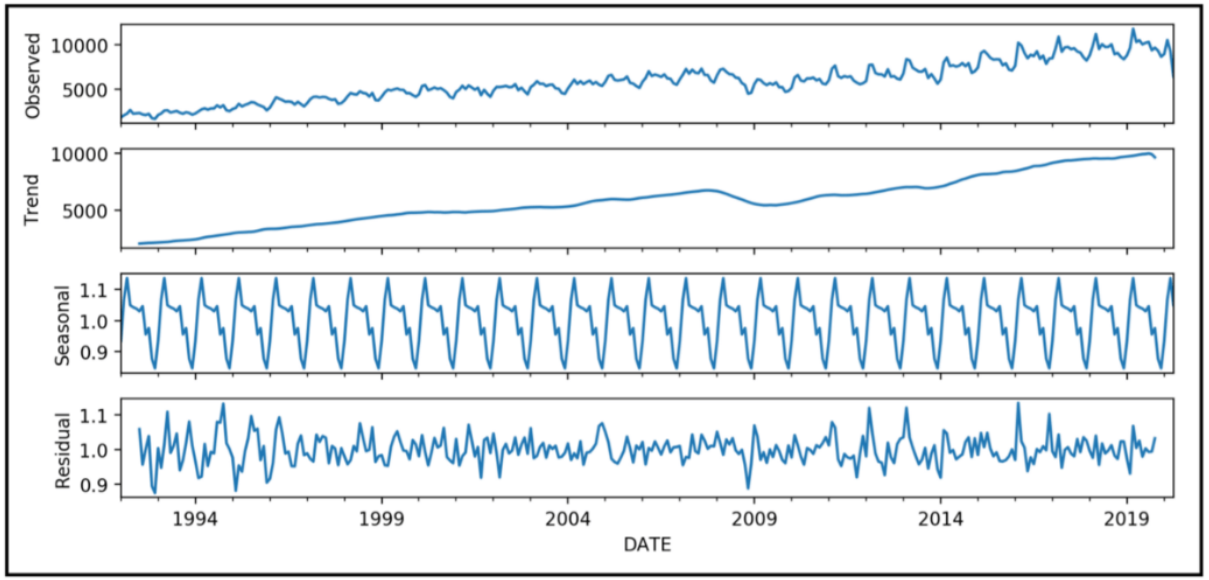
# Intro to Recurrent Neural Net (RNN) Long Short Term Memory (LSTM), Gated Recurrent Unit (GRU) for Time Series



**A Time Series** is a series of data points indexed in time order.

"***From Analyzing the Past To Predicting the Future***"

https://towardsdatascience.com/time-series-from-analyzing-the-past-to-predicting-the-future-249ab99ec52d

# Prolog

- We don't start our thinking from scratch every second.
- *As you read this sentence, you understand each word based on your understanding of previous words.*
- You don't throw everything away and start thinking from scratch again. Your thoughts have persistence.

Persistence of Memory, 1931 by Salvador **Dali**
The Museum of Modern Art (**MoMA**), New York City, NY

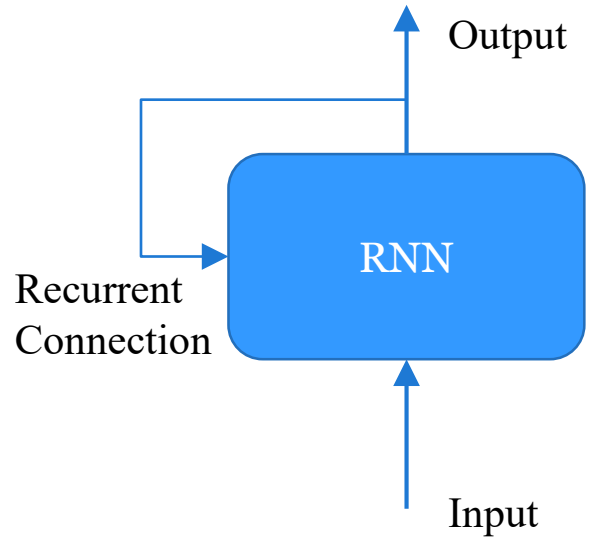
http://colah.github.io/posts/2015-08-Understanding-LSTMs

# Recurrent Neural Net (RNN)

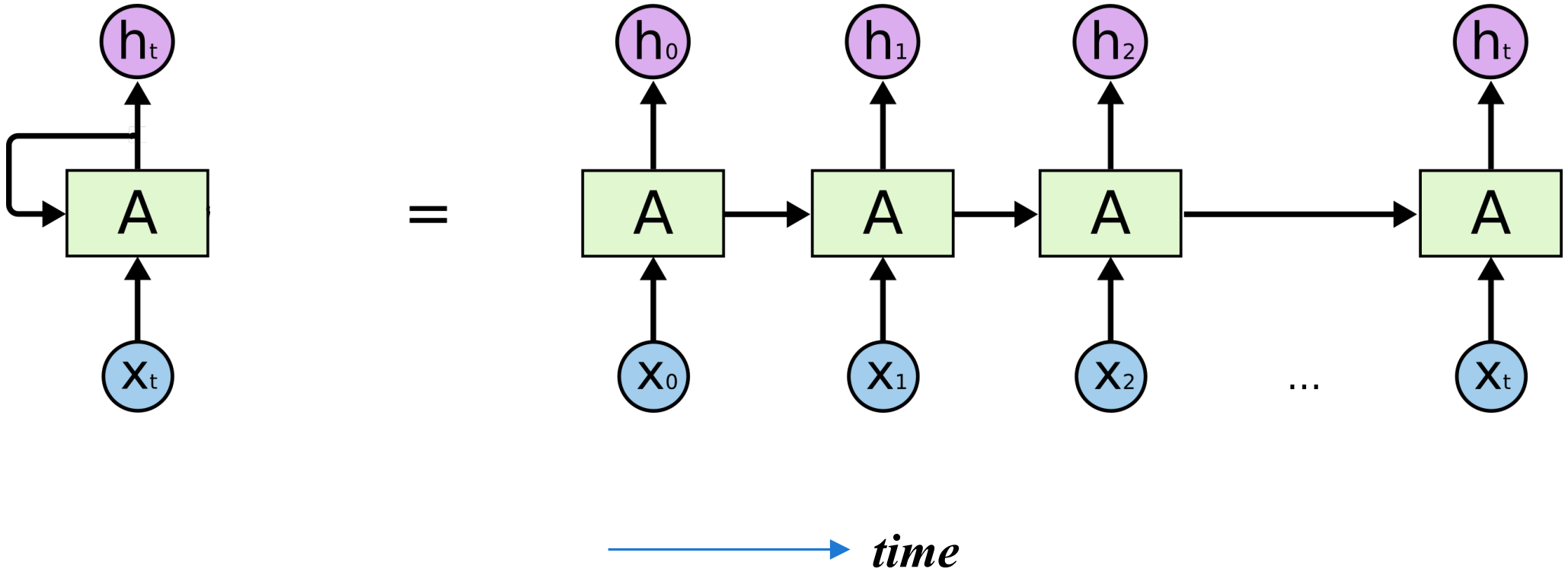
# Prolog

- Traditional feedforward neural networks can't do this, and it seems like a major shortcoming.
- For example, imagine you want to classify what kind of event is happening at every point in a movie. It's unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones.
- RNNs address this issue. They are networks with loops in them, allowing information to persist.

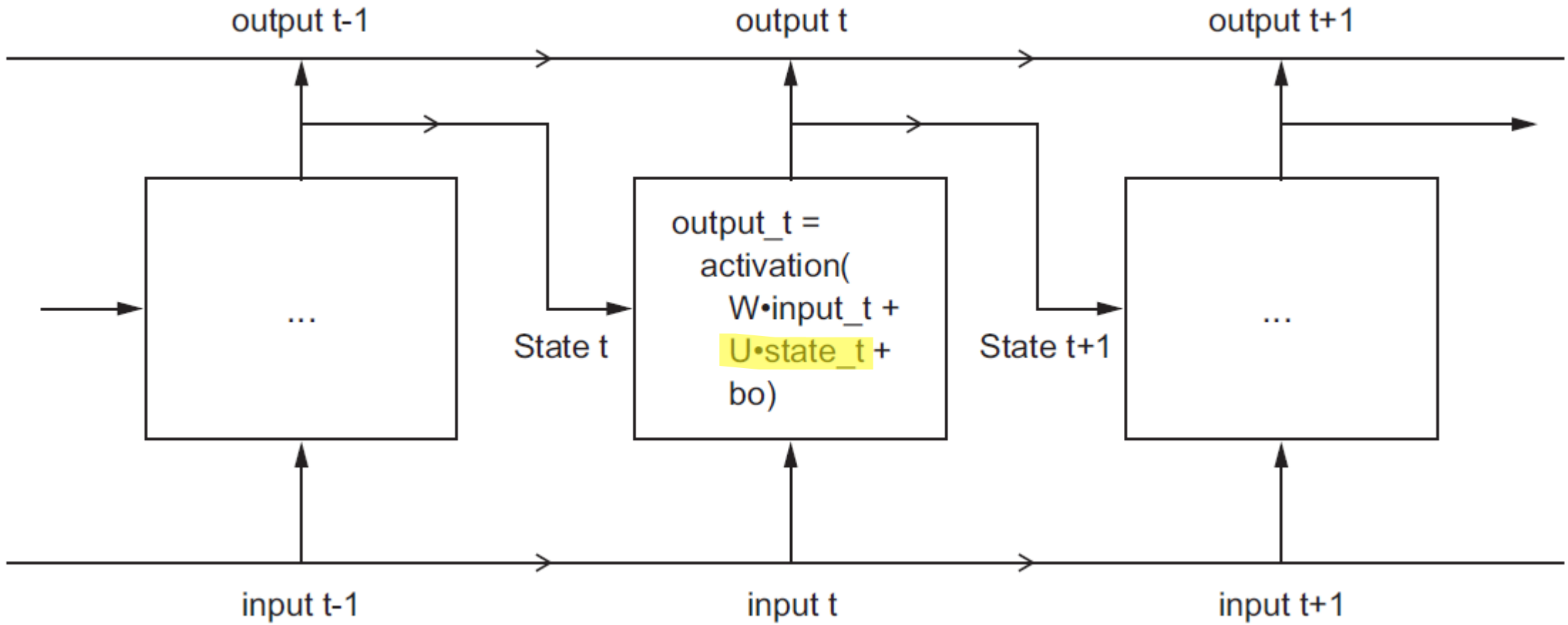
http://colah.github.io/posts/2015-08-Understanding-LSTMs



