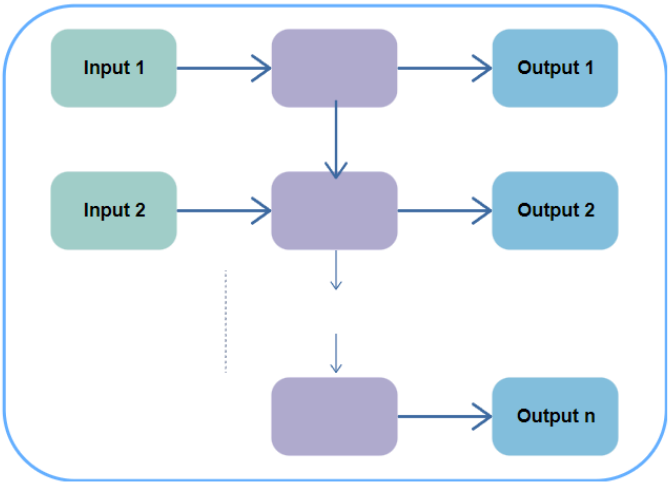
- **MANY TO ONE RNN:**
  Many-to-Many is used to generate a sequence of output data from a sequence of input units.

  This type of RNN is further divided into the following two subcategories:
  - **Equal Unit Size:** In this case, the number of both the input and output units is the same. A common application can be found in *Name-Entity Recognition*.



Many-to-Many (Equal)

  - **Unequal Unit Size:** In this case, inputs and outputs have different numbers of units. Its application can be found in *Machine Translation*.



Many-to-Many (Unequal)

ARCHITECTURE OF RNN

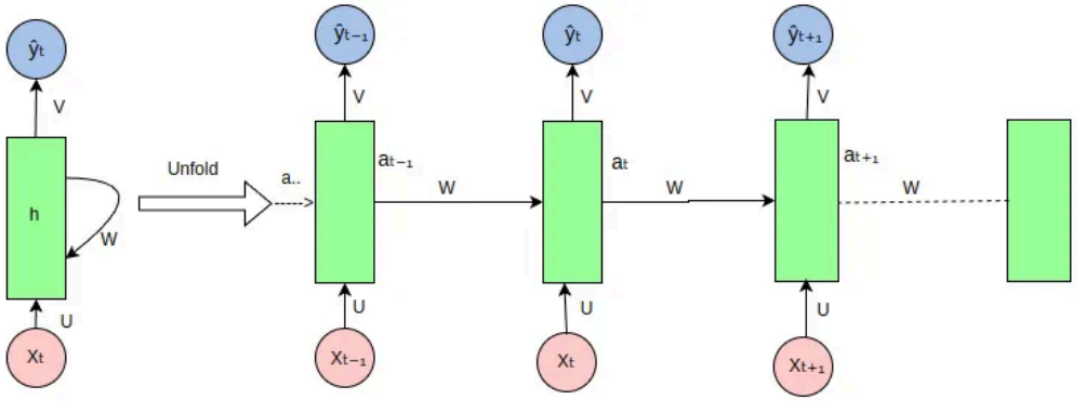


fig 3: RNN Unfolded. Image by Author.

The RNN takes an input vector X and the network generates an output vector y by scanning the data sequentially from left to right, with each time step updating the hidden state and producing an output. It shares the same parameters across all time steps. This means that the same set of parameters, represented by U, V, and W is used consistently throughout the network. `U` represents the weight parameter governing the connection from input layer X

to the hidden layer h , W represents the weight associated with the connection between hidden layers, and V for the connection from hidden layer h to output layer y. This sharing of parameters allows the RNN to effectively capture temporal dependencies and process sequential data more efficiently by retaining the information from previous input in its current hidden state.

At each time step t, the hidden state a_t is computed based on the current input x_t , previous hidden state a_{t-1} and model parameters as illustrated by the following formula:

$$a_t = f(a_{t-1}, x_t; \theta) \quad \text{— — — — — (1)}$$

It can also be written as,

$$a_t = f(U * X_t + W * a_{t-1} + b)$$

where,

- a_t represents the output generated from the hidden layer at time step t .
- x_t is the input at time step t.
- θ represents a set of learnable parameters(weights and biases).

- `U` is the weight matrix governing the connections from the input to the hidden layer; $U \in \theta$
- `W` is the weight matrix governing the connections from the hidden layer to itself (recurrent connections); $W \in \theta$
- `V` represents the weight associated with connection between hidden layer and output layer; $V \in \theta$
- `a□-1` is the output from hidden layer at time `t-1`.
- `b` is the bias vector for the hidden layer; $b \in \theta$
- `f` is the activation function.

For a finite number of time steps T=4, we can expand the computation graph of a Recurrent Neural Network, illustrated in Figure 3, by applying the equation (1) T-1 times.

