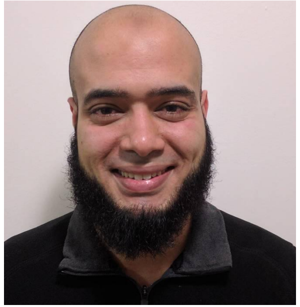
# Machine Learning

# Recurrent Neural Network

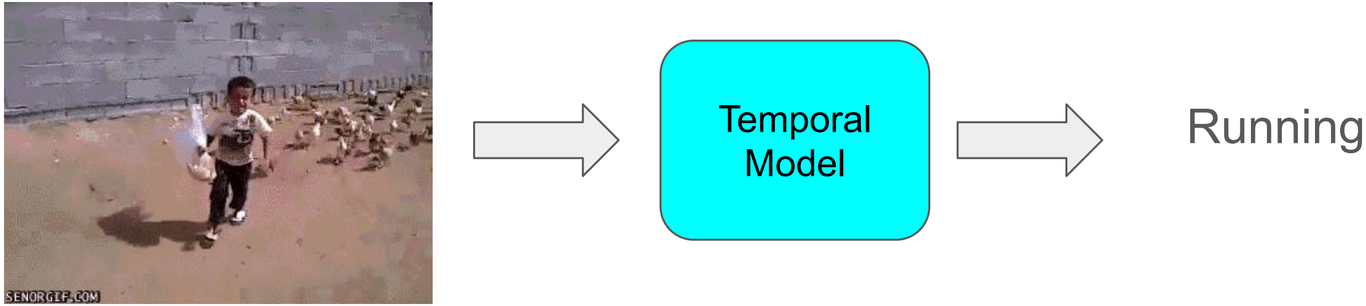

**Mostafa S. Ibrahim**
*Teaching, Training and Coaching for more than a decade!*

*Artificial Intelligence & Computer Vision Researcher*
*PhD* from Simon Fraser University - Canada
*Bachelor / MSc* from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)

# Modeling Sequential Data

- One way for this video is just to prepare as a single input
  - For example, create 3D tensor for 100 frames as 100x256x256  (3D video processing)
  - For example, for N words, just concatenate as a single string
  - For stock prices of last 17 months, the input is just all the days together
    - So the input vector is 17 x 365 (representing the whole history) features
- Good maybe not flexible for several applications and needs
  - Also computationally maybe more expensive (e.g. memory wise)



# Step-by-Step Processing

- Another interesting approach is still to process them step by step!
  - For example, for a 100-frames video, we feed 100 network inputs (each is frame)
  - For example, for 14-words sentence, we feed 14 network inputs (each is word)
- Imagine with every step, we still get our **output logits** that we can apply softmax over it to get the probabilities over C classes

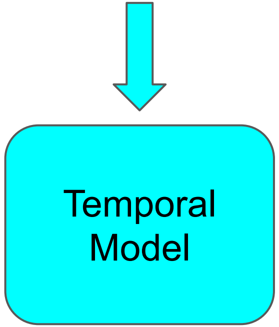
# Step-by-Step Processing

- Assume our video is 5 frames. *Below networks have the same weights*

| | | | | |
|---|---|---|---|---|
|  |  |  |  |  |
| Temporal Model | Temporal Model | Temporal Model | Temporal Model | Temporal Model |
| Running p=0.4 | Running p=0.6 | Running p=0.7 | Running p=0.85 | Running p=??? |

# Step-by-Step Processing

- But these frames are not separate IIDs.
- Each step **depends** on ALL the previous steps
  - Assume each frame is represented with a feature vector
  - The k-th frame depends on **all previous k-1 frames!**
  - *How many frames? Input could be **varying** sequence*
- Within that, the performance of the last frames still should align with most of the previous
  - Here, the probability of all the frames TOGETHER to be running is 0.95
  - However the probability of this frame independently = 0.0
- How can we process the k-th frame jointly with the last k-1 frames? Using memory of the history!

