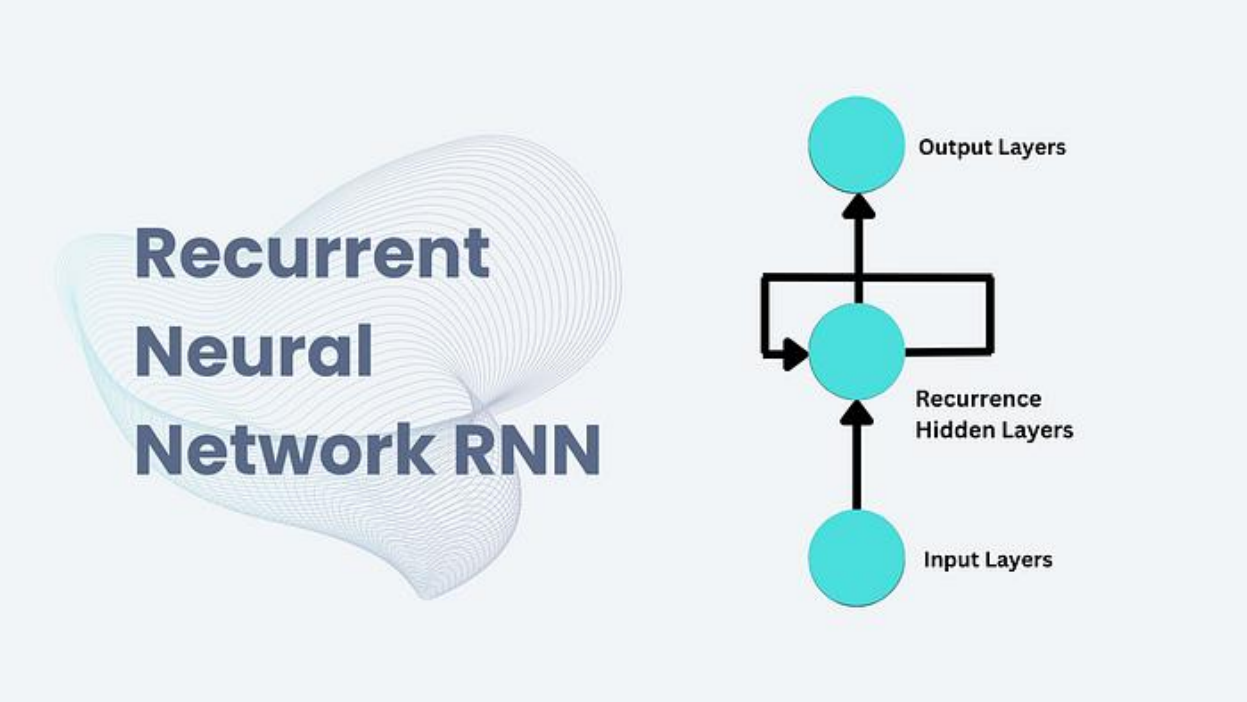


What is RNN ?

Recurrent Neural Network(RNN) is a type of Neural Network where the output from the previous step is fed as input to the current step.

- In some cases when it is required to **predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words**. Thus RNN came into existence, which solved this issue with the help of a Hidden Layer.
- The main and most important feature of RNN is its **Hidden state**, which remembers some information about a sequence. The state is also referred to as ***Memory State*** since it remembers the previous input to the network.
- RNN uses the same weights for each element of the sequence.

RNN are a class of neural networks that is powerful for modeling sequence data such as time series or natural language. Basically, main idea behind this architecture is to use sequential information.

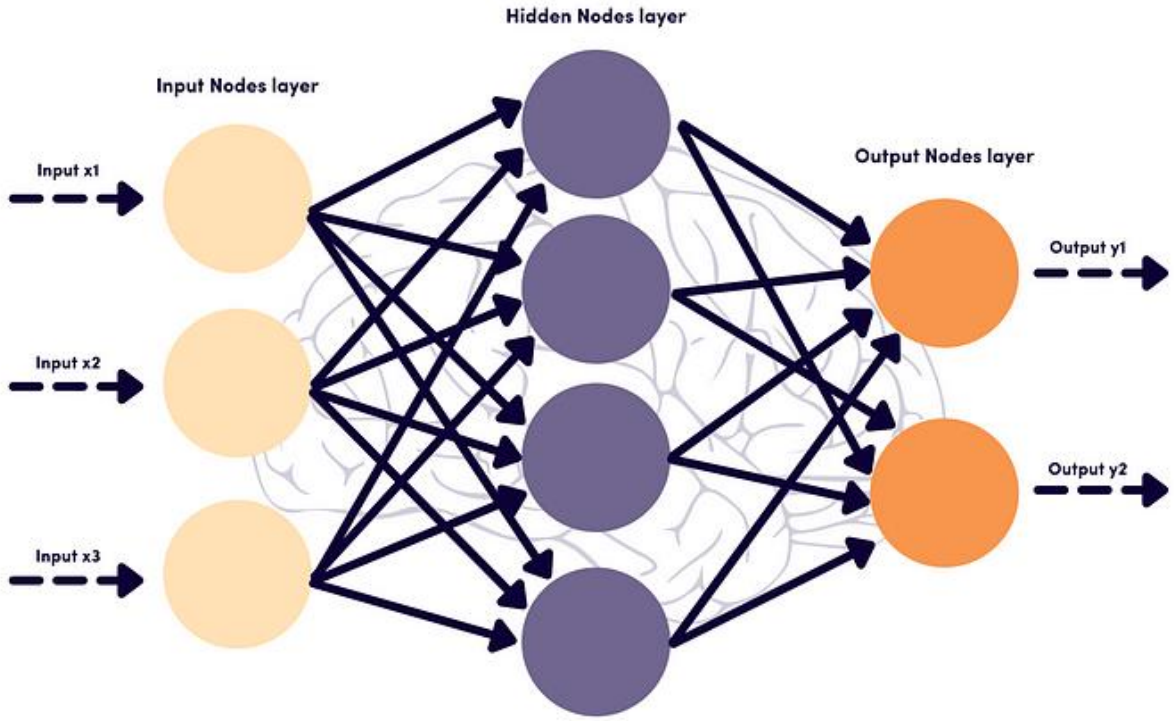
Do you know how Google's autocomplete function works???

