Introduction to the Recurrent Neural Networks

Moein Enayati

April 2019

Solve an example with NN

$\begin{bmatrix}1\\0\end{bmatrix}$

$\begin{bmatrix}0\\1\end{bmatrix}$

$\begin{bmatrix}1\\0\\0\end{bmatrix}$

Apple pie

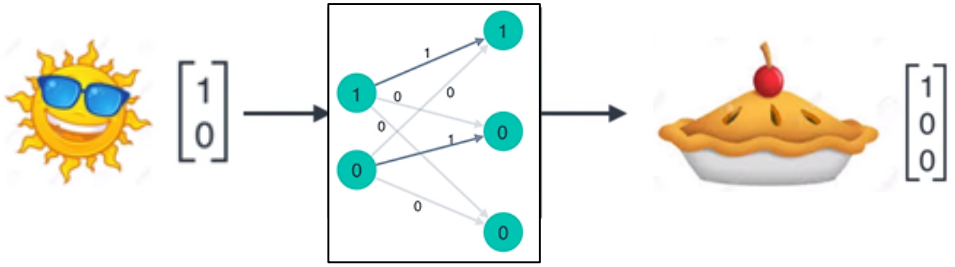
$\begin{bmatrix}0\\1\\0\end{bmatrix}$

Burger

$\begin{bmatrix}0\\0\\1\end{bmatrix}$

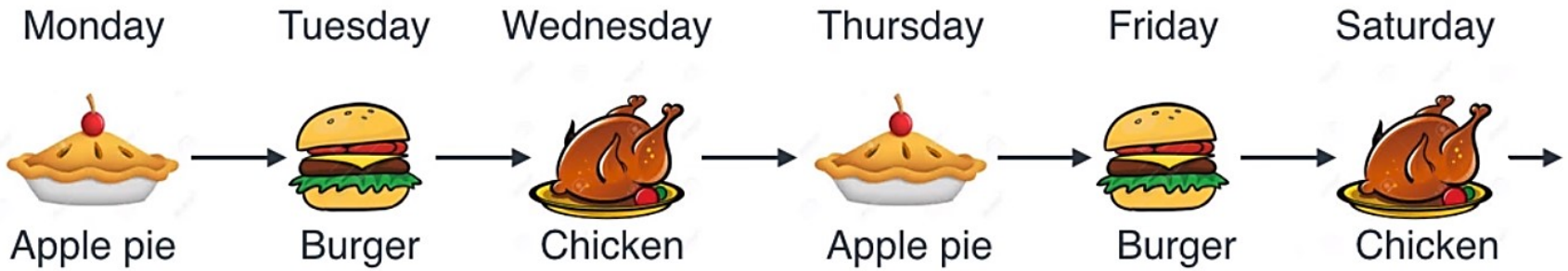
Chicken

Model it using a normal NN

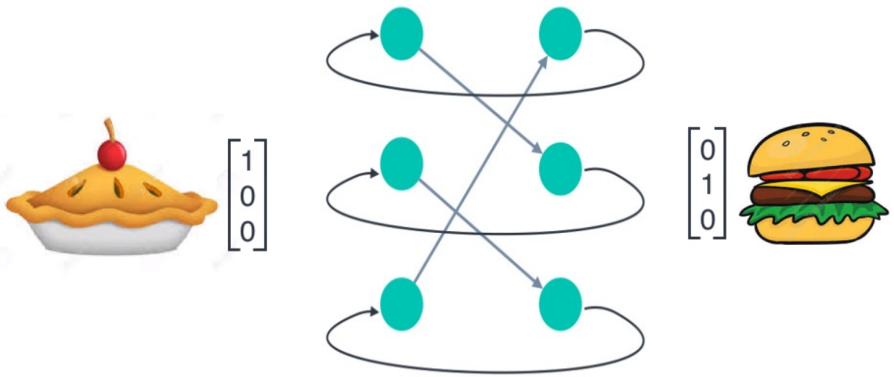


The output only depends on the weather condition

Example 2: Time dependency



Here, each step is only and only depends on the previous state,
The output goes back as the input for the next step.



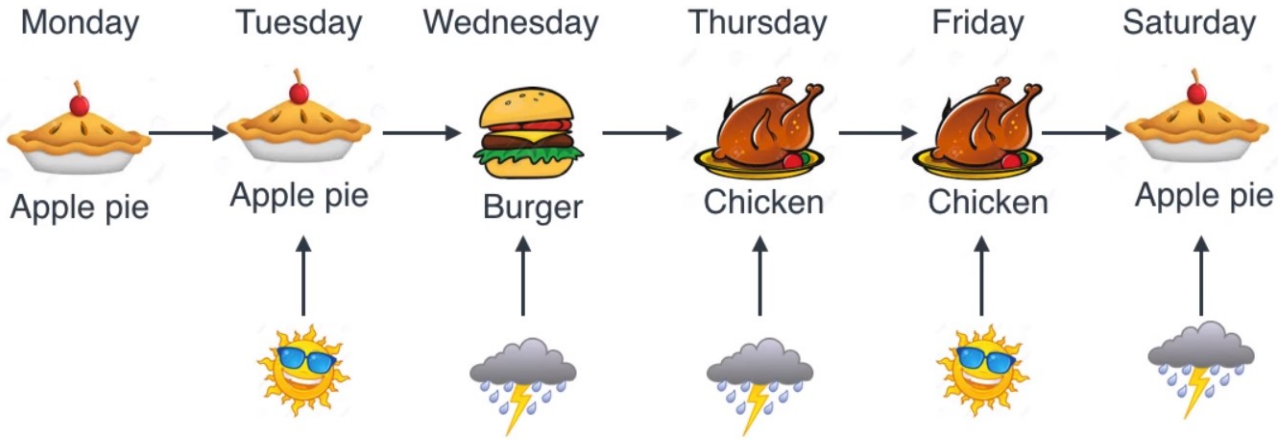
$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

