Deep Learning
CSC-Elective

Instructor : Dr Muhammad Ismail Mangrio

Slides prepared by Dr. M Asif Khan

ismail@iba-suk.edu.pk

Unit 02 NLP Week 3

Contents

- RNN (Recurrent Neural Networks)
- RNN types
- Vanishing gradient problem RNN
- LSTM (Long & Short term Memory) RNN
- GRU (Gated Recurrent Unit) RNN
- Bi-directional LSTM RNN

Recurrent Neural Network: where sequence is regarded highly

"working love learning we on deep"

Does it make sense?

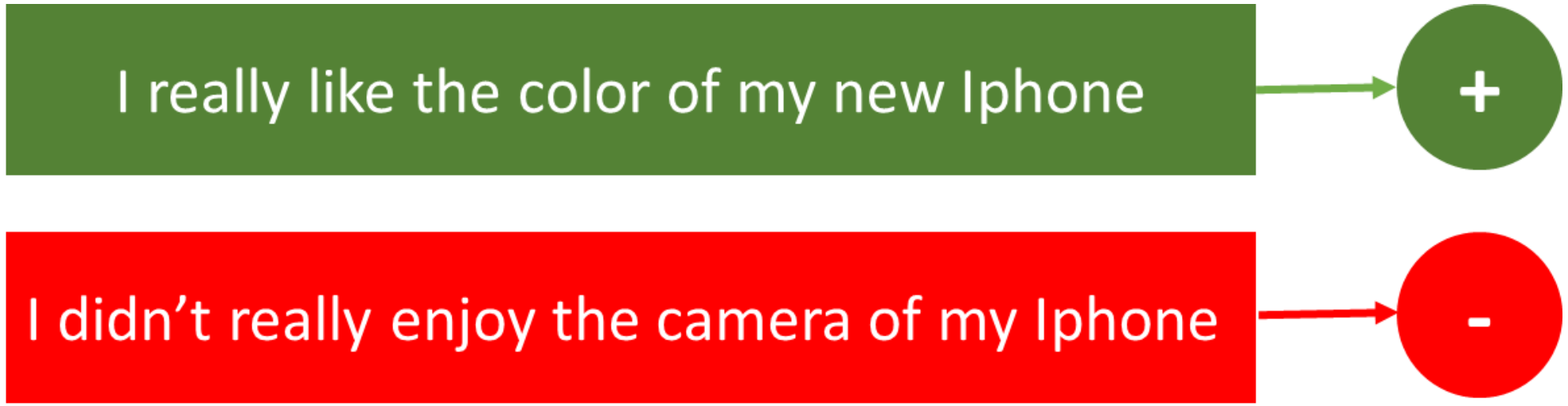
Recurrent Neural Network: where sequence is regarded highly

More sensible, isn't?

"we love working on deep learning"

When words are in correct sequence

RNN- Sentiment Classification



RNN: Image Captioning

A person riding a motorcycle on a dirt road.

Two dogs play in the grass.

A group of young people playing a game of frisbee.

Two hockey players are fighting over the puck

Language Translation

French was the official language of the colony of French Indochina, comprising modern-day Vietnam, Laos, and Cambodia. It continues to be an administrative language in Laos and Cambodia, although its influence has waned in recent years.

Translate into : French

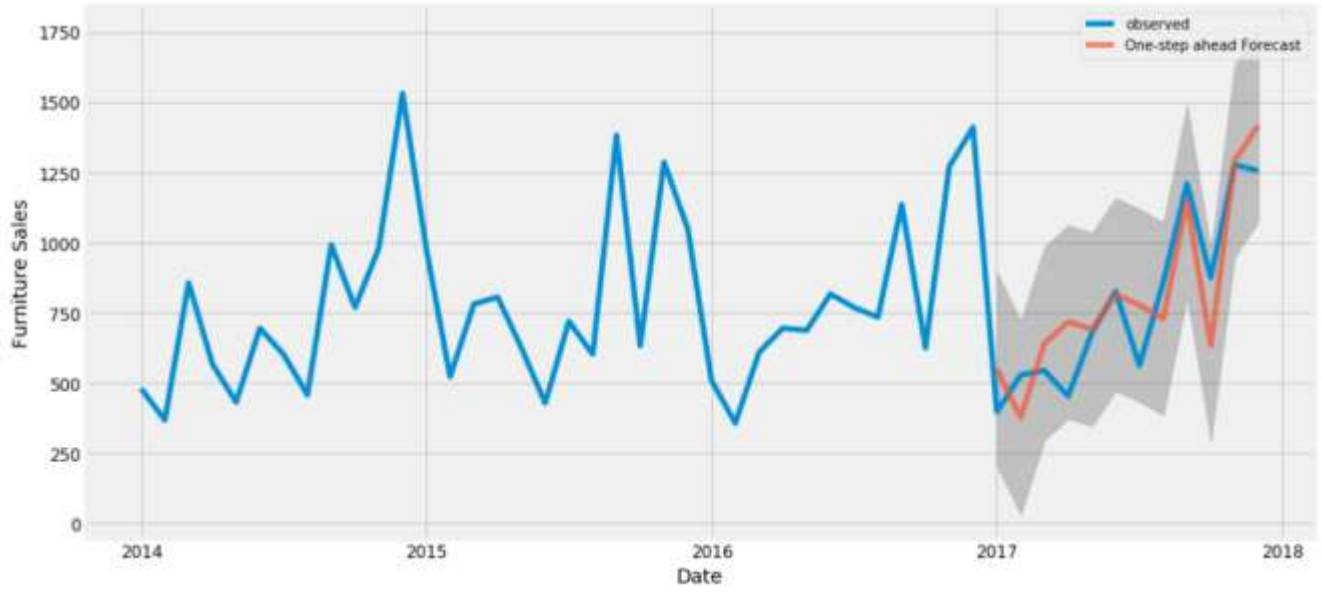
Le français était la langue officielle de la colonie de l'Indochine française, comprenant le Vietnam d'aujourd'hui, le Laos et le Cambodge. Il continue d'être une langue administrative au Laos et au Cambodge, bien que son influence a décliné au cours des dernières années.

Time series data

- **Time Series Data** refers to a **sequence of data points** collected or recorded at **regular time intervals**.
- The data points are ordered in time, and each point corresponds to a specific moment or period in the past, present, or future.
- **Key Characteristics**:
  - **Ordered by Time**: The data is indexed by timestamps, such as minutes, hours, days, months, or years.
  - **Dependence on Time**: Time series data often exhibits **temporal dependencies**, meaning the values at one point in time are related to values at earlier points.

Time series data

- **Examples of Time Series Data:**
  - Stock prices over time.
  - Daily temperature readings.
  - Sales data over a period of months.
  - Monthly unemployment rates.



RNN (Recurrent Neural Networks)

- An RNN has a **memory** (to preserve previous state and of course context)**.**
- A type of NN designed to recognize **sequential data or time-series data**.
- The natural language is also **in sequence** and there is great importance of it to understand its meaning.
- Unlike traditional feedforward neural networks, RNNs have connections that form **cycles** (recurrent, again and again), allowing information to persist.
- RNNs maintain a "**memory**" of previous inputs through their internal state (**hidden state**), making them suitable for tasks like speech recognition, language modeling, and time series forecasting.

"The weather today is very ____."

RNN (Recurrent Neural Networks)

[5, 7, 9]

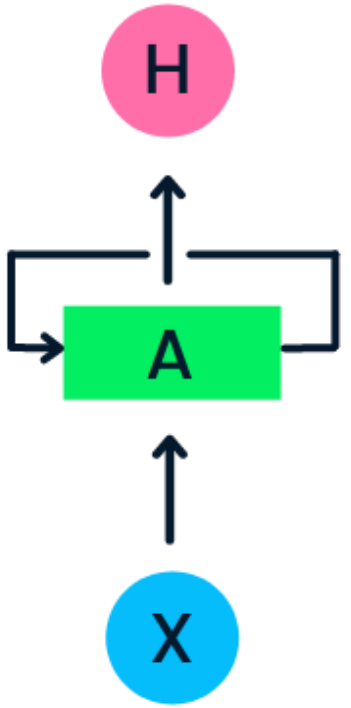
Time Step	Input	Previous Memory	Output
t1	5	none	learns 5
t2	7	remembers 5	learns pattern (5→7)
t3	9	remembers 5 & 7	learns pattern (5→7→9)

- If you ask the RNN:
- “What comes next?”

It may predict **11** because it has seen an increasing pattern.

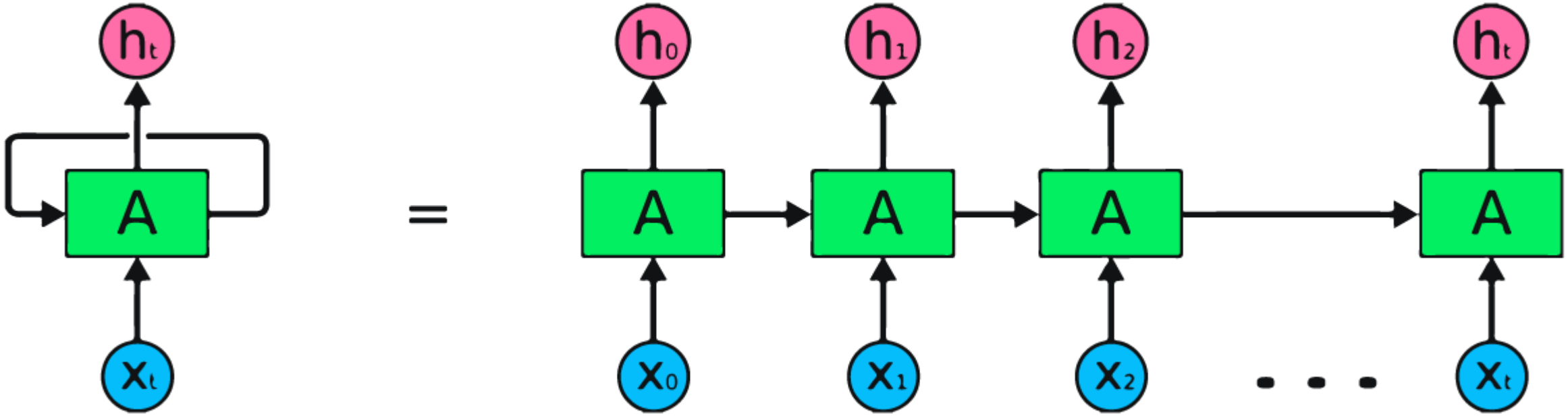
How RNN works

- In RNN, the **information cycles through the loop**, so the output is determined by the **current input** and **previously received inputs**.
- The input layer X processes the initial input and passes it to the middle layer A.
- The middle layer consists of **multiple hidden layers**, each with its **activation** functions, **weights**, and **biases**.
- These parameters are standardized across the hidden layer so that instead of creating multiple hidden layers, it will create one and loop it over.

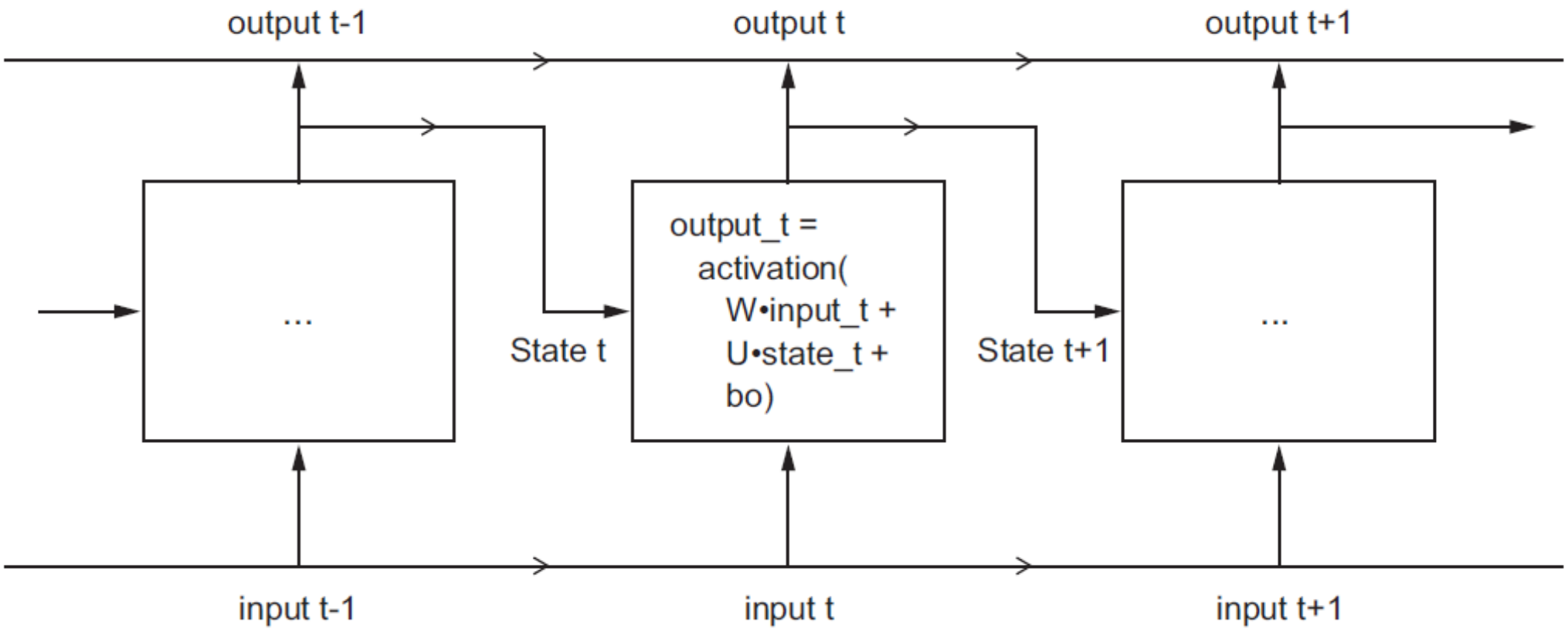


How RNN works

- Unroll/unfold RNN into multiple layer over time.