Basically, collection of large volumes of most frequently occurring consecutive words fed into ***RNN network.*** It analyze the data by finding the sequence of words occurring frequently and builds a model to predict the next word in the sentence.

So, do you see the importance of the RNN in our daily life.

Why to make it more complex by introducing a new network (RNN), hen we already have feed forward neural network(ANN)?

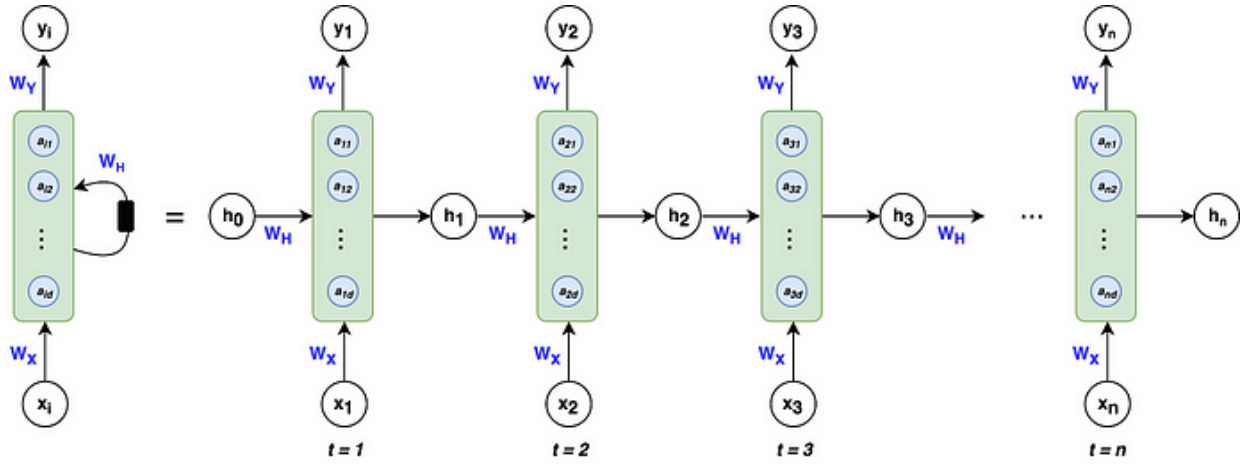
In ANN, information flows only in forward direction from the input nodes, through the hidden layers and to the output nodes. There are no cycles or loops in the network.



ANN architecutre

Issues in the feed forward neural network : -

1. Can’t handle sequential data.
2. Consider only current input.
3. Can’t memorize the previous input.



RNN architecture

Table of contents

What are RNNs used for?

Recurrent Neural Networks (RNNs) are widely used for data with some kind of sequential structure. For instance, time series data has an intrinsic ordering based on time. Sentences are also sequential, "I love dogs" has a different meaning than "Dogs I love." Simply put, if the *semantics* of your data is altered by random permutation, you have a sequential dataset and RNNs may be used for your problem! To help solidify the types of problems RNNs can solve, here is a **list of common applications**[1] :

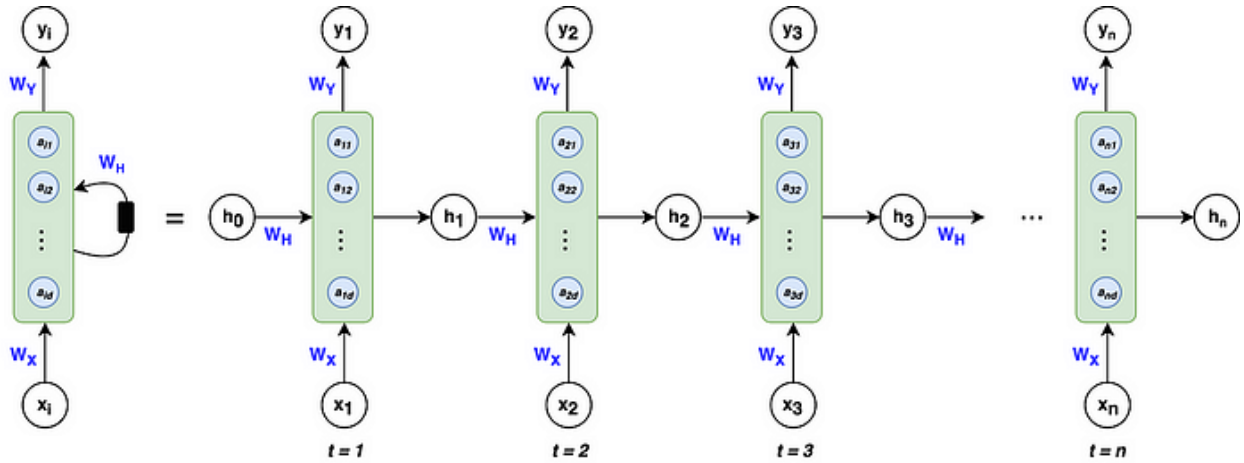
- Speech Recognition
- Sentiment Classification
- Machine Translation (i.e. Chinese to English)
- Video Activity Recognition
- Name Entity Recognition — (i.e. Identifying names in a sentence)

Great! We know the types of problems that we can apply RNNs to, now...