# An un-rolled recurrent neural network

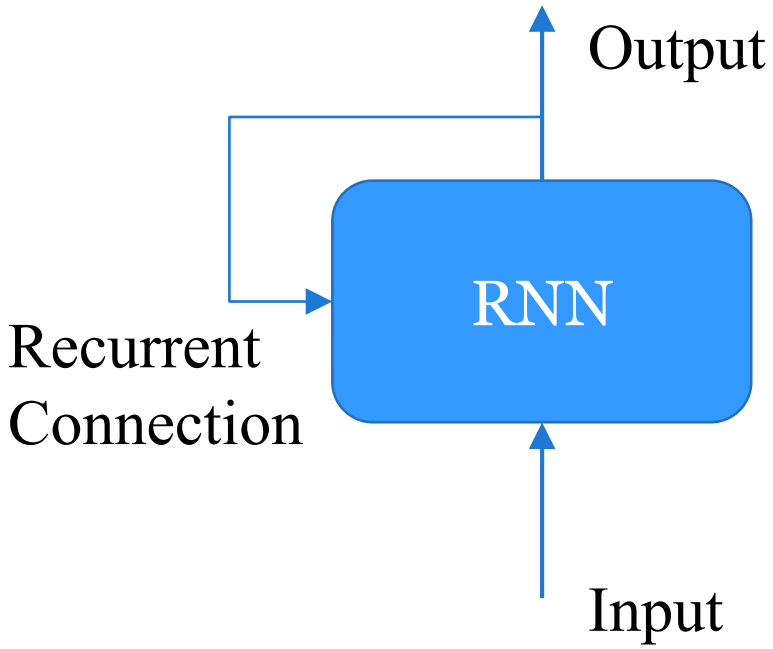


# Chollet's diagram on the textbook for RNN



# RNN Pseudo Code 0 (abstract concept)

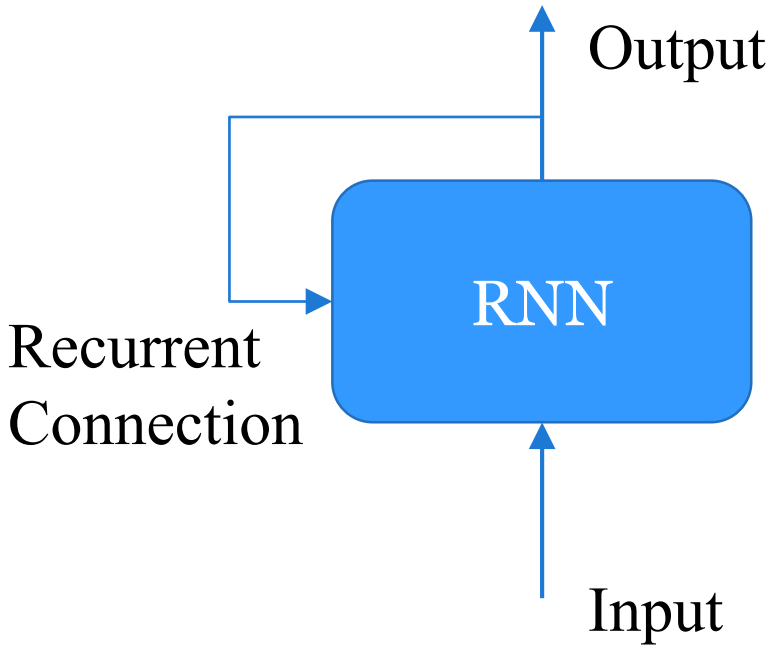
```
state_t = 0
for input_t in input_sequence:
        output_t = f (input_t, state_t)
        state_t = output_t     # previous output becomes the state for the next iteration
```



# RNN Pseudo Code 1

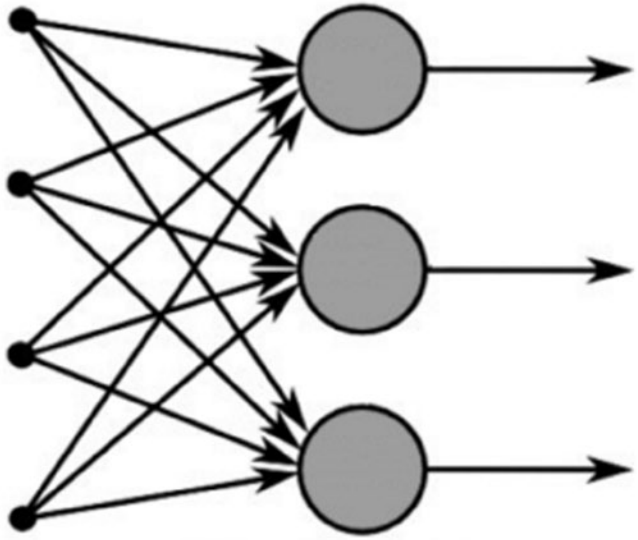
*tanh* is usually used in RNN

```
state_t = 0
for input_t in input_sequence:
        output_t = activation_func( dot(W, input_t) + dot(U, state_t) + b )
        state_t = output_t      # previous output becomes the state for the next iteration
```

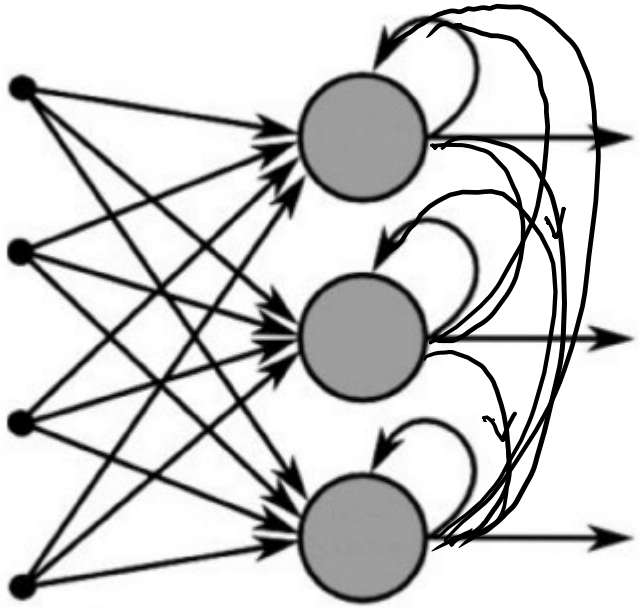


# numpy Implementation of a simple RNN

$W = 4 \times 3 = 12$



Feed-Forward Neural Network



Recurrent Neural Network

$U = 3^2 = 9$

How many parameters? 4x3+3x3+3= 24

# SimpleRNN_process.ipynb

```
import numpy as np
timesteps = 5
input_features = 4
output_features = 3
```

```
inputs = np.random.random((timesteps, input_features))
```

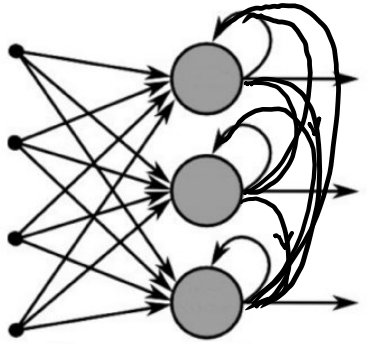
```
state_t = np.zeros((output_features,))
```

```
W = np.random.random((output_features, input_features)) # row, col
```

```
U = np.random.random((output_features, output_features))
```

```
b = np.random.random((output_features,))
```

```
successive_outputs = []
for input_t in inputs:
    output_t = np.tanh(np.dot(W, input_t)+np.dot(U, state_t)+b)
    print(output_t)
    successive_outputs.append(output_t)
    state_t = output_t
final_output_sequence = np.concatenate(successive_outputs, axis=0)
```



Recurrent Neural Network

A time series is a series of data points indexed in time order.

