



Time Series Management

Michele Linardi Ph.D.

michele.linardi@orange.fr



Syllabus

- Recap: Time Series Forecasting and Deep Learning Fundamentals
- Introduction to convolutional neural networks (CNN)
- **Introduction to recurrent neural networks (RNN)**
- **Encoder-decoder RNN model**
- **Introduction to Time Series Transformer Architecture**
- Implementation of the models with the Keras Library

Time series data... quick recap

- A **univariate time series** is a sequence of measurements of the same variable collected over time. Most often, the measurements are made at regular time intervals.



<https://www.kaggle.com/code/anushkaml/walmart-time-series-sales-forecasting>

Time Series Forecasting

- Predict Future (Values) based on the past (Historical Data)

	Date	Observation
History	2018-06-04	60
	2018-06-05	64
	2018-06-06	66
	2018-06-07	65
	2018-06-08	67
	2018-06-09	68
	2018-06-10	70
	2018-06-11	69
	2018-06-12	72
	2018-06-13	?
Future	2018-06-14	?
	2018-06-15	?

Question to ask prior to forecast

- Can we forecast?
 - How well we understand the factor influencing the future?
 - How much data we have?*
- How far in the future (horizon) we want to forecast?
- What technique, model should we use ?

Why Deep Learning models? (1/2)

Deep learning model perform well and better than other methods in many scenarios.

Model	Features	RMSE	MAPE (%)	Run time (s)
FeedForward	Date features + Covariants	10.73	4.20	53.35
AutoARIMA	Baseline	11.00	3.52	20658.95
DeepAR	Date features	11.96	5.99	156.18
FeedForward	Baseline	12.01	3.88	78.47
SeasonalNaive	Baseline	12.47	3.86	1.20
FeedForward	Date features	13.93	3.95	9.37
DeepAR	Baseline	14.01	6.12	856.51
DeepAR	Date features + Covariants	17.79	6.04	781.09
TrivialIdentity	Baseline	18.20	5.09	1.30
NPTS	Baseline	82.45	19.75	25.20

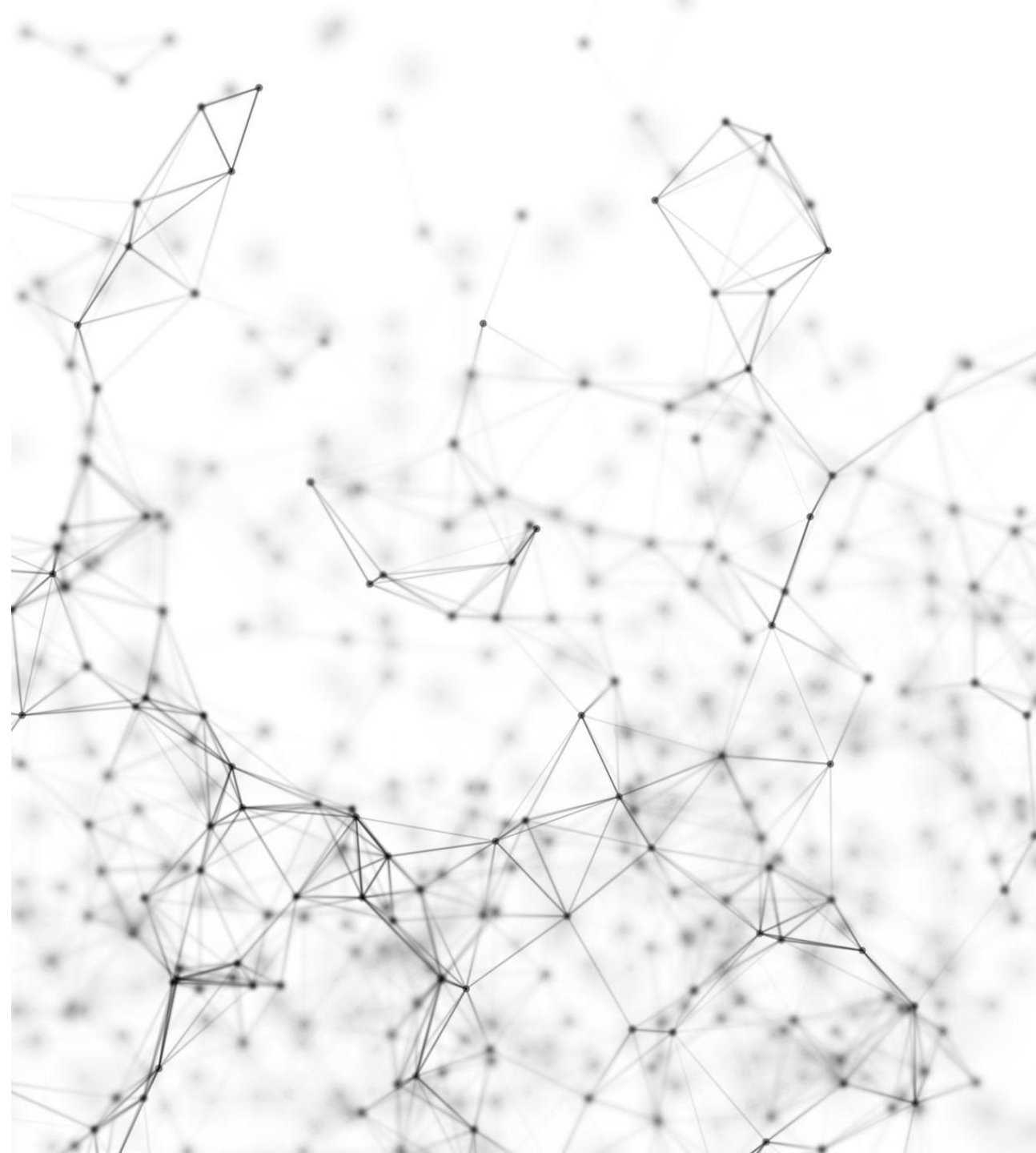
<https://blog.dataiku.com/deep-learning-time-series-forecasting>