What are RNNs and how do they work?

RNNs are different than the classical multi-layer perceptron (MLP) networks because of two main reasons: 1) They take into account what happened *previously* and 2) they *share* parameters/weights.

The architecture of an RNN



***Left**: Shorthand notation often used for RNNs*

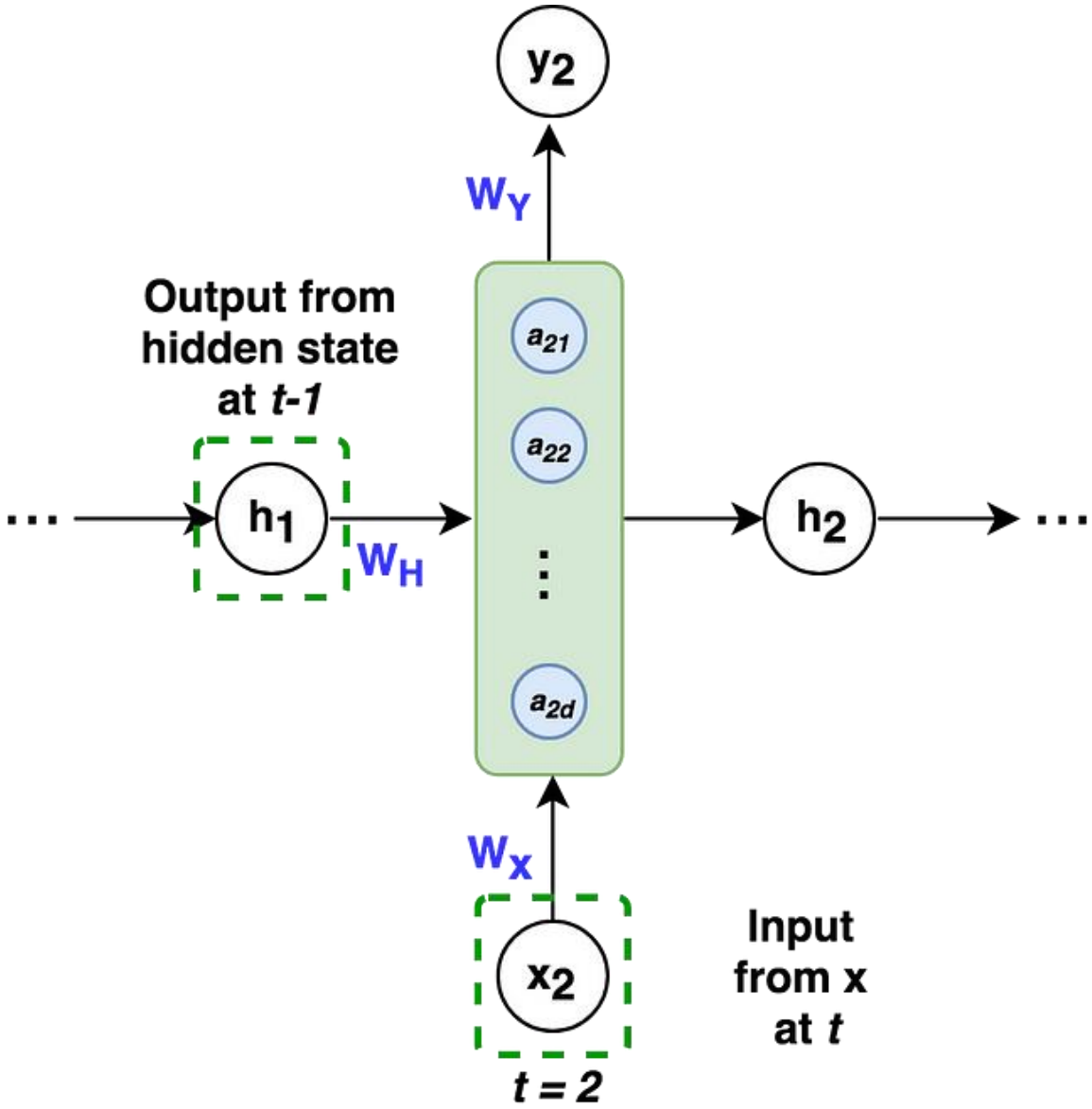
***Right**: Unfolded notation for RNNs*

Don't worry if this doesn't make sense, we're going to break down all the variables and go through a forward propagation and backpropagation in a little bit! Just focus on the flow of variables at first glance.

A breakdown of the architecture

The green blocks are called ***hidden states***. The blue circles, defined by the **vector *a*** within each block, are called ***hidden nodes*** *or* ***hidden units*** where the number of nodes is decided by the hyper-parameter ***d***. Similar to activations in MLPs, think of each green block as an activation function that acts on each blue node. **We'll talk about the calculations within the hidden states in the forward propagation section of this article.**

Vector *h* — is the output of the hidden state after the activation function has been applied to the hidden nodes. As you can see at time *t,* the architecture takes into account what happened at *t-1* by including the *h* from the *previous* hidden state as well as the input *x* at time *t*. This allows the network to account for information from previous inputs that are sequentially behind the current input. It's important to note that the zeroth *h* vector will always start as a vector of 0's because the algorithm has no information preceding the first element in the sequence.



The hidden state at t=2, takes as input the output from t-1 and x at t.

Matrices *Wx, Wy, Wh* — are the weights of the RNN architecture which are *shared* throughout the entire network. The model weights of *Wx* at *t=1* are the exact same as the weights of *Wx* at *t=2* and every other time step.