# TimeSeries_Simple.ipynb (Univariate Time Series, Horizon = 1 )

```
raw_seq = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
```

| | |
|---|---|
| [0.1 0.2 0.3] | 0.4 |
| [0.2 0.3 0.4] | 0.5 |
| [0.3 0.4 0.5] | 0.6 |
| [0.4 0.5 0.6] | 0.7 |
| [0.5 0.6 0.7] | 0.8 |
| [0.6 0.7 0.8] | 0.9 |
| **[0.7 0.8 0.9]** | ? |

Additional sample RNN program:
https://www.datatechnotes.com/2018/12/rnn-example-with-keras-simplernn-in.html

# Univariate vs. Multivariate Time Series

- Univariate time series - refers to a time series that consists of single (scalar) observations recorded sequentially over equal time increments
- Multivariate time series - refers to a dataset where multiple variables (features) are observed over time — not just one. In other words, it's a time series with more than one dependent variable, where each variable is recorded at regular time intervals, and the variables may influence or depend on each other.
  - Example:

| Time | Temperature (°C) | Humidity (%) | Power Usage (kW) |
|---|---|---|---|
| $t_1$ | 20.1 | 35 | 4.3 |
| $t_2$ | 20.4 | 36 | 4.5 |
| $t_3$ | 21.0 | 40 | 4.9 |

You can model this as a **multivariate time series** to capture their **interactions** (e.g., how humidity and temperature jointly affect power usage).

# Horizon: Can we predict multiple future values?

- When 2, 3, 4 are given, can the model output 5, 6 ? Yes

- In **time series prediction**, the term **horizon** refers to **how far into the future** you want to predict.
- Example: If your model input window is the last **10 time steps** ($t_{-9}$ … $t_o$), then:
  - **Horizon = 1:** predict $t_{+1}$
  - **Horizon = 3:** predict $t_{+3}$
  - **Horizon = [1, 2, 3]:** predict the next 3 time steps simultaneously (multi-horizon forecast)

# Keras/SimpleRNN (Old)

```
from keras.layers import SimpleRNN
model2 = Sequential()
model2.add(SimpleRNN(16, activation='tanh', input_shape=(n_steps, n_features)))
model2.add(Dense(8, activation="relu"))
model2.add(Dense(1))
model2.compile(optimizer='adam', loss='mse')
model2.fit(X, y, epochs=200, verbose=0)
```

# Keras/SimpleRNN

A sample

```
[[0.1]
 [0.2]
 [0.3]]
```

```
from keras.layers import SimpleRNN
inputs = keras.Input(shape=(n_steps, n_features))
x = SimpleRNN(64) (inputs)
output = layers.Dense(1) (x)
model21 = keras.Model(inputs = inputs, outputs=output)

model21.compile(optimizer='adam', loss='mse')
model21.fit(X, y, epochs=200, verbose=0)
```

X.shape = (13, 3, 1)  # 13 samples

# Keras/SimpleRNN – Stacking RNN layers

```
from keras.layers import SimpleRNN
inputs = keras.Input(shape=(n_steps, n_features))
# intermediate layer must return a full sequence of outputs
x = SimpleRNN(64, return_sequences=True) (inputs)
x = SimpleRNN(64, return_sequences=False) (x)
x = layers.Dense(8, activation='relu') (x)
output = layers.Dense(1) (x)
model2 = keras.Model(inputs = inputs, outputs=output)

model2.compile(optimizer='adam', loss='mse')
model2.fit(X, y, epochs=200, verbose=0) # fit model
```

# Self-Ex - Modify TimeSeries_Simple.ipynb

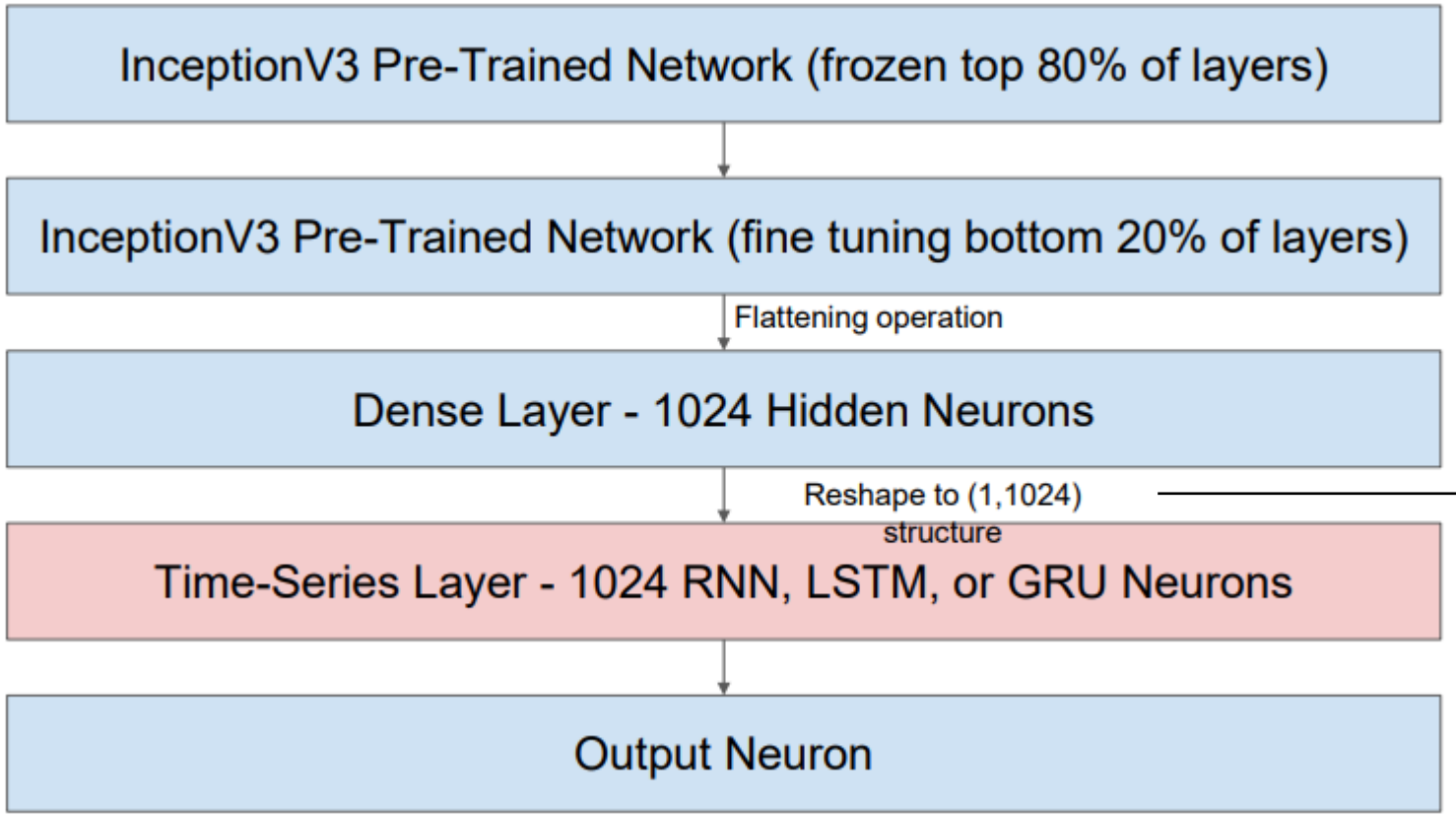
```
raw_seq = [0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55,
           0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95]
```

| | |
|---|---|
| [0. 0.05 0.1 0.15] | 0.2 |
| [0.05 0.1 0.15 0.2 ] | 0.25 |
| [0.1 0.15 0.2 0.25] | 0.3 |
| [0.15 0.2 0.25 0.3 ] | 0.35 |
| [0.2 0.25 0.3 0.35] | 0.4 |
| [0.25 0.3 0.35 0.4 ] | 0.45 |
| [0.3 0.35 0.4 0.45] | … |
| [0.35 0.4 0.45 0.5 ] | |
| [0.4 0.45 0.5 0.55] | |
| [0.45 0.5 0.55 0.6 ] | |
| [0.5 0.55 0.6 0.65] | |
| [0.55 0.6 0.65 0.7 ] | |
| [0.6 0.65 0.7 0.75] | |
| [0.65 0.7 0.75 0.8 ] | |
| [0.7 0.75 0.8 0.85] | |
| [0.75 0.8 0.85 0.9 ] | |
| **[0.8 0.85 0.9 0.95]** | **?** |

- Experiment with (1) dense net, (2) RNN (3) *another* RNN*, and (4) LSTM.
- Analyze & compare the results in a text cell at the bottom of ipynb file.