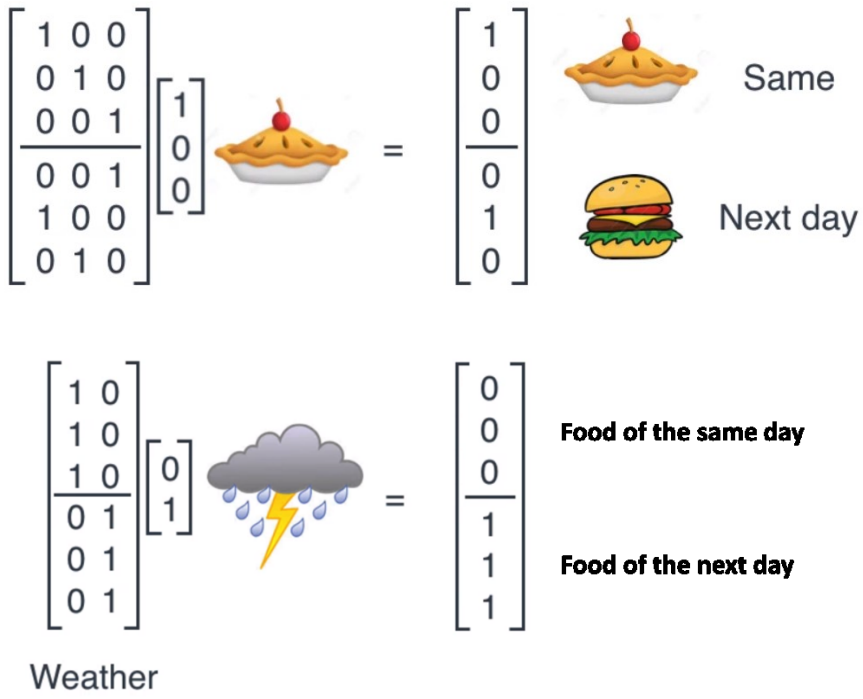
It is a simple linear transformation

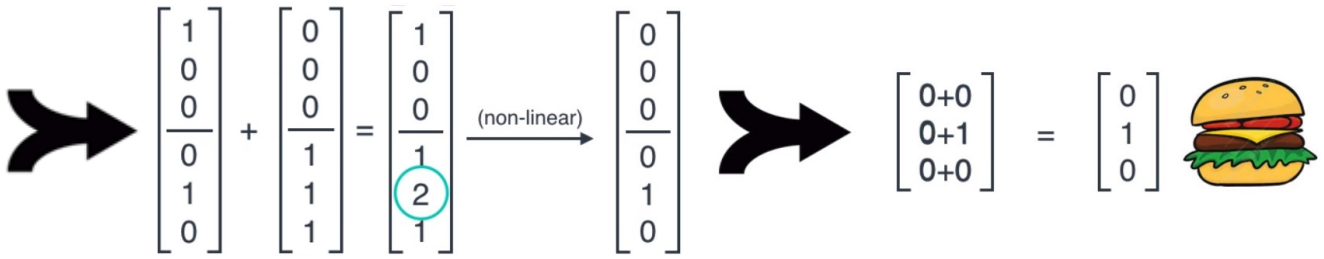
Example 3: The combined version

What if we combine the two previous conditions:

- If it's rainy, we cook the next thing in the sequence
- If it's sunny, we go out, and eat what we cooked yesterday.



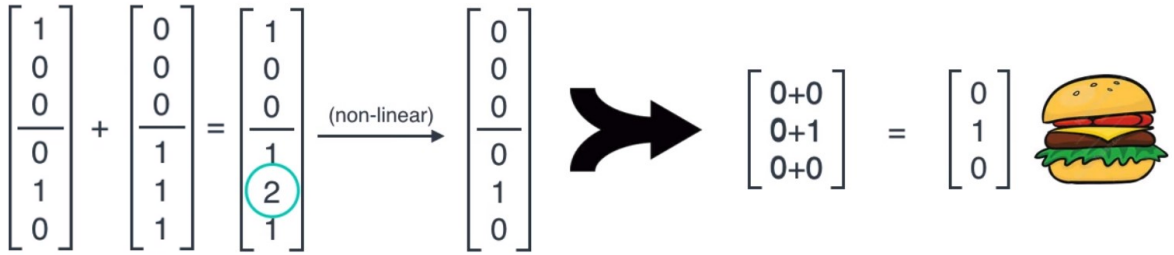


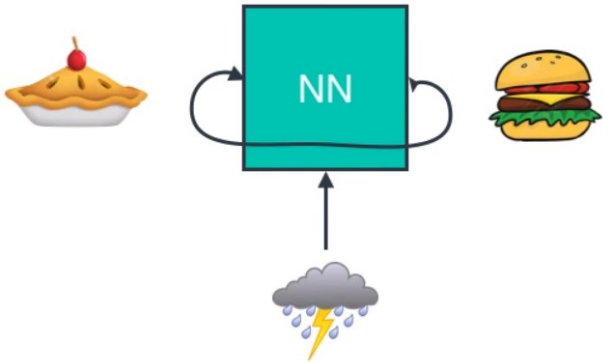


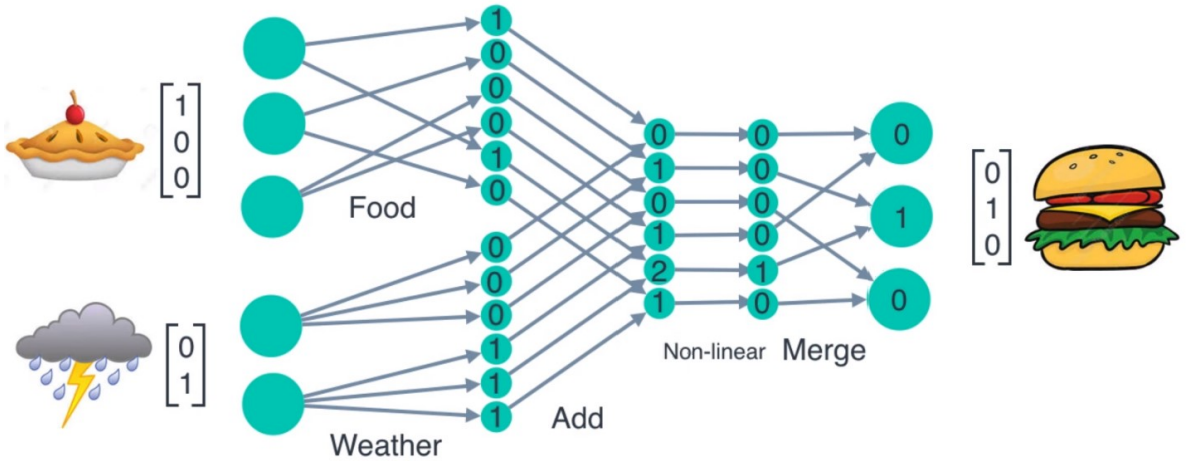
Add the output of the two NNs

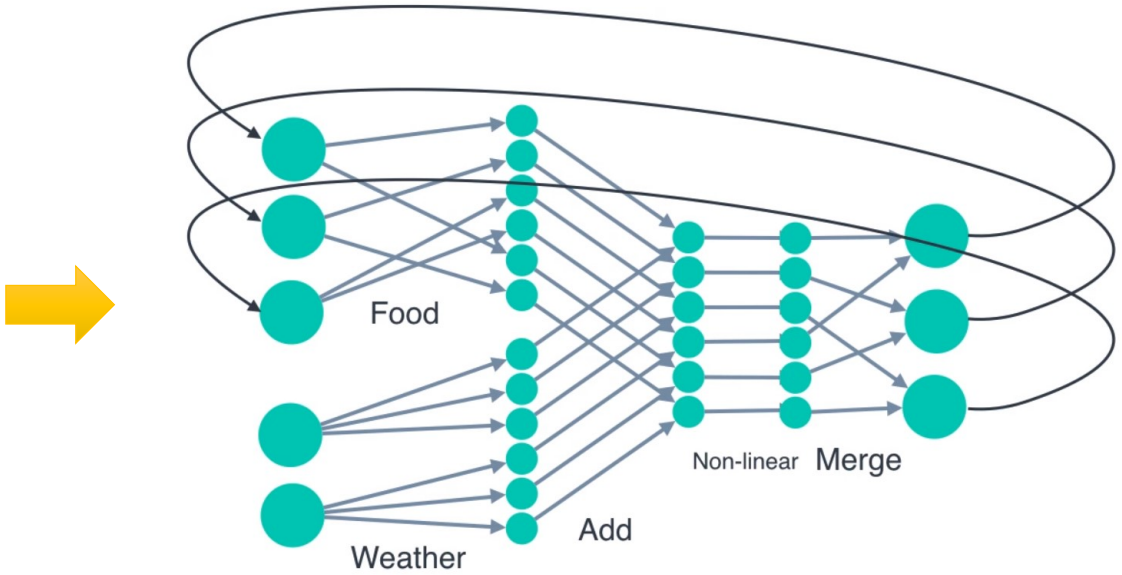
use a one hot encoding to convert the result to a new matrix

The recurrent nature of the problem









Problems with traditional NNs? Limitations | General form

- One-to-one relation between the inputs and outputs.
- Won't share features it learns from different positions.
  - Two equivalent but modified sentences.