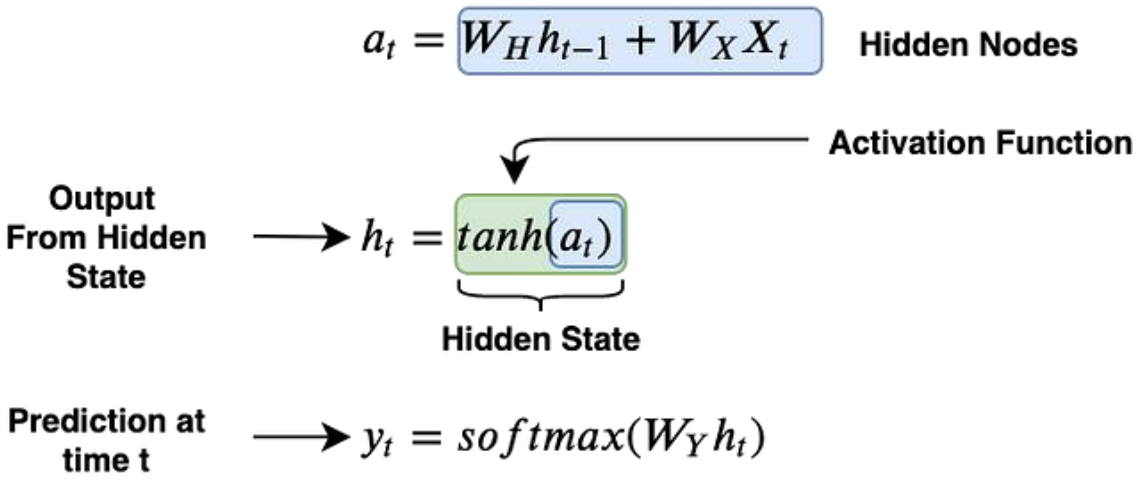
Vector x_i — is the input to each hidden state where i=1, 2,..., n for each element in the input sequence. Recall that text must be encoded into numerical values. For example, every letter in the word "dogs" would be a one-hot encoded vector with dimension *(4x1)*. Similarly, x can also be word embedding or other numerical representations.

$$d = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad o = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad g = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad s = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

One-Hot Encoding of the word “dogs”

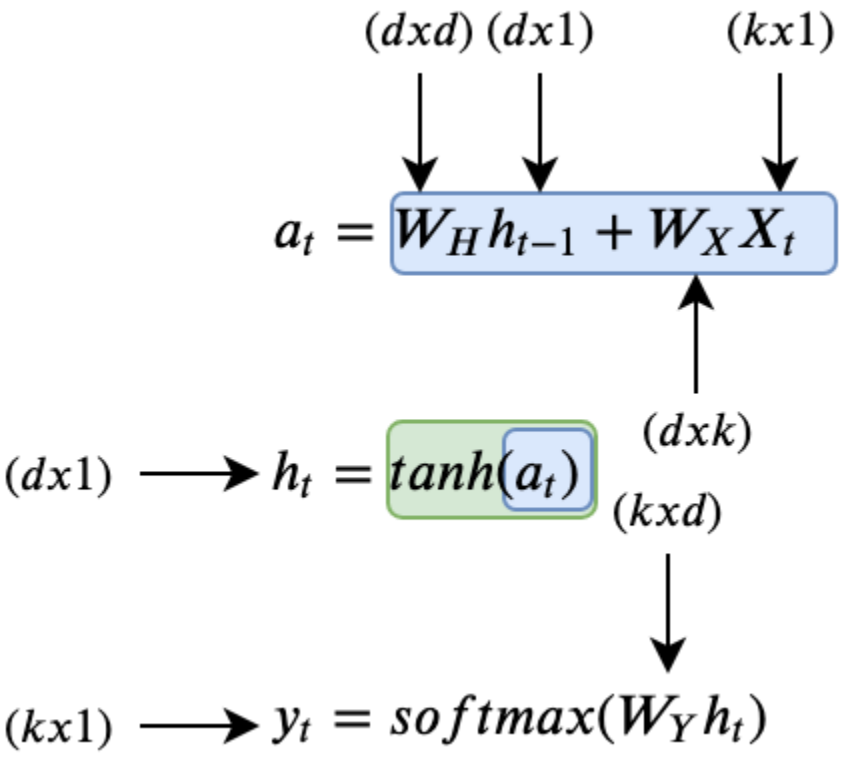
RNN Equations

Now that we know what all the variables are, here are all the equations that we’re going to need in order to go through an RNN calculation:



These are the only three equations that we need, pretty sweet! The **hidden nodes** are a concatenation of the previous state’s output weighted by the weight matrix *Wh* and the input *x* weighted by the weight matrix *Wx*. The *tanh* function is the **activation function** that we mentioned earlier, symbolized by the green block. The **output of the hidden state** is the activation function applied to the hidden nodes. To make a **prediction**, we take the output from the current hidden state and weight it by the weight matrix *Wy* with a soft max activation.