How RNN works



How RNN works

```
state_t = 0                          <---- The state at t
for input_t in input_sequence:       <---- Iterates over sequence elements
    output_t = f(input_t, state_t)
    state_t = output_t               <---- The previous output becomes the state for the next iteration.
```

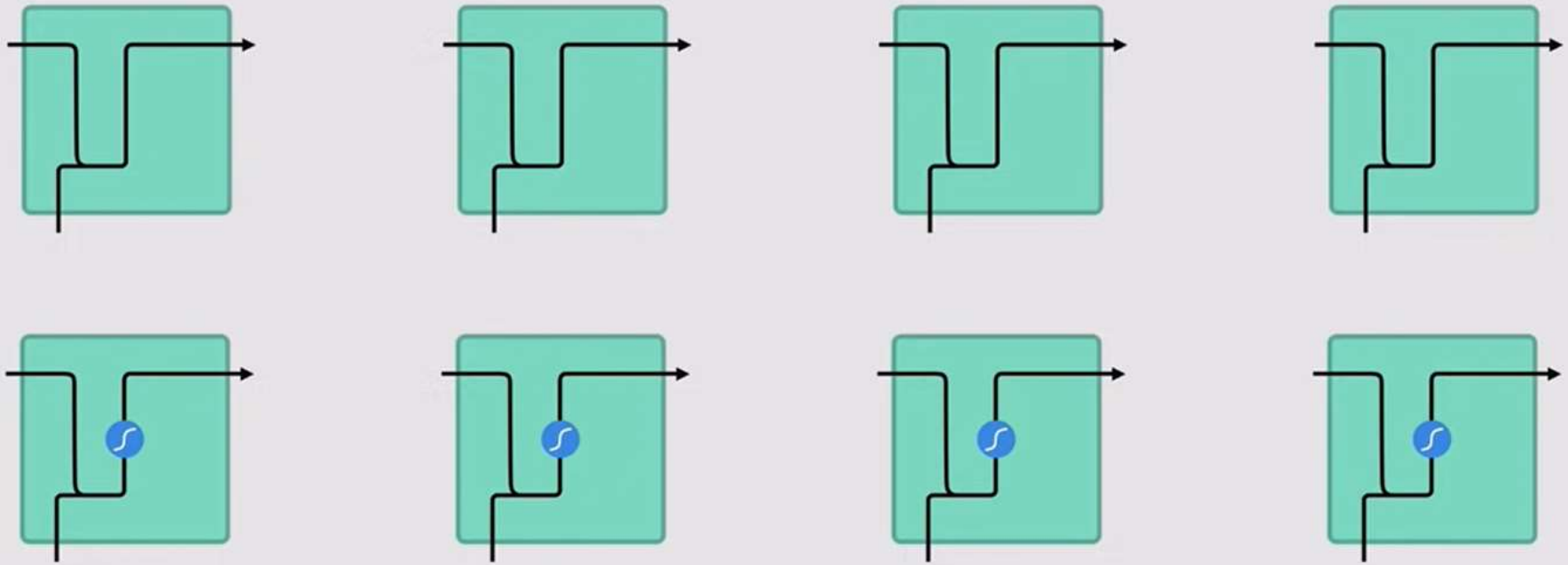
- `state_t` stores the memory or hidden state from the previous step.
- At each time step `t`, the RNN:
  1. Takes the current input (`input_t`)
  2. Combines it with the previous state (`state_t`)
  3. Passes the sum through an **activation function** (e.g., tanh or ReLU) to produce `output_t`
- The **output becomes the new state** for the next time step.

```
state_t = 0
for input_t in input_sequence:
    output_t = activation(dot(W, input_t) + dot(U, state_t) + b)
    state_t = output_t
```

405

0.81

-40.5

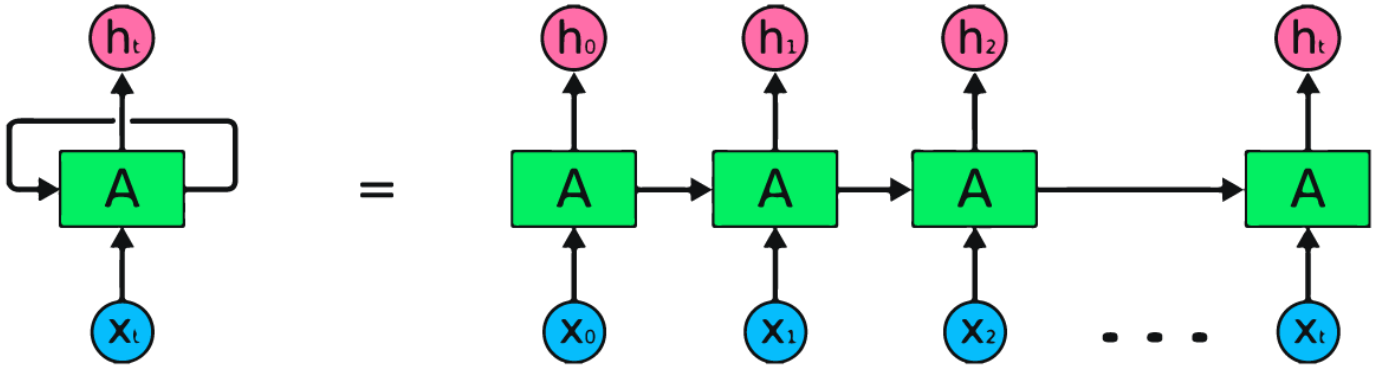


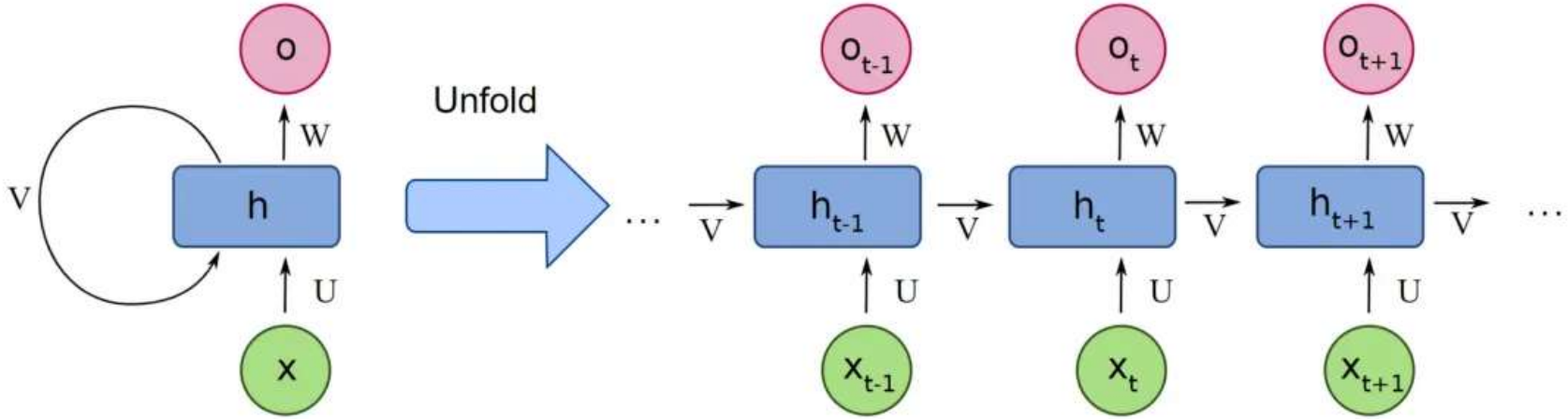
0.99

0.65

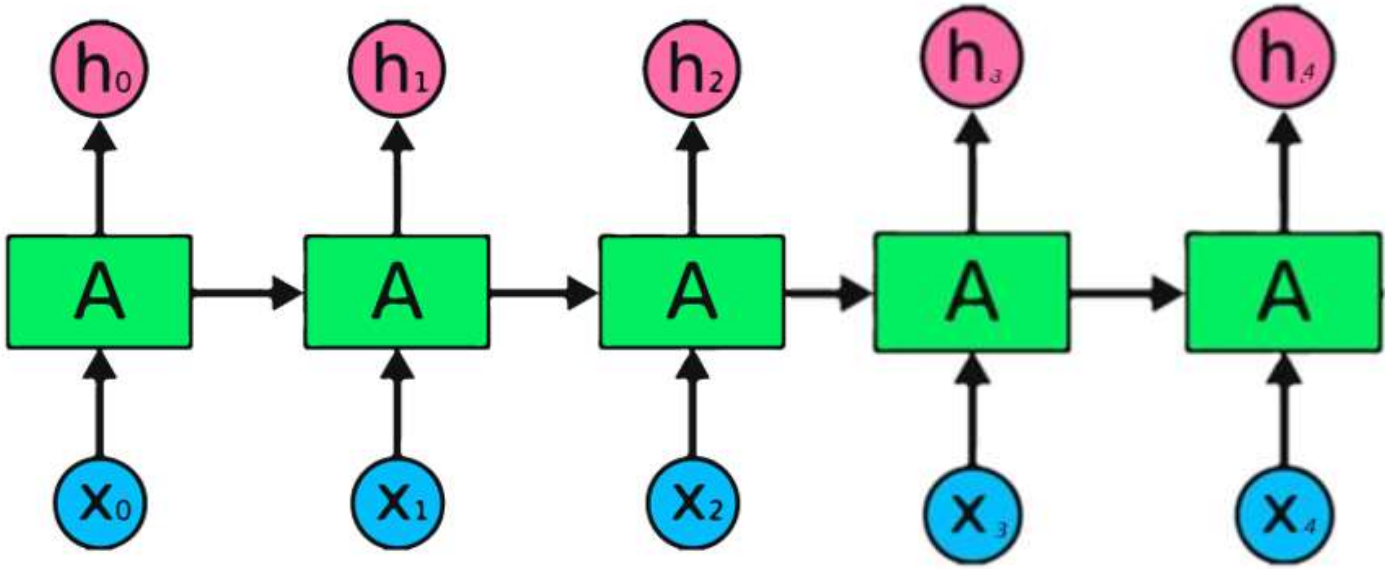
-0.99

How RNN works





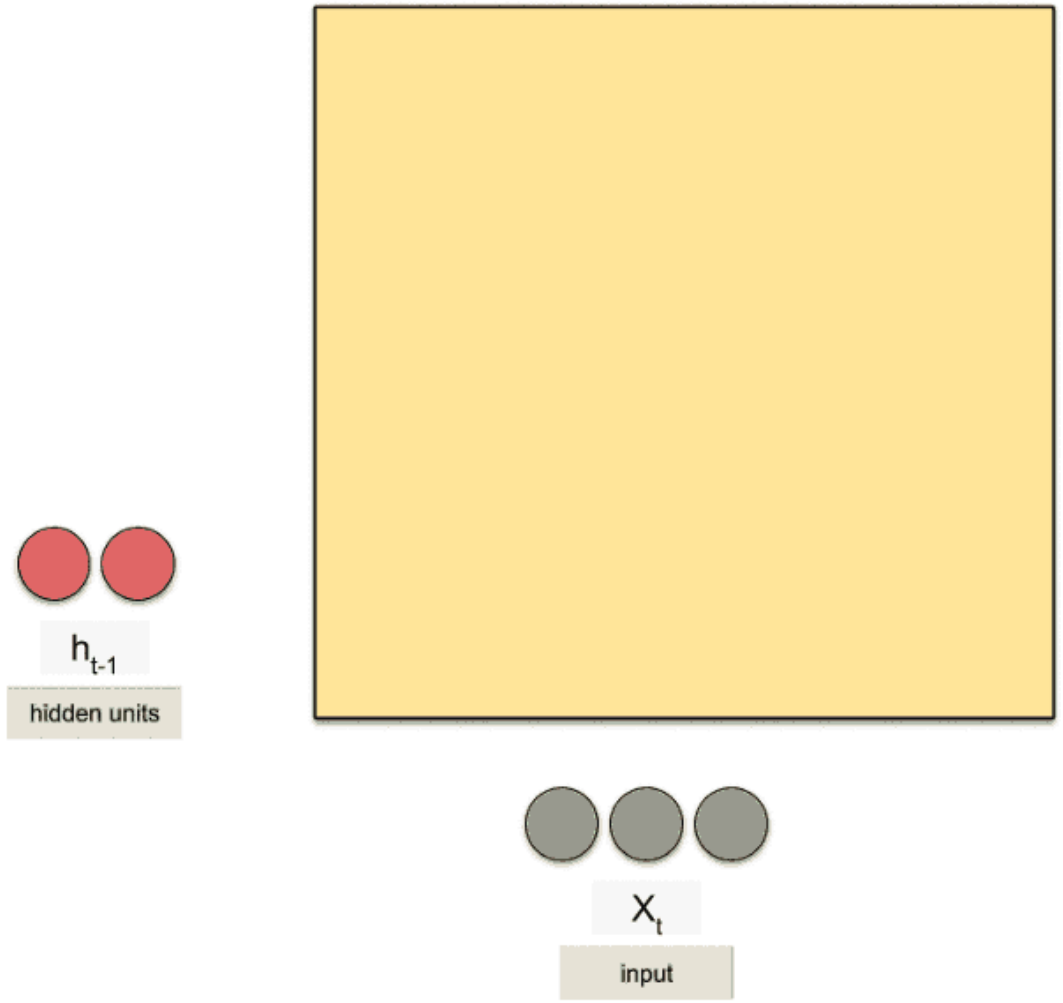
How RNN works



e.g., for sentiment analysis

This	car	is	not	good -> -ve sentiment
$<x_0$	x_1	x_2	x_3	$x_4>$

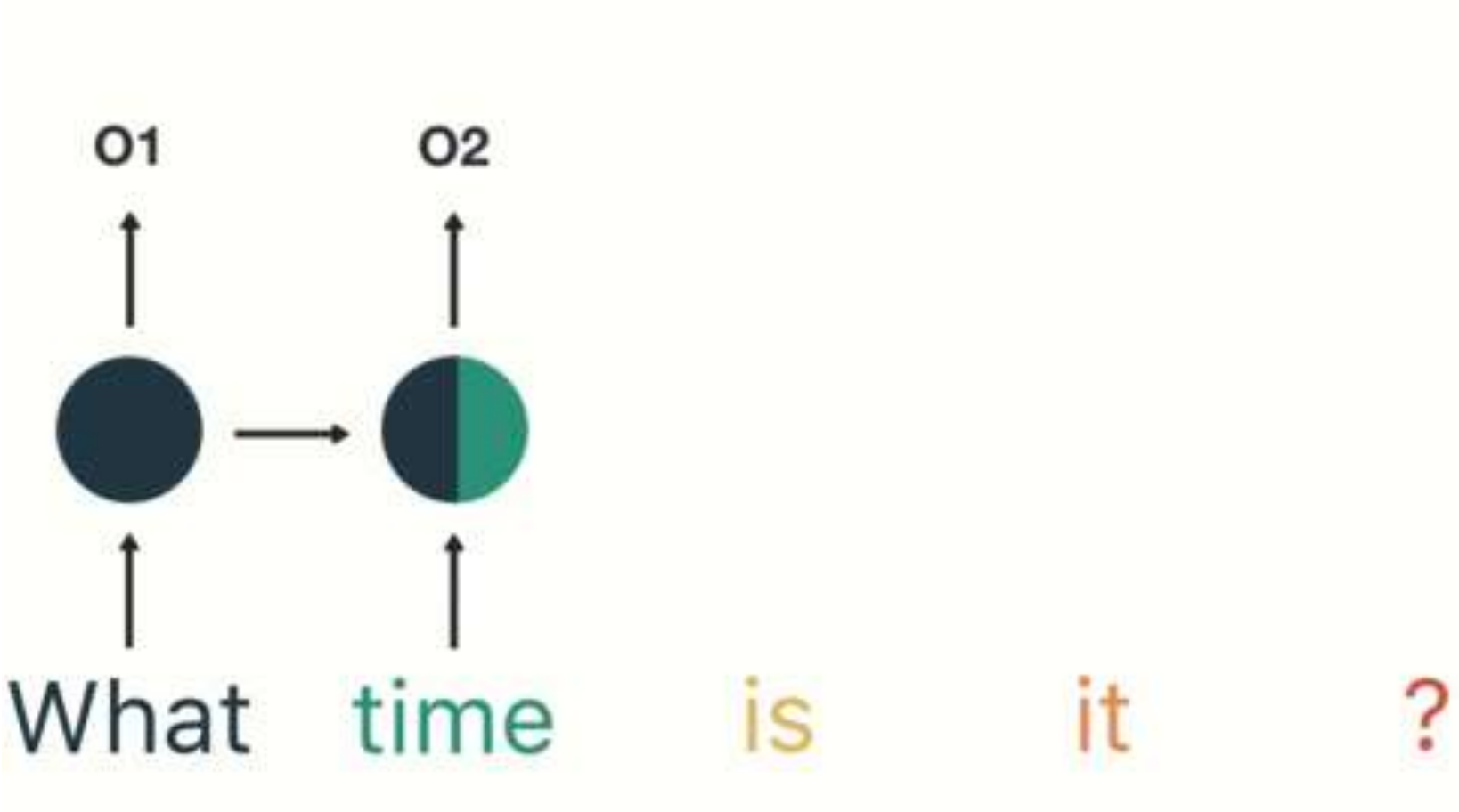
Recurrent Neural Network



Animated Example (Word tokens)

Animated Example….

Animated Example….

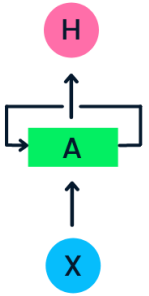


How RNN works - Example

Setup

We have:

- **Input:** temperature at time $t \rightarrow x_t = 20.0$
- **Goal:** predict next temperature y_t
- **RNN:** 8 neurons (each with tanh activation)
- **Hidden state:** previous $h_{t-1} = 0$ (since it's the first step)



Each neuron computes:

$$h_t^{(i)} = \tanh(W_x^{(i)} x_t + W_h^{(i)} h_{t-1} + b^{(i)})$$

Then all 8 $h_t^{(i)}$ combine into a vector $h_t = [h_t^{(1)}, \ldots, h_t^{(8)}]$.

Finally, output layer gives:

$$\hat{y}_t = W_y \cdot h_t + b_y$$

Example Calculation

Let's pick **small dummy values** for simplicity.

