SEQUENCE MODELS

Presentation by **Pooja Ravi** & **Aditya Shukla**

NeuRes
DS Community
SRM

Today's Presentation

What is sequential data? Why sequential models?

Recurrent Neural Networks (RNNs): Intuition & Working

- What does the code for an RNN cell look like?

Gated Recurrent Units (GRUs): Intuition & Working

- How they memorize long-range dependencies

Long Short Term Memory Networks (LSTMs): Intuition & Working

- What does the code for an LSTM cell look like?

Recent Advancements in Sequence Models

- Encoder-decoders and the Attention Mechanism

SPEAKERS

Aditya Shukla

Associate Technical Director

Pooja Ravi

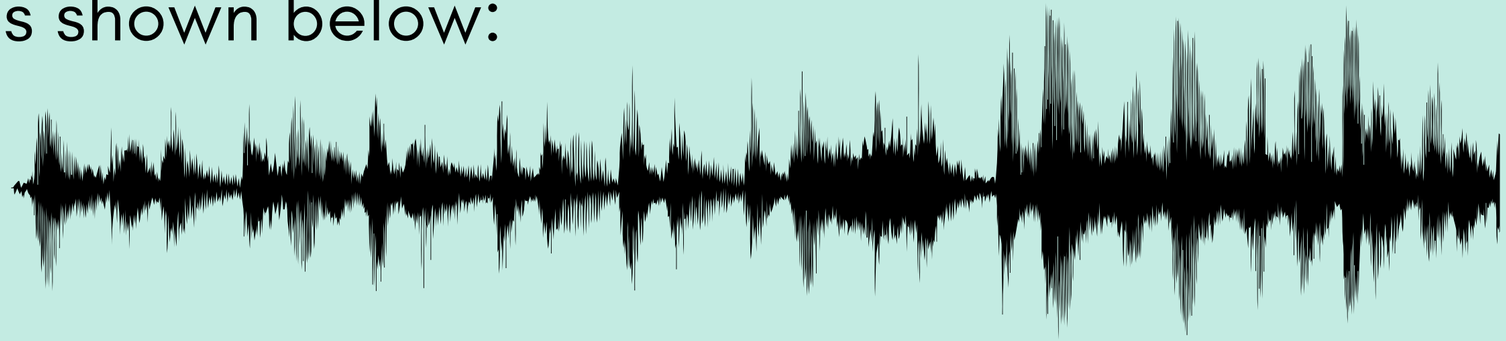
Deep Learning Researcher

What is sequential data?

- Sequential data, as the name suggests, is arranged in a certain order, without which, the data would be deemed meaningless. This order must be taken into consideration by the model.

This sentence is a sequence of words.

- Sequence models deal with sequential data. An example of the same could be text, audio clips, time series data, etc. **An audio waveform (sequential)** is shown below:



What are sequence models?

- Sequence models are deep learning architectures that are capable of making sense of sequential data.

- They need to be able to understand and retain context in data. But how? Using ‘feedback' loops.

- Such loops help the model prioritize the order of the input (which does not happen in artificial neural networks).

- To do this, they need to be able to take into consideration the current input and the output from the previous computation.

Why the need for a separate architecture?

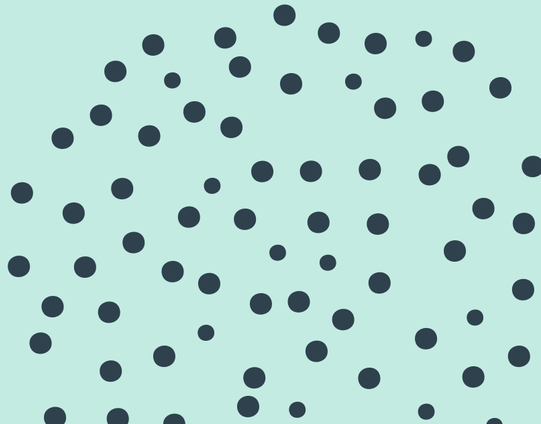
- Multi layer perceptrons, CNNs, etc. are each idiosyncratic in their own ways and are suited to one class of problems. For eg., the CNN emphasizes on the regional aspect of data i.e. the proximity of inputs.

- Sequential models, however, such as RNNs preserve the semantic information over time, a.k.a. the temporal aspect of data.

- For this reason, such models were invented to handle situations in which the sequence (temporality) is important.

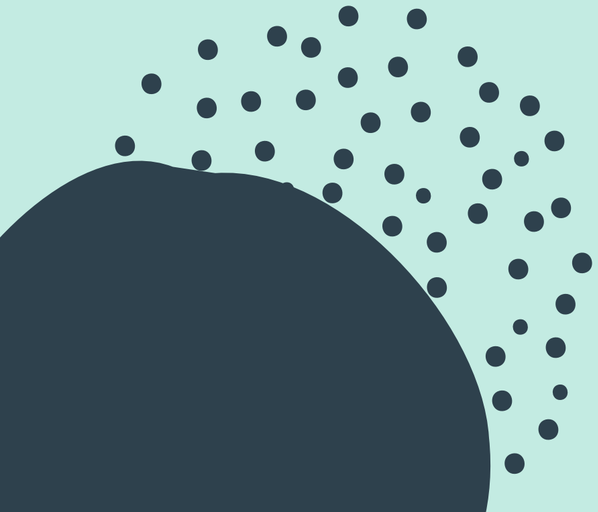
What can sequence models be used to build?

- Sentiment analysis: classifying tweets, texts, reviews, etc. as positive, neutral, negative and so on.
- Speech recognition
- Text summarization
- Time series forecasting
- Music generation
- Image captioning, etc.