It's also important to understand the dimensions of all the variables floating around. In general for predicting a sequence:



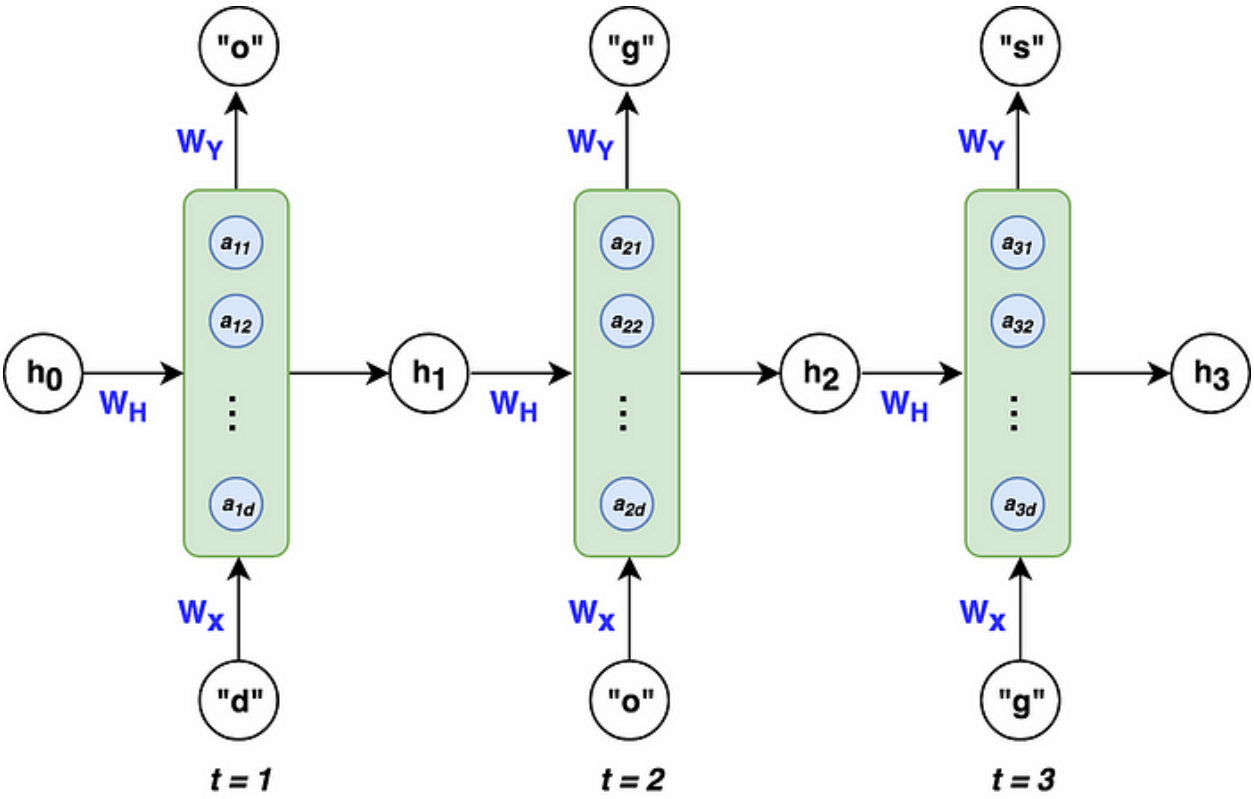
Where

- **k** is the dimension of the input vector x_i
- **d** is the number of hidden nodes

Now we're ready to walk through an example!

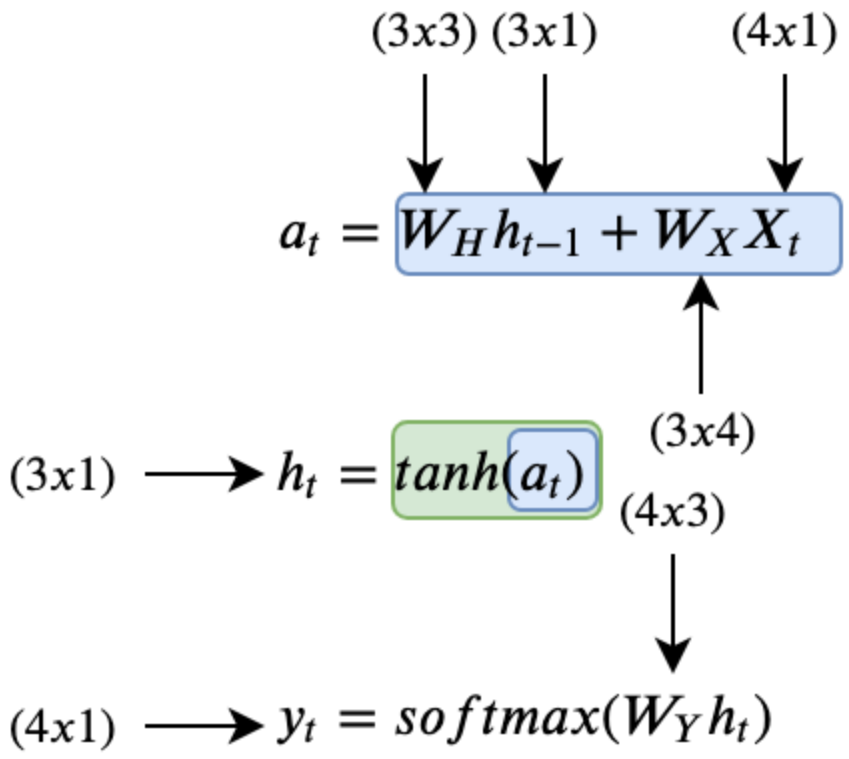
A trivial example

Take the word **"dogs,"** where we want to train an RNN to predict the letter **"s"** given the letters **"d"-"o"-"g"**. The architecture above would look like the following:



RNN architecture predicting the letter “s” in “dogs”

To keep this example simple, we’ll use 3 hidden nodes in our RNN *(d=3)*. The dimensions for each of our variables are as follows:



where k = 4, because our input *x* is a 4-dimensional one-hot vector for the letters in "dogs."

Forward Propagation

Let's see how a forward propagation would work at time *t=1*. First, we have to calculate the hidden nodes ***a***, then apply the activation function to get ***h***, and finally calculate the **prediction**. Easy!