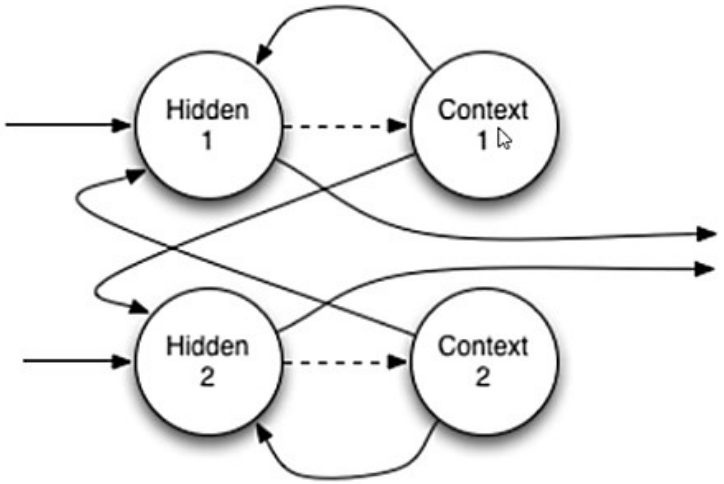
RNN is the solution:

- **Round 1:**
  - Read the first word X1
  - feed it to the RNN (learn)
  - try to predict Y1
- **Round 2:**
  - Read the second word X2
  - Also use some of the learned information from the previous time step
  - Predict Y2

Before RNNs

Analysis of time series using neural networks required feature engineering , over time
(average of a sequence)
being used as the input of NN

General Form



RNN is basically a neuron with some concept of memory, A.K.A. the "context". It can be as simple as a copy of the cell's output at the previous time stamp.

What RNNs are used for? Time|Order|Sequence

Image Classification

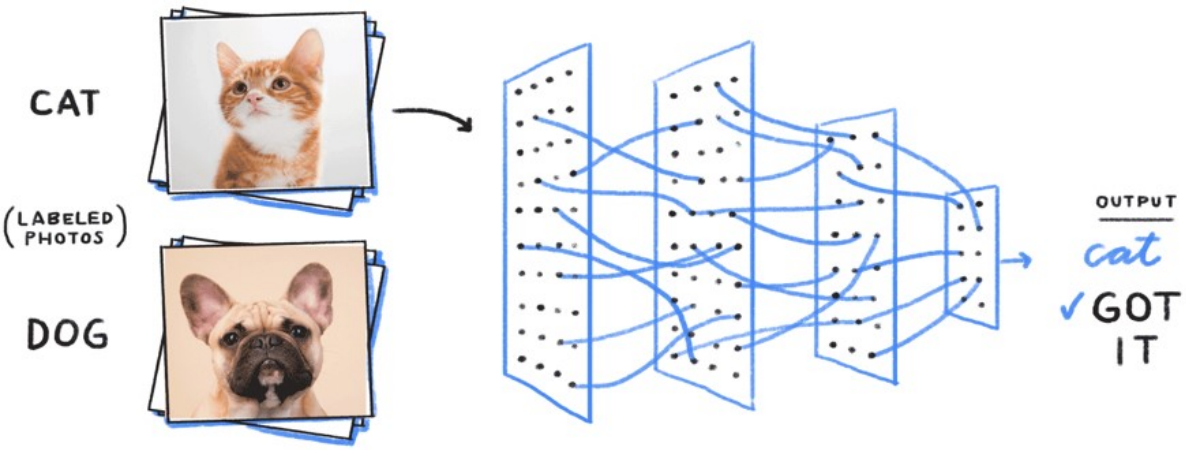


Image Captioning

Detected Objects:
cat: 1, **suitcase**: 0.96, bag: 0.89, luggage: 0.65, black: 0.63

LSTM-C: a cat laying on a **suitcase**

Detected Objects:
cat: 1, **bottle**: 0.98, wine: 0.84, black: 0.58, standing: 0.58

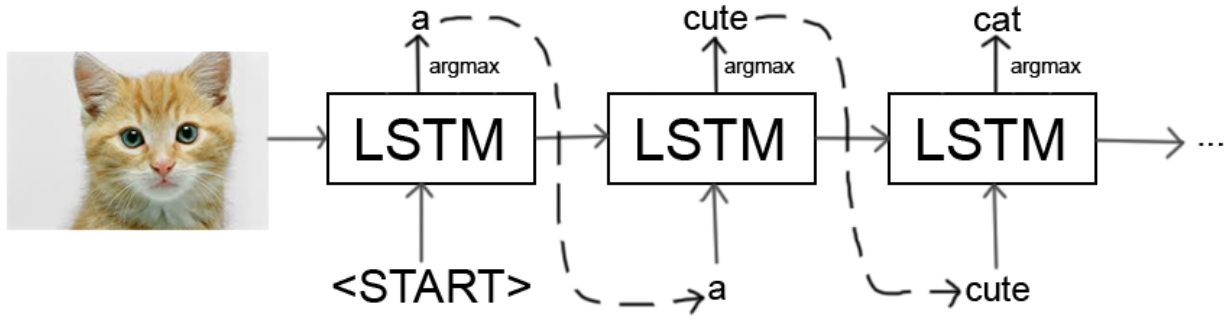
LSTM-C: a cat sitting on a table next to a **bottle** of wine

Video Captioning

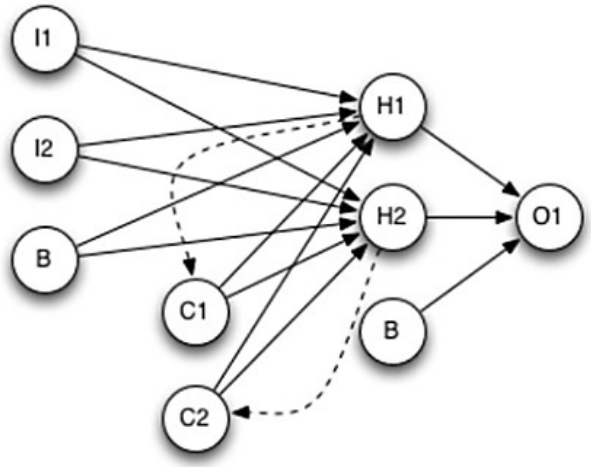
how different objects (frames) interact over time

It knows the "language model",

how different objects (words) interact in a sentence



RNN Background | Elman Network

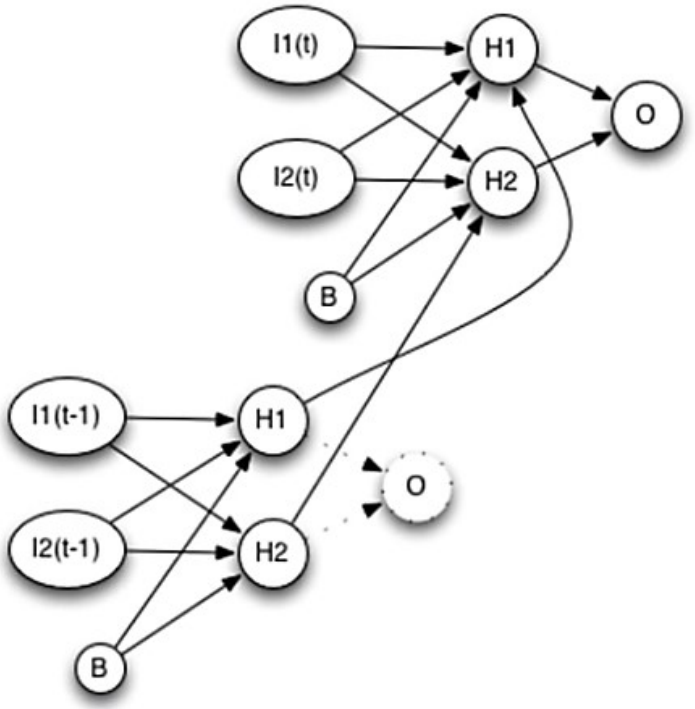


- Has only one layer.
- Always has bias values.
- Number of context neurons = Number of hidden neurons.
- The value from hidden neurons -> Copy to the context neurons.
- Use them as extra inputs during the next call/step/time

Result: Short term memory

Backpropagation through time

Unfold Elman Net and turn it to a new NN which constantly unfolds while visiting more samples of the sequence.