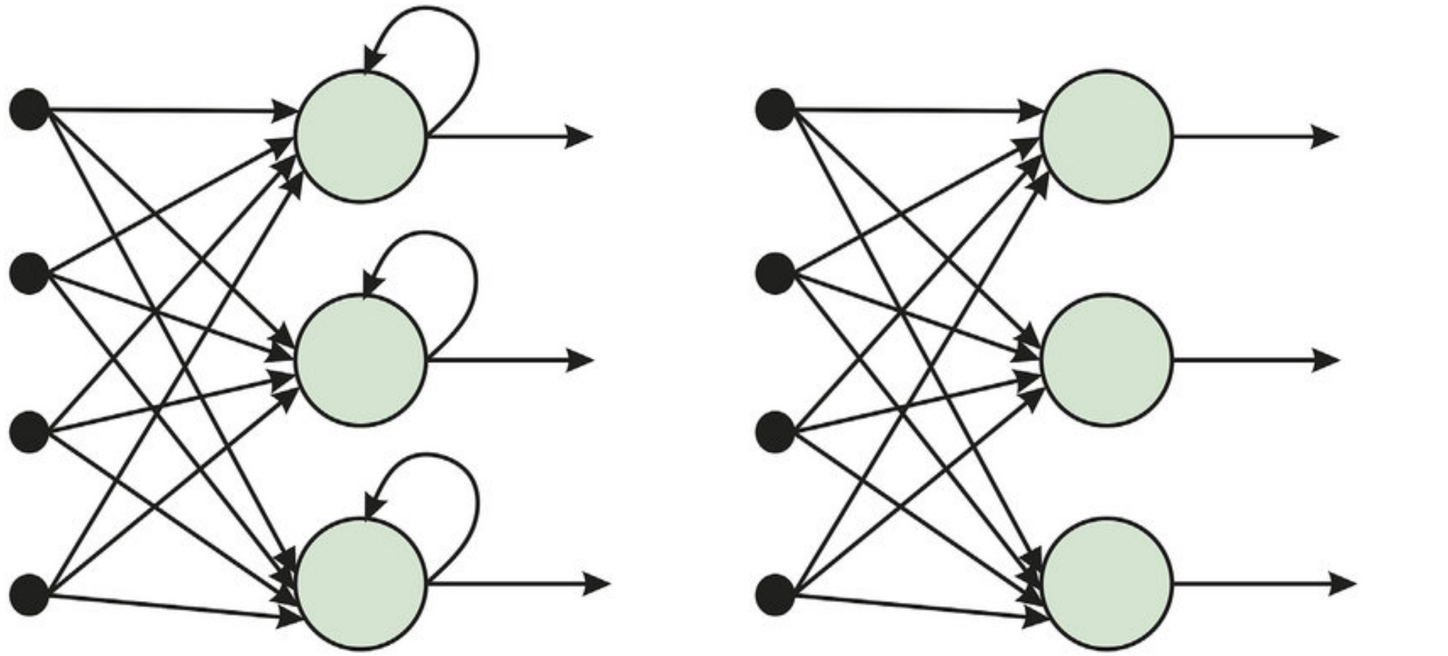
Recurrent Neural Networks (RNNs)

Introducing the concept of "memory"



Introduction

Recurrent neural networks can retain memory or context of earlier inputs by making use of ‘feedback' loops to make better predictions when dealing with sequential data.



(a) Recurrent Neural Network

(b) Feed-Forward Neural Network

The hidden state 😮

- The hidden state (exclusive to Recurrent Neural Networks) is the ‘memory component’ of RNNs. It carries information through time.

- It is usually put through a nonlinear activation function such as tanh so that its value stays comfortably between -1 and 1.

- This hidden state is forwarded to all the time steps and carries the so called ‘memory’ which equips RNNs with the ability to retain information.

Intuition behind RNNs ☹

- We have 3 weight matrices: **Wxh** - input weights, **Whh** - hidden state weights and **Wyh** - output weights. These are the trainable parameters and are shared across time.
- Recurrent neurons employ a **hidden state h_t** which is passed from one neuron to another i.e. throughout all the 'time steps'. The equation for the same is represented as:

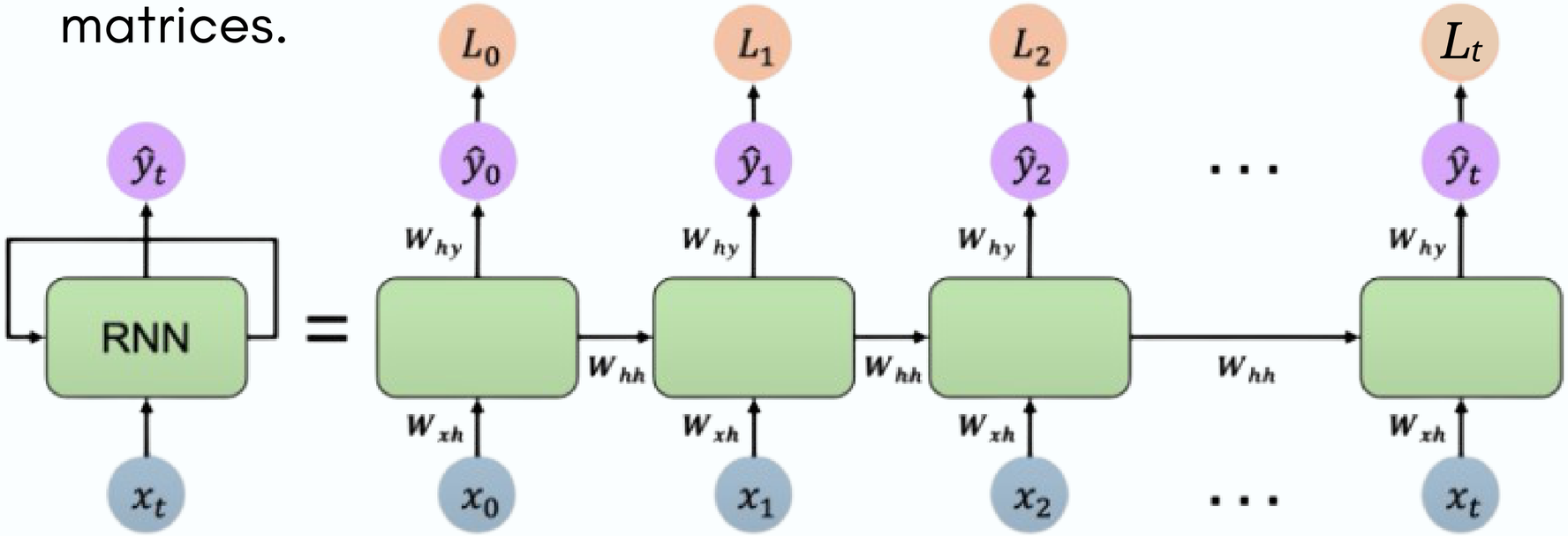
$$h_t = tanh(W_{xh}.x_t + W_{hh}.h_{t-1})$$