Parameter	Symbol	Value
Input	x_t	20.0
Previous hidden state	h_{t-1}	0
Input weights	W_x	[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8]
Biases	b	[0.1 each]
Output weights	W_y	[0.05 each]
Output bias	b_y	0.1

How RNN works - Example

Step 1 — Hidden activations

For each neuron i:

$$h_t^{(i)} = \tanh(W_x^{(i)} x_t + b^{(i)})$$

Let's compute:

Neuron	$W_x^{(i)} \times 20 + b^{(i)}$	tanh()	$h_t^{(i)}$
1	0.1×20 + 0.1 = 2.1	0.97	
2	0.2×20 + 0.1 = 4.1	0.9994	
3	0.3×20 + 0.1 = 6.1	0.99999	
4	0.4×20 + 0.1 = 8.1	0.999999	
5	0.5×20 + 0.1 = 10.1	1.0	
6	0.6×20 + 0.1 = 12.1	1.0	
7	0.7×20 + 0.1 = 14.1	1.0	
8	0.8×20 + 0.1 = 16.1	1.0	

✅ So the hidden state vector:

$$h_t = [0.97, 0.9994, 0.99999, 0.999999, 1, 1, 1, 1]$$

Step 2 — Output layer

$$\hat{y}_t = W_y \cdot h_t + b_y$$

$$= 0.05(0.97 + 0.9994 + 0.99999 + 0.999999 + 1 + 1 + 1 + 1) + 0.1$$

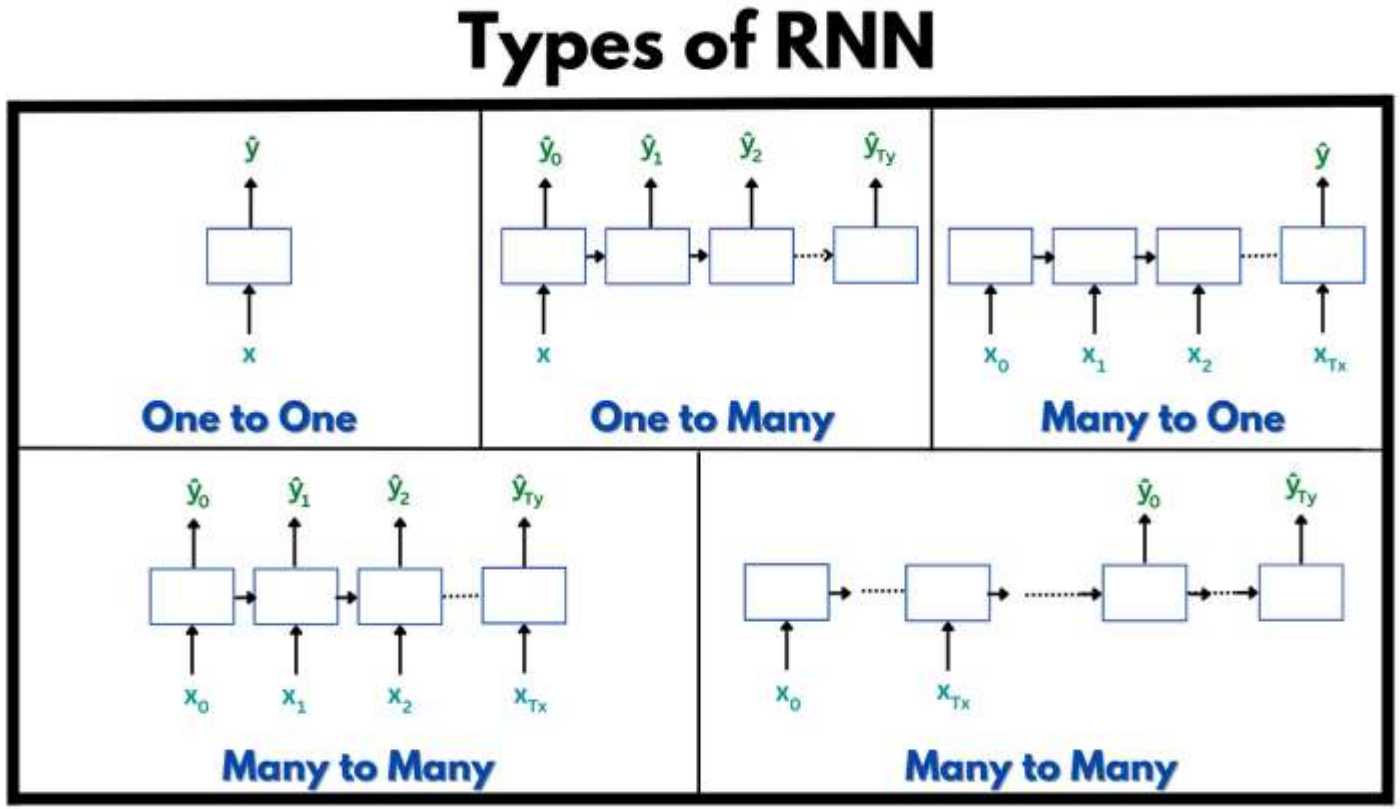
$$= 0.05(7.97) + 0.1 = 0.3985 + 0.1 = \boxed{0.4985}$$

✅ The predicted value (normalized) is 0.4985

(you'd later denormalize it to get the real temperature, e.g., 25°C)

Types of RNN

- The type of RNN are:
  - One to one RNN
  - One to many RNN
  - Many to one RNN
  - Many to many RNN

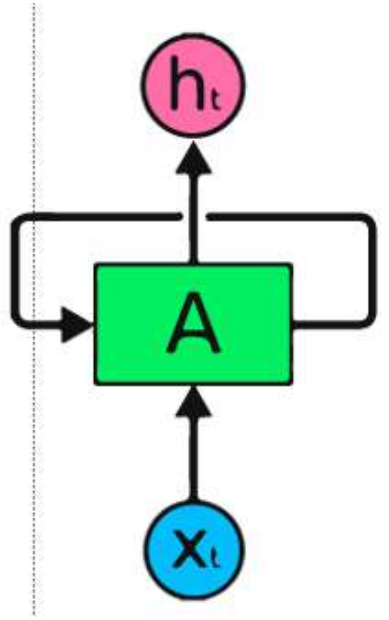


Types of RNN: One to One

- There is **one input for each time step** and **one output for each time step**, but there is no sequence of inputs or outputs being passed to the network
- Basic RNN used for tasks like classification or regression where each input corresponds to a single output.
- **Example**:
  - Let's say we want to use a one-to-one RNN to predict a single continuous value based on an input sequence (**like predicting the next temperature based on the current data**).

Types of RNN: One to One

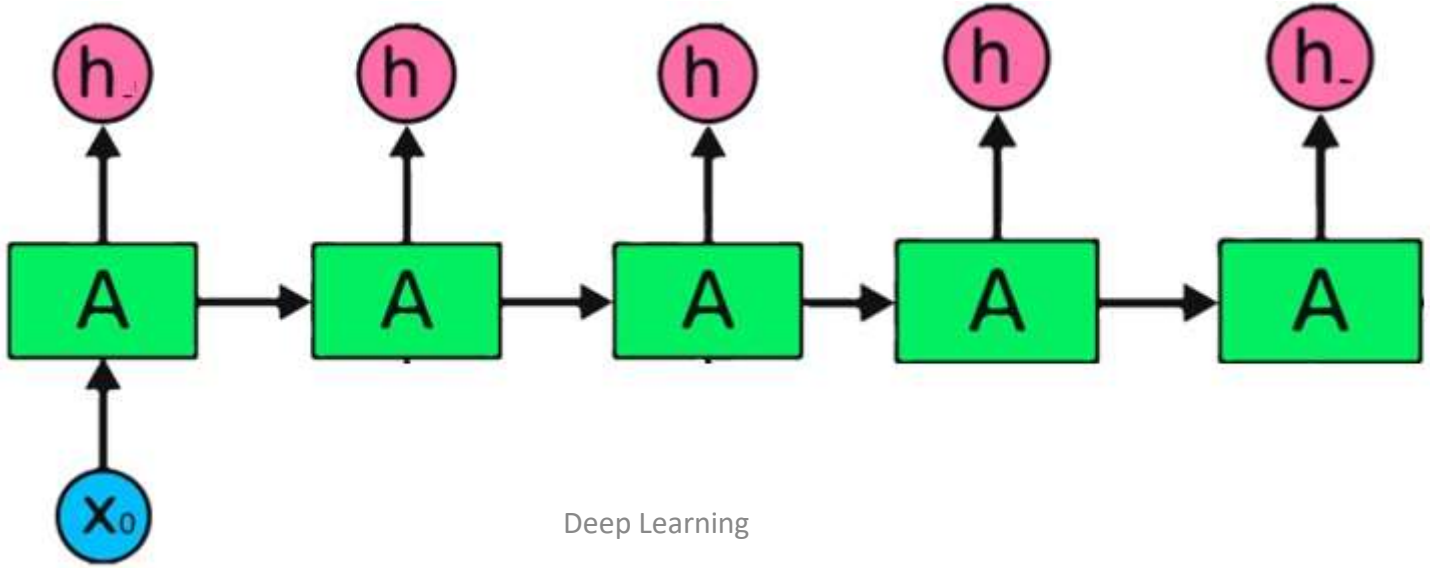
```python
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
from tensorflow.keras.optimizers import Adam

# Sample data: Let's assume we have time series data
# 10 time steps, and each time step has one feature (this could be temperature, stock price, etc.)
X = np.array([[i] for i in range(10)])  # Example input: 10 time steps, 1 feature
y = np.array([10])  # Example output: the regression target for this sequence
print("X = ",X)
print("Y = ",y)
# Reshape X to be [samples, timesteps, features], as expected by RNN
X = X.reshape((1, 10, 1))  # 1 sample, 10 time steps, 1 feature
print("Now X = ", X)
# Define the RNN model
model = Sequential()

# Add a SimpleRNN layer
model.add(SimpleRNN(50, activation='relu', input_shape=(10, 1)))

# Output layer for regression (a single value)
model.add(Dense(1))

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.001), loss='mse')

# Train the model (dummy example, usually you'd use more data)
model.fit(X, y, epochs=100)

# Predict using the trained model
prediction = model.predict(X)
print(f"Predicted output: {prediction}")
```




```
Epoch 99/100
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - loss: 2.2495e-04
Epoch 100/100
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - loss: 3.1655e-04
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 129ms/step
Predicted output: [[9.981146]]
```

Types of RNN: One to Many

- Model processes **an input sequence and produces multiple outputs**, usually over time.
- Commonly used in tasks such as sequence labeling or time series prediction, where you might want to **predict a sequence of outputs based on a single input sequence**.



Types of RNN: One to Many

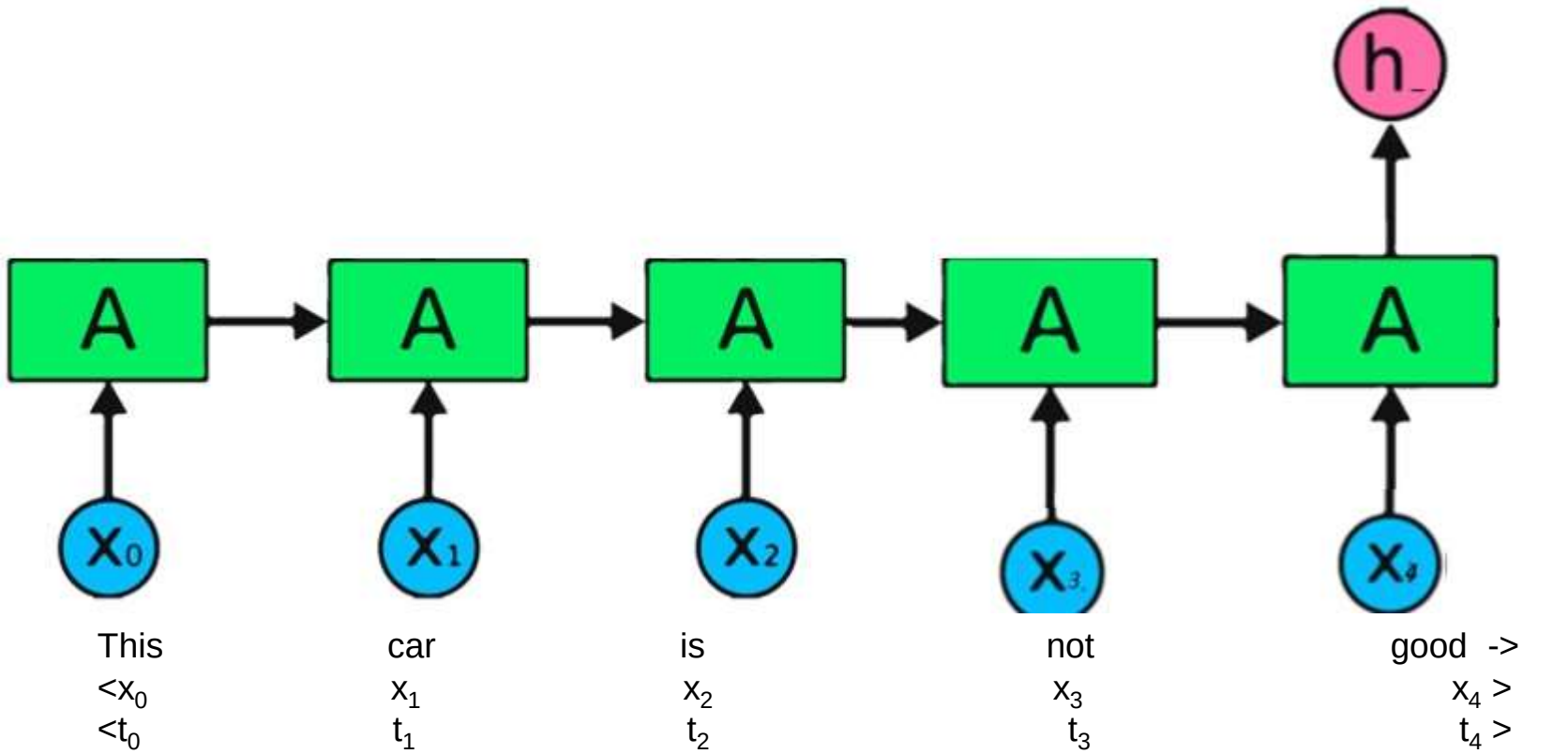
- Suppose you have a sentence where the input is the first word, and the output is a sequence of words generated by the model.
- For instance, if the input is the word "Hello", the RNN might generate a sequence like "Hello, how are you doing today?". The network generates multiple outputs, based on one initial input.
- Examples:
  - Google search suggestions
  - Text generation: Generating a sequence of words or characters.
  - Music generation: Predicting a sequence of musical notes based on the first few notes.
  - Image captioning: Generating a sequence of words to describe the content of an image.
  - Speech generation: Producing a sequence of sound samples or phonemes based on initial speech input.

Types of RNN: Many to One

- **Multiple inputs** (a sequence of data) are processed and mapped into a **single output**.
- **Sequence Processing**: The model takes an entire sequence as input and produces **a single output** after processing all the input steps through the recurrent layers.
- Examples:
  - Sentiment analysis (e.g., classifying the sentiment of a sentence).
  - Time series forecasting (e.g., predicting future stock prices based on historical data).
  - Text classification (e.g., spam detection or topic categorization).

Types of RNN: Many to One

- E.g., sentiment analysis



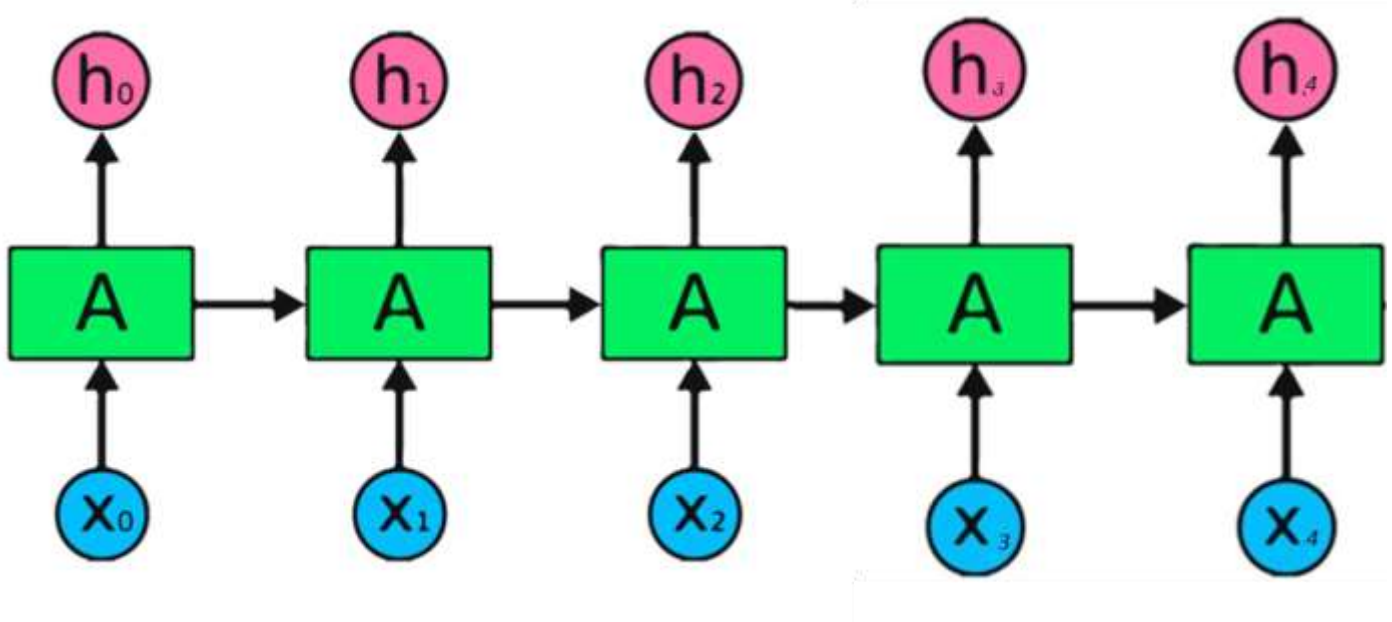
Types of RNN: Many to Many

- **Both** the **input** and the **output are sequences**.
- The model takes in a sequence of data and produces a sequence of outputs, making it ideal for tasks where the input-output relationship is sequential and the number of input and output steps are the same.
- **How it works**:
  - The RNN processes a sequence of inputs (X_1, X_2, ..., Xn).
  - It generates a sequence of outputs (Y_1, Y_2, ..., Yn), with each output potentially dependent on the entire input sequence or just the prior outputs and states.