Train Just like the normal backpropagation, With a new constraint on weights:
Weight vectors are equivalent over time and don't change. (Shared weights)

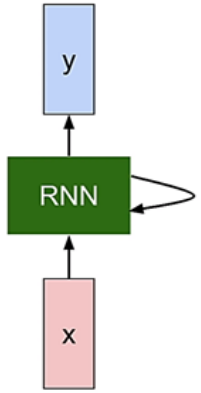
$$To\ constrain: \quad w_1 = w_2 \qquad we\ need: \quad \Delta w_1 = \Delta w_2$$

$$\dashrightarrow compute: \quad \frac{\partial E}{\partial w_1} \quad and \quad \frac{\partial E}{\partial w_2} \dashrightarrow use \quad \frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2} \quad for\ w_1\ and\ w_2$$

Vanilla RNN | Formulation

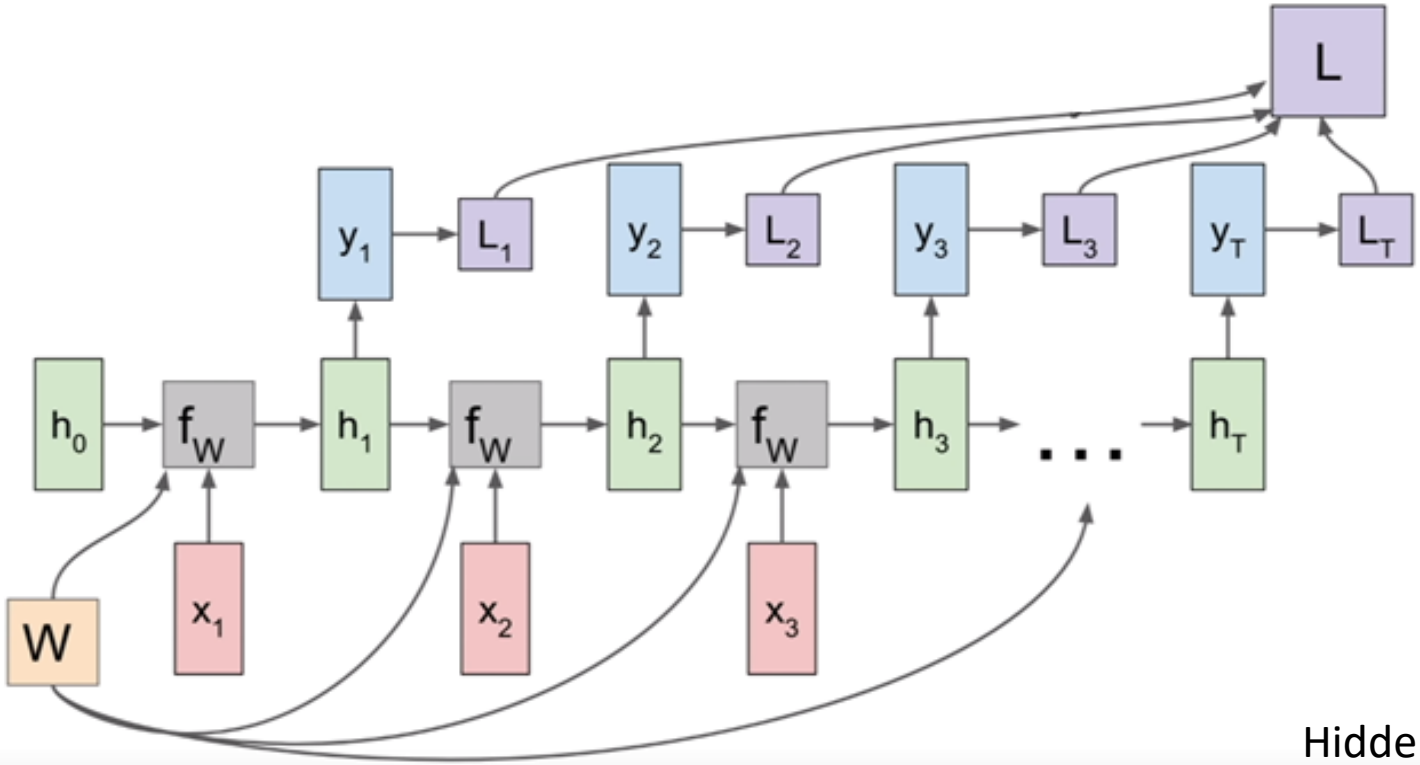
At every step in time, uses

- the new sample from the sequence
- the previous context or state



$$h_t = f_W(h_{t-1}, x_t)$$

new state — some function with parameters W — old state — input vector at some time step



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

$$y_t = W_{hy}h_t$$

Input
Hidden state

X weight matrix -> Pass through non-linearity

Actually are using the same W and F for the entire graph

Vanilla RNN | Formulation

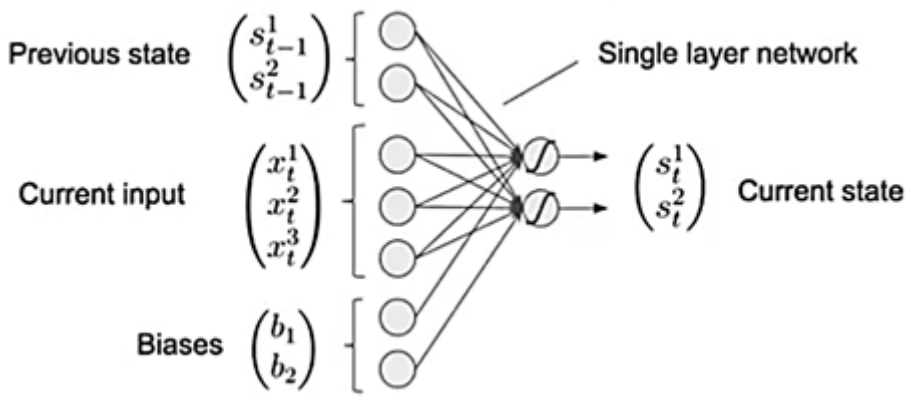
$$s_t = \varphi(W s_{t-1} + U x_t + b)$$

$$\begin{pmatrix} s_t^1 \\ s_t^2 \end{pmatrix} = \varphi\left(\begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix}\begin{pmatrix} s_{t-1}^1 \\ s_{t-1}^2 \end{pmatrix} + \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \end{pmatrix}\begin{pmatrix} x_t^1 \\ x_t^2 \\ x_t^3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}\right)$$

$$= \varphi\left(\begin{pmatrix} w_{11}s_{t-1}^1 + w_{12}s_{t-1}^2 \\ w_{21}s_{t-1}^1 + w_{22}s_{t-1}^2 \end{pmatrix} + \begin{pmatrix} u_{11}x_t^1 + u_{12}x_t^2 + u_{13}x_t^3 \\ u_{21}x_t^1 + u_{22}x_t^2 + u_{23}x_t^3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}\right)$$

$$= \varphi\left(\begin{pmatrix} w_{11}s_{t-1}^1 + w_{12}s_{t-1}^2 + u_{11}x_t^1 + u_{12}x_t^2 + u_{13}x_t^3 \\ w_{21}s_{t-1}^1 + w_{22}s_{t-1}^2 + u_{21}x_t^1 + u_{22}x_t^2 + u_{23}x_t^3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}\right)$$

$$= \varphi\left(\begin{pmatrix} w_{11} & w_{12} & u_{11} & u_{12} & u_{13} \\ w_{21} & w_{22} & u_{21} & u_{22} & u_{23} \end{pmatrix}\begin{pmatrix} s_{t-1}^1 \\ s_{t-1}^2 \\ x_t^1 \\ x_t^2 \\ x_t^3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}\right)$$