  Here, **x_t : current input** and **h_{t-1} : previous hidden state**.
- The output for that time step is then calculated using an activation function over the hidden state along with the output weights.

Intuition behind RNNs ☹

- Finally, the loss for each time step is calculated using y-hat and the ground truth (y), using an appropriate loss function.

- All such loss values are summed up from each time step to obtain the final loss. Our end goal is to minimize this loss by updating the 3 weight matrices.



Backpropagation Through Time (BPTT) ☹ ☹

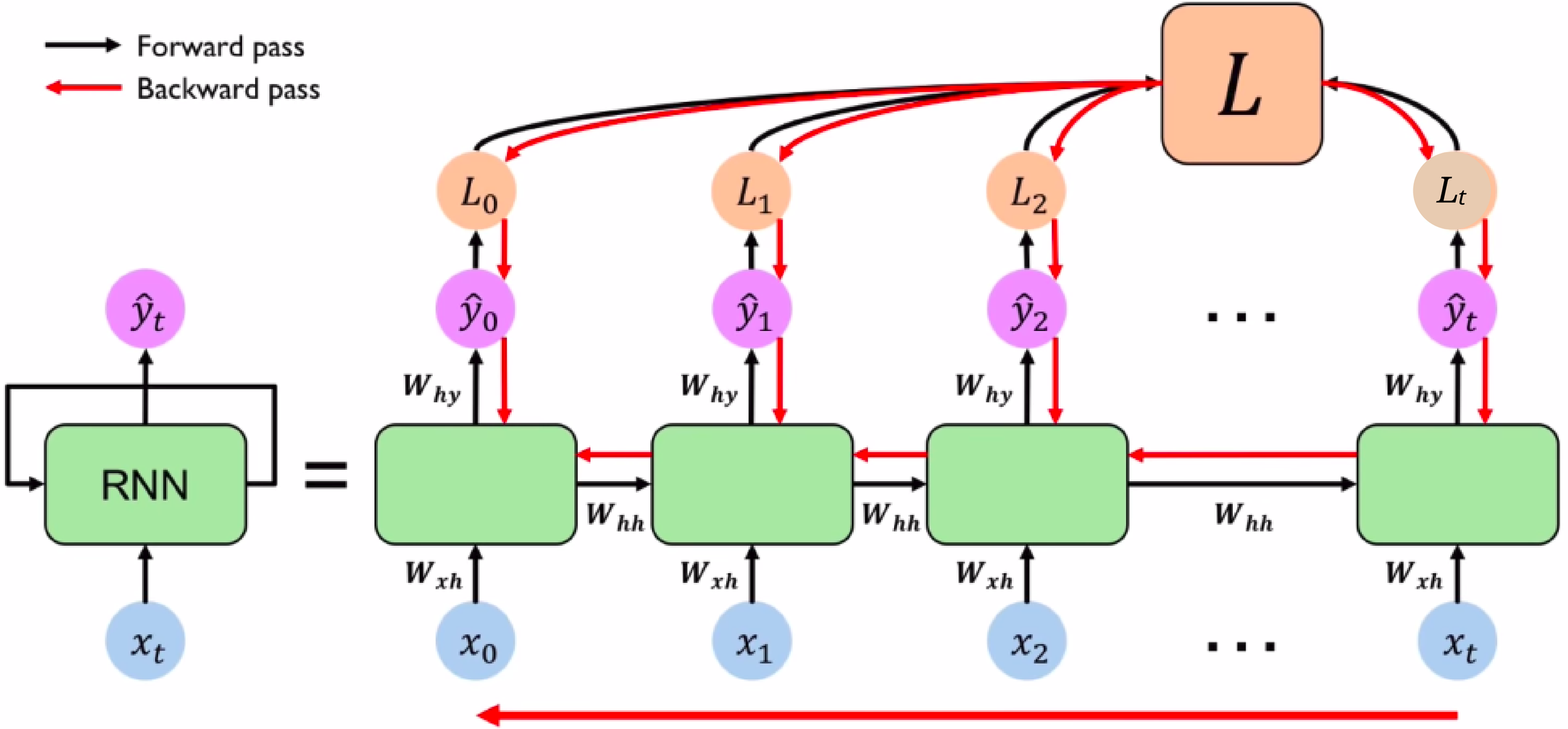
- RNNs make use of a concept called Backpropagation Through Time in order to update the weight matrices and perform better.

- Initially, the feed forward mechanism computes all the predicted outputs, from which the losses are then calculated and summed up.

- Then, the gradient of the final loss needs to be calculated and this is where BPTT begins. It follows a method in which derivatives are calculated (using chain rule) and sent backwards through each time step in order to update the input, cell state and output weight matrices separately.

- Recall that all the weight matrices (for each time step) are shared between layers and hence the updation of weights takes place uniformly.