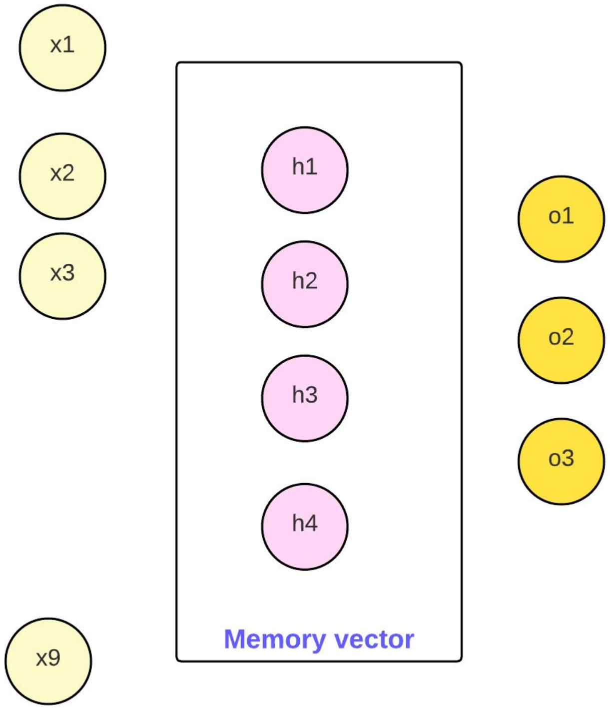

Running p=0.95

# Step-by-Step Processing using **Memory**

- One interesting approach is to summarize the whole history in a single vector
  - This state vector represents our history so far
  - It represents the model **memory** (of the history). We call it the state vector
  - The length of the state feature vector is typically of the **same** length as the step vector
- Then at each step we have **2 inputs** only
  - The k-th feature vector for the k-th frame (word / input)
  - A **state feature** vector representing the frames from 1 to k-1
  - After each k-th step, the state vector will be **updated** to represent inputs from 1 to k!
- This way we kept our processing simple
  - We think of our history as a **single state** vector NOT k-1 vectors

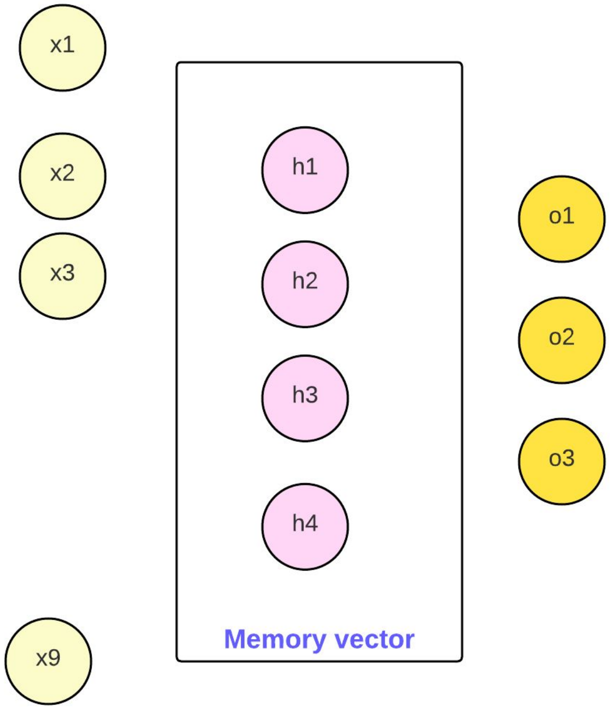
# RNN Hidden Layer

- RNN is **Recurrent** Neural Network
- An RNN layer is just like a normal hidden layer
  - If you feed N vectors, you get N vectors
  - The only difference, it keeps an internal **memory** to fuse in the normal hidden outputs
  - You either utilize **all** the outputs, or the **last** output!
- If the hidden layer size is 4 features, then also the memory vector size is 4
  - Initially, we **set** the memory to **zeros** for each sequence
  - Then with each time step, the memory is updated
  - The output of k-th step depends on both the previous memory (k-1) and the current input