(*) change the location of RNN layer or add additional RNN layer

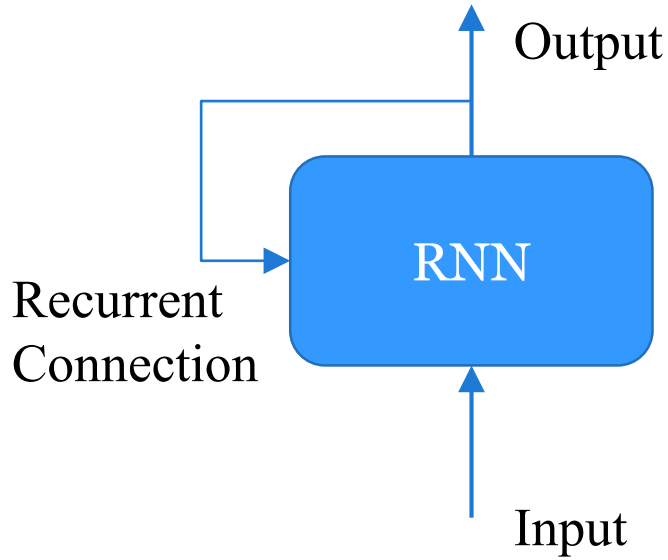
# Example Project, DeepSteer – An Application with RNN




https://stackoverflow.com/a/63168811

LTU research day Poster - http://qbx6.ltu.edu/chung/papers/43_Kocherovsky_DeepSteer.pdf

# Review Question




state_t = 0

***for*** input_t ***in*** input_sequence:

output_t = ***f*** (input_t, ? )

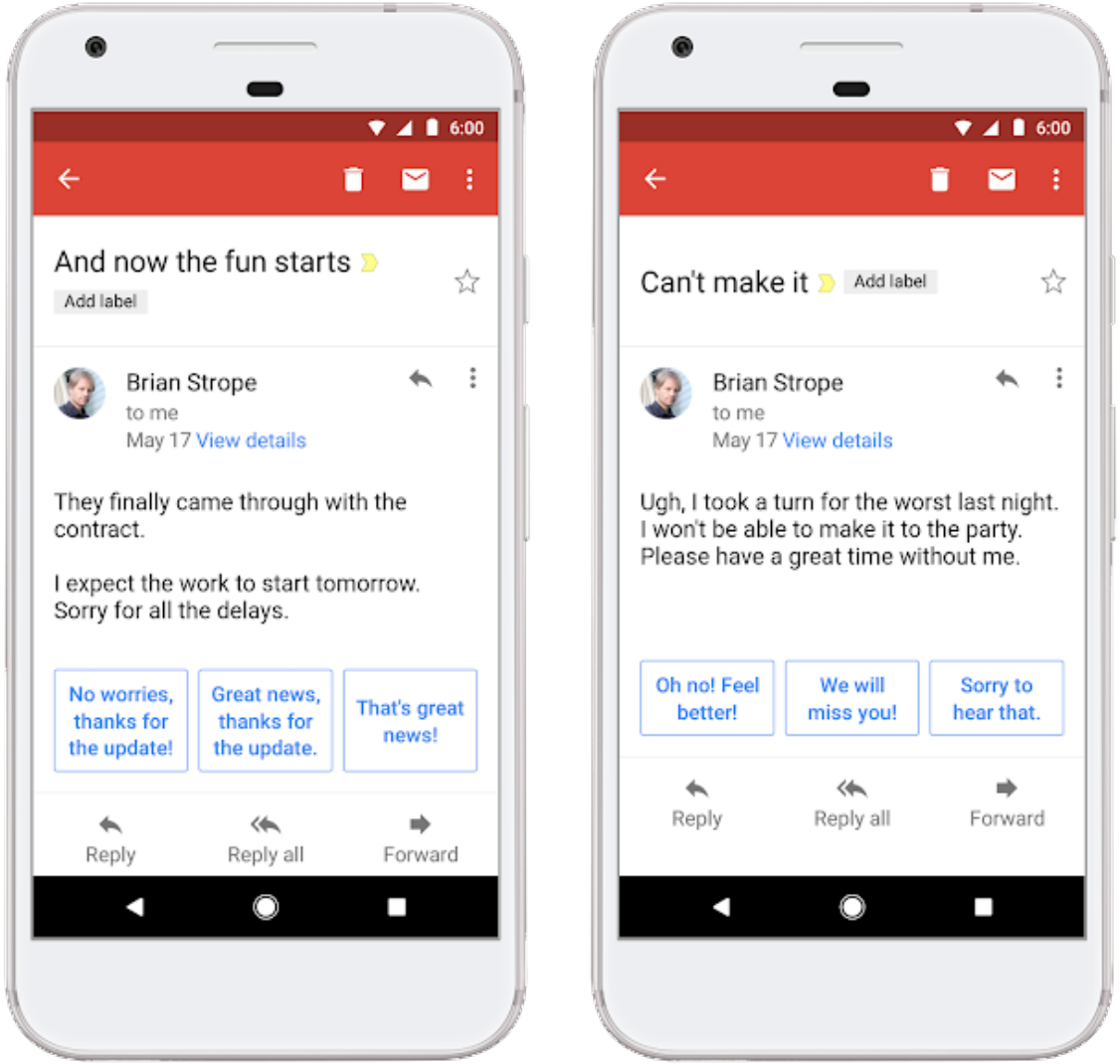
**state_t** = output_t # previous output becomes the state for the next iteration

# Long Short Term Memory (LSTM)

# You are using LSTM nearly everyday!

## Gmail's smart reply

https://ai.googleblog.com/2017/05/efficient-smart-reply-now-for-gmail.html



# Review: RNN

```
raw_seq = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
```

| | |
|---|---|
| [0.1 0.2 0.3] | 0.4 |
| [0.2 0.3 0.4] | 0.5 |
| [0.3 0.4 0.5] | 0.6 |
| [0.4 0.5 0.6] | 0.7 |
| [0.5 0.6 0.7] | 0.8 |
| [0.6 0.7 0.8] | 0.9 |

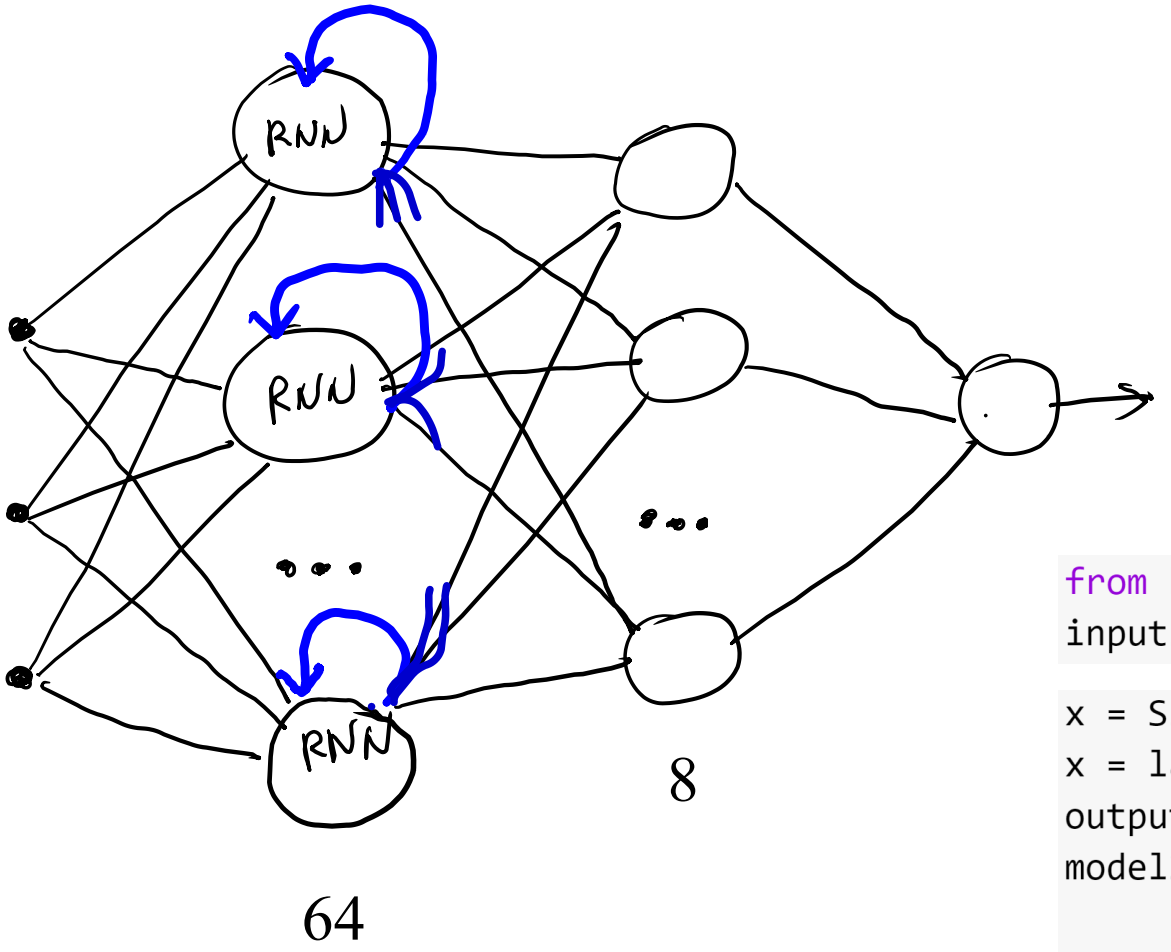
3 n-steps

1

```
from keras.layers import SimpleRNN
inputs = keras.Input(shape=(n_steps, n_features))
x = SimpleRNN(64) (inputs)
output = layers.Dense(1) (x)
model21 = keras.Model(inputs = inputs, outputs=output)

model21.compile(optimizer='adam', loss='mse')
model21.fit(X, y, epochs=200, verbose=0)
```