Z. Tang, P.A. Fishwik, [Feedforward Neural Nets as Models for Time Series Forecasting](#), November 1993

Why Deep Learning models? (2/2)

- Non-parametric
- Flexible and expressive
- Easily inject exogenous features into the model
- Learn from large time series datasets

Recurrent Neural Network (RNN) for Time series Forecasting



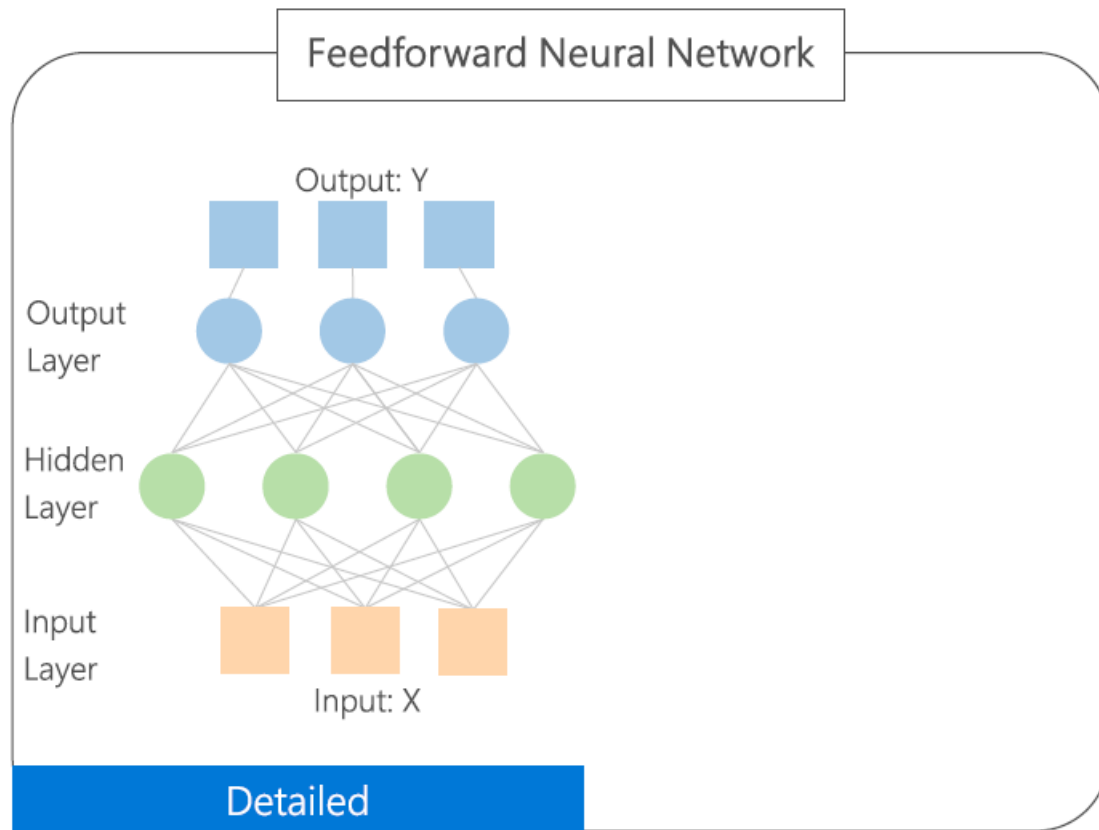
Roadmap

- What are RNNs?
- How RNNs are trained: Backpropagation through time (BPTT).
- Vanilla RNN and its gradient problems.
- Other RNN units.