$a_4 = f(a_3, x_4; \theta)$ — — — — — (2)

Equation (2) can be expanded as,

$a_4 = f(U * X_4 + W * a_3 + b)$

$a_3 = f(U * X_3 + W * a_2 + b)$

$a_2 = f(U * X_2 + W * a_1 + b)$

The output at each time step t, denoted as y_t is computed based on the hidden state output a_t using the following formula,

$\hat{y}_t = f(a_t; \theta)$ — — — — — (3)

Equation (3) can be written as,

$\hat{y}_t = f(V * a_t + c)$

when t=4, $\hat{y}_4 = f(V * a_4 + c)$

where,

- $\hat{y}_t$ is the output predicted at time step t.
- V is the weight matrix governing the connections from the hidden layer to the output layer.
- c is the bias vector for the output layer.

BACKPROPAGATION

Backpropagation involves adjusting the model's parameters (weights and biases) based on the error between predicted output and the actual target value. The goal of backpropagation is to improve the model's performance by minimize the loss function. Backpropagation Through Time is a special variant of backpropagation used to train RNNs, where the error is propagated backward through time until the initial time step t=1. Backpropagation involves two key steps: forward pass and backward pass.

1. Forward Pass: During forward pass, the RNN processes the input sequence through time, from `t=1` to `t=n`, where `n` is the length of input sequence. In each forward propagation, the following calculation takes place

$a_t = U * X_t + W * a_{t-1} + b$

$a_t = \tanh(a_t)$

$\hat{y}_t = \text{softmax}(V * a_t + c)$

After processing the entire sequence, RNN generates a sequence of predicted outputs $\hat{y} = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}\square]$. Loss is then computed by comparing predicted output ŷ at each time step with actual target output y. Loss function given by,

L(y, ŷ) = (1/t) * Σ(y□ — ŷ□)² — — —— — -> MSE Loss

2. Backward Pass: The backward pass in BPTT involves computing the gradients of the loss function with respect to the network's parameters (`U`, `W`, `V` and biases) over each time step.

Let's explore the concept of backpropagation through time by computing the gradients of loss at time step t=4. The figure below also serves as an illustration of backpropagation for time step 4.

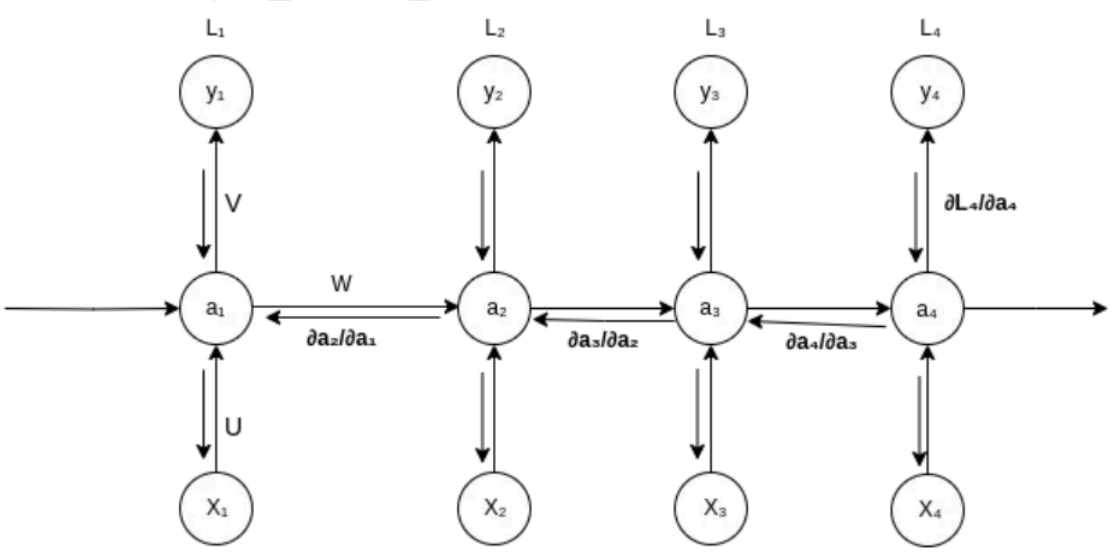


fig 4: Back Propagation Through Time (BPTT). Image by Author

Derivative of loss L w.r.t V

Loss L is a function of predicted value $\hat{y}$, so using the chain rule $\partial L/\partial V$ can be written as,

$$\partial L/\partial V = (\partial L/\partial \hat{y}) * (\partial \hat{y}/\partial V)$$

Derivative of loss L w.r.t W

Applying the chain rule of derivatives $\partial L/\partial W$ can be written as follows: The loss at the 4th time step is dependent upon $\hat{y}$ due to the fact that the loss is calculated as a function of $\hat{y}$, which is in turn dependent on the current time step's hidden state a_4, a_4 is influenced by both $W$and a_3, and again $a_3$is connected to both $a_2$and W,and a_2 depends on $a_1$and also on W.

$$\partial L_4/\partial W = (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial W) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial W) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial W) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial a_1 * \partial a_1/\partial W)$$

Derivative of loss L w.r.t U

Similarly, $\partial L/\partial U$ can be written as,

$$\partial L_4/\partial U = (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial U) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial U) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial U) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial a_1 * \partial a_1/\partial U)$$

Here we're summing up the gradients of loss across all time steps which represents the key difference between BPTT and regular backpropagation approach.

BLOGS TO READ:

- BLOG-1
- BLOG-2