Notation summary for vanilla RNN:

$$s_t = \varphi(W s_{t-1} + U x_t + b)$$

sometimes written as

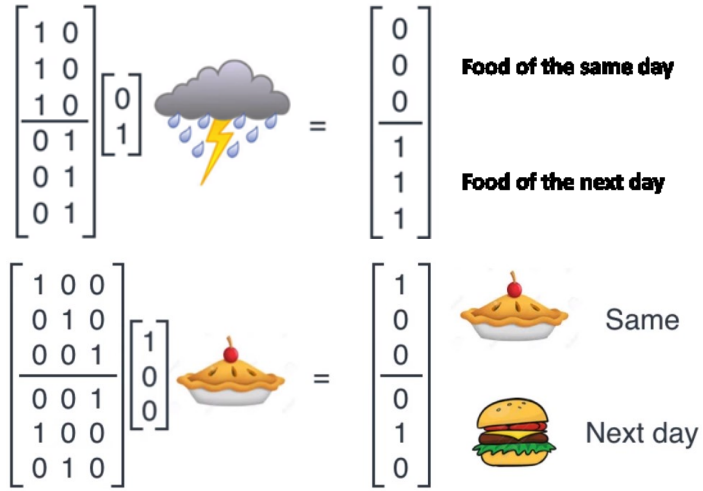
$$s_t = \varphi(W_c[s_{t-1}, x_t] + b)$$