RNN architectures:

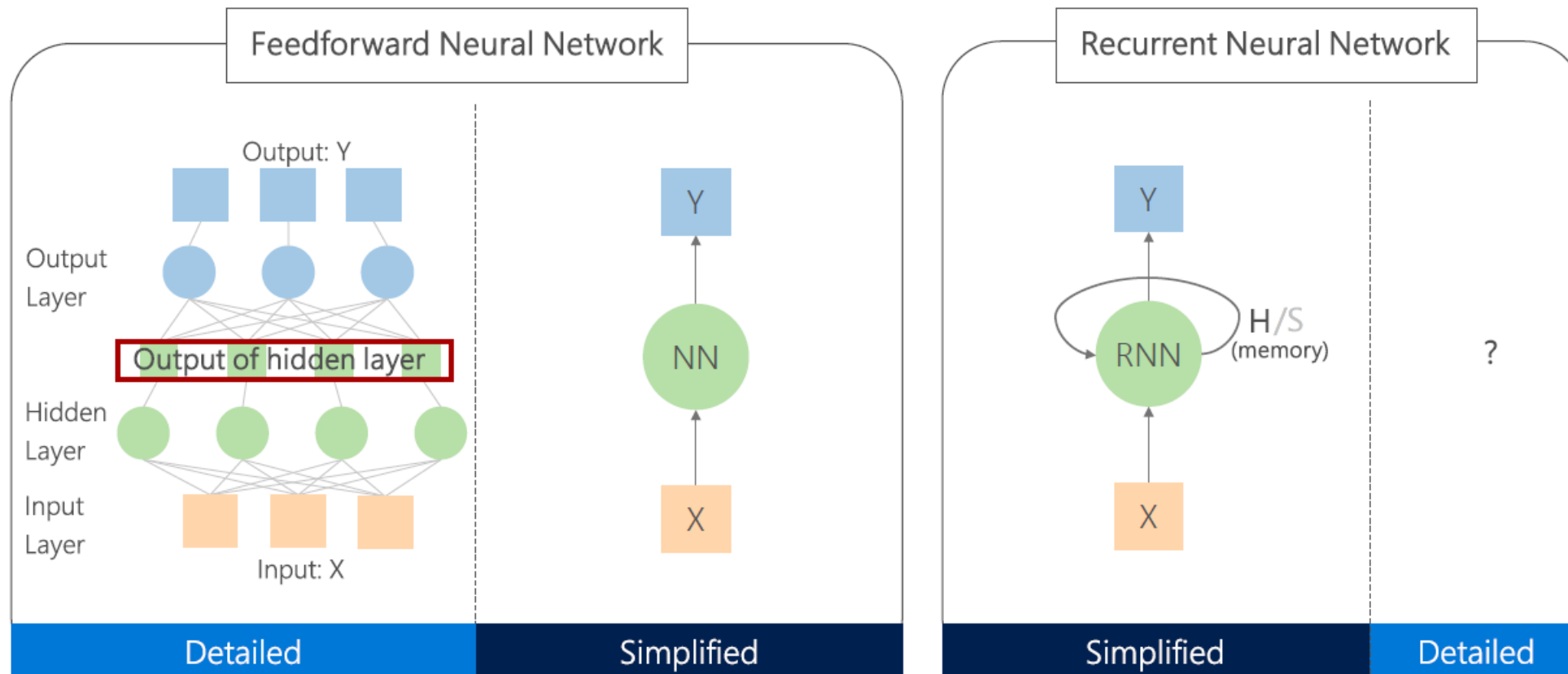
- GRU
- LSTM
- RNN stacking
- RNN for one step time series forecasting
- Encode-decoder RNN for multi-step time series forecasting

Recurrent Neural Networks (RNN)



