Sequential Data

Sequential data refers to a type where the order and sequence of individual elements are significant. It is a collection of observations or events arranged in a specific sequence or time series.

Examples of sequential data include:

1. Time series data: Data collected at successive time points, such as stock prices, weather measurements, or sensor readings.
2. Sequences of text: Sentences, paragraphs, or documents where the order of words matters, such as natural language sentences, genetic sequences, or code.
3. Music and audio signals: Musical notes or audio samples arranged in a specific order, such as melodies or speech recordings.
4. Video frames: Sequential images forming a video stream, such as frames of a movie or surveillance footage.
5. Clickstream data: Logs of user actions on a website or application, capturing the order of clicks and interactions.
6. DNA sequences: The arrangement of nucleotides in a strand of DNA, which carries genetic information.

Recurrent Neural Networks (RNNs) are a popular type of neural network architecture commonly used for processing sequential data. They are designed to handle sequential data by processing it one element at a time while maintaining an internal memory of the previous elements. This memory allows them to capture dependencies and patterns in the sequence.

Other methods of analyzing and modeling sequential data often involves techniques such as time series analysis, sequence prediction, hidden Markov models (HMMs), etc.

RNN models can be of different types based on their input and output configurations:

1. Many-to-Many (Seq2Seq). Eg: Language translation .
2. Many-to-One. Eg: Sentiment Analysis.
3. One-to-Many. Eg: Image Captioning, metadata extraction.
4. Many-to-Many with Attention: Speech Recognition
5. One-to-one. Eg : Classification or Characterization.

Recurrent Neural Networks (RNN)

What differentiates Recurrent Neural Networks from Feedforward Neural Networks also known as Multi-Layer Perceptrons (MLPs) is how information gets passed through the network. While Feedforward Networks pass information through the network without cycles, the RNN has cycles and transmits information back into itself. This enables them to extend the functionality of Feedforward Networks to also take into account previous inputs $\mathbf{X}_{0:t-1}$ and not only the current input $\mathbf{X}_t$.