RNNs: backpropagation through time



```
class RNNCell(tf.keras.layers.Layer):
    def __init__(self, rnn_units, input_dim, output_dim):
        super(RNNCell, self).__init__()

        # Initialize weight matrices
        self.W_xh = self.add_weight([rnn_units, input_dim])
        self.W_hh = self.add_weight([rnn_units, rnn_units])
        self.W_hy = self.add_weight([output_dim, rnn_units])

        # Initialize hidden state to a matrix of zeros
        self.h = tf.zeros([rnn_units, 1])

    def call(self, x):
        # Compute the new hidden state
        self.h = tf.math.tanh(self.W_xh * self.x + self.W_hh * self.h)

        # Compute the output y
        self.y_hat = self.W_hy * self.h

        return y_hat, self.h
```

The Problem with RNNs

- As we have seen so far, the essence of the RNNs was to be able to store memory i.e. information about the past that can inform future decisions.
- But the problem with RNNs is that of vanishing gradients. To summarize this problem: when we back-propagate through time, our gradients become smaller and smaller very rapidly. Gradients are the values which we use to update our parameters, and if they're too small, then parameter updates are painfully slow, and hence learning is slowed down. Visualize this:

  new parameter = parameter - learning_rate * gradient_for_parameter
- So the further back we look in time, the less and less information from then is being remembered, meaning that older information is not given as much importance as newer information.

- This is a big problem; let me elaborate why. Have a look at the following sentence:

  The cats, which already ate all their food, were full. □ □
- The past-tense of the verb 'were' here was informed by the plural 'cats'. Hence, the 2nd word of the sentence influences the 9th word.
- If an RNN was asked to autocomplete what word should have been in the 9th position, it most probably will have made an error. This is because the context that would help it correctly predict 'were' was to be found all the way back at the 2nd timestep.

This precisely, is where the more robust GRUs (and later LSTMs) come into play. GRUs are adept at handling long-range dependencies.

With the GRU, we begin to emphasize this idea of holding onto memory over a long period of time. Let's discuss the intuition behind this.

Gated Recurrent Units (GRUs)

Holding on only to useful memory

Intuition behind GRUs ☹

- GRUs, as the name suggests, introduce the concept of a gate in a recurrent unit. We also reintroduce the memory cell, whose job it is to hold on to useful data which may come in handy in the future.

  But, the novel idea here is this:
- At each timestep, the respective recurrent unit generates a new candidate for the memory cell and offers it to the update gate. It can be thought of as that unit offering some insight from its particular input that can be stored in the memory cell to potentially be used later on.

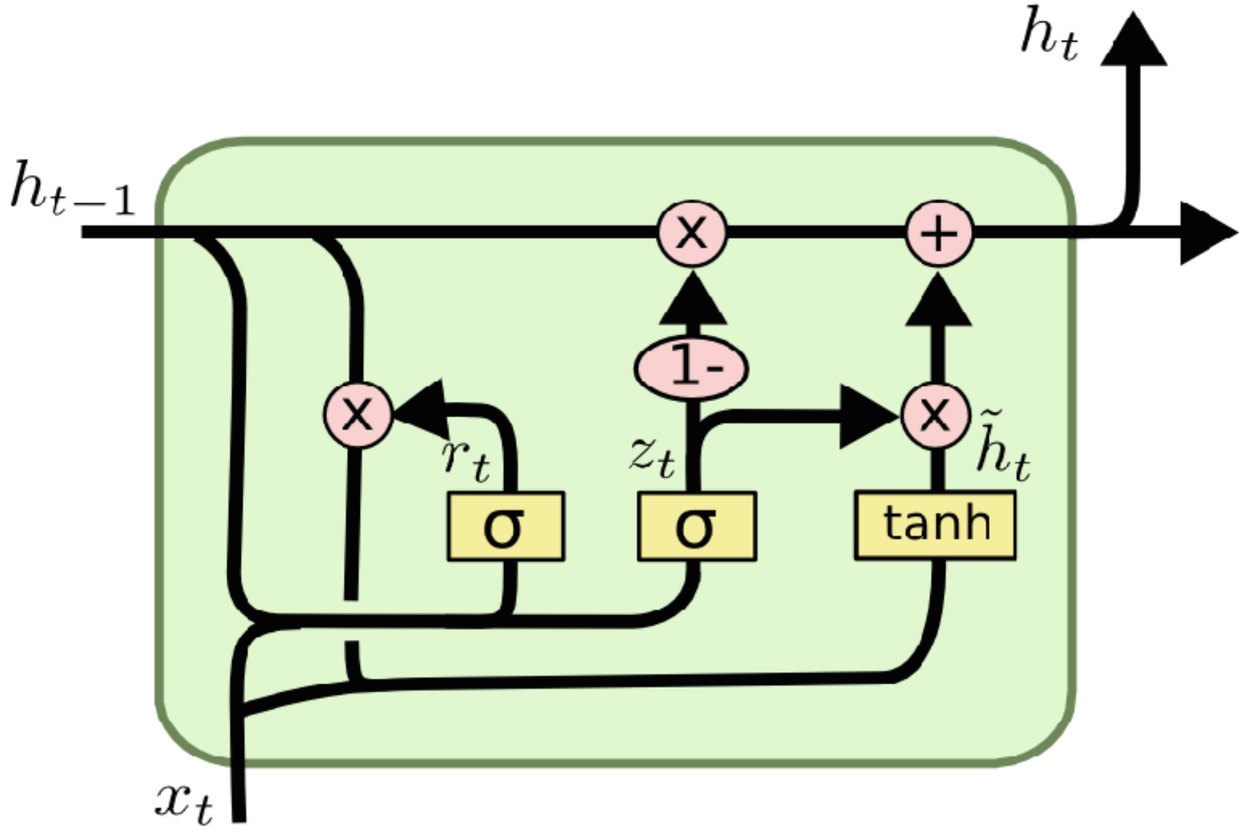
- Let's go back to our cat sentence 😸😸:

  The cats, which already ate all their food, were full.

- Here, each word will be fed to a recurrent cell. The 2nd cell ("cats") will offer some insight from its input (the word "cats") to be stored in the memory cell, and our model (which is trained and now very smart) shall store this insight in the memory cell.
- As we traverse through the rest of the words "which....food, ", each recurrent cell offers its update gate some insight through its respective word, but the update gate declines this offer each time, and finally its memory in the form of "cats" comes in handy when it correctly predicts "...food, were full."