At **t=1**

$$a_1 = \begin{pmatrix} W_{H,11} & W_{H12} & W_{H,13} \\ W_{H,21} & W_{H,22} & W_{H,23} \\ W_{H,31} & W_{H,32} & W_{H,33} \end{pmatrix} \begin{pmatrix} h_{0,1} \\ h_{0,2} \\ h_{0,3} \end{pmatrix} + \begin{pmatrix} W_{X,11} & W_{X,12} & W_{X,13} & W_{X,14} \\ W_{X,21} & W_{X,22} & W_{X,23} & W_{X,24} \\ W_{X,31} & W_{X,32} & W_{X,33} & W_{X,34} \end{pmatrix} \begin{pmatrix} x_{1,1} \\ x_{1,2} \\ x_{1,3} \\ x_{1,4} \end{pmatrix} = \begin{pmatrix} a_{1,1} \\ a_{1,2} \\ a_{1,3} \end{pmatrix}$$

$$h_1 = tanh(\begin{pmatrix} a_{1,1} \\ a_{1,2} \\ a_{1,3} \end{pmatrix}) = \begin{pmatrix} h_{1,1} \\ h_{1,2} \\ h_{1,3} \end{pmatrix}$$

$$y_1 = softmax(\begin{pmatrix} W_{Y,11} & W_{Y,12} & W_{Y,13} \\ W_{Y,21} & W_{Y,22} & W_{Y,23} \\ W_{Y,31} & W_{Y,32} & W_{Y,33} \\ W_{Y,41} & W_{Y,42} & W_{Y,43} \end{pmatrix} \begin{pmatrix} h_{1,1} \\ h_{1,2} \\ h_{1,3} \end{pmatrix}) = \begin{pmatrix} y_{1,1} \\ y_{1,2} \\ y_{1,3} \\ y_{1,4} \end{pmatrix}$$

To make the example concrete, I've initialized random weights for the matrices ***Wx, Wy,*** and **Wh** to provide an example with numbers.

$$a_1 = \begin{pmatrix} 0.1 & 0.5 & 0.1 \\ 0.5 & 0.9 & 0.3 \\ 0.3 & 0.2 & 0.1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0.6 & 0.8 & 0.4 & 0.8 \\ 0.2 & 0.2 & 0.8 & 0.7 \\ 0.9 & 0.8 & 0.1 & 0.2 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.6 \\ 0.2 \\ 0.9 \end{pmatrix}$$

$$h_1 = tanh(\begin{pmatrix} 0.6 \\ 0.2 \\ 0.9 \end{pmatrix}) = \begin{pmatrix} 0.54 \\ 0.20 \\ 0.72 \end{pmatrix}$$

$$y_1 = softmax(\begin{pmatrix} 0.9 & 0.8 & 0.3 \\ 0.2 & 0.3 & 0.4 \\ 0.6 & 0.9 & 0.1 \\ 0.5 & 0.0 & 0.3 \end{pmatrix} \begin{pmatrix} 0.54 \\ 0.20 \\ 0.72 \end{pmatrix}) = \begin{pmatrix} 0.32 \\ 0.21 \\ 0.24 \\ 0.22 \end{pmatrix}$$

At t=1, our RNN would predict the letter "d" given the input "d". This doesn't make sense, but that's ok because we've used untrained random weights. This was just to show the workflow of a forward pass in an RNN.

At t=2 and t=3, the workflow would be analogous except that the *vector h* from *t-1* would no longer be a vector of 0's, but a vector of non-zeros based on the inputs before time *t*. *(As a reminder, the weight matrices Wx, Wh, and Wy remain the same for t=1,2, and 3.*)

It's important to note that while the RNN *can* output a prediction at every single time step, it isn't necessary. If we were just interested in the letter after the input "dog" we could just take the output at t=3 and ignore the others.

Now that we understand how to make predictions with RNNs, let's explore how RNNs learn to make *correct predictions*.

Backpropagation through time (BPTT)

Like their classical counterparts (MLPs), RNNs use the backpropagation methodology to learn from sequential training data. Backpropagation with RNNs is a little more challenging due to the ***recursive nature of the weights and their effect on the loss which spans over time.*** We'll see what that means in a bit.

To get a concrete understanding of how backpropagation works, let's lay out the general workflow:

1. Initialize weight matrices ***Wx, Wy, Wh*** randomly
2. Forward propagation to compute predictions
3. Compute the loss
4. Backpropagation to compute gradients
5. Update weights based on gradients
6. Repeat steps 2–5

Note: *that the output h from the hidden unit is not learned, it is merely the information gained by concatenating the learned weights to previous output h and current input x.*