Types of RNN: Many to Many

- **Machine Translation**: Converting a sequence in one language (input) to a sequence in another language (output), e.g., English to Urdu translation.
- **Speech-to-Text**: Converting an audio waveform (sequence) to a sequence of words.
- **Video Captioning**: Generating a sequence of words to describe the content of a video
- Other: Chatbot, Question answers etc.
- **Example of translation to German from English**:
  - Input: "I am learning RNNs." (sequence of words).
  - Output: "Ich lerne RNNs." (translated sequence in German).

Types of RNN: Many to Many



Forward Propagation RNN

- Forward propagation refers to the process of passing data through the network, from the input to the output, in order to compute the predictions.
- RNNs are particularly designed for sequential data, where each output depends not only on the current input but also on the previous outputs or hidden states.
- Key Components:
  - Input at each time step (x□): The data that is fed into the network at each time step.
  - Hidden state (h□): The memory of the network, which stores information about the sequence seen so far.
  - Output (y□): The prediction made by the network for the current time step
- Example: at time stamp t =1, 2, 3, ..:
  - If the input is x1 and h1 is hidden state then: $h_1 = f(W \cdot x_1 + U \cdot h_0 + b)$
  - In general, if the input is xt and ht is hidden state then: $h_t = f(W \cdot x_t + U \cdot h_{t-1} + b)$

Forward Propagation RNN

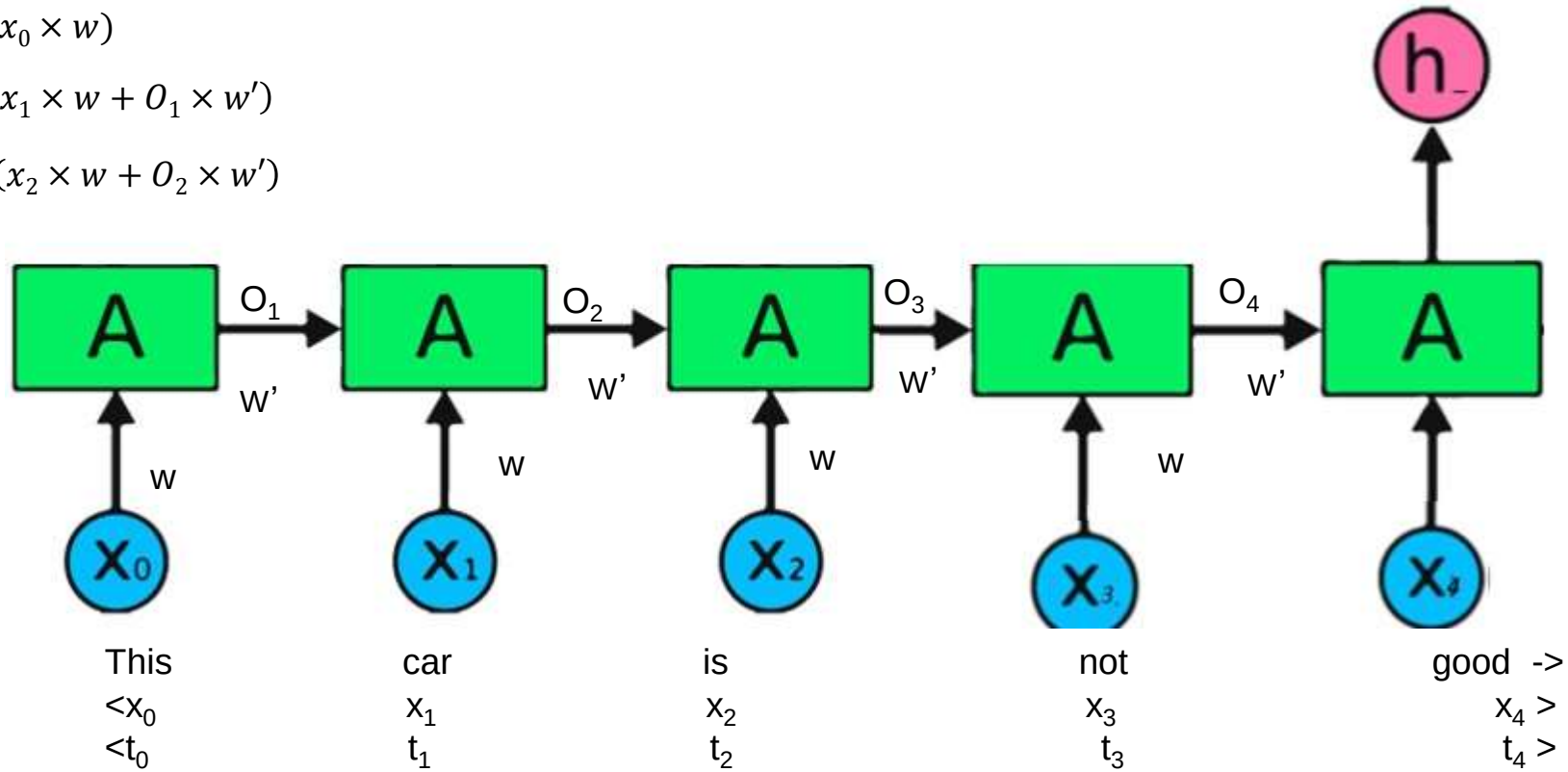
- E.g., sentiment analysis

-ve sentiment Sigmoid/softmax

$$O_1 = f(x_0 \times w)$$

$$O_2 = f(x_1 \times w + O_1 \times w')$$

$$O_3 = f(x_2 \times w + O_2 \times w')$$



Backward Propagation RNN

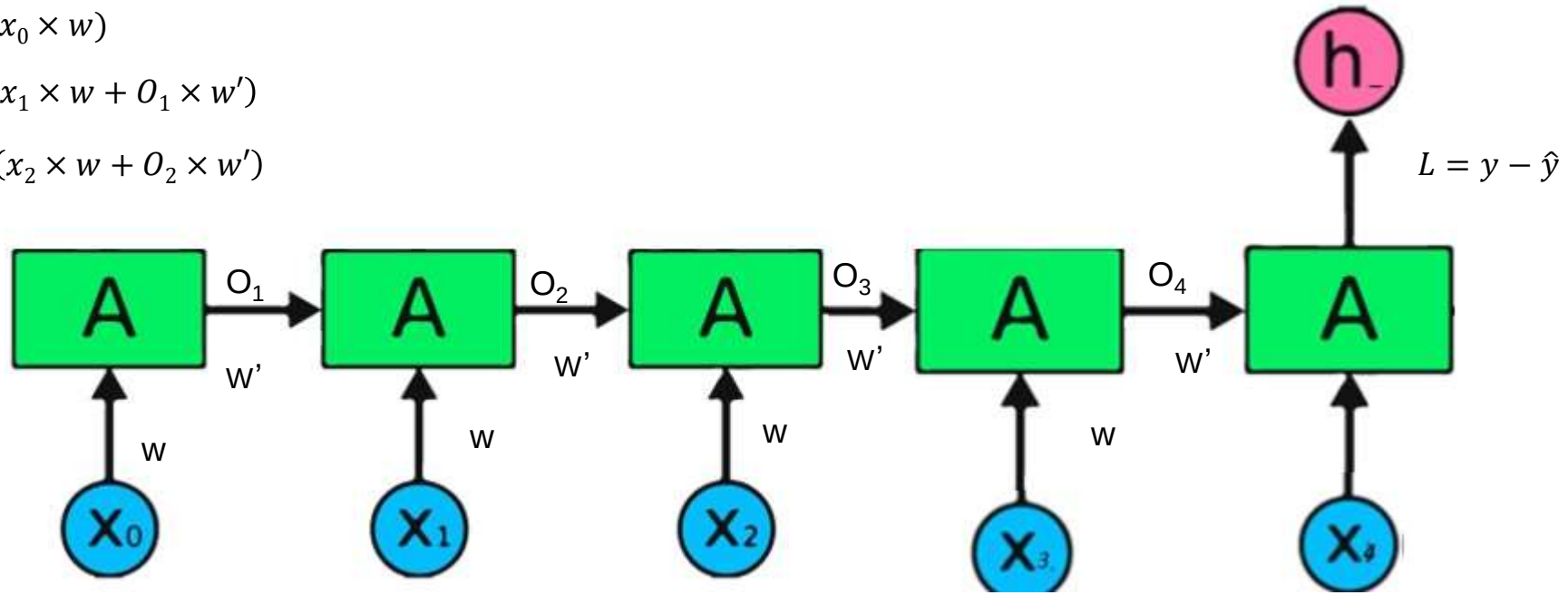
- E.g., sentiment analysis

-ve sentiment Sigmoid/softmax

$$O_1 = f(x_0 \times w)$$

$$O_2 = f(x_1 \times w + O_1 \times w')$$

$$O_3 = f(x_2 \times w + O_2 \times w')$$



This <x_0 <t_0 | car x_1 t_1 | is x_2 t_2 | not x_3 t_3 | good -> x_4 > t_4 >

$$w_{new} = w_{old} - \eta \frac{\partial L}{\partial x}$$

$$w'_{new} = w'_{old} - \eta \frac{\partial L}{\partial w'_{old}}$$

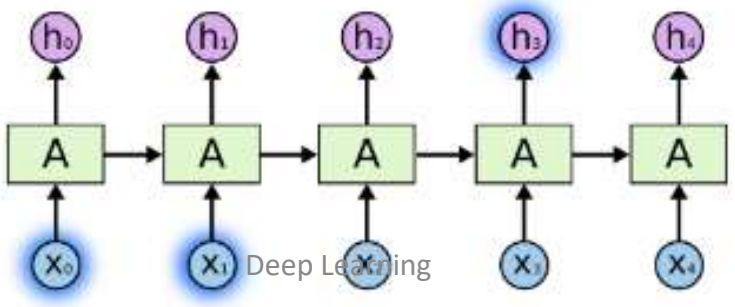
$$\frac{\partial L}{\partial w'_{old}} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial w'_{old}}$$

Vanishing gradient problem with RNN

- During training, **gradients of the loss function with respect to weights can become very small** as they are propagated backward through time.
- **Key Issue**: Gradients that become **exceedingly small** lead to slow or no learning, especially in long sequences.
- RNNs use a **chain of multiplications** across time steps.
- If the weights of the network have values less than 1, the gradients shrink exponentially as they are propagated back.
- This leads to the vanishing of the gradients.
- Solution to Vanishing gradient problem is LSTM RNN.

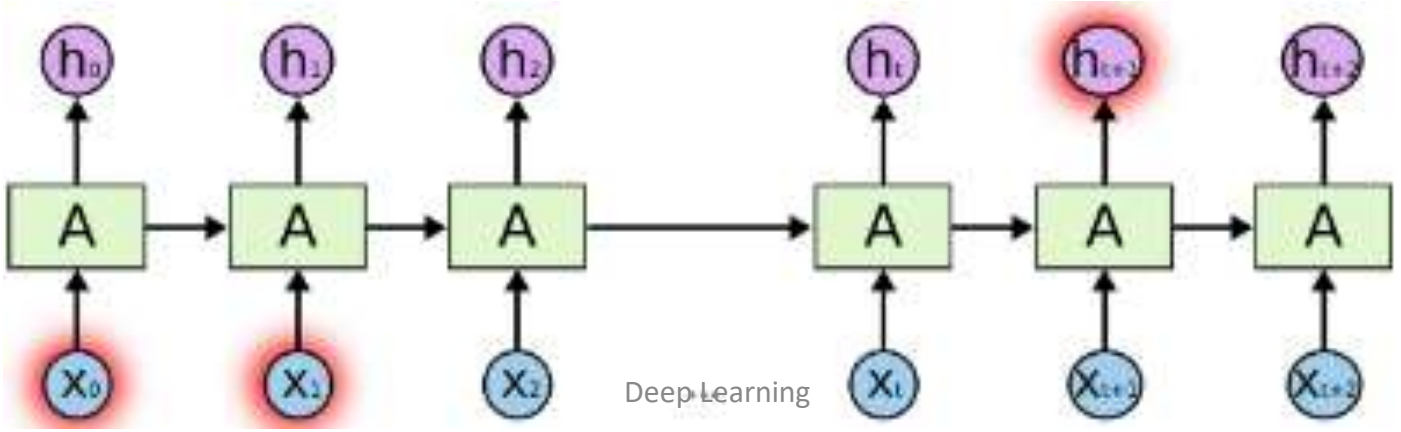
The Problem of Long-Term Dependencies

- RNNs might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame.
- If RNNs could do this, they'd be extremely useful. But can they? It depends.
- We need to look at recent information to perform the present task.
- For example, consider a language model trying to predict the next word based on the previous ones.
- If we are trying to predict the last word in "the clouds are in the sky," we don't need any further context – it's pretty obvious the next word is going to be sky.
- In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.



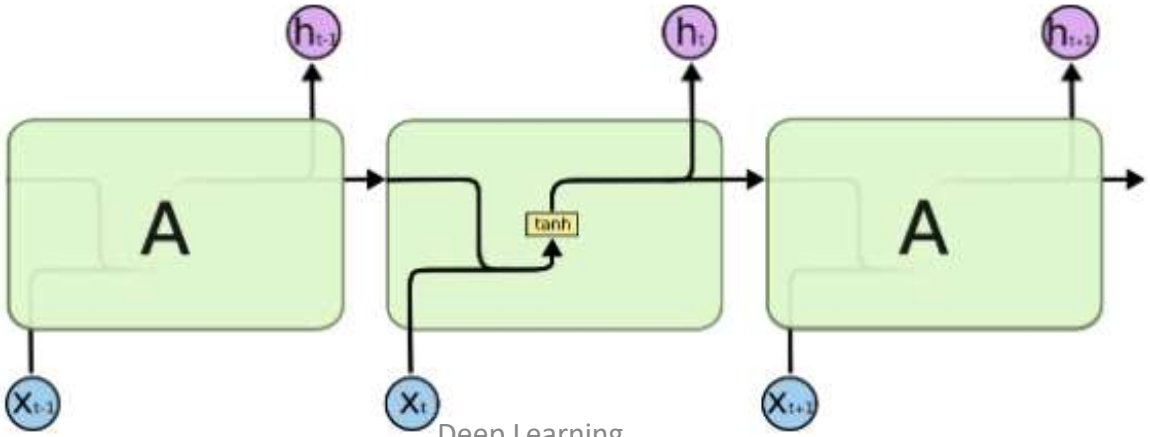
The Problem of Long-Term Dependencies

- There are also cases where we need more context.
- Consider trying to predict the last word in the text "I grew up in France… I speak fluent French."
- In the context of RNNs, context refers to the **information from previous time** steps that is maintained and used by the model **to make predictions** or decisions at the current time step.
- Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the *context of France*, from further back.
- It's entirely possible **for the gap between the relevant information** and the point where it is needed to become very large.
- Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.