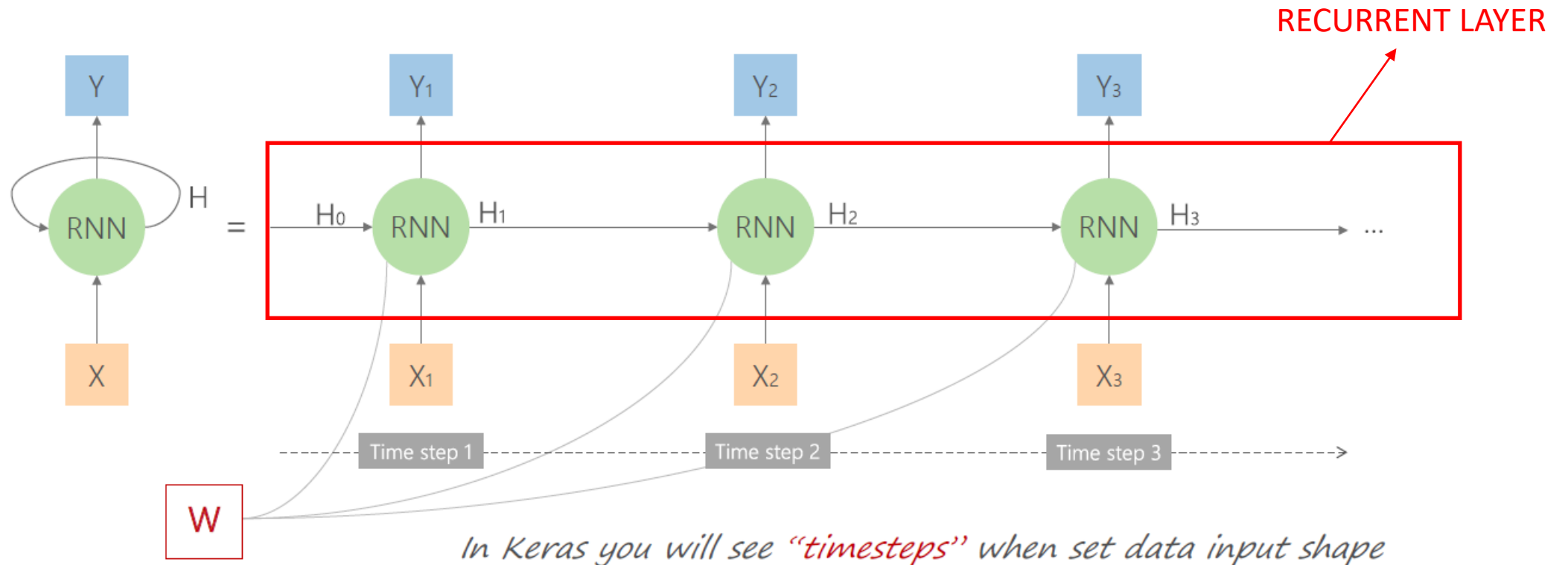
Recurrent Neural Networks (RNN)

RNN has an internal state (H), which can be fed back to the network



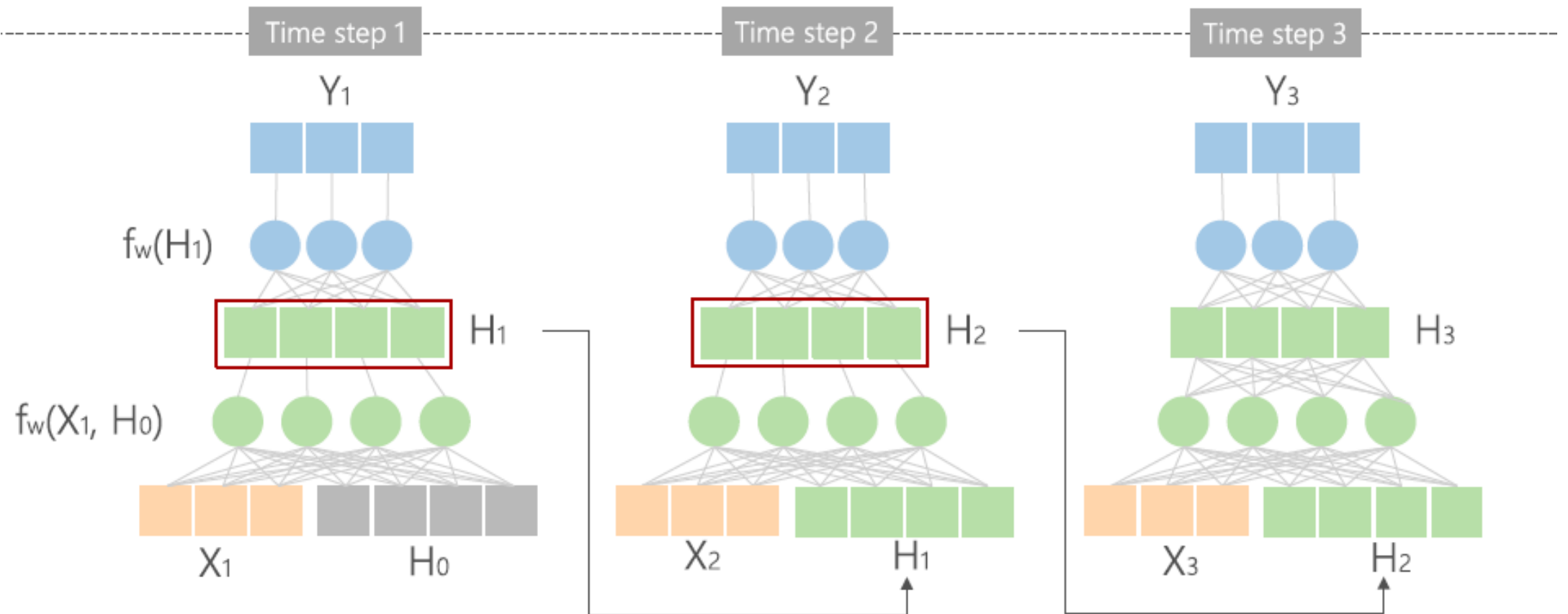
Recurrent Neural Networks (RNN)