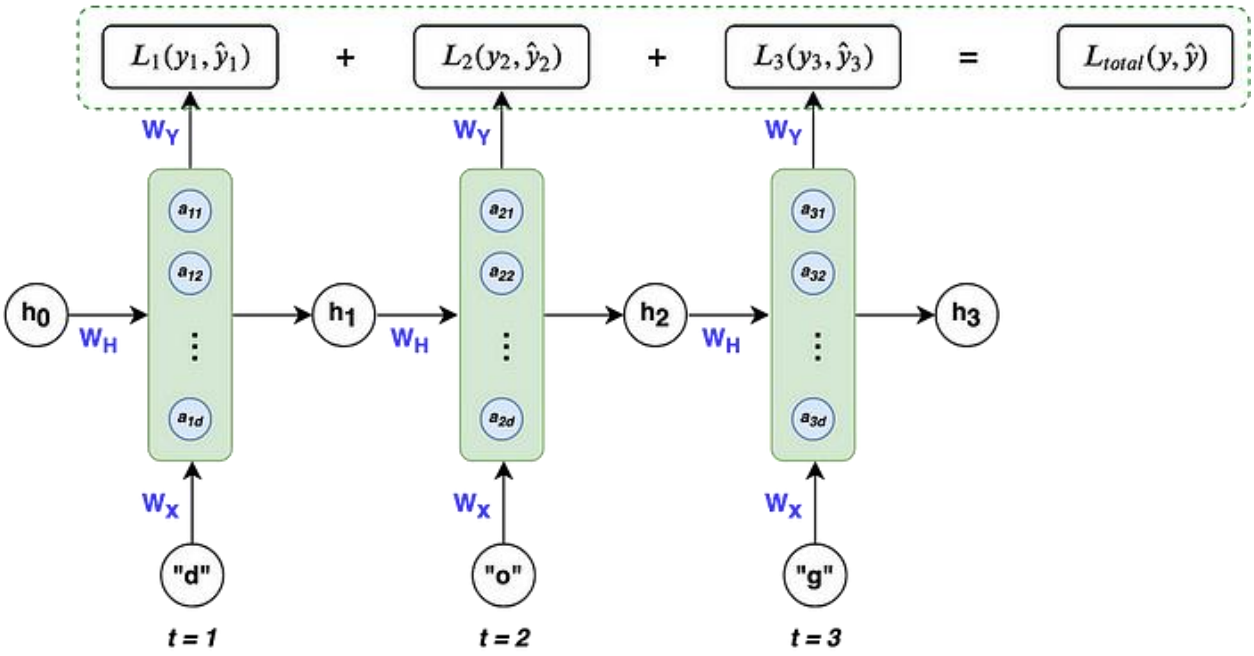
Because this example is a classification problem where we're trying to predict four possible letters ("d-o-g-s"), it makes sense to use the **multi-class cross entropy loss function**:

$$L_t(y_t, \hat{y}_t) = -y_t log(\hat{y}_t)$$

Taking into account all time steps, the **overall loss is**:

$$L_{total}(y, \hat{y}) = -\sum_{t=1}^{n} y_t log(\hat{y}_t)$$

Visually, this can be seen as:



Given our loss function, we need to calculate the gradients for our three weight matrices *Wx, Wy, Wh,* and update them with a learning rate η. Similar to normal backpropagation, the gradient gives us a sense of how the loss is changing with respect to each weight parameter. We update the weights to minimize loss with the following equation:

$$W_i := W_i - \eta \frac{\partial L_{total}(y, \hat{y})}{\partial W_i}$$

where i = x, y, and h as a shorthand for the 3 weight matrices

Now here comes the tricky part, calculating the gradient for *Wx, Wy, and Wh*. We'll start by calculating the gradient for *Wy* because it's the easiest. As stated before, the effect of the weights on loss spans over time. The weight gradient for *Wy* is the following:

Function dependencies with respect to W_Y

$$L_t = -y_t log(\hat{y}_t) \rightarrow \hat{y}_t = softmax(z_t) \rightarrow z_t = W_Y h_t$$

Chain rule with respect to W_Y

$$\frac{\partial L_t}{\partial W_Y} = \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial z_t} \frac{\partial z_t}{\partial W_Y}$$

Gradient for W_Y

$$\frac{\partial L_{total}}{\partial W_Y} = \frac{\partial L_1}{\partial \hat{y}_1} \frac{\partial \hat{y}_1}{\partial z_1} \frac{\partial z_1}{\partial W_Y} + \frac{\partial L_2}{\partial \hat{y}_2} \frac{\partial \hat{y}_2}{\partial z_2} \frac{\partial z_2}{\partial W_Y} + \frac{\partial L_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial z_3} \frac{\partial z_3}{\partial W_Y} = \sum_{t=1}^{n} \frac{\partial L_t}{\partial W_Y}$$

That's the gradient calculation for *Wy*. Hopefully, pretty straight forward, **the main idea is chain rule and to account for the loss at each time step.**

The weight matrices *Wx* and W*h* are analogous to each other, so we'll just look at the gradient for *Wx* and leave *Wh* to you. One of the trickiest parts about calculating *Wx* is the recursive dependency on the previous state, as stated in line (2) in the image below. We need to account for the derivatives of the current error with respect to each of the previous states, which is done in (3). Finally, we again need to account for the loss at each time step (4).

Function Dependencies with respect to W_X

$$L_t = -y_t log(\hat{y}_t) \rightarrow \hat{y}_t = softmax(z_t) \rightarrow z_t = W_Y h_t \rightarrow h_t = tanh(W_H h_{t-1} + W_X X_t) \quad (1)$$

Chain rule with respect to W_x

$$\frac{\partial L_t}{\partial W_Y} = \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial W_x}$$ but note, that within h_t, h_{t-1} also contains W_x, thus we need to recurisvely chain rule h_{t-1} until we reach h_0 (2)

$$\frac{\partial L_t}{\partial W_Y} = \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial W_x} + \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial W_x} + \ldots + \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial h_{t-n}}\frac{\partial h_{t-n}}{\partial W_x} = \sum_{k=0}^{n}\frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial W_x} \quad (3)$$

Gradient for W_X

$$\frac{\partial L_{total}}{\partial W_X} = \sum_{t=1}^{n}\sum_{k=0}^{n}\frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial W_x} \quad (4)$$

And that's backpropagation! Once we have the gradients for *Wx, Wh, and Wy*, we update them as usual and continue on with the backpropagation workflow. Now that you know how RNNs learn and make predictions.

One major problem: vanishing gradients

A problem that RNNs face, which is also common in other deep neural nets, is the **vanishing gradient problem**. Vanishing gradients make it difficult for the model to learn long-term dependencies. For example, if an RNN was given this sentence:

The brown and black dog, which was playing with the cat, was a german shepherd.