GRU Cell

😵😵😵 (ughh, Maths again)



$$z_t = \sigma(W_z.[h_{t-1}, x_t])$$
$$r_t = \sigma(W_r.[h_{t-1}, x_t])$$
$$\tilde{h}_t = \tanh(W .[r_t * h_{t-1}, x_t])$$
$$h_t = z_t * \tilde{h}_t + (1 - z_t) * h_{t-1}$$

Reset Gate

- We concatenate the memory state from the previous cell, **ht-1** and the input into the current cell, **xt**. Their product is then matrix-multiplied by a **reset weight matrix Wr** to generate the output for the reset gate. This output is then sigmoid'ed and decides how much of the old memory state should be retained as useful information (and the rest discarded).

Update Gate

- We again do the same concatenation between **ht-1** and **xt**. This value is now matrix-multiplied by an **update weight matrix Wz** to generate the output for the update gate. This output is then passed through a sigmoid activation function and decides how much of the new candidate offered by the current input will be taken as the new memory state.

- We then multiply the previous memory state by the reset gate's output, effectively only taking some of its value. We then matrix-multiply this value by our current input, apply a nonlinearity such as Tanh and boom! We've got our brand new candidate **ĥ**.

- Now, with the output our update gate has given us, we plug the respective calculated values into the equation for calculating the final output memory **hidden state h**.

- What we have effectively done is take **zt%** of the new candidate and **(1 - zt)%** of the old memory state.

Long Short Term Memory Networks (LSTMs)

Emphasizing long-range dependencies

Intuition behind LSTMs ☹

Here, we have a total of 3 gates: forget gate, input gate, and the output gate.

Cell State

- In LSTMs, we have a value called cell state in addition to the hidden state. It is important to distinguish between the two because the hidden state is present in RNNs as well but the cell state is an addition of the LSTM model.
- The cell state stores the long-term information about a sequence, that is, it is the primary long-term memory element of the network, whereas the hidden state stores only short-term working memory (as we have seen in RNNs).

Forget Gate

- This gate is similar to the Reset Gate of the GRU. The forget gate decides which data from the previous cells should be kept and which should be discarded or "forgotten".
- We feed in the previous hidden state **ht-1** and current input **xt**, and pass them through a sigmoid. Recall that the closer the values output by the gate are to 0, the more we are going to forget, and the closer they are to 1, the more we are going to remember.

Input Gate

- The input gate decides which data from the current cell input is to be stored in our cell state. So, we take the previous hidden state and the current input, concatenate them and pass them once to a sigmoid, and once to a Tanh.
- The Tanh part gives us the data squished between -1 and 1 (so that values don't explode) while the Sigmoid part which is between 0 and 1 tells us which important part of the Tanh's output is to be stored in the cell state.