sometimes as

$$s_t = \varphi(W_b[s_{t-1}, x_t])$$

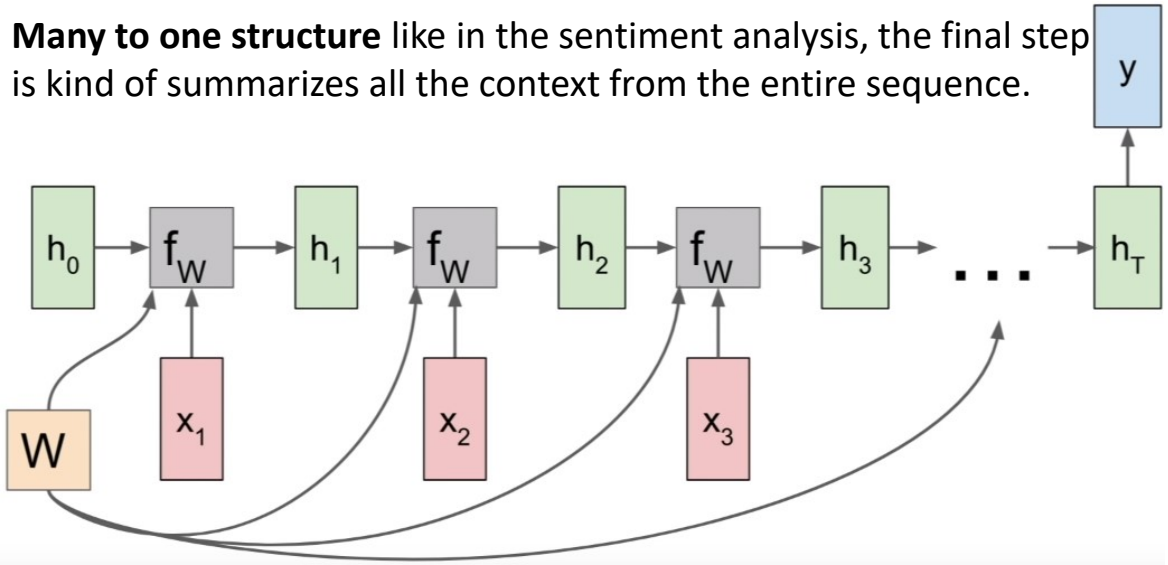
Remember our example

Concatenation of the state and inputs

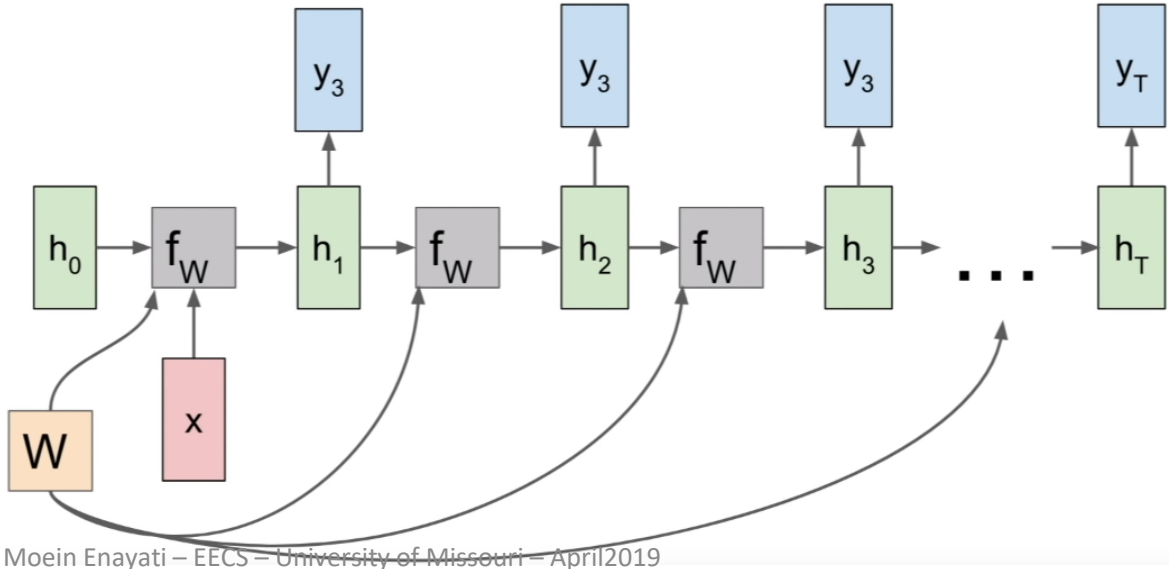


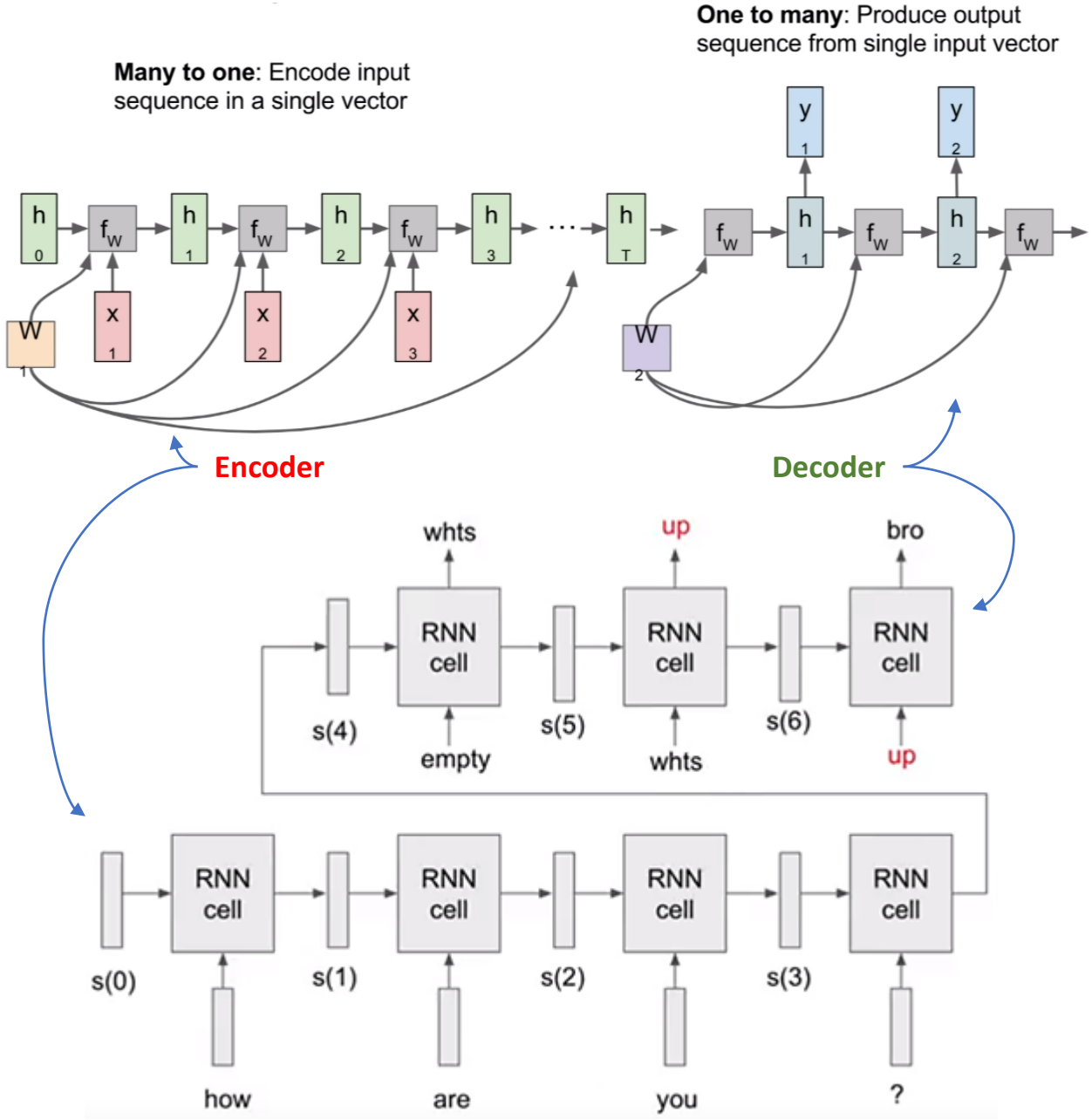
Types of RNNs

Many to one structure like in the sentiment analysis, the final step is kind of summarizes all the context from the entire sequence.



One to many structure where we have a single input to produce multiple outputs (image captioning) This could be something to ask students

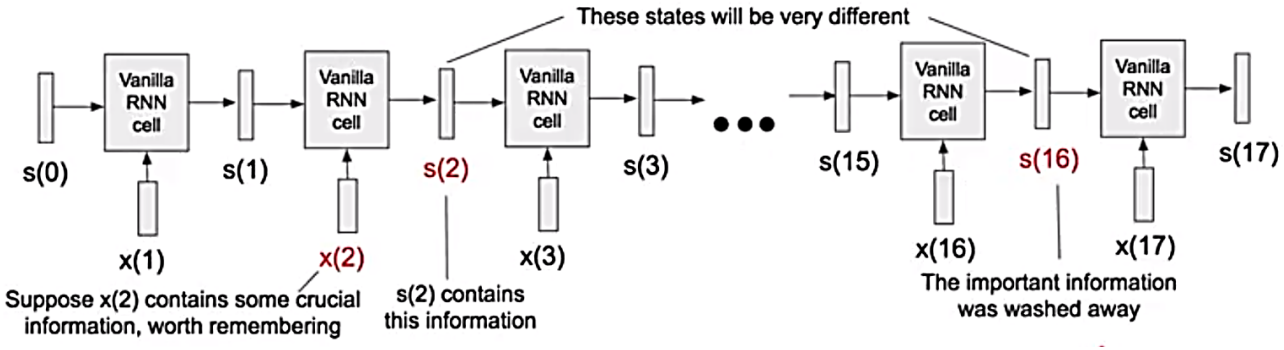




Vanilla RNN | problem

Information Morphing: Inability to keep important memory content for more than few time steps.

If we unroll the vanilla RNN, suppose X2 has some important information needed to make decision at X17. S2 encodes this information. In practice S2 and S16 will be quite different. So the information from S2 is not kept until S16.



Why this happens?

- **Nonlinearity** will collapse the memory.
- **No memory protection** against the nonlinearities being applied.

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

Solution is:

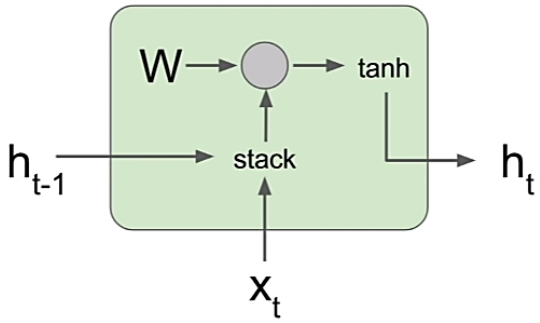
- **1st :** selectively update the memory
- **2nd :** use +/- for the update, instead of multiplication

$$\begin{bmatrix}\text{New state candidate}\end{bmatrix} = \text{FUNC}\left[\begin{bmatrix}\text{Previous state}\end{bmatrix}, \begin{bmatrix}\text{Current input}\end{bmatrix}\right]$$

Adding the new information, won't corrupt all previous memories.

New state candidate is a nonlinear function of previous state and the current input.

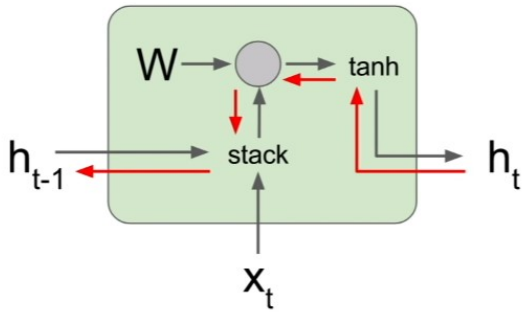
Vanilla RNN | diminishing gradient flow



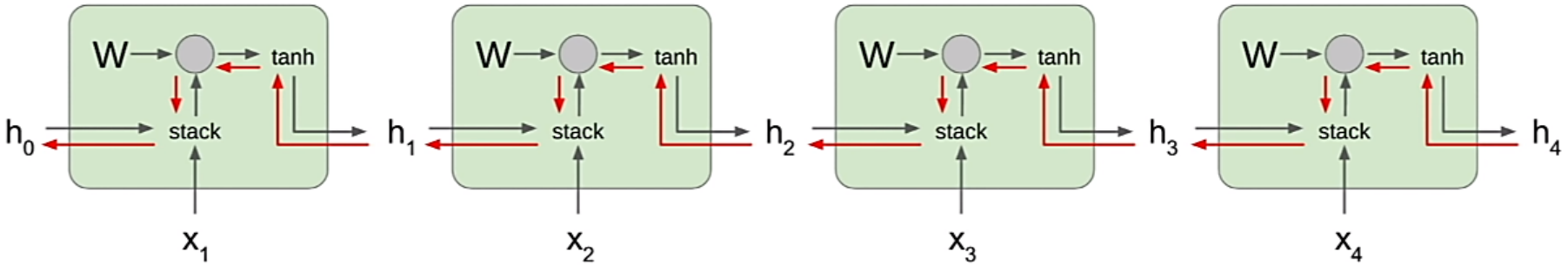
Vanilla RNN, forward pass

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$
$$= \tanh\left((W_{hh} \quad W_{hx})\begin{pmatrix}h_{t-1}\\x_t\end{pmatrix}\right)$$
$$= \tanh\left(W\begin{pmatrix}h_{t-1}\\x_t\end{pmatrix}\right)$$

compute the gradient of loss with respect to Ht-1



Vanilla RNN, backward pass



Every time we backpropagate, we **multiply** it by some part of the weight matrix. To compute the update of multiple cells, we will multiply many many times this W matrix. This self-multiplication either moves the values toward **zero (vanishing)** or toward **infinity** (**exploding**).

LSTM | solution for the gradient flow

Solutions for the exploding gradients?