# Review: RNN



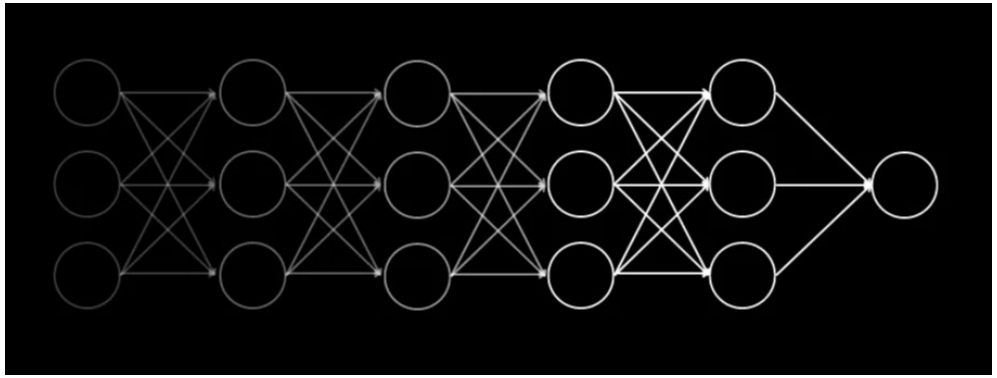

```
from keras.layers import SimpleRNN
inputs = keras.Input(shape=(n_steps, n_features))
```

```
x = SimpleRNN(64, return_sequences=False) (x)
x = layers.Dense(8, activation='relu') (x)
output = layers.Dense(1) (x)
model2 = keras.Model(inputs = inputs, outputs=output)

model2.compile(optimizer='adam', loss='mse')
model2.fit(X, y, epochs=200, verbose=0) # fit model
```

# Problems of RNN: *vanishing gradient problem*

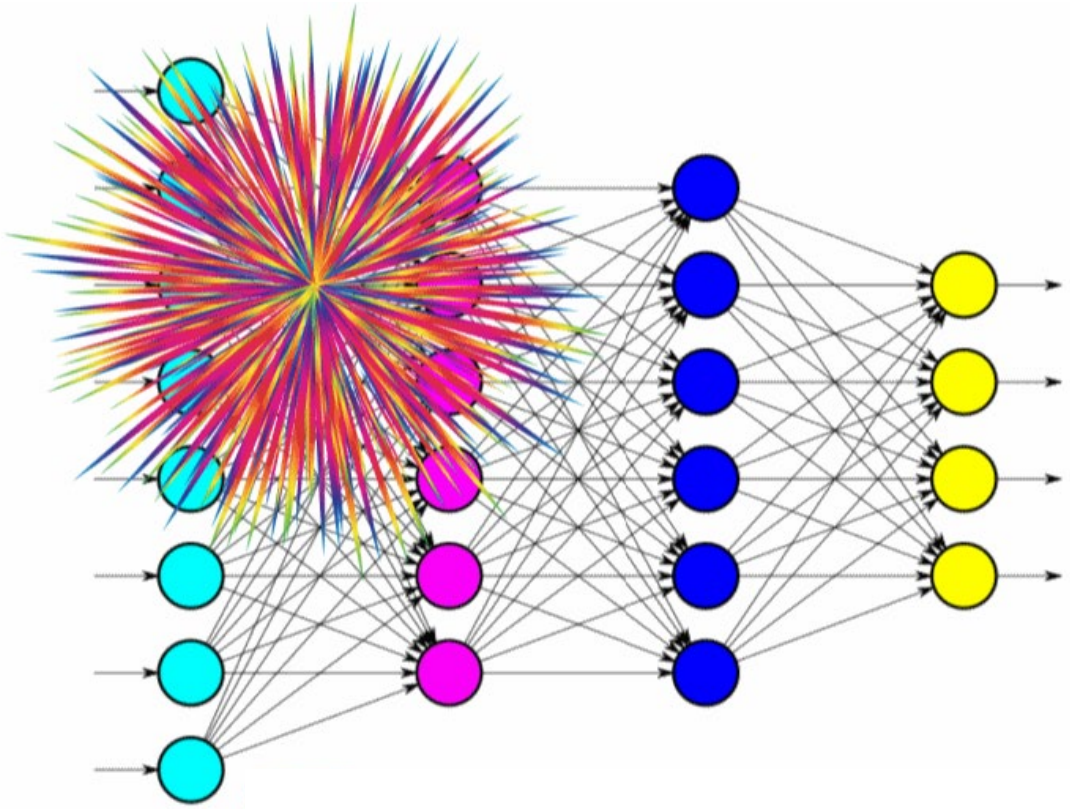
- SimpleRNN has a major issue: although it should theoretically be able to retain at time ***t*** information about inputs seen many timesteps before, in practice, such long-term dependencies are impossible to learn.
- This is due to the *vanishing gradient problem*, an effect that is similar to what is observed with non-recurrent networks (feedforward networks) that are many layers deep: as you keep adding layers to a network, the network eventually becomes untrainable.
- https://youtu.be/SKMpmAOUa2Q (The *vanishing gradient problem*)



https://levelup.gitconnected.com/rnns-are-coming-back-to-take-over-transformers-yes-for-real-51697943bc67

# Exploding gradient problem

- Things can go wrong the other way as well, with the gradients of the loss function becoming too large during backpropagation, leading to unstable training.