Forget Gate

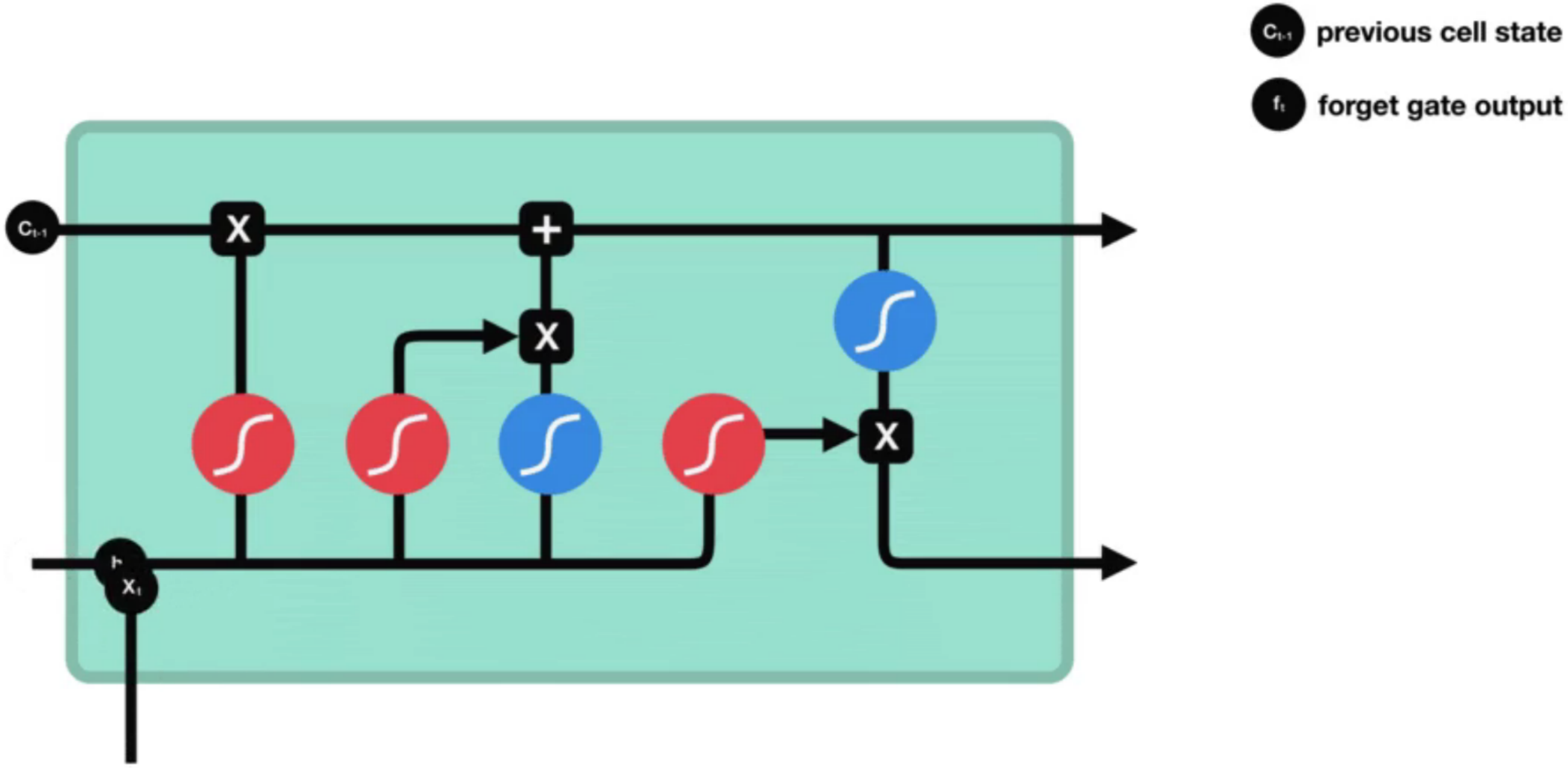
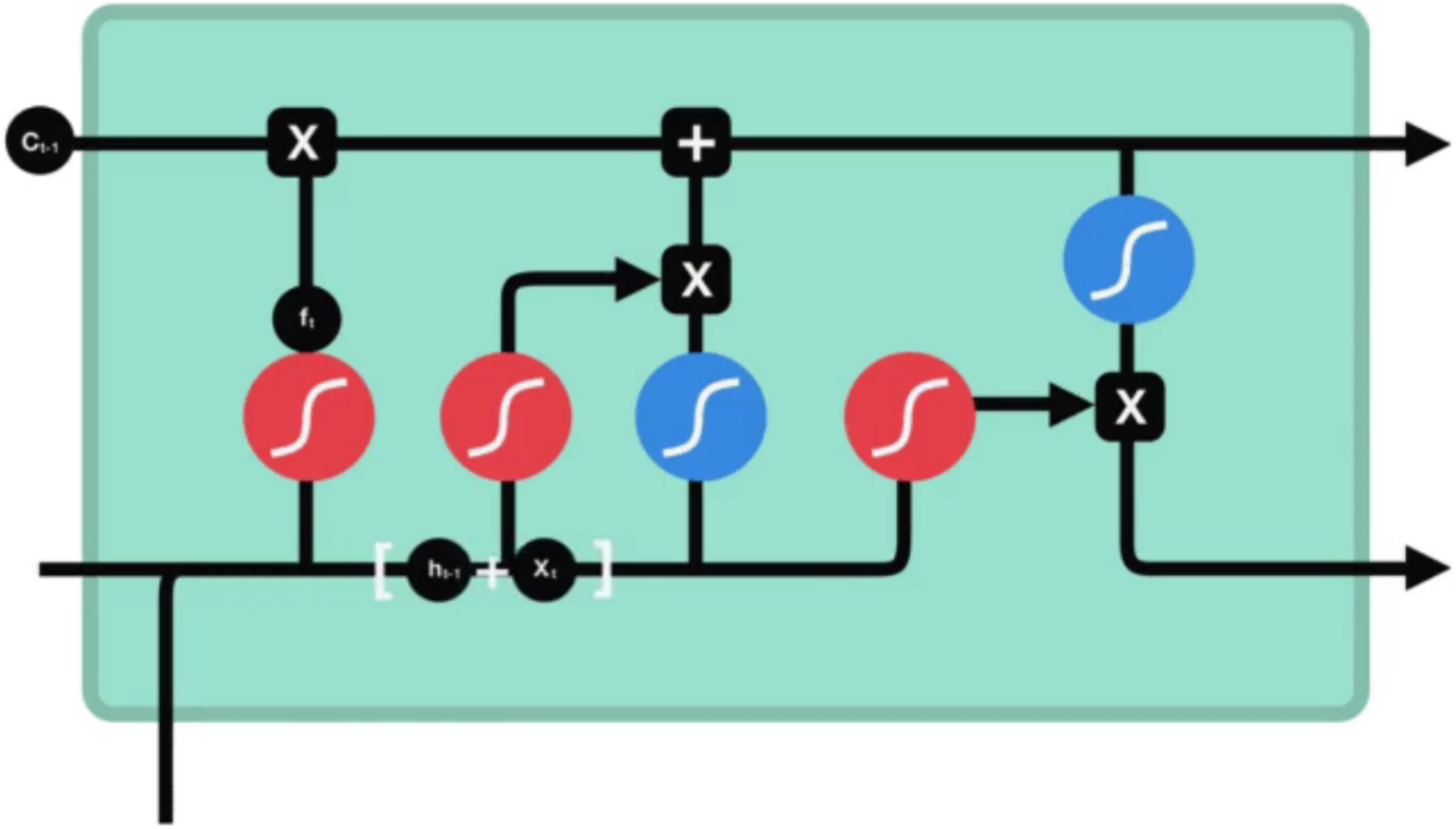