- The same weight and bias shared across all the network steps

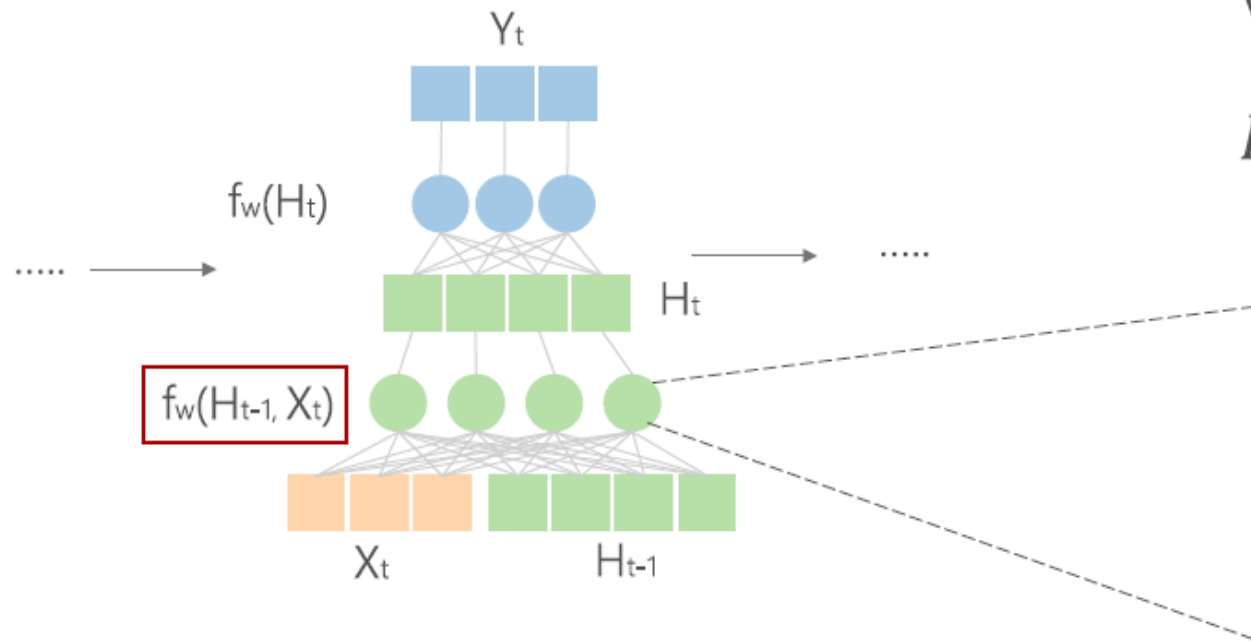


RNN – First Complete Picture

Internal state (a.k.a. Hidden state at time t $H_t \in \mathbb{R}^u$); $u := \text{units}$



Vanilla RNN Unit



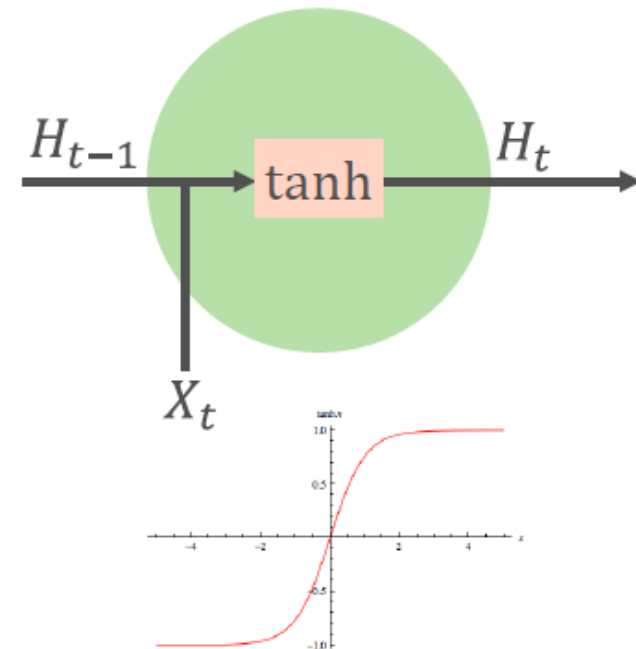
some function
with parameter W

$$H_t = f_W(H_{t-1}, X_t)$$

new state old state Input vector at
some time step

Vanilla RNN:

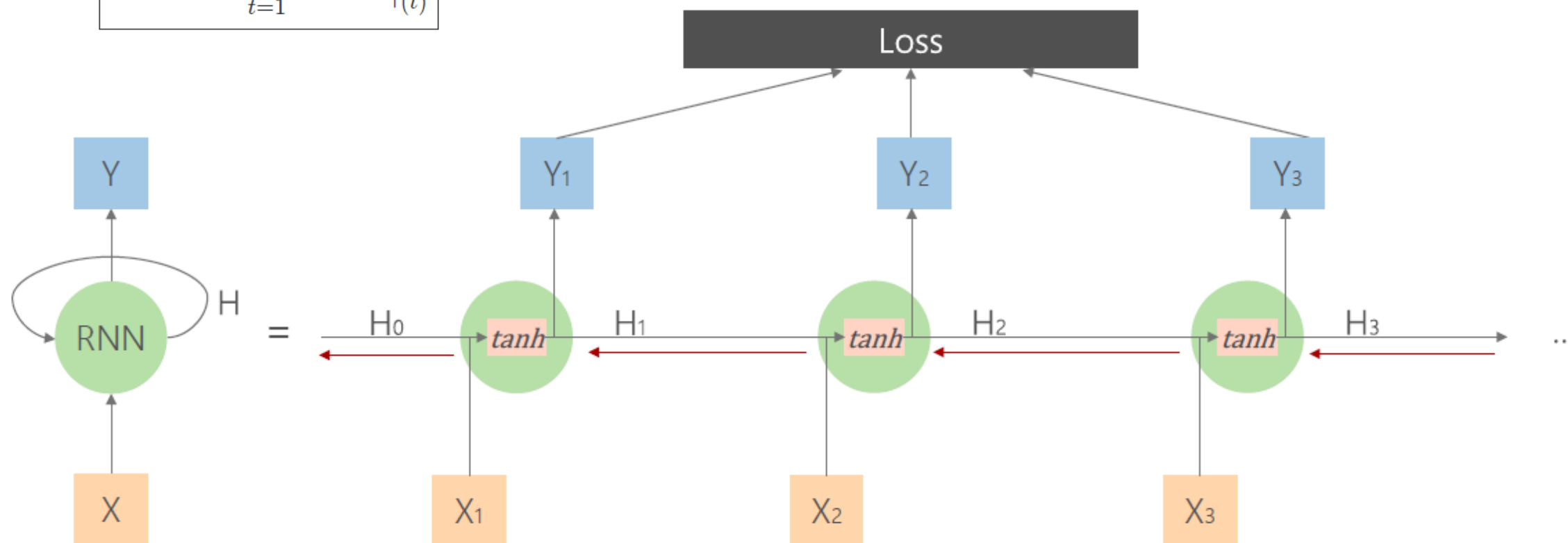
$$H_t = \tanh(W_h H_{t-1} + W_x X_t)$$
$$= \tanh(\mathbf{W} \cdot [H_{t-1}, X_t])$$