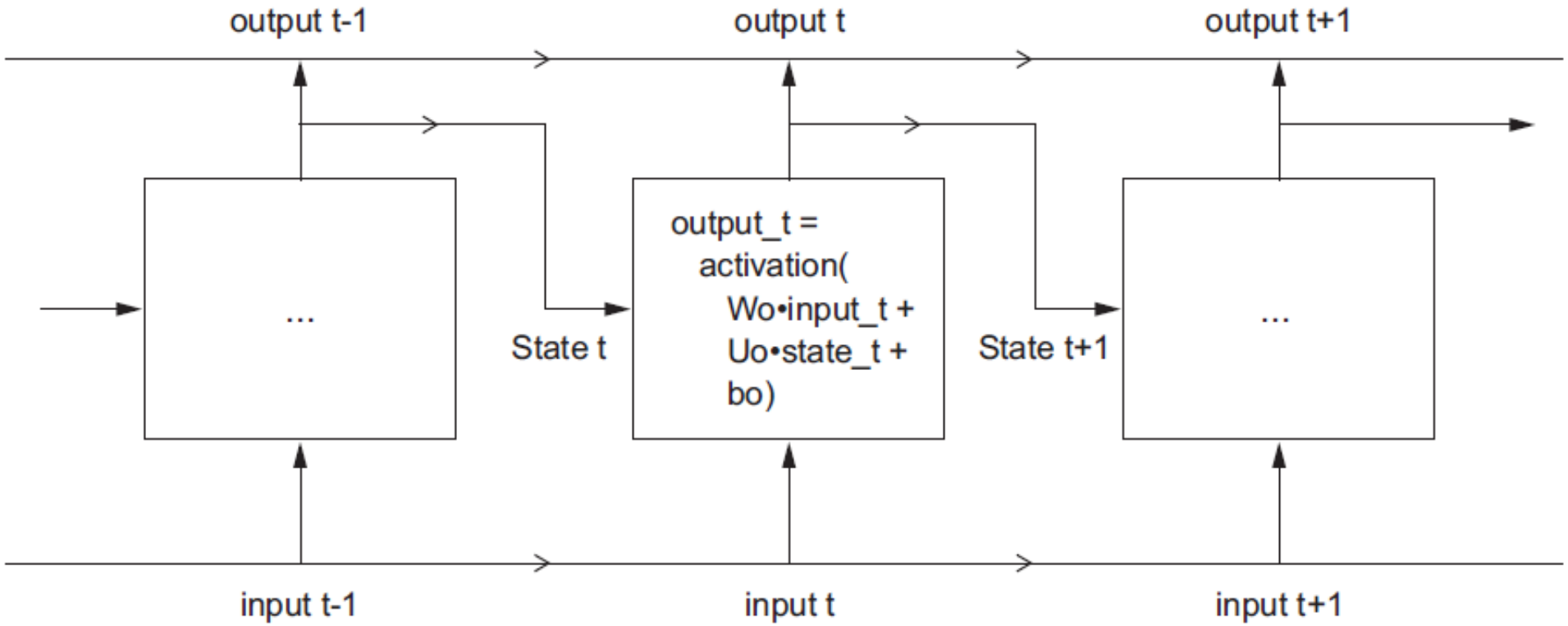


youtube.com/deeplizard

# LSTM (Long Short Term Memory) Networks

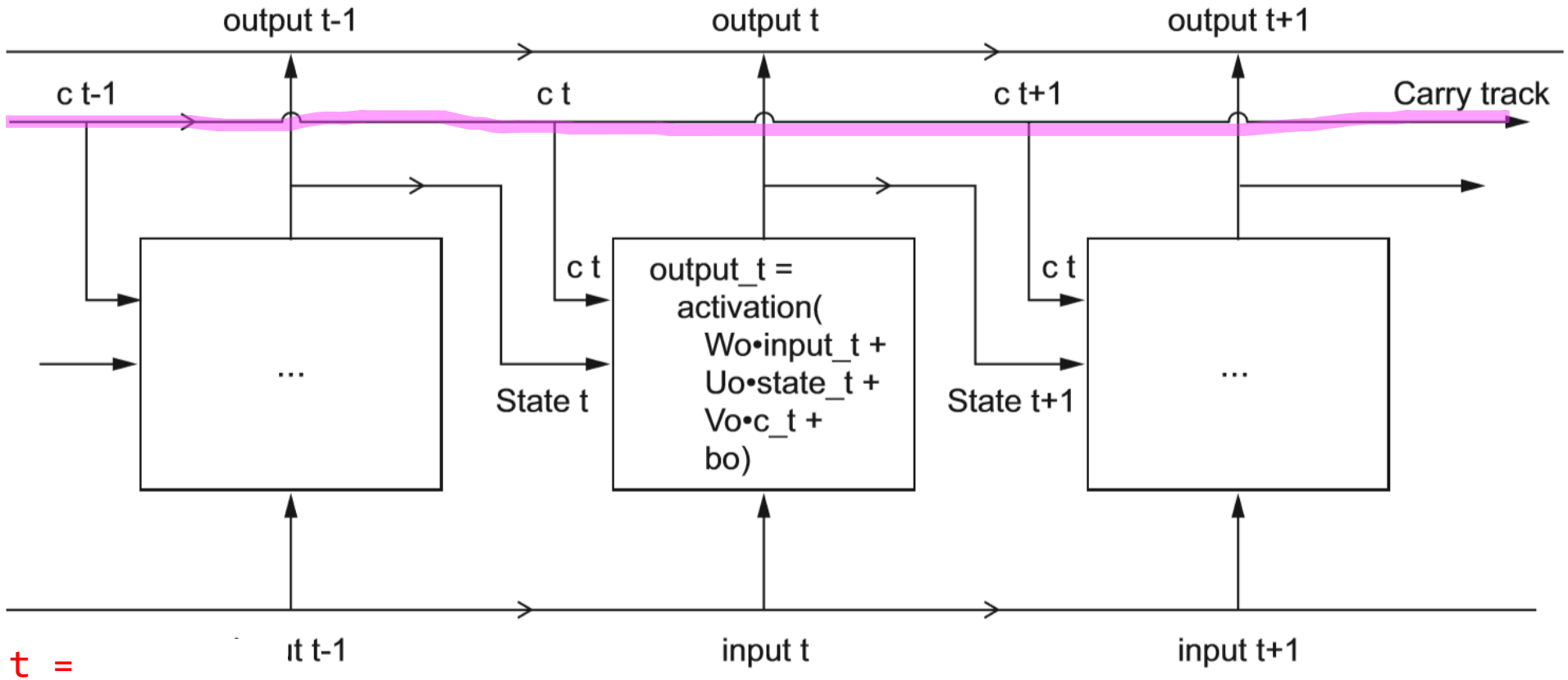
- A special kind of RNN, capable of learning long-term dependencies
- It adds a way to carry information across many timesteps
- Imagine a **conveyor belt** running parallel to the sequence you're processing.
  - Information from the sequence can jump onto the conveyor belt at any point, be transported to a later timestep, and jump off, intact, when you need it.
  - This is essentially what LSTM does: it saves information for later, thus preventing older signals from *gradually vanishing* during processing.

# *(Review)* The starting point of an **LSTM** layer: a **SimpleRNN**



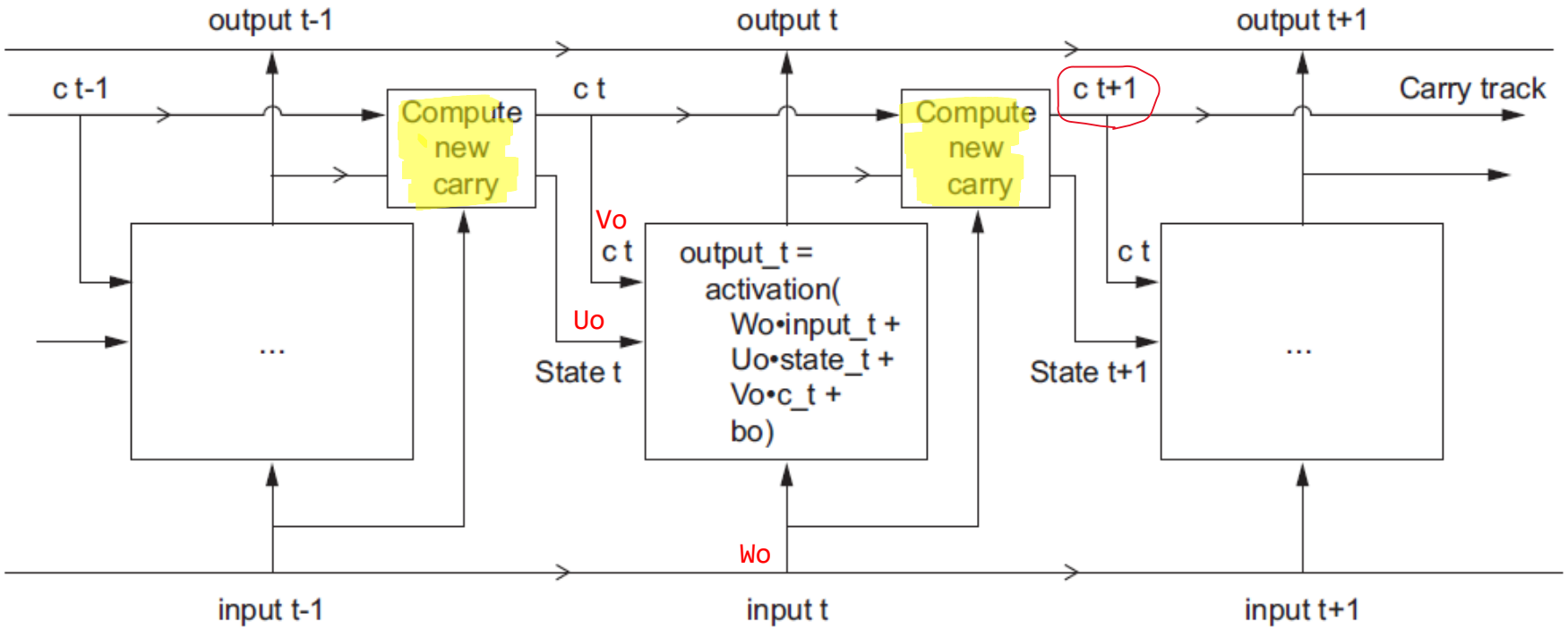

```
output_t = activation(W•input_t + U•state_t + b)
```

# Going from a SimpleRNN to LSTM: adding a **carry track** (`ct`)




$$output\_t = activation(W_o \bullet input\_t + U_o \bullet state\_t + V_o \bullet c\_t + b_o)$$

# Anatomy of an **LSTM** with **ct** (Carry Track)




$$output\_t = activation(W_o \bullet input\_t + U_o \bullet state\_t + V_o \bullet c\_t + b_o)$$

Next new carry:

```
c_t+1 = i_t*k_t + c_t*f_t
```

To present and update new info in the carry (to keep)

To deliberately **forget** irrelevant info in the carry track

```
i_t = activation( Ui•state_t + Wi•input_t + bi)
f_t = activation( Uf•state_t + Wf•input_t + bf) # forget
k_t = activation( Uk•state_t + Wk•input_t + bk) # keep
```