Image Credit: Michael Phi on Medium

Input Gate



C_{t-1} previous cell state

f_t forget gate output

i_t input gate output

$\tilde{C}_t$ candidate

Image Credit: Michael Phi on Medium

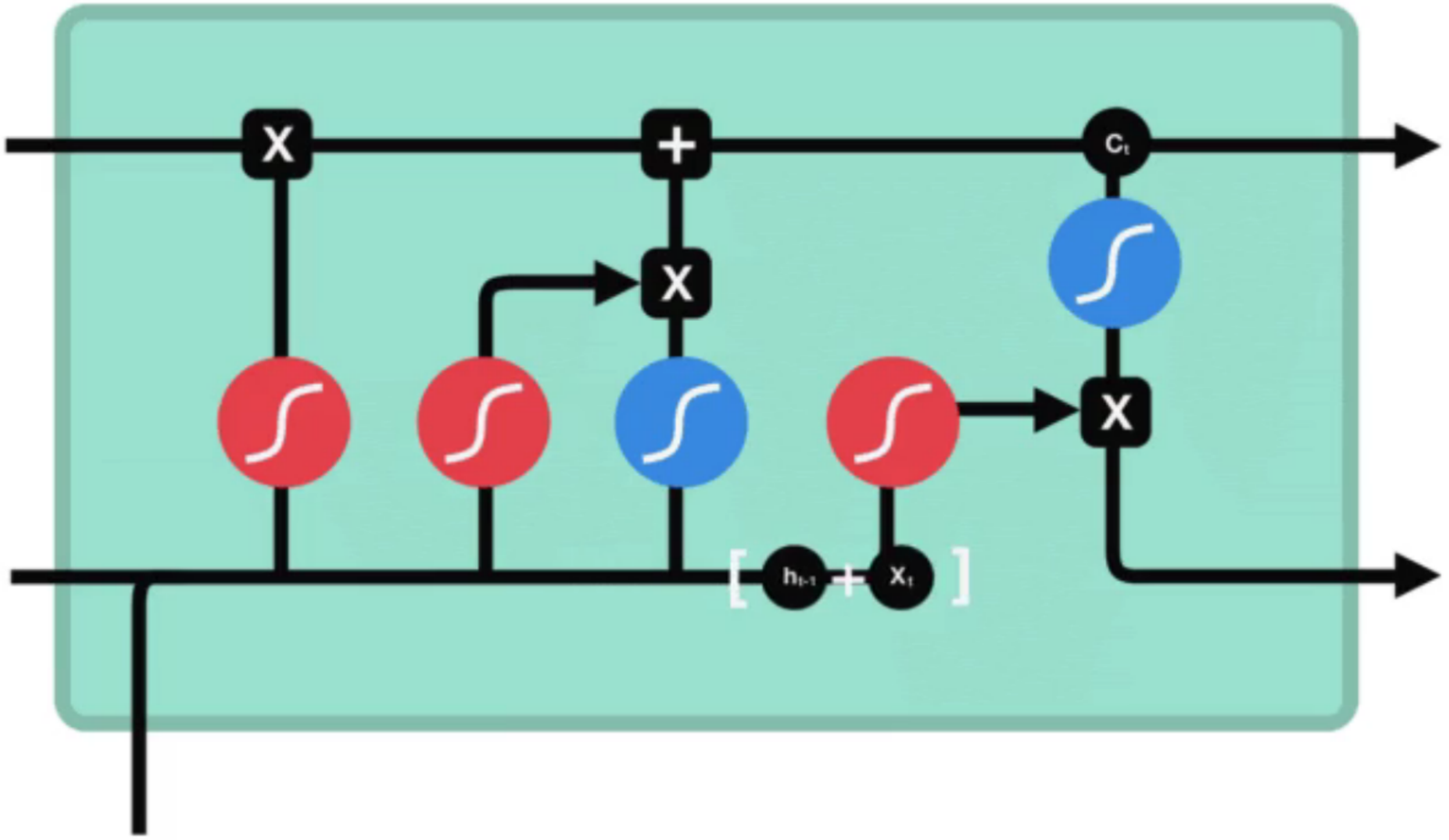
Next Memory State

- The output of the forget gate and the input gate are added together element-wise and this will become the new memory cell state. Recall that this contains data about the long-term dependencies.
- This value will be routed straightaway to the next cell.

Output Gate

- Finally, we have the output gate. Here, we take the current cell input and the previous hidden state, and the cell state which has just been calculated and multiply the two.
- This is the value of the hidden state that we're going to send to the next cell.