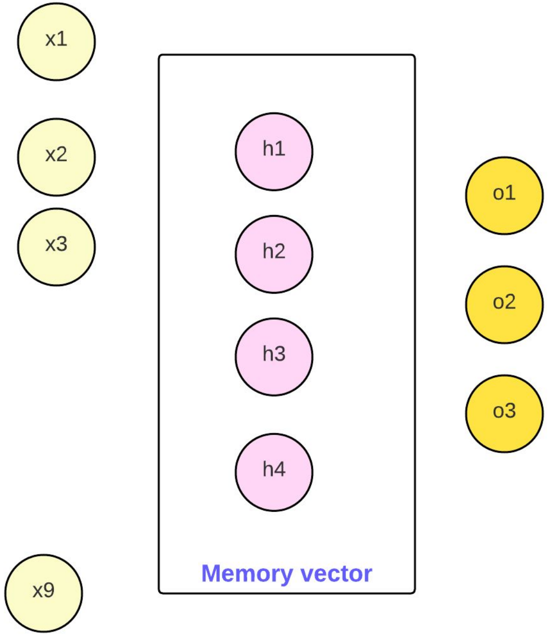


# RNN Hidden Layer

- In a normal hidden layer
  - We will have 2 weight matrices and their biases
    - Input to hidden (our focus) and hidden to output
    - Let's call them W_xh and W_ho
- One can represent the history as the sum of the hidden states so far
  - Let the previous memory is vector s (initially zeros)
  - Feed kth input X to W_xh to get h_k vector
  - Update s to s = s + h_k  (sum of vectors)
  - Apply non-linear transformation: s = tanh(s)
- However, this is so naive memory representation



- Let’s create a simple network
- We will init the 2 weight matrices
- Assume each frame is 9 features
- Assume we have 1000 videos
    - Each video length (N) varies from 10 to 50
        - Doesn’t matter!
- Assume our network outputs 3 nodes for 3 multi-class categories




```python
class SimpleRNN:
    def __init__(self, input_size=9, hidden_size=4, output_size=3):
        # W_xh and W_ho are like old ones.
        self.W_xh = np.random.randn(hidden_size, input_size)    # input to hidden      4x9
        self.W_ho = np.random.randn(output_size, hidden_size)   # hidden to output     3x4
        self.b_xh = np.zeros((hidden_size, 1))  # hidden bias
        self.b_ho = np.zeros((output_size, 1))  # output bias
```

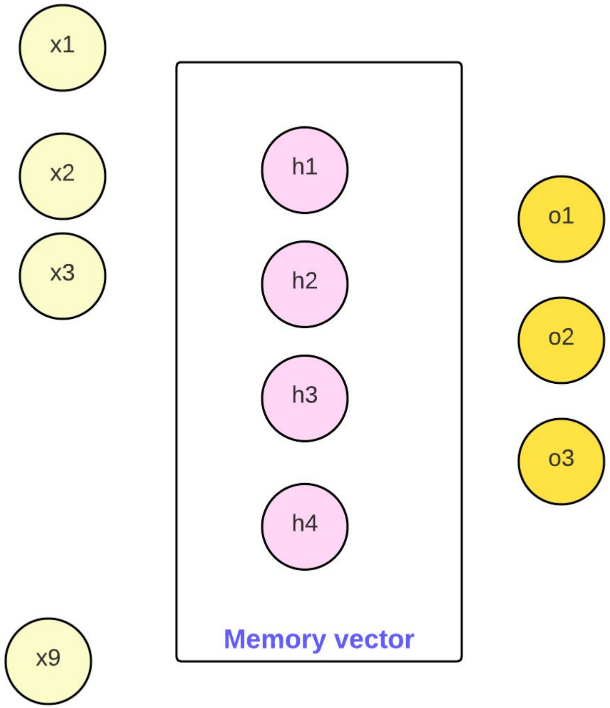
- Inputs is a single sequence of N steps, each has 9 features

```python
def forward(self, inputs):
    steps_output, hidden_states = {}, {}
    hidden_states[-1] = np.zeros((self.W_xh.shape[0], 1))   # no history at idx -1

    # feed each input while utilizing its history
    for t in range(len(inputs)):
        x = np.array(inputs[t]).reshape(-1, 1)               # 9x1
        # Normal input to hidden transformation (embedding)
        hidden_cur = np.dot(self.W_xh, x) + self.b_xh        # 4x9 * 9x1 + 4x1 = 4x1


        hidden_states[t] = hidden_states[t-1]
        hidden_states[t] += hidden_cur                   # Element-wise addition cur + old
        hidden_states[t] = np.tanh(hidden_states[t])# Non-linear transformation
                                                             # to enhance addition

        # Normal hidden to output transformation
        steps_output[t] = np.dot(self.W_ho, hidden_states[t]) + self.b_ho

    return steps_output, hidden_states
```