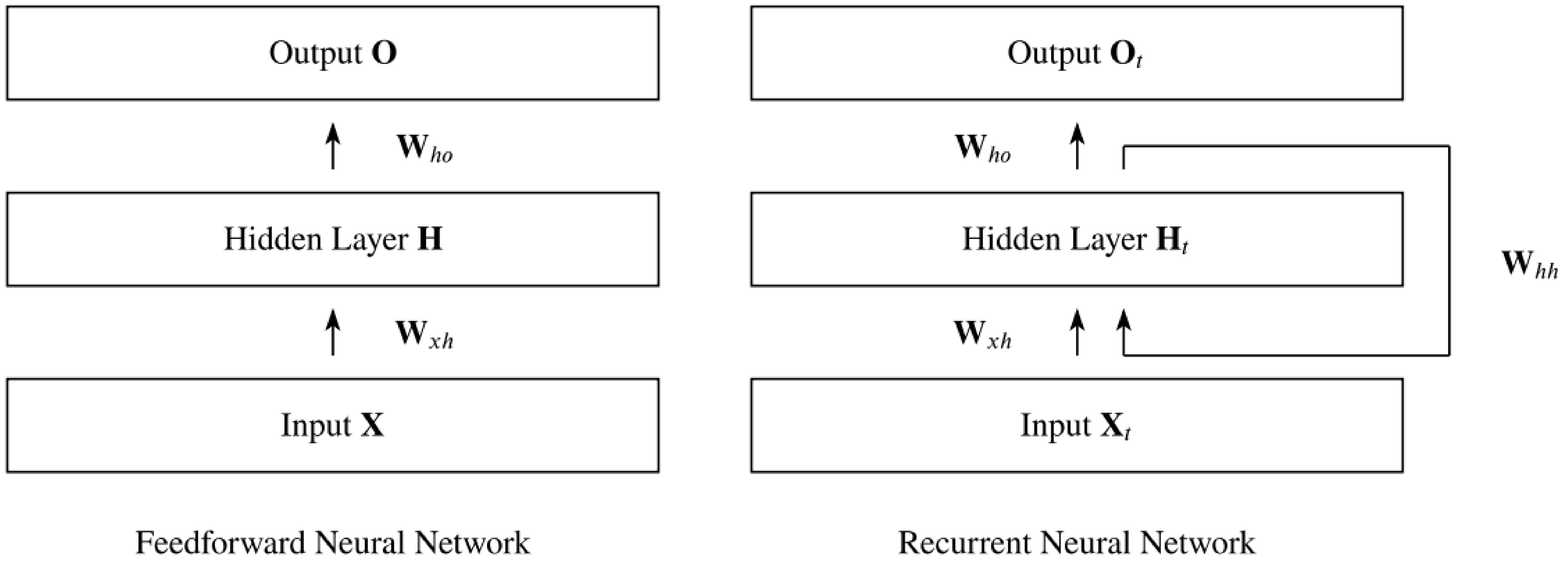
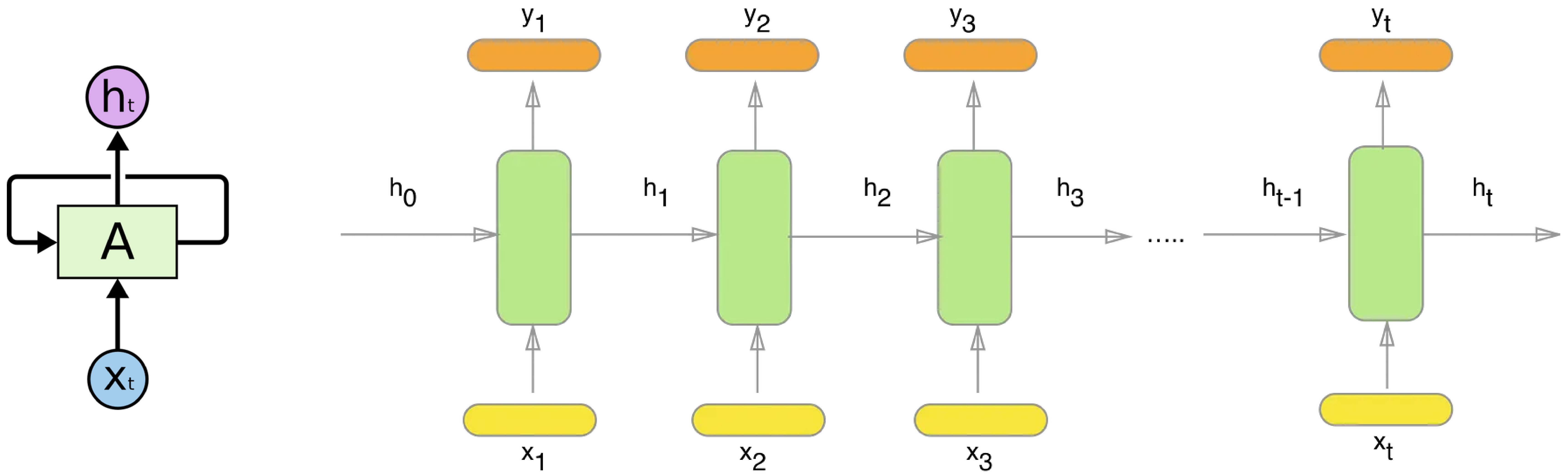


Figure 1: Visualisation of differences between Feedfoward NNs und Recurrent NNs

To understand the architecture of an RNN, it can be helpful to consider a simple example, such as predicting the next word in a sentence. In this case, the RNN would take in a sequence of words as input, ***one at a time***. The network would process the current input word and information from the previous time step(s) at each time step. This information would be passed through the network via the loop, allowing the network to maintain a memory of the previous words in the sequence.



The basic building block of an RNN is the recurrent neuron, which is similar to a standard neural network neuron, but with additional input from the previous time step. This input is typically referred to as the hidden state or the memory of the neuron. The hidden state is updated at each time step, based on the current input and the previous hidden state, and is then passed forward to the next time step.

How do we input a word or sentence to a neuron ?

Since plain text cannot be used in a neural network, we need to encode the words into vectors. There are two ways that we can follow :

1. The best approach is to use word embeddings using libraries like word2vec or GloVe or encoders of popular HuggingFace transformer models.
2. Or we can make a vocabulary of words in our text and make one-hot encoded vectors for each word. The tokenize function of tensorflow does that. Suppose there are V number of words in our vocabulary. Then for each word, we can generate vectors of shape (V,1) where all the values are 0, except the one at the i-th position where i is the position of the word in the vocabulary. For example, if our vocabulary is apple, apricot, banana, …, king, … zebra and the word is banana, then the vector is [0, 0, 1, …, 0, …, 0].