Output Gate



c_{t-1} previous cell state

f_t forget gate output

i_t input gate output

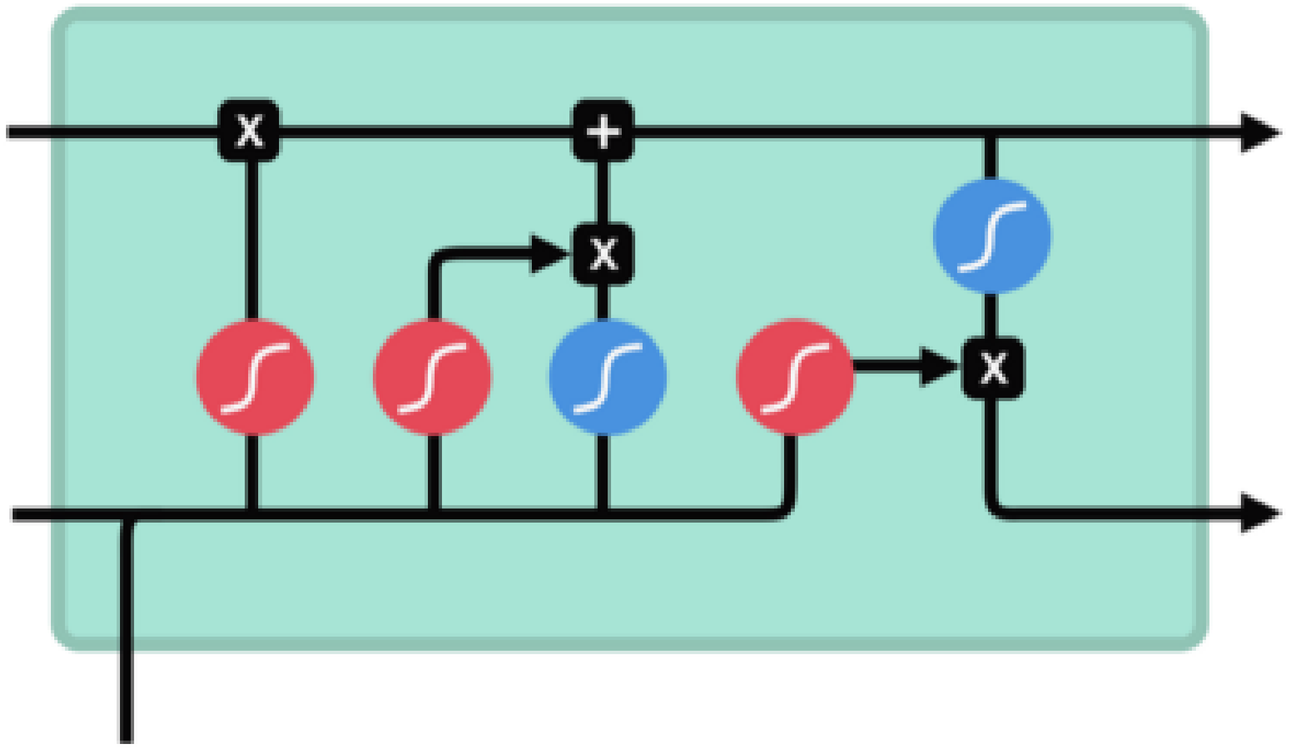
$\tilde{c}_t$ candidate

c_t new cell state

o_t output gate output

h_t hidden state

Image Credit: Michael Phi on Medium

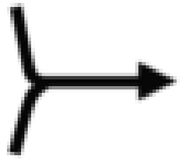


sigmoid

tanh

pointwise multiplication

pointwise addition



vector concatenation

LSTM Cell from Scratch

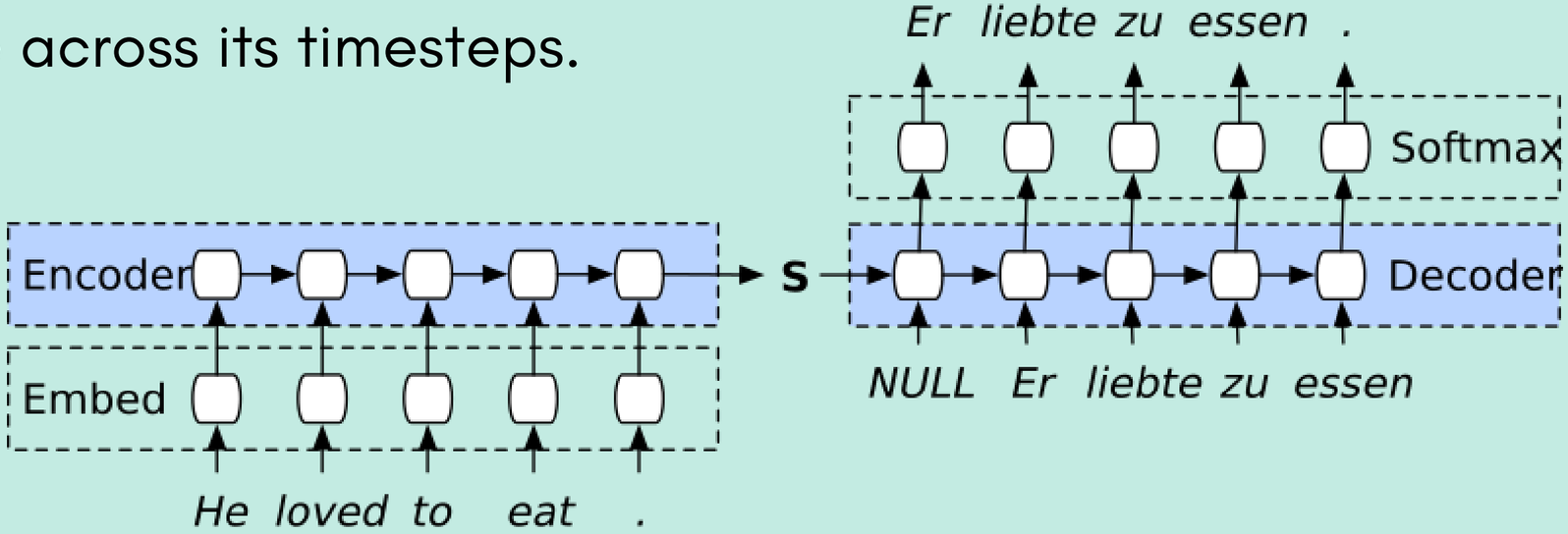
```
def one_lstm_cell():

    #forget gate
    in1          = np.dot(W_forget,[input+prev_h])
    forget_gate = sigmoid(in1)
    memory       = np.multiply(forget_gate,Cprev)

    #input gate
    in2                   = sigmoid(np.dot(W_input,[input+prev_h])
    cell_state_calc      = np.tanh(np.dot(W_cell,[input+prev_h])
    calc1                 = np.multiply(in2,cell_state_calc)
    current_cell_state  = memory + calc1

    #output gate
    in3           = np.dot(W_output,[input+prev_h])
    output        = sigmoid(in3)
    calc2         = np.tanh(current_cell_state)
    hidden_state = np.multiply(output,calc2)
```

```
W_forget - forget gate weight matrix
input    - Xt
prev_h   - previous hidden state (Ht-1)
Cprev    - previous cell state
W_input - input gate weight matrix
W_cell   - cell state weight matrix
current_cell_state - Ct
w_output - output weight matrix
```

Recent Advancements in Sequence Models

Encoder-Decoder Models (Seq2Seq) ➡ ⤴ ➡

- Encoder-decoder models are extensions to the recurrent models we have seen so far. They map a sequence to sequence, both of which may be unequal in length i.e. many-to-many mapping.
- Such models have 3 principle parts: the encoder which receives the input sequence, the intermediate context vector which stores some important semantic information from that input sequence in a compressed representation, and finally the decoder which takes this latent representation and tries to give the output sequence across its timesteps.



Attention Mechanism ☹

- The context vector generated in the encoder-decoder architecture is highly influenced by the latter states i.e. the information it was presented with most recently and fails to pay attention to semantic information earlier back.

- The attention mechanism tries to rectify this by giving relative importance to different parts of the sentence which carry the most semantic meaning, rather than only the last few words. This means that it "pays attention" while reading the input and stores these important insights in the context vector that will be given to the decoder.

And that's it! ☹☹

Thank you for giving us your precious time!

Proudly presented by

The Data Science Community SRM