The Problem of Long-Term Dependencies

- Thankfully, LSTMs don't have this problem!
- Long-term dependencies refer to the ability of the network to capture relationships or patterns between inputs that are **far apart in the sequence**.
- In other words, it is the capacity of the RNN to remember and use information from **earlier time steps** (which could be many steps before) when making predictions or decisions at later time steps.
- **Long Short Term Memory networks** – usually just called "LSTMs" – are a special kind of RNN, capable of learning **long-term dependencies**.
- LSTMs are explicitly designed to avoid the long-term dependency problem.
- Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

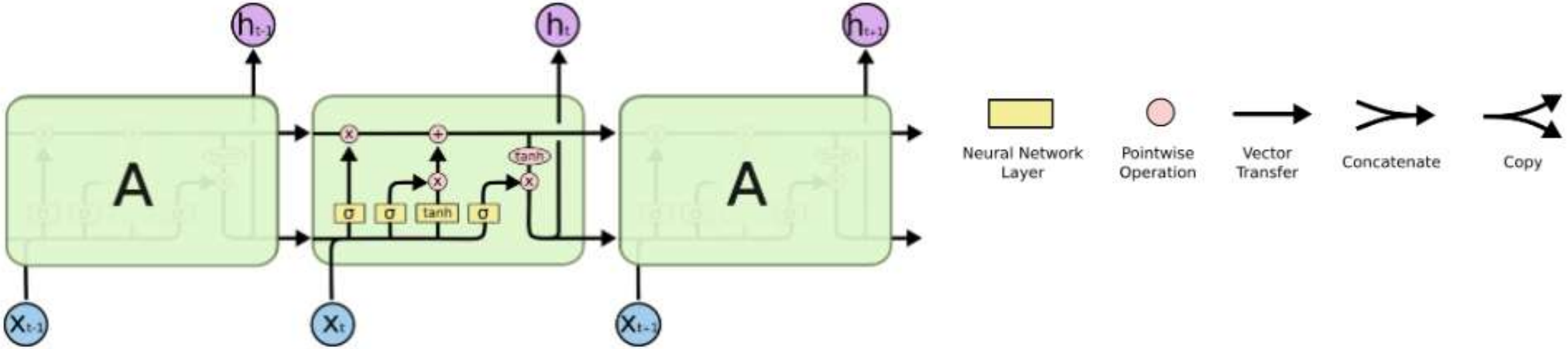


The repeating module in a standard RNN contains a single layer.


NICE TO MEET YOU
I ALREADY
FORGOT YOUR NAME

The Problem of Long-Term Dependencies

- LSTMs also have this **chain-like structure**, but the repeating module has a different structure (means not like RNN with single activation function but multiple activations).
- Instead of having a single neural network layer, there are **four, interacting** in a very special way.



The repeating module in an LSTM contains four interacting layers.

Reference: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM (Long Short Term Memory)

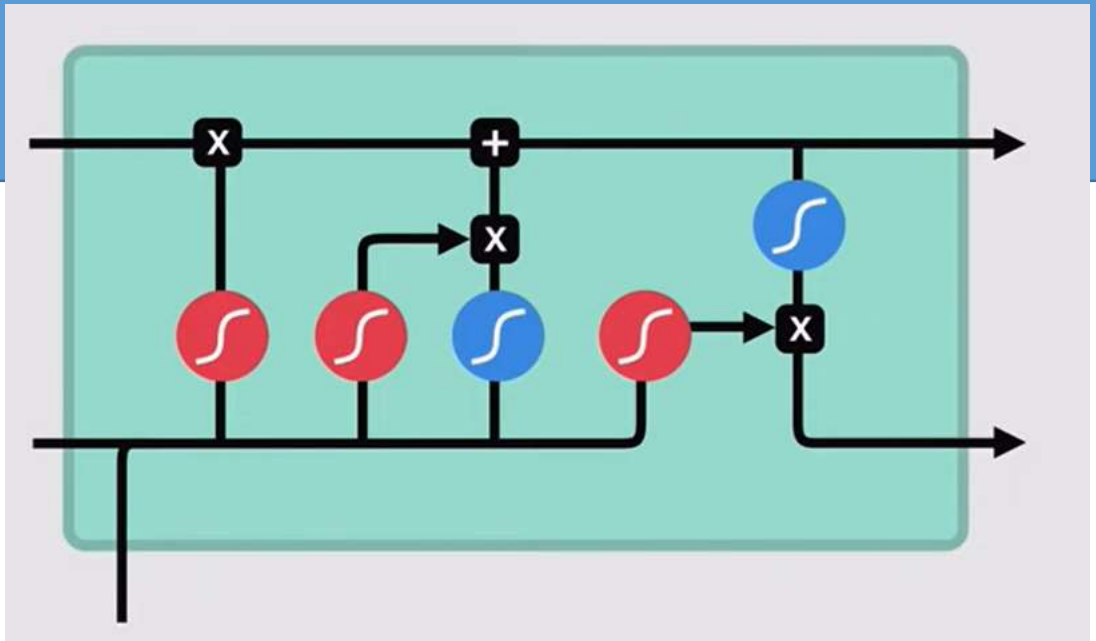
- LSTMs were specifically designed to address these issues and improve learning in sequential data.
- LSTM is a type of Recurrent Neural Network (RNN) that introduces gates to regulate the flow of information.
- It allows the network to remember or forget information across long sequences, enabling better performance on tasks with long-term dependencies.

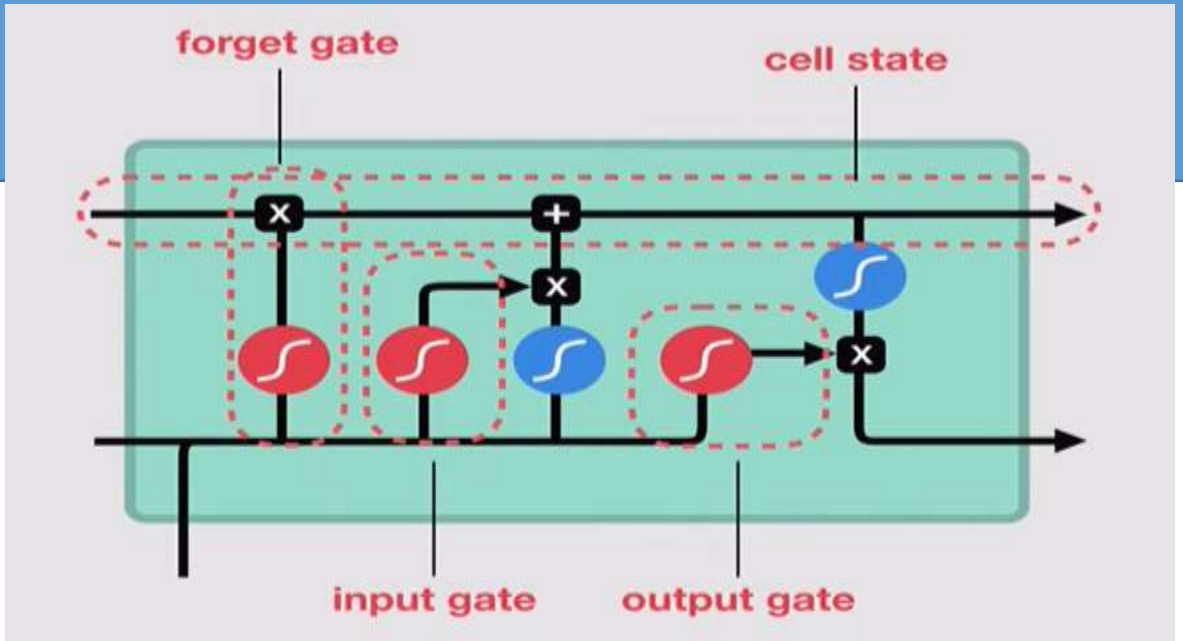
LSTM (Long Short Term Memory)

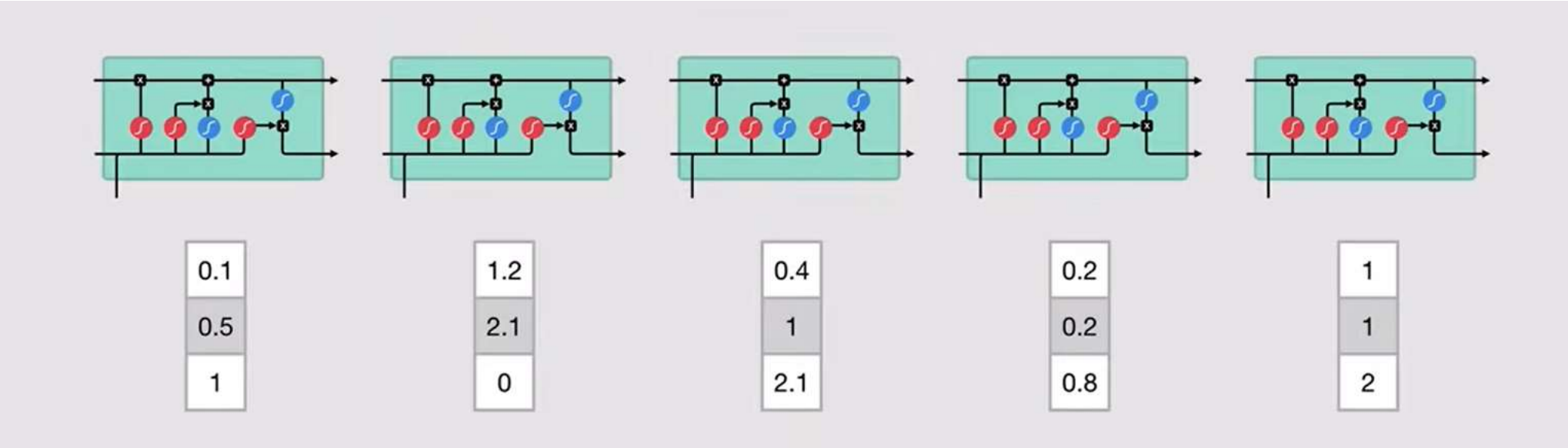
- Key Characteristics of LSTMs:
  - **Memory Cell**: The core component that stores information over time.
  - **Gates**: Mechanisms that control the flow of information:
    1. **Forget Gate**: Decides what information to discard.
    2. **Input Gate**: Controls what new information to store in the memory cell.
    3. **Output Gate**: Determines the output based on the memory cell and current input.

LSTMs add a **memory management system** that helps the network:

- Remember important information for long durations
- Forget unimportant information
- Control the flow of information




forget gate
cell state
input gate
output gate

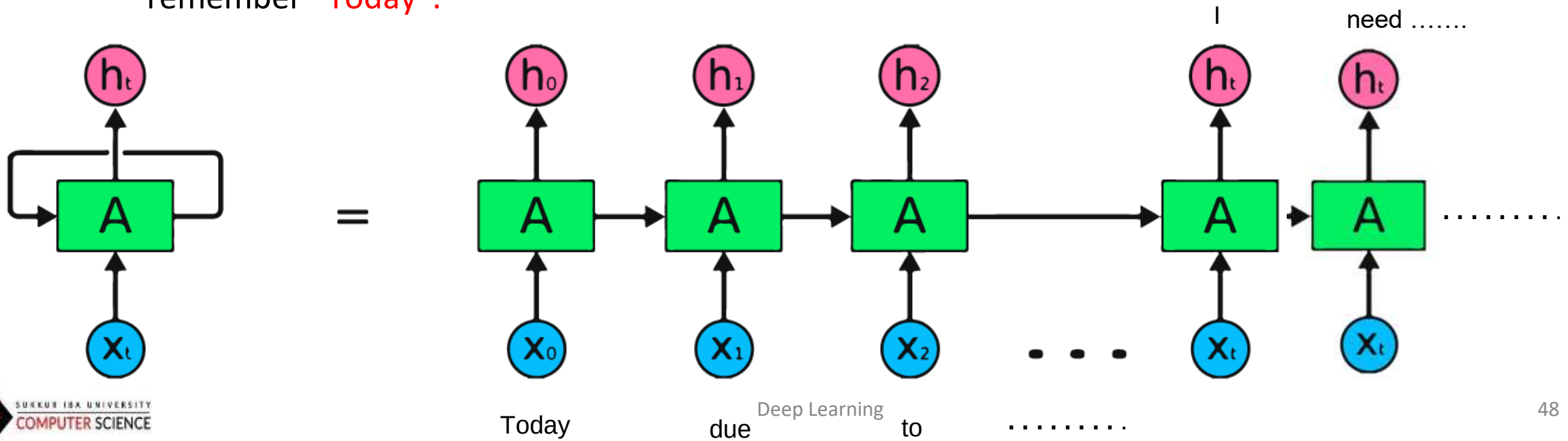

0.1
0.5
1
1.2
2.1
0
0.4
1
2.1
0.2
0.2
0.8
1
1
2

LSTM (Long Short Term Memory)

- Examples:
- Today, due to my current job situation and family conditions, I need to take a loan.
- Last year, due to my current job situation and family conditions, I had to take a loan.
- The decision of filling “I need to take a loan” & “I had to take a loan” was based on word “Today” & “Last year ” respectively.

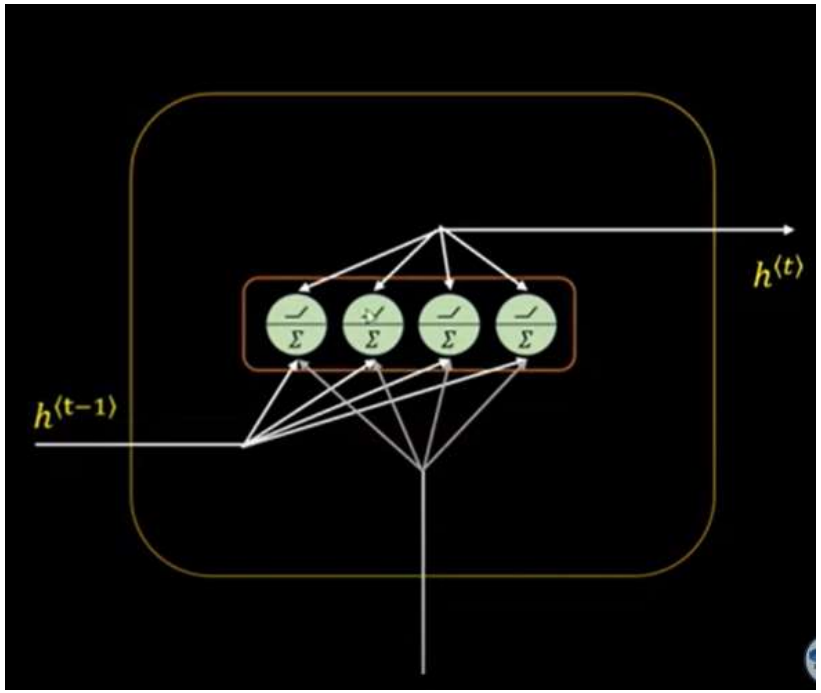
LSTM (Long Short Term Memory)

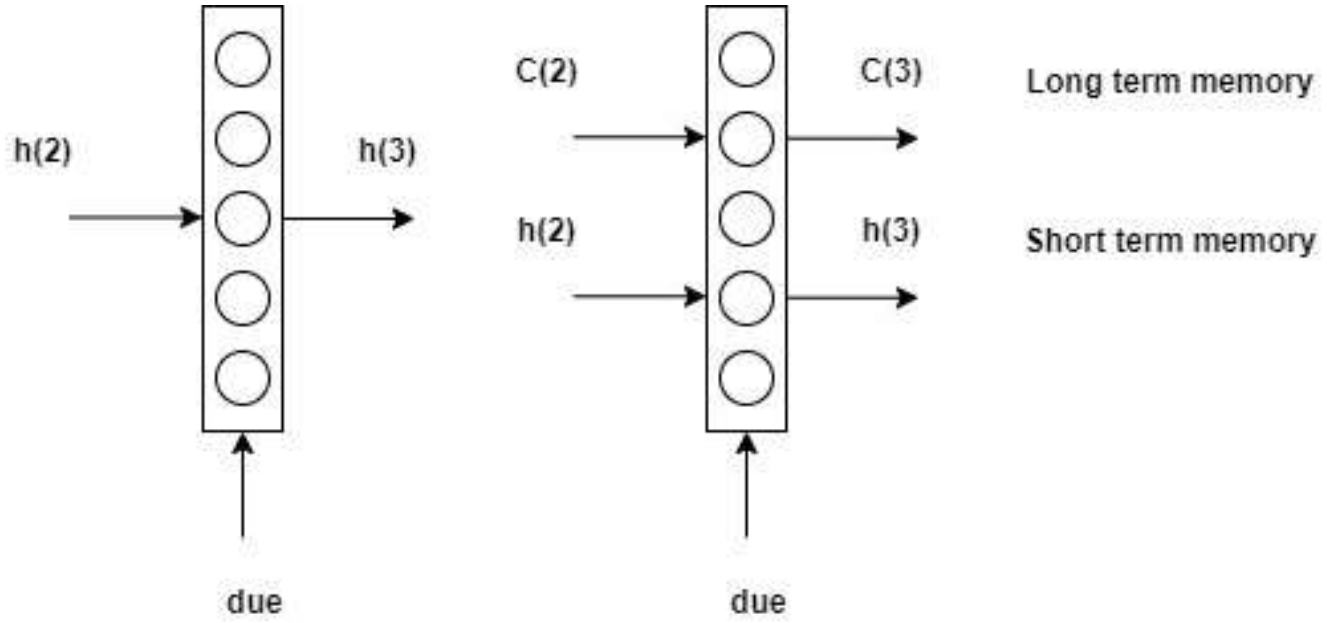
- Today, due to my current job situation and family conditions, I need to take a loan. Unroll RNN for this sentence:
- Below is the same/one layer but represented/unrolled in different times.
- To predict the word I need, RNN should remember Today.
- As RNN has short term memory of few words near by due to vanishing gradient, it cannot remember "Today".