- **Clipping** the gradient
- **Adaptive learning rate**(RMSprop)
- **Truncated** BP trough time (limited number of steps)

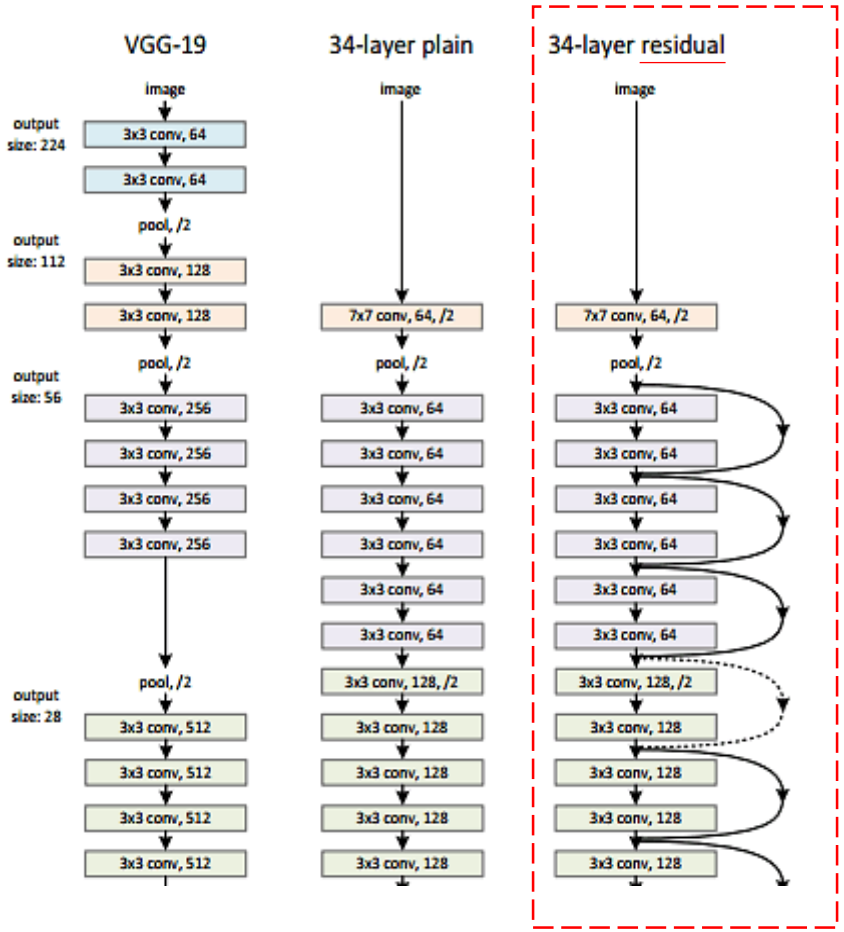
Solutions for the vanishing gradients?

- Smart weight initialization
- Activation functions such as ReLu
- Optimization algorithms such as RMSprop

* The most useful approach is **LSTM**, **GRU**, which are especially designed to handle the problem of vanishing gradient.

In **LSTM** we keep most of the hidden states' information and update only **part of it** using the addition. This additive interaction instead of a nonlinear interaction would allow us to do **an easier backpropagate**. This is very similar to the additive interactions in the **resnet**.

Same problem in deep CNNs



Solution:
Resnet with some shortcuts to pass the gradient. They make a kind of high way for gradients to flow backward through the entire network.

LSTM | selectivity through gates

We want to be **selective** about what to **forget** and what to **remember**. We also may not need the entire **memory** to create the new state candidate (read gate).

Example:

- To say a word here, I don't need all my childhood memory, (read gate).
- I also don't want that part of memory being changed as a result of our conversations here!(write gate, prevents unnecessary updates).
- Forgetting is required, as the memory capacity is limited.

Vanilla RNN

$$h_t = \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

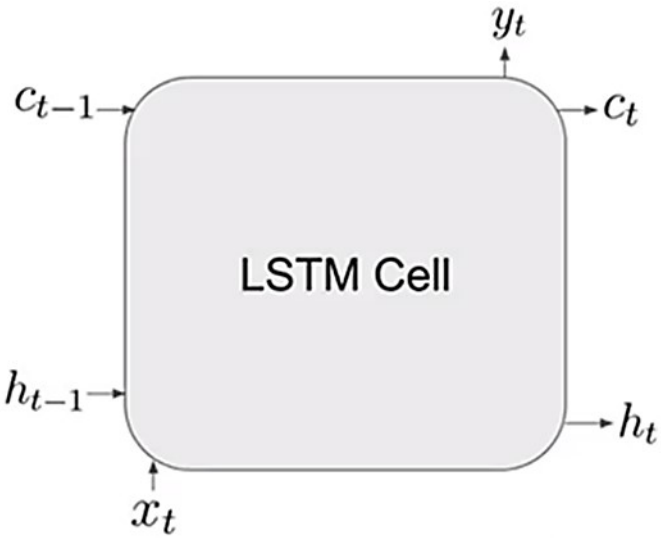
The state is a pair of vectors:

c_t — Memory Cell

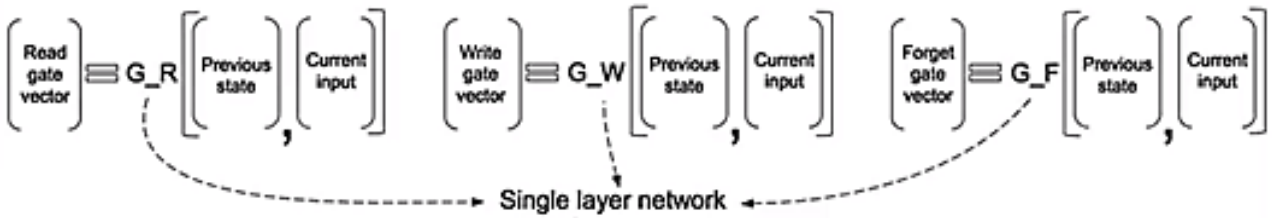
h_t — Shadow State (gated version of memory cell)

The output is a part of the state:

$y_t = h_t$ — Cell output



Each gate is a separate single layer network with 2 parameters:



The **memory protection** is all applied to the **Ct**.
Ht is the filtered/gated version of the memory.
Yt Output of the cell would be equal to ht.

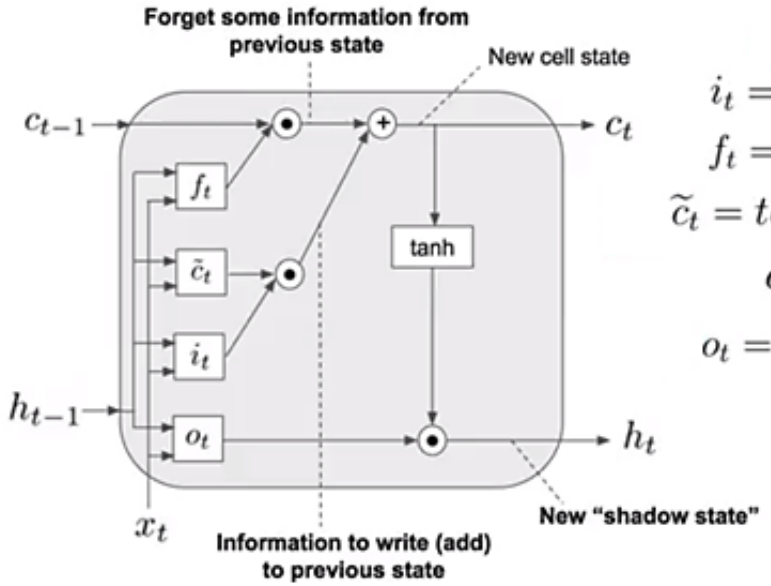
LSTM | how gates work

Gates will put weights on each element (of memory) and define what (or how much) to remember, or forget, or write. In other words, in generating a new state candidate we may not need all previous values:

0-1 gates: (Most) gates are being defined by sigmoid functions. For simplicity, imagine just the extreme values of 0 and 1 for each

- **Read gate,** prevents loading unnecessary old memories.
- **Write gate,** prevents updating some useful memories of the past.
- **Forget gate** is a way around the limitations on the memory capacity.