# RNN Hidden Layer

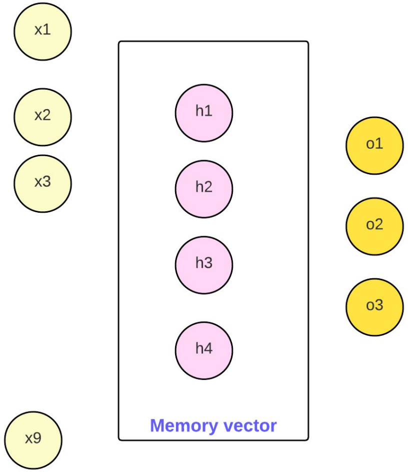
- To avoid a trivial sum operation, we need to introduce more complex transformations
  - *Not so trivial, still has tanh*
- One way is to introduce a new weight matrix, let’s call it W_hh, which is an internal weight matrix to transform the old memory to the current time step
  - So its size is hidden x hidden
  - It connects the hidden layer to **itself**
  - This is a **recurrent** link

```
self.W_hh = np.random.randn(hidden_size, hidden_size)
self.b_hh = np.zeros((hidden_size, 1))
```



# RNN Hidden Layer

- Only one line of code change
  - We transform first the previous hidden state to current step
  - Then add to that the current hidden output
  - Then use with tanh
- Now the hidden state after the k-th step is series of transformations and accumulations of the hidden outputs from W_xh
  - Learned Transformation. Accumulate. Non-linear Activation




```
# Transform the previous hidden state to the current timestep (avoid naive addition)
hidden_states[t] = np.dot(self.W_hh, hidden_states[t-1]) + self.b_hh
hidden_states[t] += hidden_cur                    # Element-wise addition cur + old
hidden_states[t] = np.tanh(hidden_states[t])# Non-linear transformation
                                                   # to enhance addition
```

# RNN Hidden Layer: As an Equation

- This layer can be written mathematically as following

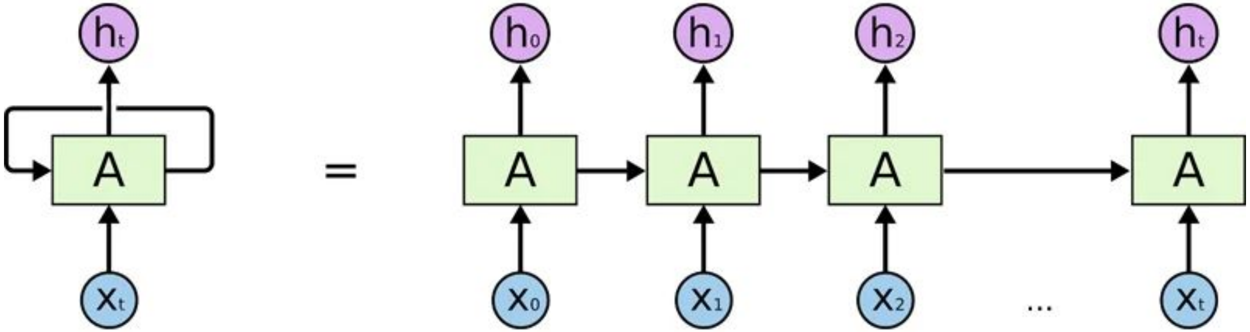
For each element in the input sequence, each layer computes the following function:

$$h_t = \tanh(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh})$$

where $h_t$ is the hidden state at time *t*, $x_t$ is the input at time *t*, and $h_{(t-1)}$ is the hidden state of the previous layer at time *t-1* or the initial hidden state at time *0*. If `nonlinearity` is `'relu'`, then $\text{ReLU}$ is used instead of $\tanh$.

# RNN Hidden Layer: As a Diagram

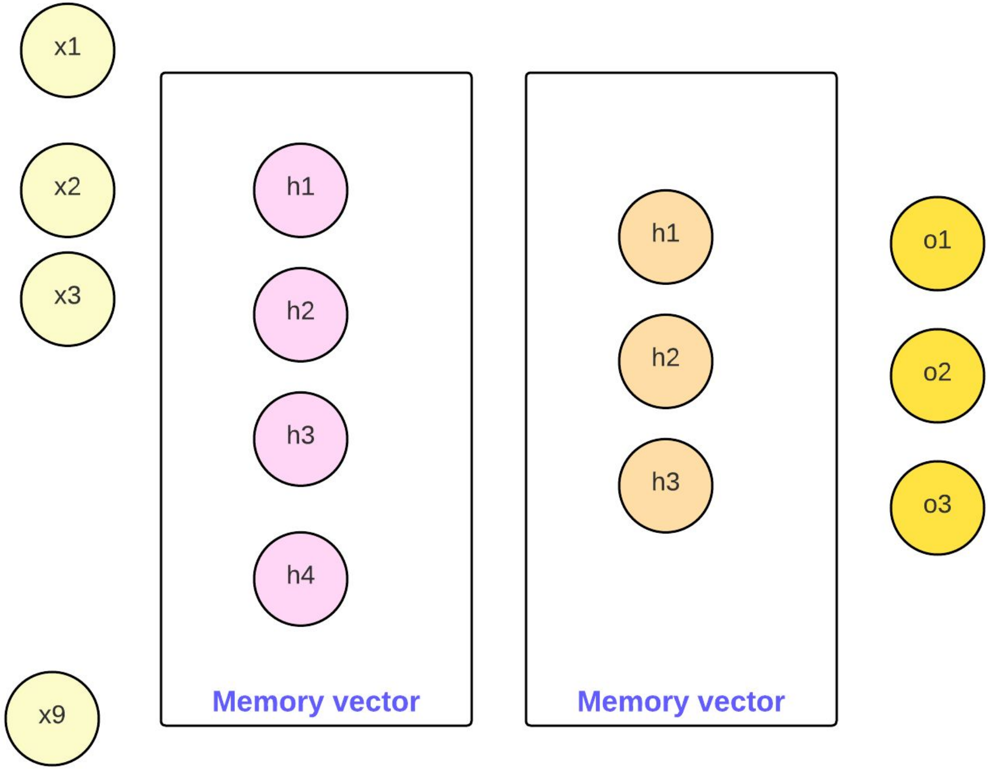
- There are 2 common ways to visualize an RNN
- Left side; Rolled version where the RNN operation requires 2 inputs
  - X_t, the X input at time t.
  - A recurrent link that keeps feeding the previous hidden state
- Right side: unroll it over steps
  - Again the same 2 inputs




Img src

# Multi-RNNs

- We can design a network with e.g. 5 hidden layers
- Similarly,  we can have e.g. 5 RNN layers
  - As black box, you give vector and take vector that utilizes its own memory
  - Each RNN has its own extra weight matrices
  - Each RNN layer receives the final output from the previous RNN



"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."