LSTM (Long Short Term Memory)

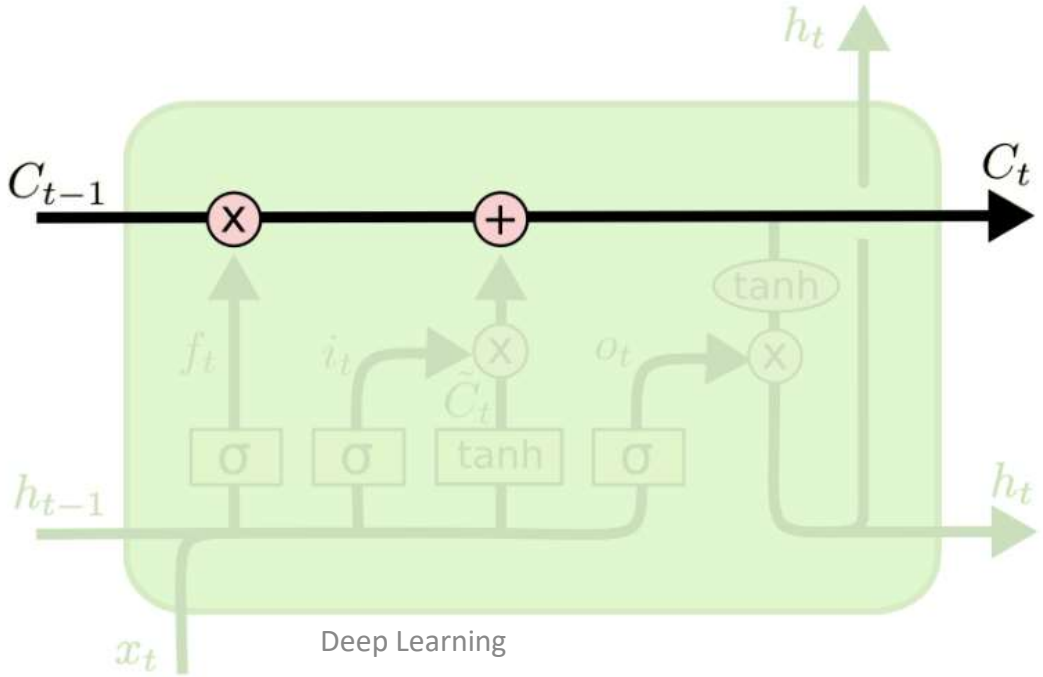
- Memory cell in LSTM





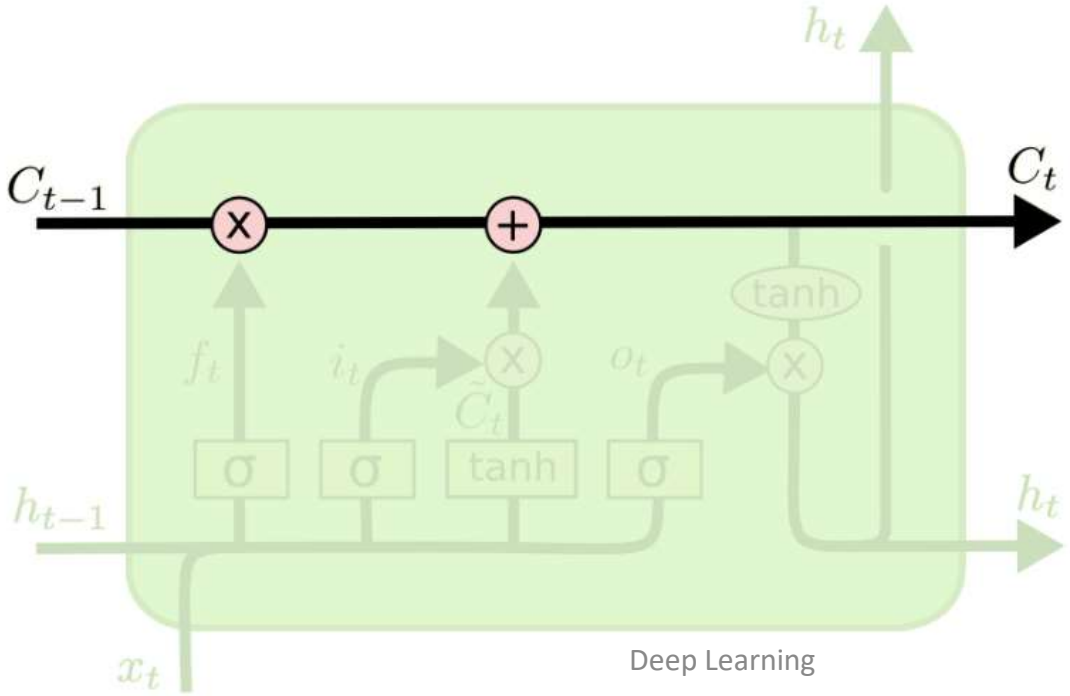
LSTM (Long Short Term Memory)

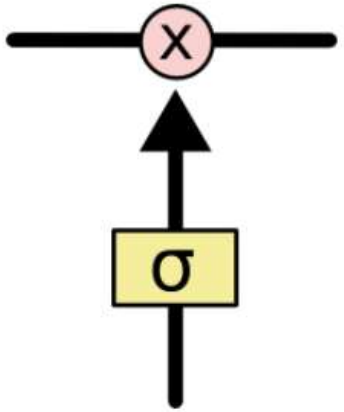
- The key to LSTMs is the **cell state**, the horizontal line running through the top of the diagram.
- The cell state is kind of **like a conveyor belt**.
- It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged. (This refers to how the cell state flows through the LSTM across all time steps, from the beginning of the sequence to the end. Unlike the hidden state, which is updated at each time step.)
- The LSTM does have **the ability to remove or add information to the cell state**, carefully regulated by structures called gates.



LSTM (Long Short Term Memory)

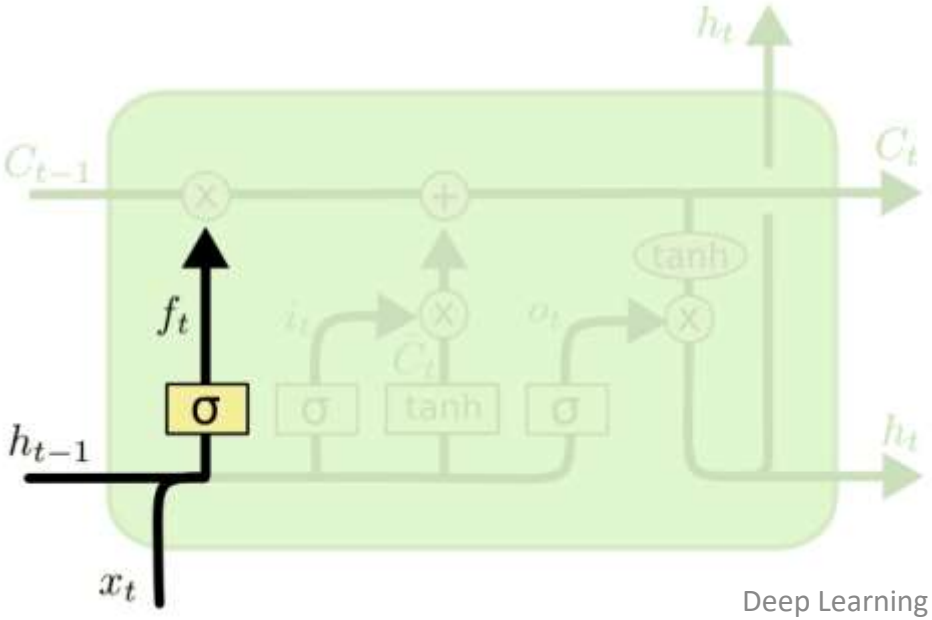
- Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.
- The **sigmoid** layer outputs numbers between zero and one, describing how much of each component should be let through.
- A value of zero means “let nothing through,” while a value of one means “let everything through!”





LSTM (Long Short Term Memory)

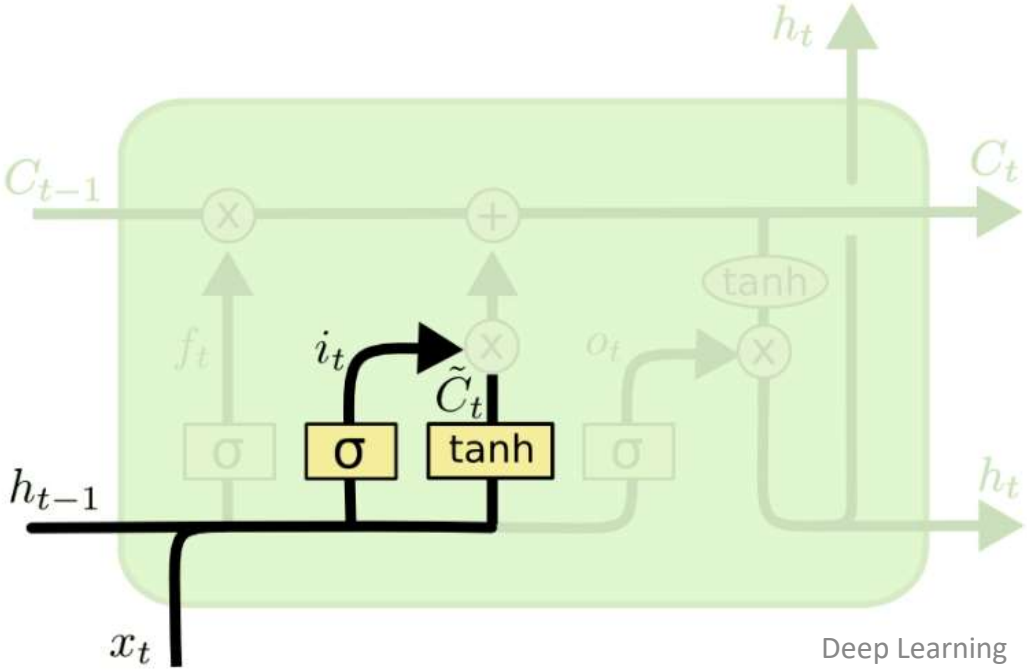
- The first step in our LSTM is to decide **what information** we're going to throw away from the cell state.
- This decision is made by a sigmoid layer called the "**forget gate layer**."
- It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}.
- A **1 represents** "completely keep this" while a **0 represents** "completely get rid of this."



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

LSTM (Long Short Term Memory)

- The next step is to decide **what new information we're going to store** in the cell state.
- This has two parts. First, a sigmoid layer called the "**input gate layer**" decides which values we'll update.
- Next, a tanh layer creates a vector of new candidate values, $\widetilde{C}_t$, that could be added to the state. In the next step, we'll combine these two to create an update to the state.

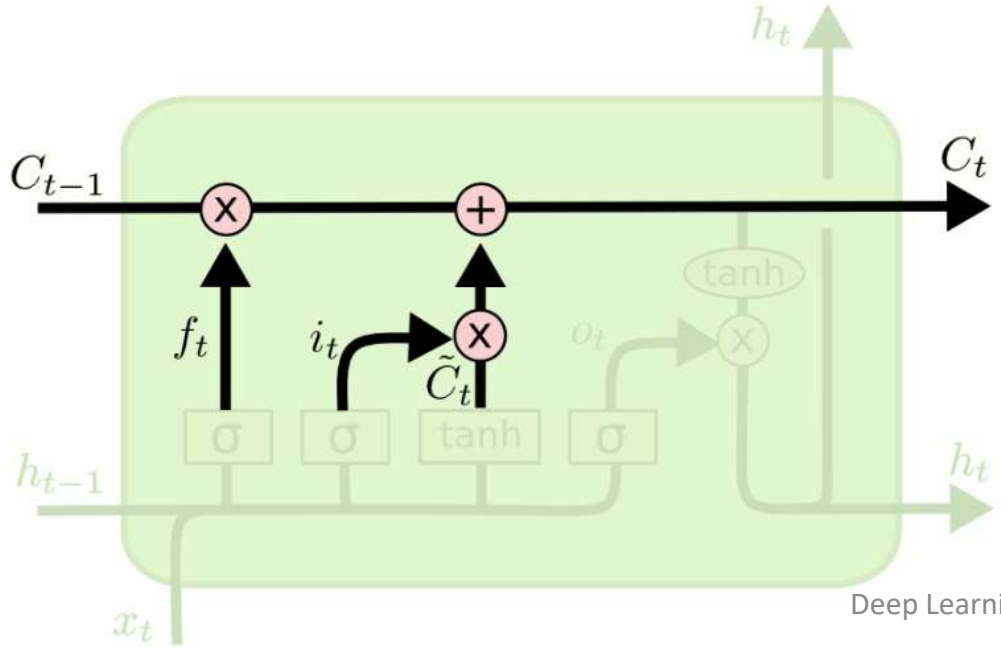


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

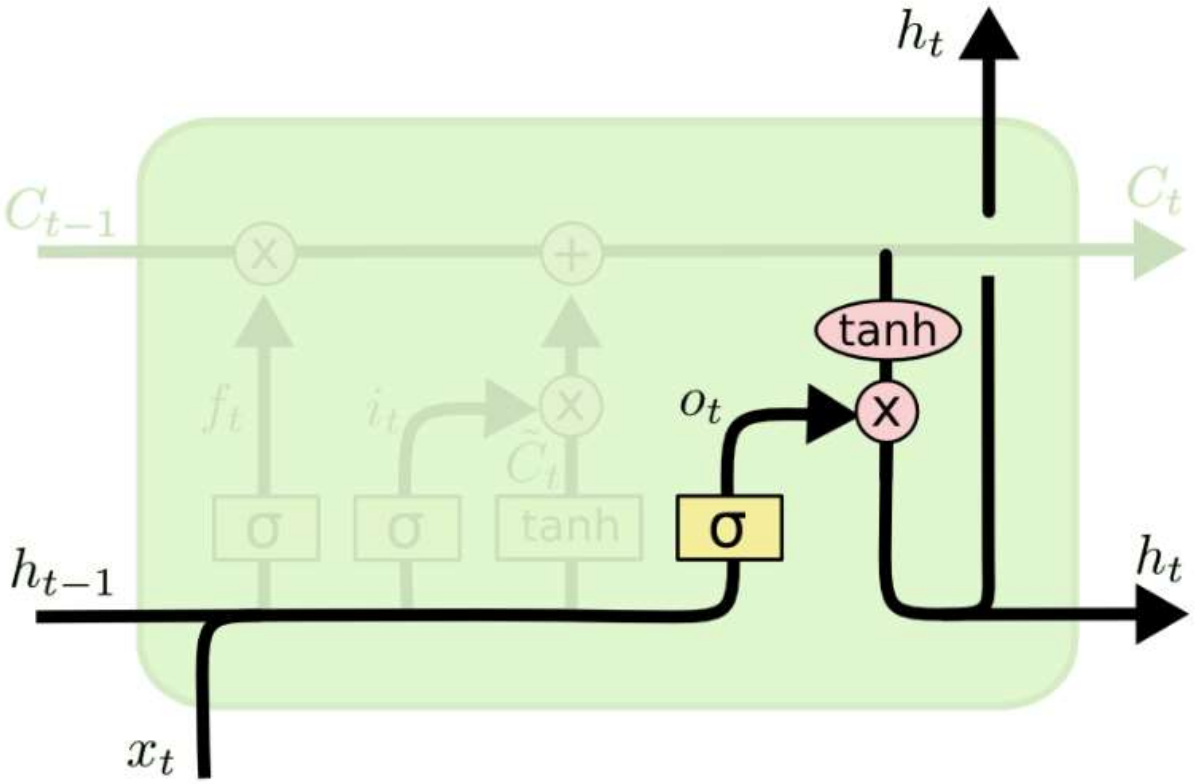
LSTM (Long Short Term Memory)