- **Gates** and **state candidate** are functions of X1, and Ht-1
- Next is to **filter the candidate** using the **write/input gate**, here in LSTM we call it input gate.
- Then **forget unnecessary** information from the previous memory.
- **Add these two** together to get a **new memory** content.
- Then apply **nonlinearity to the memory** (tanh) and gate it with an **read/output gate**.
- The result is Ht: the gated version of memory content.



LSTM equations

$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$$ Input gate

$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f)$$ Forget gate

$$\widetilde{c}_t = tanh(W h_{t-1} + U x_t + b)$$ Memory cell candidate

$$c_t = f_t \circ c_{t-1} + i_t \circ \widetilde{c}_t$$ Memory cell

$$o_t = \sigma(W_o h_{t-1} + U_o x_t + b_o)$$ Output gate

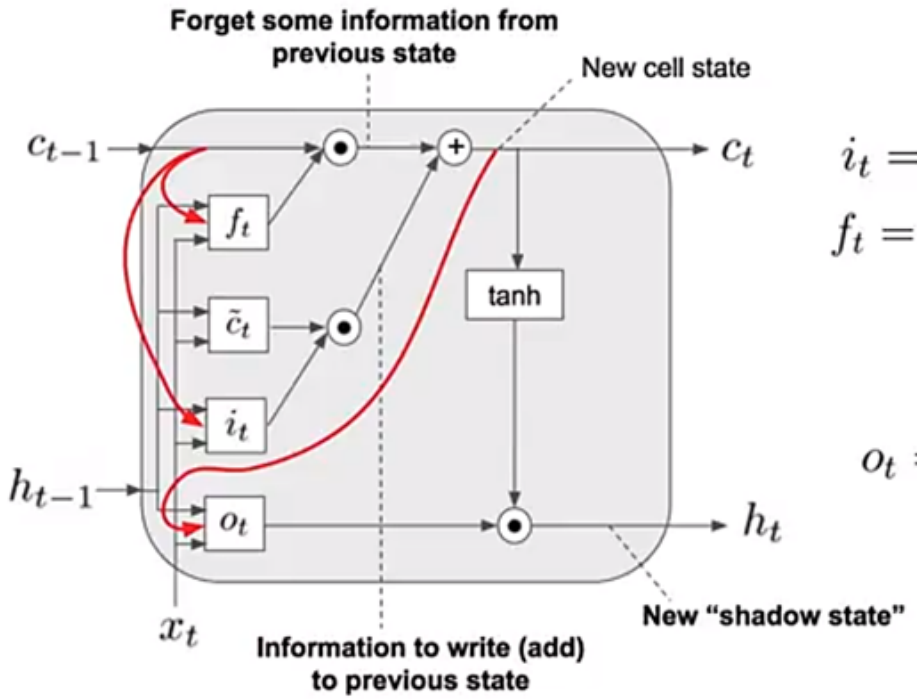
$$h_t = o_t \circ tanh(c_t)$$ Shadow state

$$y_t = h_t$$ Cell Output

LSTM | with Peephole connection

In LSTM as we use a gated version of the state/cell to compute the next candidate state, we potentially loose information. Ht-1 is not the entire information from the past.

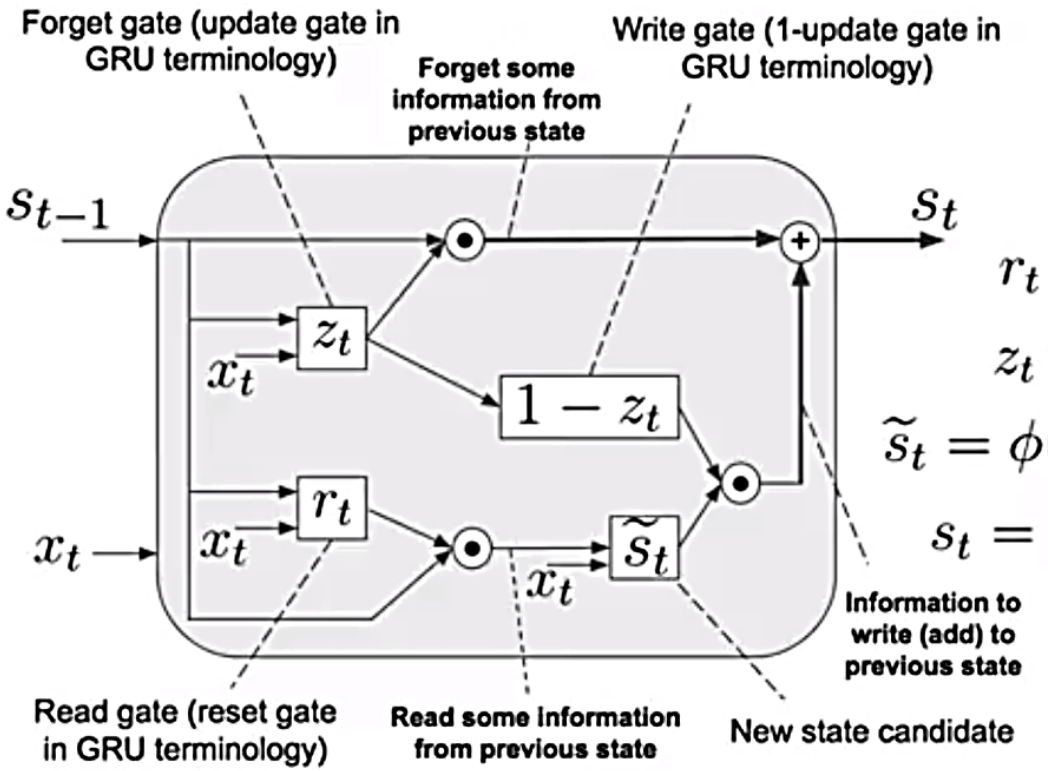
Here, additional terms appear in the computation of the gates, so that we make sure the entire state information is available for computing the gates, not only the filtered version Ht.



LSTM with Peephole connections

$$i_t = \sigma(W_i h_{t-1} + U_i x_t + P_i c_{t-1} + b_i)$$
$$f_t = \sigma(W_f h_{t-1} + U_f x_t + P_f c_{t-1} + b_f)$$
$$\tilde{c}_t = tanh(W h_{t-1} + U x_t + b)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$
$$o_t = \sigma(W_o h_{t-1} + U_o x_t + P_o c_t + b_o)$$
$$h_t = o_t \circ tanh(c_t)$$
$$y_t = h_t$$

GRU | Gated recurrent units



GRU equations

$$r_t = \sigma(W_r s_{t-1} + U_r x_t + b_r)$$ Read gate

$$z_t = \sigma(W_z s_{t-1} + U_z x_t + b_z)$$ Forget gate

$$\tilde{s}_t = \phi(W\ (r_t \circ s_{t-1}) + U x_t + b)$$ State candidate

$$s_t = z_t \circ s_{t-1} + \underbrace{(1 - z_t)}_{\text{Write gate}} \circ \tilde{s}_t$$ Current State

References:

- Late 1980s, **backpropagation through time** to train **vanilla RNN**.
- 1997, LSTM, S. Hochreiter, J. Schmidhuber
- 2000, LSTM with peepholes, F.A. Gers, J. Schmidhuber, Recurrent nets that time and count.
- 2014, GRU, Kyunghyun Cho, Yosua Bengio, Learning Phrase Representations using RNN Encoder-Decoder for statistical Machine Translation.