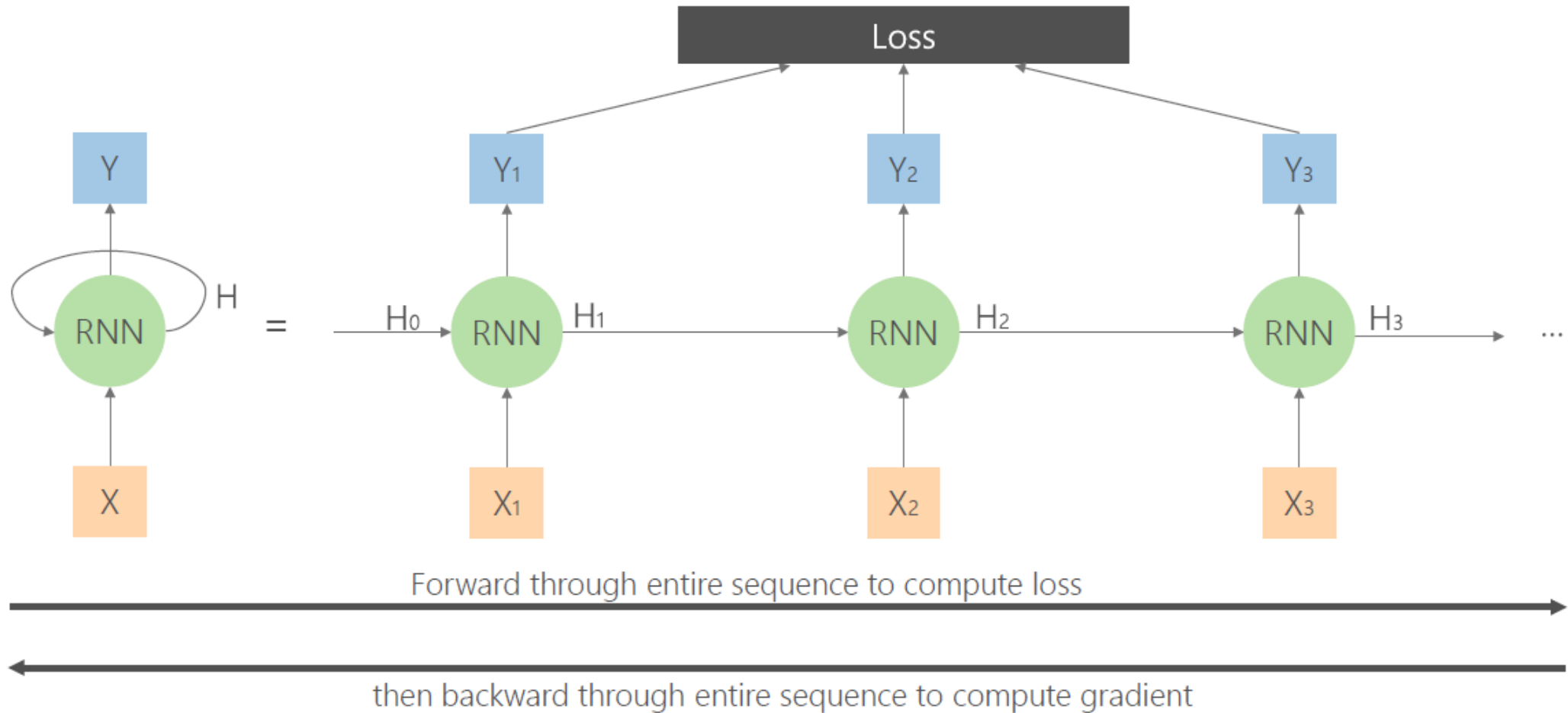
Back propagation (through time) of Hidden states

$$\frac{\partial \mathcal{L}^{(T)}}{\partial W} = \sum_{t=1}^T \frac{\partial \mathcal{L}^{(T)}}{\partial W} \Big|_{(t)}$$