# A time series is a series of data points indexed in time order.

## Review: **TimeSeries_Simple.ipynb**

```
raw_seq = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
```

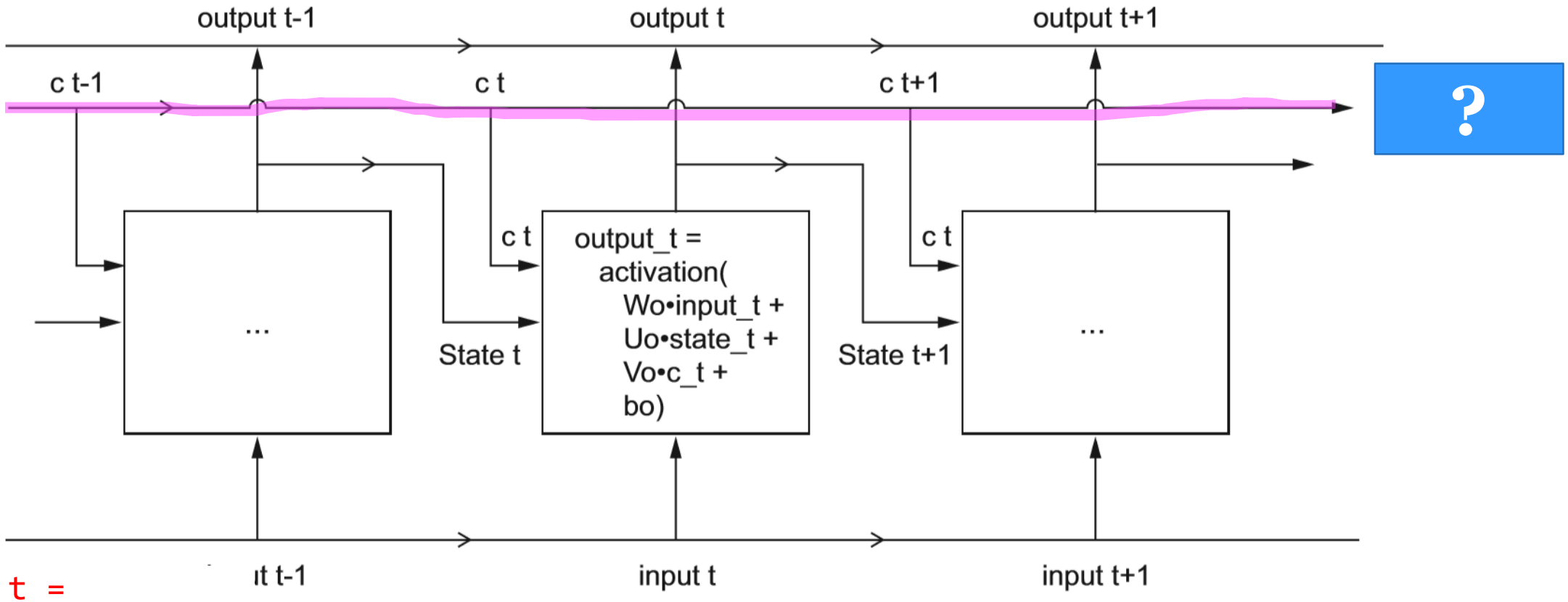
| | |
|---|---|
| [0.1 0.2 0.3] | 0.4 |
| [0.2 0.3 0.4] | 0.5 |
| [0.3 0.4 0.5] | 0.6 |
| [0.4 0.5 0.6] | 0.7 |
| [0.5 0.6 0.7] | 0.8 |
| [0.6 0.7 0.8] | 0.9 |
| [0.7 0.8 0.9] | ? |

```
inputs = keras.Input(shape=(n_steps, n_features))
x = LSTM(50) (inputs)
output = layers.Dense(1) (x)
model = keras.Model(inputs = inputs, outputs=output)

model.compile(optimizer='adam', loss='mse')
model.fit(X, y, epochs=200, verbose=0)
```

# Review: What does LSTM stand for?

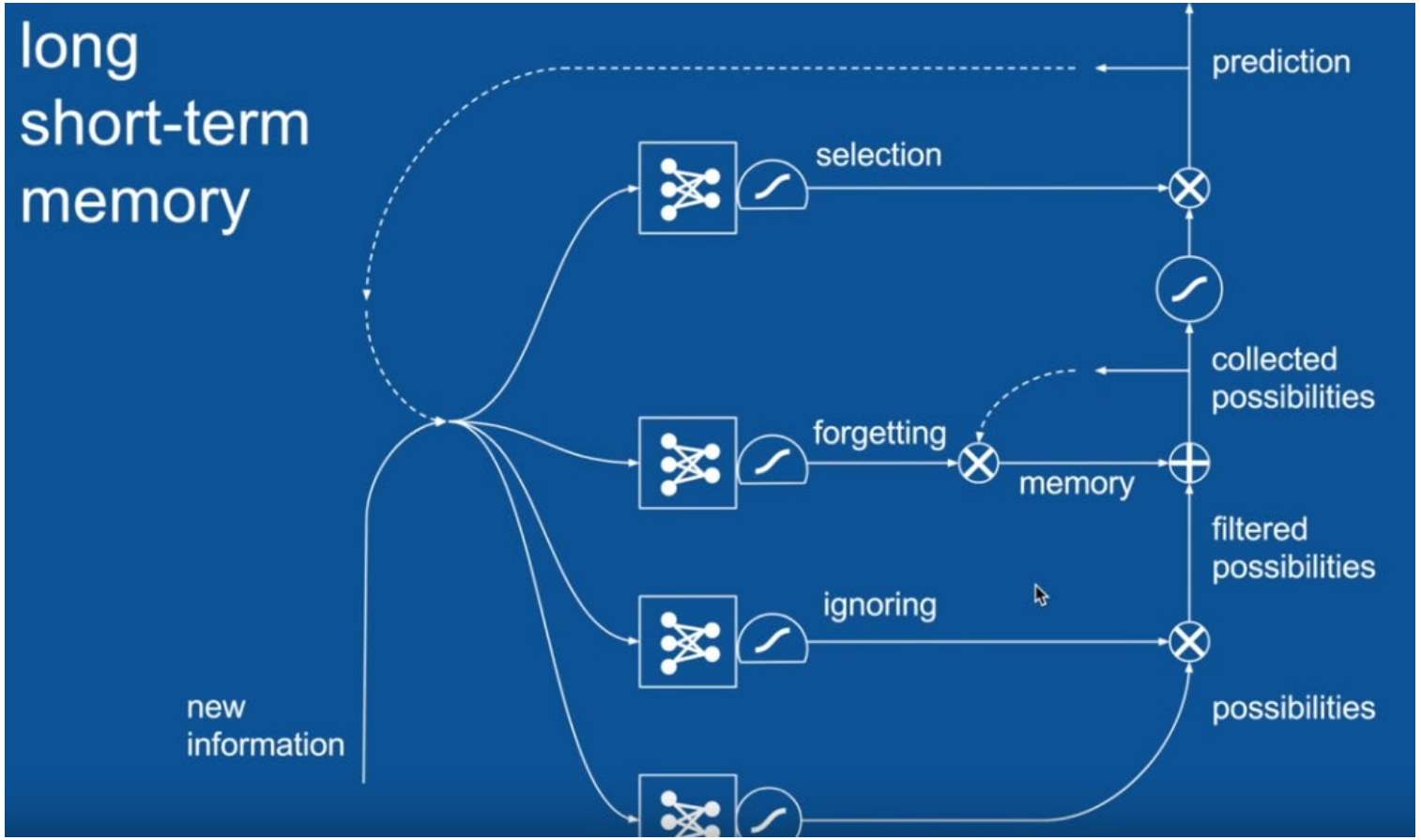
# Review: What is the name of the pink line?




$$output\_t = activation(W_o \bullet input\_t + U_o \bullet state\_t + V_o \bullet c\_t + b_o)$$