- It's now time to **update the old cell state, C_{t-1}**, into the **new cell state C_t**.
- The previous steps already decided what to do, we just need to actually do it.
- We multiply the old state by f_t, forgetting the things we decided to forget earlier.
- Then we add $i_t \times \tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

LSTM (Long Short Term Memory)



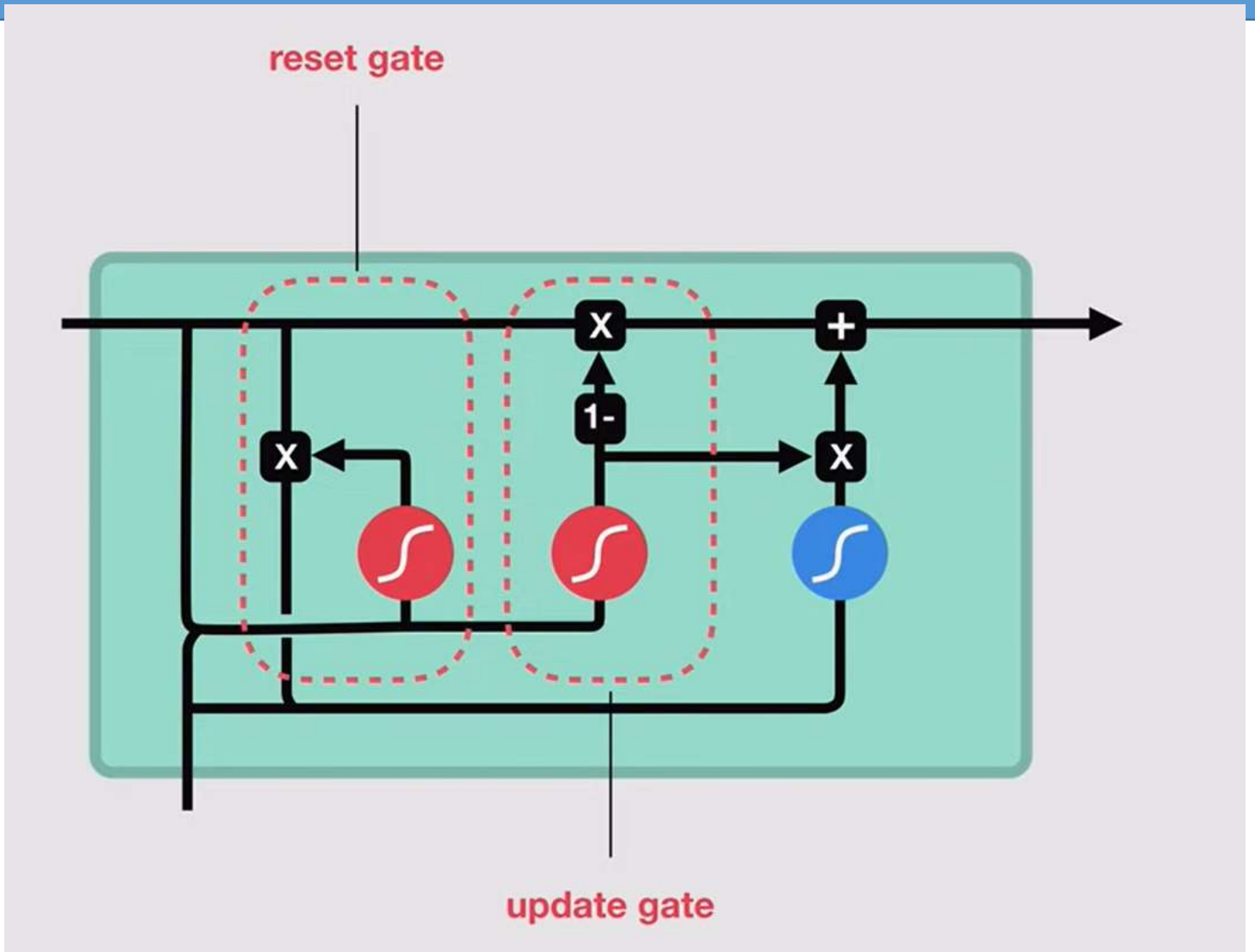
$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

Code is available at elearnnig with file NLP15_FakeNewsClassifierUsingLSTM.ipynb

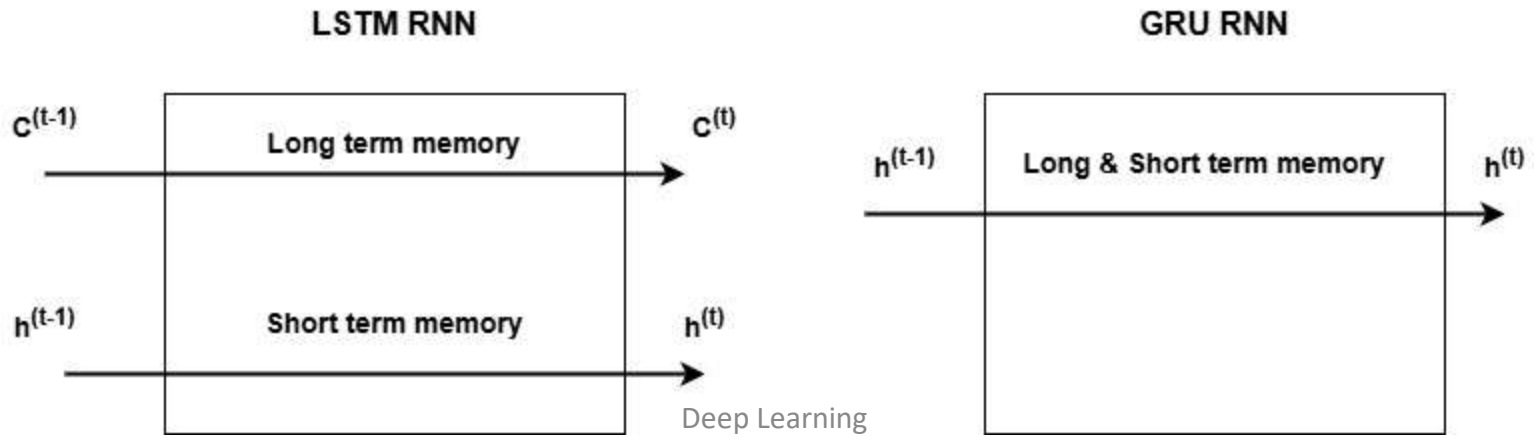
GRU (Gated Recurrent Unit) RNN

- **GRU** (Gated Recurrent Unit) is a type of Recurrent Neural Network (RNN).
- It can be called **lighter version of LSTM**.
- Designed to **address** issues like the **vanishing gradient** problem and **long-term dependency** learning.
- GRU simplifies the architecture compared to LSTM (Long Short-Term Memory) while **achieving similar results**.
- Key Features:
  - **Update Gate**: Controls how much of the previous hidden state should be passed to the current state.
  - **Reset Gate**: Determines how much of the previous memory to forget.
  - **Fewer Parameters**: Compared to LSTMs, GRUs use fewer gates, making them computationally more efficient.


reset gate
X
+
1-
X
X
update gate

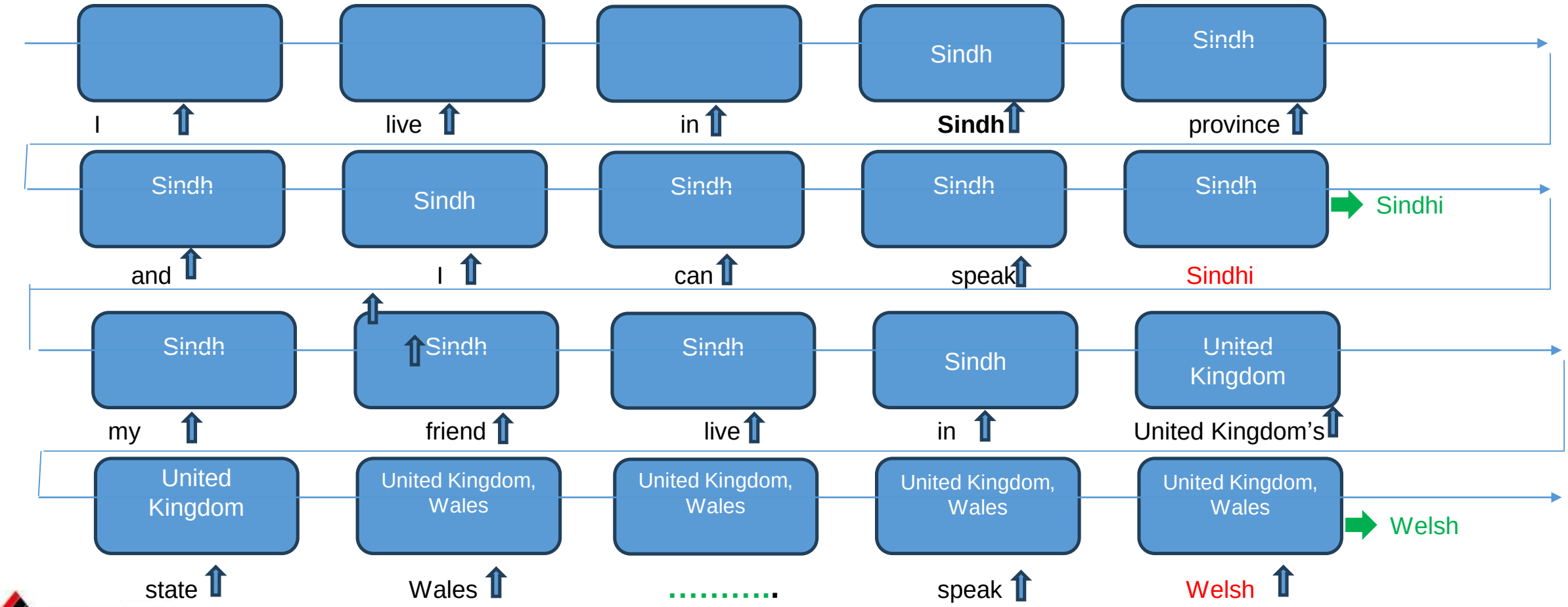
GRU (Gated Recurrent Unit) RNN

- **Update Gate** (z): Decides **how much** of the previous memory should be **carried forward**.
- It ranges between 0 and 1. If z is 1, it keeps the previous memory; if 0, it forgets the previous memory.
- **Reset Gate** (r): Controls **how much** of the past hidden state should **be ignored**.
- A value of 0 means no memory from the past; 1 means full memory.
- **Candidate Hidden State**: A new candidate state is generated (based on the reset gate) and combined with the update gate to create the final hidden state.



GRU (Gated Recurrent Unit) RNN

- To show the difference between update and reset gate, let's consider the following sentence: I live in **Sindh** province and I can speak _Sindhi_, whereas; my friend live in **United Kingdom's** state **Wales** and he can speak _Welsh_ ?



GRU (Gated Recurrent Unit) RNN

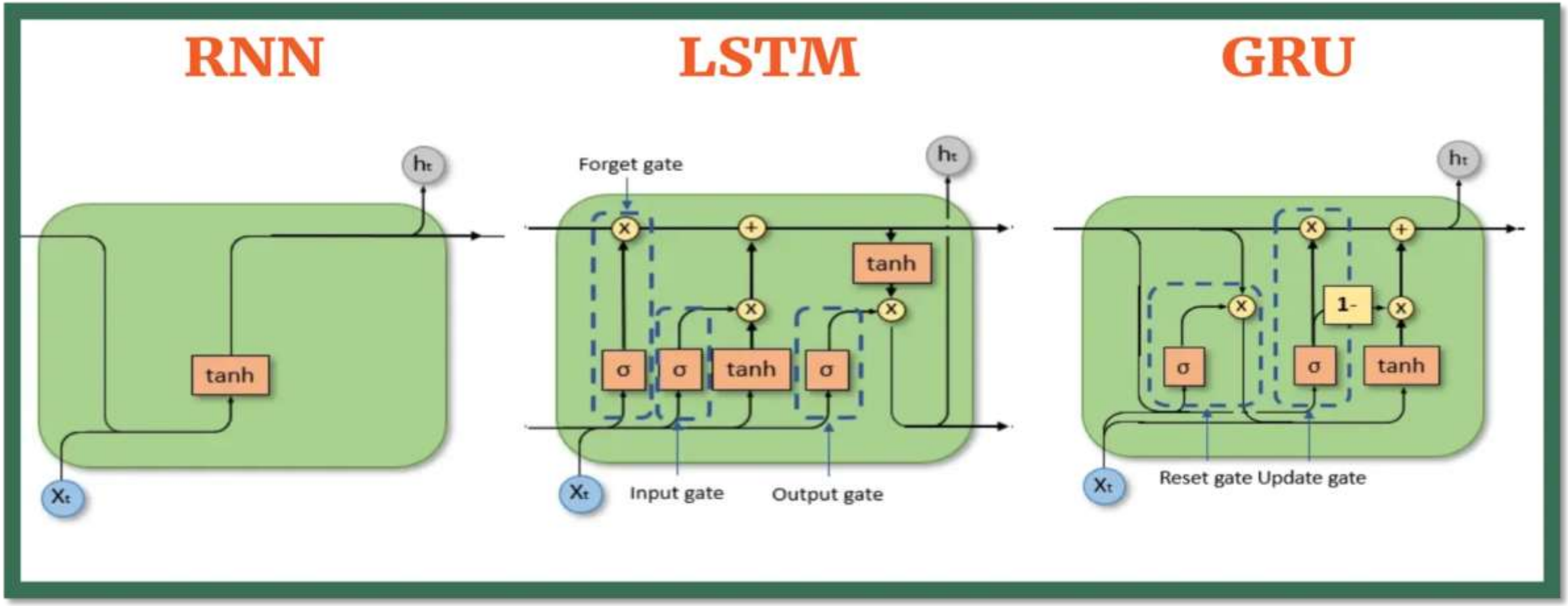
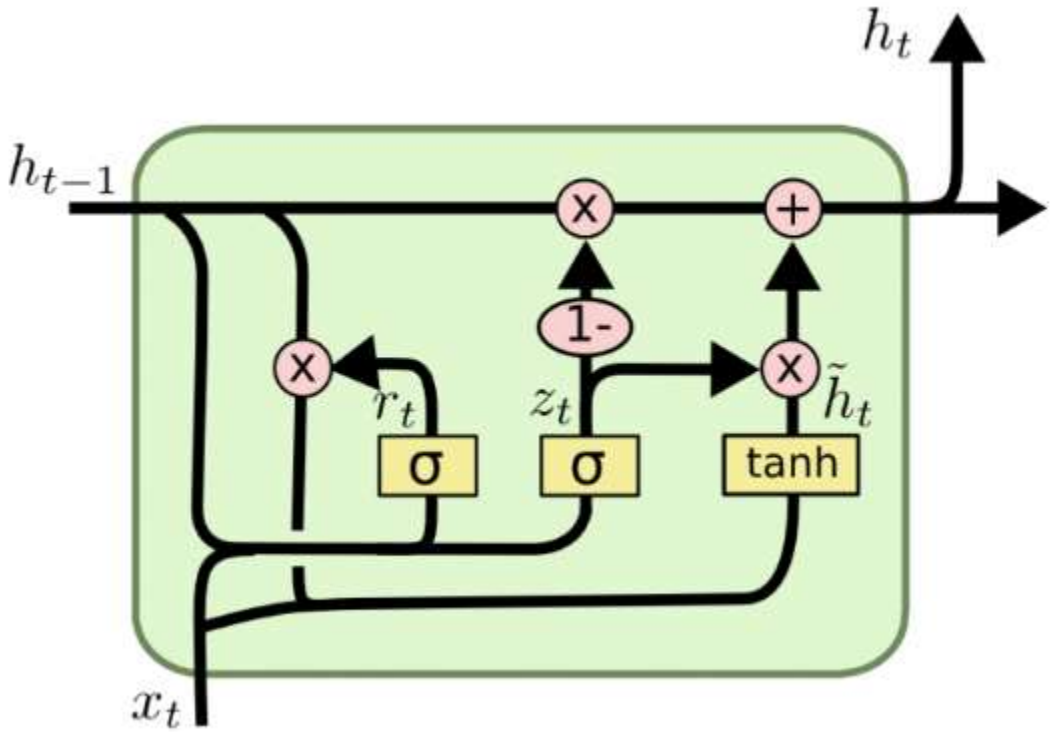