(x_2) (x_4) (x_5) (x_{14}) (x_{15})

and had to predict the last two words "german" and "shepherd," the RNN would need to take into account the inputs "brown", "black", and "dog," which are the nouns and adjectives that describe a german shepherd. However, the word "brown" is quite far from the word "shepherd." From the gradient calculation of *Wx* that we saw earlier, we can break down the backpropagation error of the word "shepherd" back to "brown" and see what it looks like:

$$\frac{\partial L_{15}}{\partial \hat{y}_{15}} \frac{\partial \hat{y}_{15}}{\partial z_{15}} \frac{\partial z_{15}}{\partial h_{15}} \frac{\partial h_{15}}{\partial h_2} \frac{\partial h_2}{\partial W_x}$$

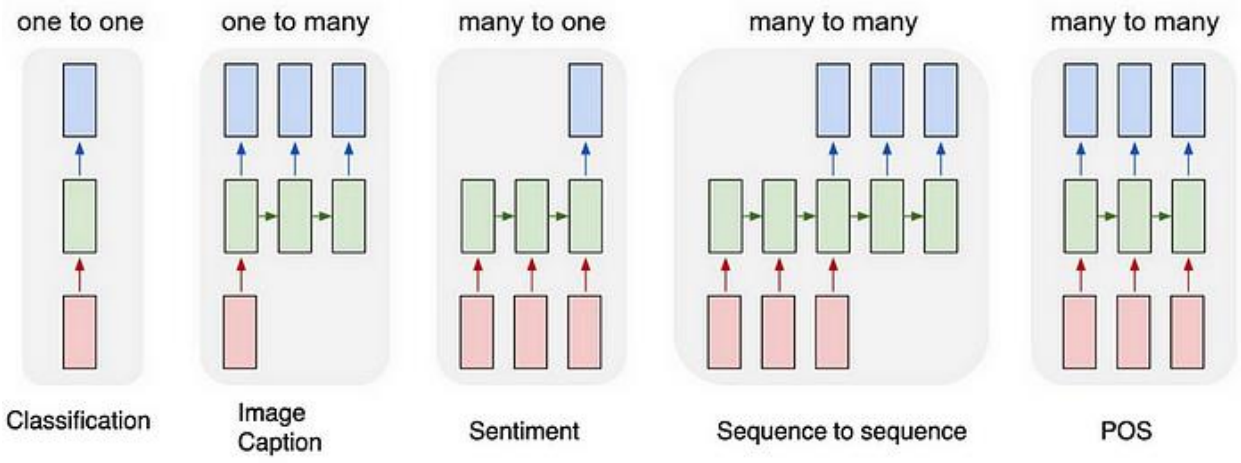
The partial derivative of the state corresponding to the input “shepherd” respective to the state “brown” is actually a chain rule in itself, resulting in:

$$\frac{\partial h_{15}}{\partial h_2} = \frac{\partial h_{15}}{\partial h_{14}} \frac{\partial h_{14}}{\partial h_{13}} \cdots \frac{\partial h_2}{\partial h_1}$$

That’s a lot of chain rule! These chains of gradients are troublesome because if less than 1 they can cause the loss from the word shepherd with respect to the word brown to approach 0, thereby *vanishing*. This makes it difficult for the weights to take into account words that occur at the start of a long sequence. So the word “brown” when doing a forward propagation, may not have any effect in the prediction of “shepherd” because the weights weren’t updated due to the vanishing gradient. This is one of the major disadvantages of RNNs.

However, there have been advancements in RNNs such as gated recurrent units (GRUs) and long short term memory (LSTMs) that have been able to deal with the problem of vanishing gradients.

Types of RNN architectures:



One to One

This type of RNN behaves the same as any simple Neural network it is also known as Vanilla Neural Network. In this Neural network, there is only one input and one output.

One To Many

In this type of RNN, there is one input and many outputs associated with it. One of the most used examples of this network is Image captioning where given an image we predict a sentence having Multiple words.

Many to One

In this type of network, Many inputs are fed to the network at several states of the network generating only one output. This type of network is used in the problems like sentimental analysis. Where we give multiple words as input and predict only the sentiment of the sentence as output.

Many to Many

In this type of neural network, there are multiple inputs and multiple outputs corresponding to a problem. One Example of this Problem will be language translation. In language translation, we provide multiple words from one language as input and predict multiple words from the second language as output.

Advantages and Disadvantages of Recurrent Neural Network

Advantages

1. An RNN remembers each and every piece of information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long short term memory.
2. Recurrent neural networks are even used with convolutional layers to extend the effective pixel neighborhood.

Disadvantages

1. Vanishing and exploding gradient problems.
2. Training an RNN is a very difficult task.
3. It cannot process very long sequences if using Tanh or Relu as an activation function.

Applications of Recurrent Neural Network

1. Language Modelling and Generating Text
2. Speech Recognition
3. Machine Translation
4. Image Recognition, Face detection

5. Time series Forecasting

Variation Of Recurrent Neural Network (RNN)

To overcome the problems like vanishing gradient and exploding gradient descent several new advanced versions of RNNs are formed some of these are as;

1. Bidirectional Neural Network (BiNN)
2. Long Short-Term Memory (LSTM)

Bidirectional Neural Network (BiNN)

A BiNN is a variation of a Recurrent Neural Network in which the input information flows in both direction and then the output of both direction are combined to produce the input. BiNN is useful in situations when the context of the input is more important such as NLP tasks and Time-series analysis problems.

Long Short-Term Memory (LSTM)

Long Short-Term Memory works on the read-write-and-forget principle where given the input information network reads and writes the most useful information from the data and it forgets about the information which is not important in predicting the output. For doing this three new gates are introduced in the RNN. In this way, only the selected information is passed through the network.