# Recurrent Neural Networks (RNN) and Long Short-Term Memory (LSTM) by Brandon Rohrer

- https://youtu.be/WCUNPb-5EYI

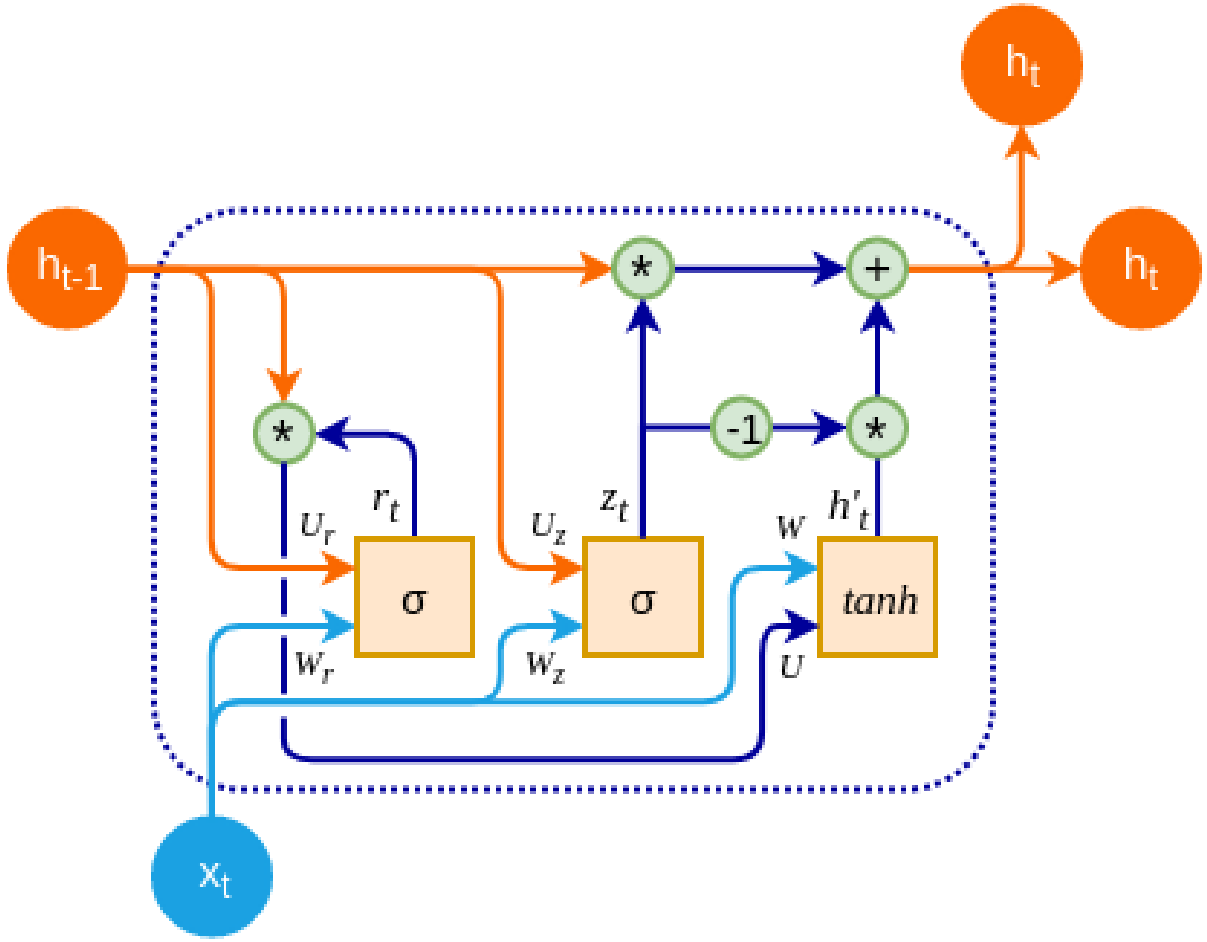



Slightly Different from Chollet, Keras

# Gated Recurrent Unit (GRU)

# GRU (Gated Recurrent Unit)

- A type of RNN architecture that is similar (simpler) to LSTM (Long Short-Term Memory).
- Like LSTM, GRU is designed to model sequential data by allowing information to be selectively remembered or forgotten over time
  - **Update gate ($z_t$):** combines input + forget gate functions.
  - **Reset gate ($r_t$):** controls how much past information to use when computing the new candidate hidden state.




https://medium.com/@anishnama20/understanding-gated-recurrent-unit-gru-in-deep-learning-2e54923f3e2

# GRU (Gated Recurrent Unit) with Keras

```
from keras.layers import GRU

inputs = keras.Input(shape=(n_steps, n_features))
x = GRU(50) (inputs)
output = layers.Dense(1) (x)
modelG = keras.Model(inputs = inputs, outputs=output)

modelG.compile(optimizer='adam', loss='mse')
modelG.fit(X, y, epochs=200, verbose=0)
```

**TimeSeries_Simple.ipynb**

# Other Info/issues/tasks related to RNN, LSTM, and/or GRU

# Comparisons

- RNN: simplest but forgets long-term info. Has gradient problems
- LSTM: powerful, remembers long-term info. Gradient problems mitigated.
- GRU: simpler and faster than LSTM, with similar performance. Gradient problems mitigated.

# Stock Market Prediction – Many are using LSTM or RNN

- https://www.datacamp.com/community/tutorials/lstm-python-stock-market
- https://towardsdatascience.com/neural-networks-to-predict-the-market-c4861b649371
- http://intelligentonlinetools.com/blog/2018/04/03/machine-learning-stock-market-prediction-lstm-keras/
- https://bcourses.berkeley.edu/files/70257274/download
- https://www.quantinsti.com/blog/artificial-neural-network-python-using-keras-predicting-stock-price-movement
- https://medium.com/@kingsubham27/stock-market-prediction-using-deep-learning-b71ae6fea740
- https://towardsdatascience.com/aifortrading-2edd6fac689d
- …

# Should we use GPU or CPU?

- For RNN: Multicore CPU would be OK
- For LSTM or GRU:
  - GPU seems better.
  - Small horizon size, CPU would be OK
  - https://pubs.aip.org/aip/acp/article-abstract/2510/1/030017/2915915/Benchmarking-CPU-vs-GPU-performance-in-building

# Self-Ex: Modify TimeSeries_Simple.ipynb *(Review)*

```
raw_seq = [0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55,
           0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95]
```

```
[0.    0.05 0.1  0.15]
[0.05 0.1  0.15 0.2 ]
[0.1  0.15 0.2  0.25]
[0.15 0.2  0.25 0.3 ]
[0.2  0.25 0.3  0.35]
[0.25 0.3  0.35 0.4 ]
[0.3  0.35 0.4  0.45]
[0.35 0.4  0.45 0.5 ]
[0.4  0.45 0.5  0.55]
[0.45 0.5  0.55 0.6 ]
[0.5  0.55 0.6  0.65]
[0.55 0.6  0.65 0.7 ]
[0.6  0.65 0.7  0.75]
[0.65 0.7  0.75 0.8 ]
[0.7  0.75 0.8  0.85]
[0.75 0.8  0.85 0.9 ]
```

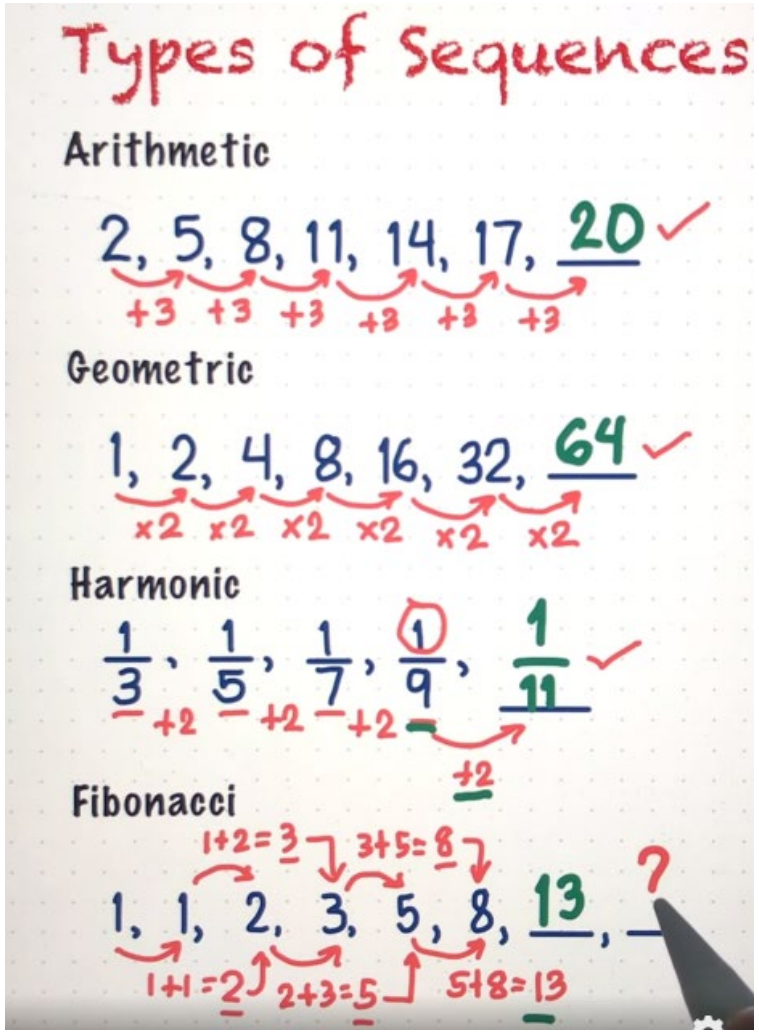
0.2
0.25
0.3
0.35
0.4
0.45
…

**[0.8, 0.85, 0.9, 0.95]** **?**

- Experiment with (1) DNN, (2) RNN (3) *another* RNN*, (4) LSTM, and (5) GRU
- Analyze & compare the results in a text cell at the bottom of ipynb file

(*) change the location of RNN layer or add additional RNN layer

# Self-Ex 2: Use LSTM and GRU



# Review Exercise

3 5 7 9 11 13 15 17 19 21 ?

Create training dataset for this timeseries:

- Num of features is 1
- Timestep (window size) is 4
- Horizon (forecast ahead) is 1

| Sample | X (Input sequence of 4 timesteps) | y (Next value) |
|---|---|---|
| 1 | [3, 5, 7, 9] | 11 |
| 2 | [5, 7, 9, 11] | 13 |
| 3 | [7, 9, 11, 13] | 15 |
| 4 | [9, 11, 13, 15] | 17 |
| 5 | [11, 13, 15, 17] | 19 |
| 6 | [13, 15, 17, 19] | 21 |
| 7 | [15, 17, 19, 21] | 23 |

# Review Exercise

3 5 7 9 11 13 15 17 19 21 ?

Create training dataset for this timeseries:

- Num of features is 1
- Timestep (window size) is 4
- Horizon (forecast ahead) is **2**

| Sample | X (4 timesteps) | y (next 2 values) |
|---|---|---|
| 1 | [3, 5, 7, 9] | [11, 13] |
| 2 | [5, 7, 9, 11] | [13, 15] |
| 3 | [7, 9, 11, 13] | [15, 17] |
| 4 | [9, 11, 13, 15] | [17, 19] |
| 5 | [11, 13, 15, 17] | [19, 21] |
| 6 | [13, 15, 17, 19] | [21, 23] |