Diagram credit: https://python.plainenglish.io/introducing-gru-rnn-and-lstm-a-beginners-guide-to-understanding-these-revolutionary-deep-35b509a34a5a

GRU (Gated Recurrent Unit) RNN

- Optional details:



$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

$$\tilde{h}_t = \tanh(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h)$$

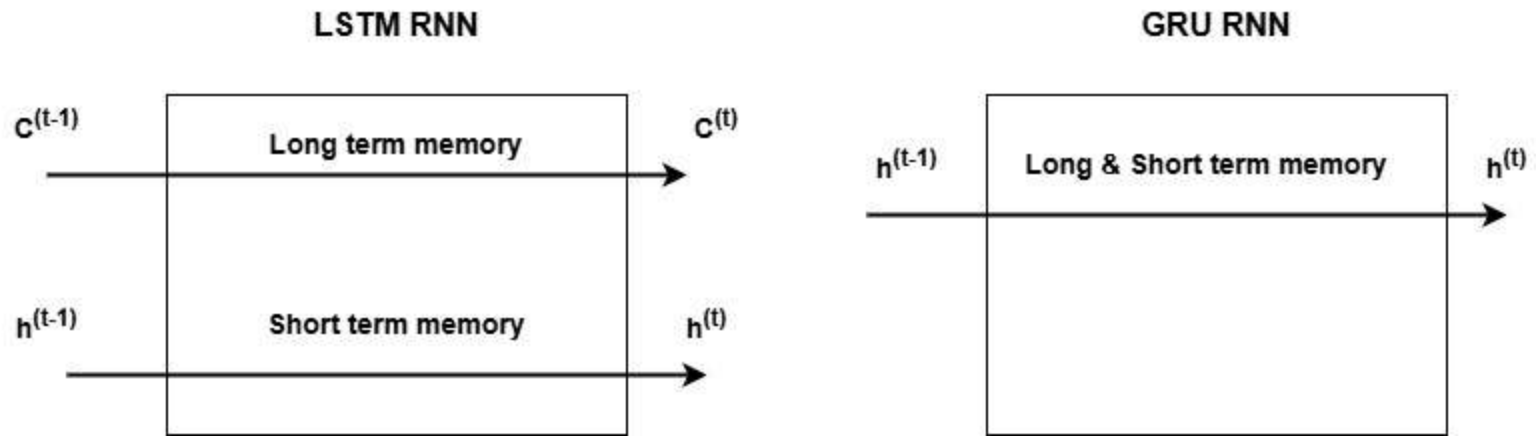
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

Diagram credit: http://dprogrammer.org/rnn-lstm-gru

GRU (Gated Recurrent Unit) RNN

- **Difference between LSTM & GRU**

LSTM	GRU
It has three gates input, output and update gate	It has two gates reset and update gate
More accurate in longer sequence but less efficient	Accurate mostly like LSTM however faster in computations
It has a cell state and carry hidden state	It has merged both cell & hidden into one cell state



GRU (Gated Recurrent Unit) RNN

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import GRU, Dense
import numpy as np

# Simulated data (1000 samples, 10 timesteps, 5 features)
X = np.random.rand(1000, 10, 5)
y = np.random.rand(1000, 1)  # Regression task

# Train-test split
train_size = int(0.8 * len(X))
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Define the model
model = Sequential([
    GRU(32, input_shape=(10, 5)),  # GRU with 32 units
    Dense(1)  # Output layer for regression
])

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))

# Evaluate the model
loss = model.evaluate(X_test, y_test)
print(f"Test Loss: {loss}")
```

Bi-directional LSTM

- Problems in Name Entity Recognition

Faheem loves apple, it keeps him healthy and happy

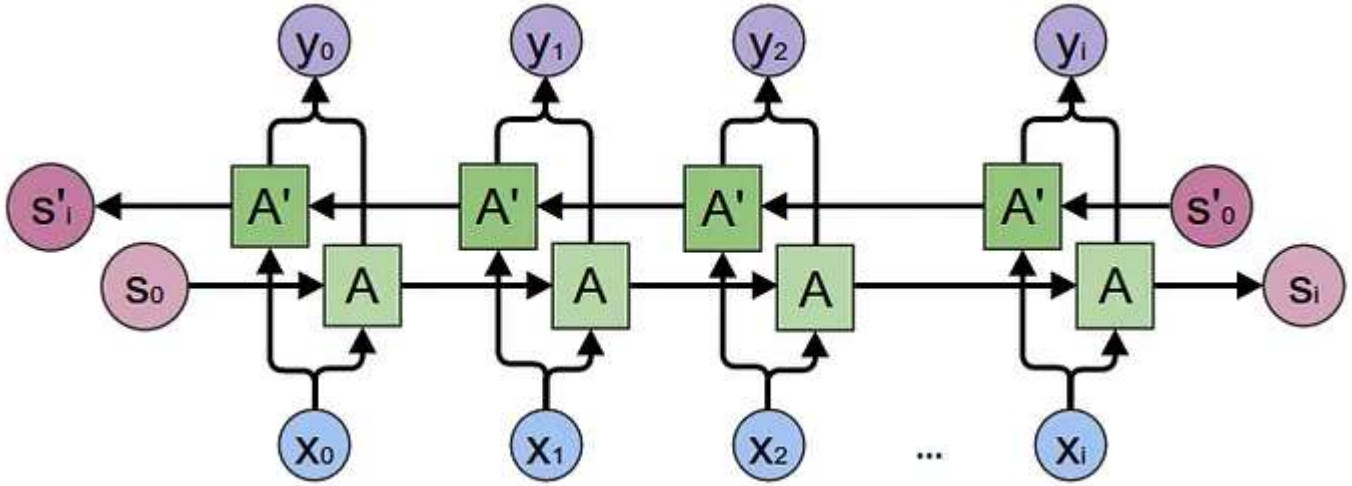
Ismail loves apple, the company produces best phones in the world.

Now what your RNN call apple and apple?

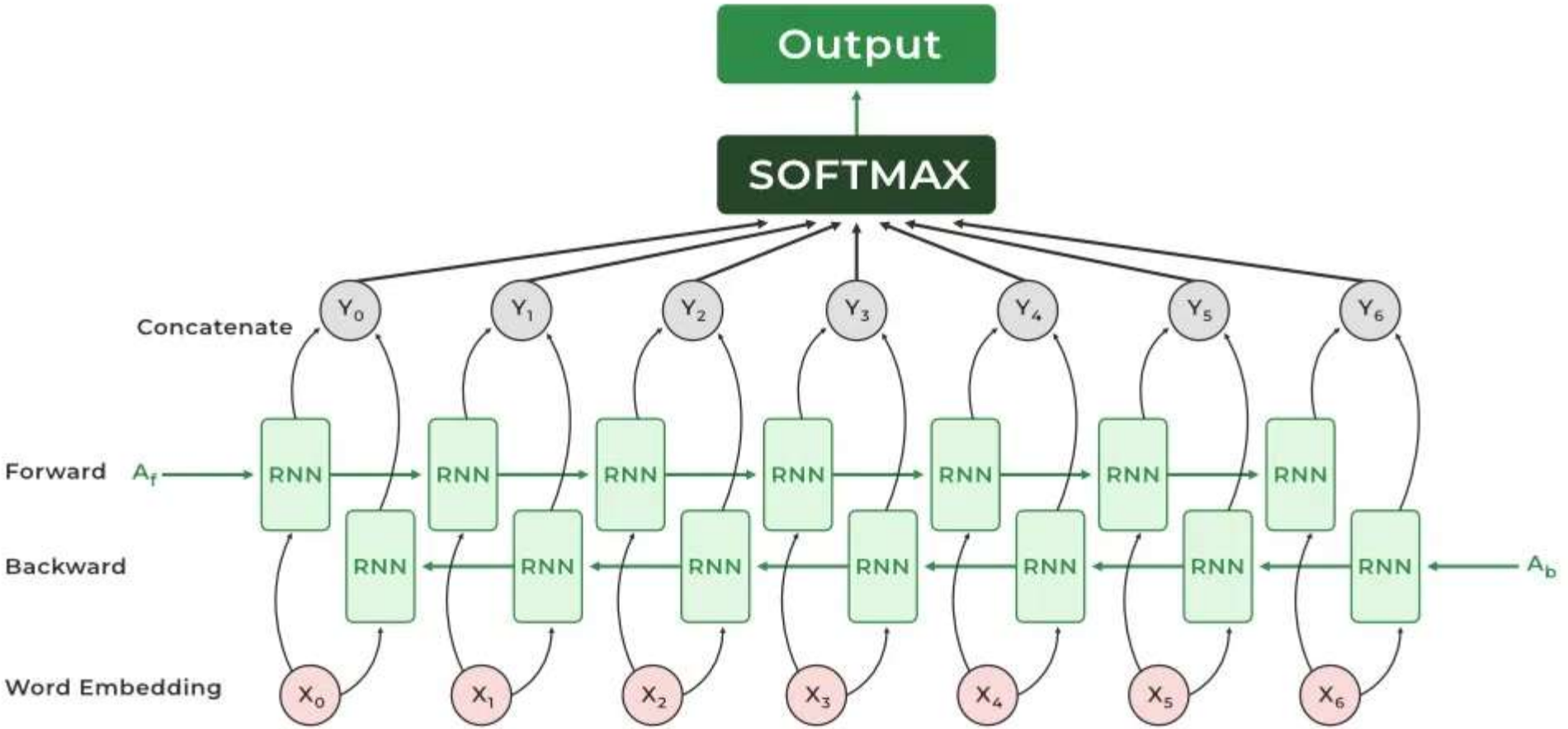
Create an LSTM with four neurons in hidden layer and unroll for both documents.

The RNN can make correct decision on apple from the words appearing in sequence after apple & apple.

Bi-directional LSTM



Bi-directional LSTM



Bi-directional LSTM

- Bi-directional LSTM (Bi-LSTM):
  - **Enhances LSTM** by processing the sequence in **both forward and backward directions**.
  - **Two LSTMs** are trained simultaneously:
    - **Forward LSTM** reads the sequence from left to right.
    - **Backward LSTM** reads the sequence from right to left.
- Why Bi-LSTM?
  - **Contextual Understanding**: It allows the model to capture context from both past and future elements in the sequence, which is critical for tasks like **speech recognition**, **machine translation**, and **named entity recognition**.
- Advantages of Bi-LSTM:
  - **Improved Accuracy**: By looking at both past and future contexts, Bi-LSTMs improve performance over traditional uni-directional models.
  - **Better Sequence Modeling**: More powerful in handling tasks where future context influences the interpretation of past inputs.
  - **Parallel Processing**: Both directions can be processed in parallel for efficient training.

Code is available at elearnnig with file NLP16_imdbClassifier_Bi-directional_LSTM.ipynb

Embedding layer

- An embedding layer is a component of neural networks used to map words or tokens to vectors of real numbers, allowing them to be processed by deep learning models.
- This transformation converts discrete, symbolic representations (words) into continuous vectors in a lower-dimensional space.
- The goal is to capture the semantic meaning of words, where similar words are represented by similar vectors.
- Why Use an Embedding Layer?
  - Reduces dimensionality compared to using one-hot encoding.
  - Encodes semantic information, allowing models to understand relationships between words.

Embedding layer working

1. **Input Representation**:
   - Each word is represented by an integer index corresponding to its position in a vocabulary.
2. **Embedding Matrix**:
   - The embedding layer contains a matrix of weights, where each row corresponds to a word index and each column corresponds to a dimension in the embedding space.
   - Typically, these embeddings are initialized randomly and are fine-tuned during training.
3. **Process**:
   - The input sequence (list of word indices) is passed through the embedding layer, which converts each index into its corresponding word vector.
   - These vectors are used as input to subsequent layers (e.g., LSTMs, CNNs) for further processing.

Embedding layer working

1. **Input Representation**:
   - Each word is represented by an integer index corresponding to its position in a vocabulary.
2. **Embedding Matrix**:
   - The embedding layer contains a matrix of weights, where each row corresponds to a word index and each column corresponds to a dimension in the embedding space.
   - Typically, these embeddings are initialized randomly and are fine-tuned during training.
3. **Process**:
   - The input sequence (list of word indices) is passed through the embedding layer, which converts each index into its corresponding word vector.
   - These vectors are used as input to subsequent layers (e.g., LSTMs, CNNs) for further processing.

- **Example of Word Embedding:**
  - Word: "King"Embedding Vector (dimension=3): [0.3, 0.7, 0.4]
- **Advantages of Using Embeddings:**
  - Semantic Relationships: Embeddings help capture relationships like synonyms, antonyms, and word analogies (e.g., "King - Man + Woman = Queen").
  - Efficient Computation: Embedding layers reduce the dimensionality compared to methods like one-hot encoding.

Code is available at elearnnig with file NLP14_Word_embedding_Layer.ipynb

Embedding layer training

- How Does the Embedding Learn?
- Points:
  - Initialized randomly or from pre-trained models (Word2Vec, GloVe).
  - Updated during model training (e.g., sentiment classification).
  - Backpropagation adjusts word vectors to minimize loss.
  - Example: “Good” moves closer to “Excellent” in vector space.
- Using Pre-Trained Word Embeddings
- Points:
  - Pre-trained on large corpora → better semantic understanding.
  - Common models: Word2Vec, GloVe, FastText.
  - Embedding layer can load weights instead of training from scratch.

Code is available at elearnnig with file NLP14_Word_embedding_Layer.ipynb

Embedding layer training

```
#
# 1. Load Dataset
# ------------------------------
# IMDb dataset (preprocessed into integers)
vocab_size = 10000  # use top 10,000 words
max_length = 200    # truncate / pad sequences to 200 words

(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=vocab_size)

print("Training samples:", len(X_train))
print("Test samples:", len(X_test))

# ------------------------------
# 2. Preprocess Data (Padding)
# ------------------------------
# Pad sequences so they all have the same length
X_train = sequence.pad_sequences(X_train, maxlen=max_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_length)

print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)

# ------------------------------
# 3. Define Model with Embedding Layer
# ------------------------------
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=32, input_length=max_length),
    Flatten(),            # flatten embeddings before Dense layers
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')
])
```

Code

How can I install Python lib

seq2seq models

- Seq2Seq is a deep learning model architecture used for transforming one sequence of data into another sequence.
- It is commonly used in NLP tasks where both the input and output are sequences, such as:
- Machine Translation (e.g., English → French)
- Text Summarization
- Speech Recognition Image Captioning

Encoders & Decoders

- **Encoder**:
  - It processes the input sequence (e.g., a sentence in a language) and transforms it into a fixed-size context vector that summarizes the information.
  - It is responsible for capturing the relevant features of the input to pass on to the decoder for generating output.
- **Components of the Encoder**:
  - **Input Sequence**:
    - A sequence of tokens (words or characters) from the input text.
    - Example: "Hello, how are you?“
  - **Embedding Layer**:
    - Converts words into continuous vector representations (embeddings) to make them understandable for the model.
  - **Recurrent Neural Networks (RNNs), GRUs, LSTMs**:
    - Common architectures used in the encoder to process input sequences.
    - RNNs, GRUs, and LSTMs are capable of handling sequential data and learning long-range dependencies.
  - **Context Vector**:
    - The output of the encoder is a fixed-length vector (often referred to as the context vector), which contains all the encoded information from the input sequence.

Encoders & Decoders

- **Decoder**:
  - The Decoder takes the context vector from the encoder and generates the output sequence (e.g., a translated sentence, summary, or response)..
- **Components of the Encoder**:
  - **Initial input (Context Vector)**:
    - The decoder receives the context vector from the encoder, representing the input sequence's summary.**Embedding Layer**:
    - Converts words into continuous vector representations (embeddings) to make them understandable for the model.
  - **Recurrent Neural Networks (RNNs), GRUs, LSTMs**:
    - Common architectures used in the encoder to process input sequences.
    - RNNs, GRUs, and LSTMs are capable of handling sequential data and learning long-range dependencies.
  - **Context Vector**:
    - The output of the encoder is a fixed-length vector (often referred to as the context vector), which contains all the encoded information from the input sequence.

Summary

- Introduced RNN
- Discussed problems with RNN
- LSTM introduced to overcome vanishing gradient of RNN
- GRU RNN introduced