Forward Propagation in RNN

Equations :

1) $h_t = f(W^{(hh)}h_{t-1} + W^{(hx)}x_t)$

2) $y_t = softmax(W^{(S)}h_t)$

3) $J^{(t)}(\theta) = \sum_{i=1}^{|V|} (y'_{t_i} log\, y_{t_i})$

1. Holds information about the previous words in the sequence. As you can see, h_t is calculated using the previous h_(t-1) vector and current word vector x_t. We also apply a non-linear activation function f (usually *tanh* or *sigmoid*) to the final summation. It is acceptable to assume that h_0 is a vector of zeros.
2. Calculates the predicted word vector at a given time step t. We use the *softmax function* to produce a (V,1) vector with all elements summing up to 1. This probability distribution gives us the index of the most likely next word from the vocabulary.
3. Uses the *cross-entropy* loss function at each time step t to calculate the error between the predicted and actual word.

Suppose we are doing a sentiment analysis task, as inputs we have sentences of different number of words and as outputs we have either 0 or 1. Let's consider a sentence of 4 words. At first, we will separate out the words and create embedding vectors for each word. Let's say, we used word2vec model, so our vectors have shape (1,300). We will represent the word vector as a column (take transpose) of shape (300,1). We have X0, X1, X2, X3 : inputs.

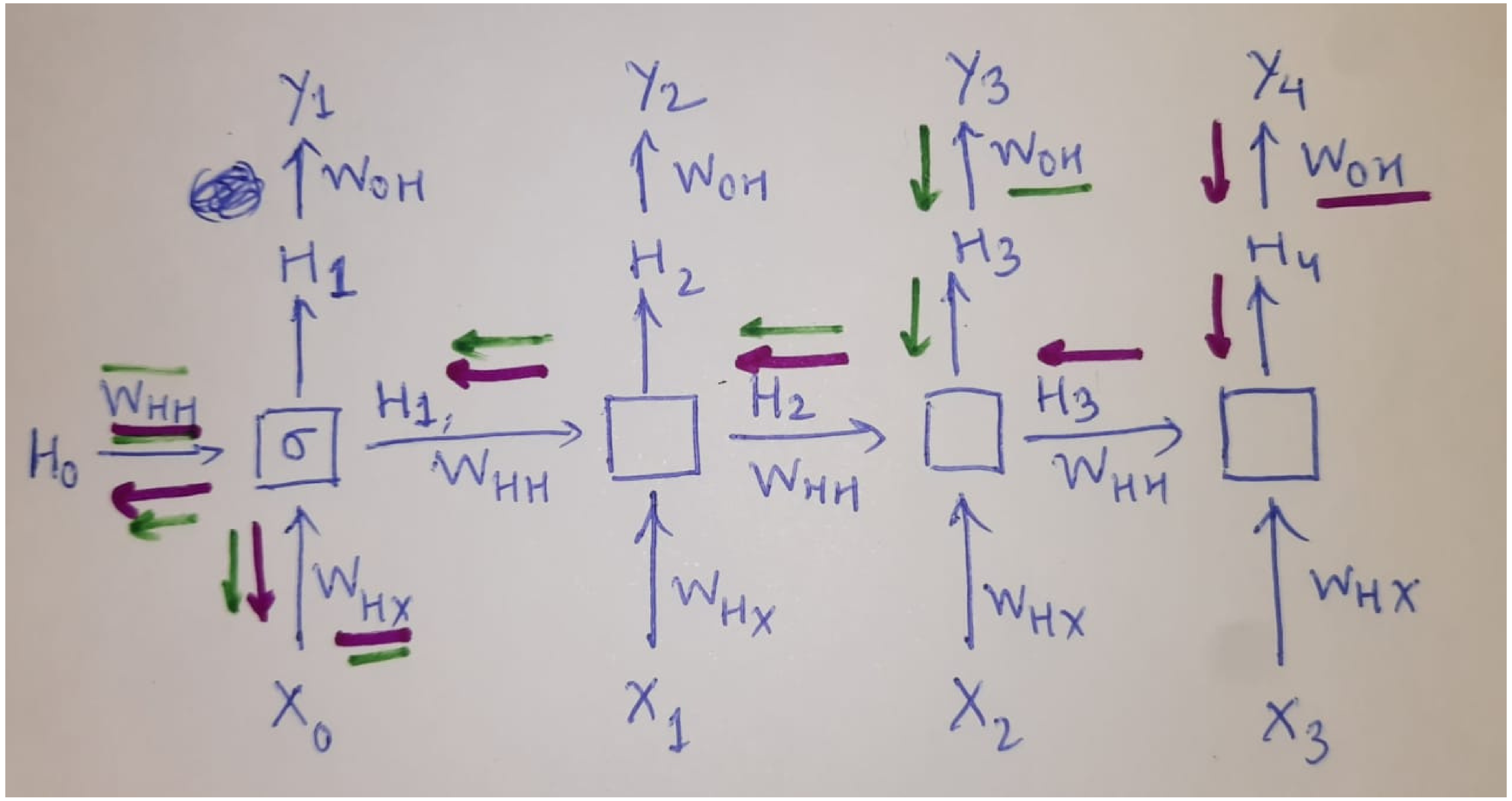
Now let's look at the sizes of the different weight matrices. First we need to consider a size for the output of the neuron. Let it be 128. So, the input-to-hidden weight matrix Whx will be of size (128,300) . Whx * Xi will give (128,1) columns.
The outputs (Hi) will have shape (128,1) . So the recurrent weight matrix Whh will have shape (128,128). Whh * Hi will give (128,1) columns. Now they are added. Then an activation function, suppose sigmoid acts on the sum. The output is H(i+1) .

This output will be multiplied with a hidden-to-output matrix Woh of shape (2,128) and then softmax activation is applied. We get a column matrix with 2 values. If the highest value is the first one, then result is 0, else 1.

Going for forward propagation , the equations will be :

1. ***H1 = sigmoid (Whx * X0 + Whh*H0) , y1 = softmax (Woh*H1). E1 = LL(y,y1)***
2. ***H2 = sigmoid (Whx * X1 + Whh*H1) , y2 = softmax (Woh*H2). E2 = LL(y,y2)***
3. ***H3 = sigmoid (Whx * X2 + Whh*H2) , y3 = softmax (Woh*H3). E3 = LL(y,y3)***
4. ***H4 = sigmoid (Whx * X3 + Whh*H3) , y4 = softmax (Woh*H4). E4 = LL(y,y4)***

To find gradients using backpropagation, we will iterate through each time step:



Back Propagation through time in RNN

The errors at each time step is calculated and summed up to get the total error.
Initially , dC/dWhx = zeros of shape Whx , same for Woh and Whh.

For the final time step,

1. Cost = - (y ln(y4) + (1-y) ln(1 - y4). , y4 =softmax (Woh * H4) .
2. Now , dC/dy4 = (y4-y)/y4(1-y4) .
3. Interestingly, **dC/dWoh += (y4-y)*H4.T** [(2,128) = (2,1)*(1,128)]
4. dC/dH4 = Woh.T * (y4-y) [(128,1) = (128,2)*(2,1)]

5. delta = dC/dH4 * H4(1-H4)= Woh.T * (y4-y) *H4(1-H4)

6. dC/dWhx += dC/dH4 * [dH4/dWhx + dH4/dH3*dH3/dWhx +
dH4/dH2*dH2/dWhx + dH4/dH1*dH1/dWhx]

6. dC/dWhh += dC/dH4 * [dH4/dWhh + dH4/dH3*dH3/dWhh +
dH4/dH2*dH2/dWhh + dH4/dH1*dH1/dWhh]

To simplify this : run a loop from current time step t to 1 , and inside it :

1. dC/dWhh + = delta*H(t-1).T
2. dC/dWhx += delta*X(t-1).T
3. delta = (Whh* delta)* H(t-1)*(1- H(t-1))

For the 3rd,2nd and 1st first time step, similarly repeat the steps .

And finally update the Whx, Whh and Woh and run next epoch .
Too complicated !!!

Problems with RNN

The common problems with traditional RNNs are the vanishing gradient and the exploding gradient problems, where the gradients used in backpropagation become very small or very large as they are propagated back through the network over time. Which of the two phenomena occurs depends on whether the weight of the recurrent edge $|W_{hh}| > 1$ or $|W_{hh}| < 1$ and on the activation function in the hidden node. Given a sigmoid activation function, the vanishing gradient problem is more pressing, but with a rectified linear unit max(0, x), it is easier to imagine the exploding gradient.

Truncated backpropagation through time (TBPTT) is a solution to the exploding gradient problem, but it sacrifices the ability to learn long-range dependencies.

For the vanishing gradient problem, variations of RNNs such as, Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) is a solution. They have mechanisms to better control the flow of information through the network and mitigate the vanishing gradient problem.

NICE TO MEET YOU
I ALREADY
FORGOT YOUR NAME

Vocabulary of Words

Notebook : https://colab.research.google.com/drive/1tJ348aedj7SD9yKgGJ_ZFIY4oQ_zp23d?usp=sharing

Tutorial : https://youtu.be/TsXR7_vtusQ

Word2Vec

Notebook : https://colab.research.google.com/drive/1dFA7h9BH3DyVKcRva4TZ-3fD37xe-HGZ?usp=sharing