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}(\hat{y}^{<t>}, y^{<t>})$$



RNN - Backpropagation



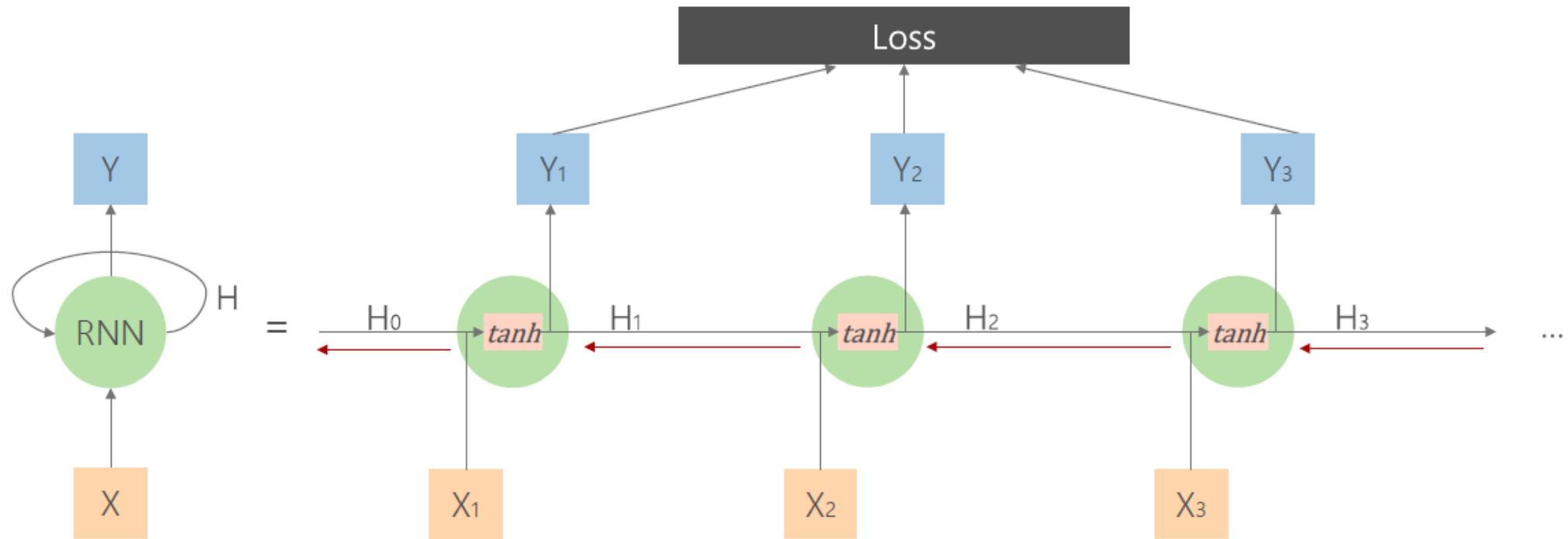
Back propagation of Hidden states

Vanilla RNN Gradient Problems

Computing gradient of H_0 involves repeated tanh and many factors of W which causes:

Exploding gradient (e.g. $5*5*5*5*5*5*.....$)

Vanishing gradients (e.g. $0.7*0.7*0.7*0.7*0.7*0.7*.....$)



Explosion of the gradient (e.g. $5*5*5*5*5*5*$)

Exploding gradients are obvious:

Your gradients will become a large number (e.g., NaN (not a number))

Solution: **Gradient clipping**

Clip the gradient when it goes higher than a threshold

Gradient Vanishing

(e.g. $0.7 \times 0.7 \times 0.7 \times 0.7 \times 0.7 \times 0.7 \times \dots$)

Vanishing gradients are more problematic because it's not obvious when they occur or how to deal with them.

Solutions:

- Change activation function to ReLU
- Proper initialization
- **Regularization := [Cost function = Loss + Regularization terms]**
- **Change architecture to LSTM or GRU**

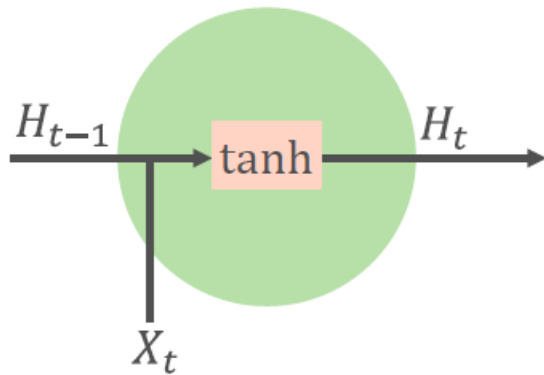
<https://keras.io/api/layers/regularizers/>

```
from keras import regularizers
model.add(Dense(64, input_dim=64,
                  kernel_regularizer=regularizers.l2(0.01))
```

Gated Recurrent Unit (GRU)

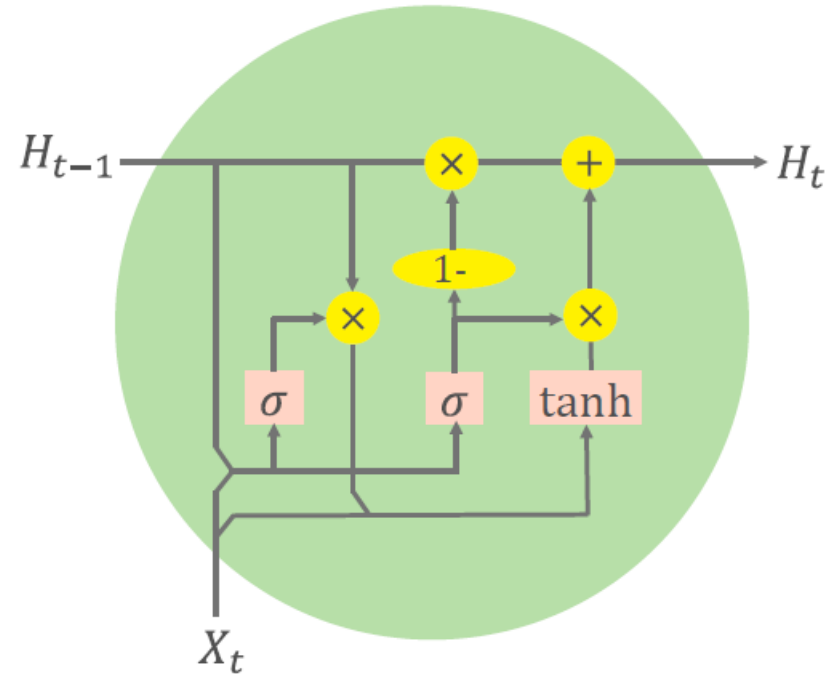
Vanilla RNN:

$$H_t = \tanh(\mathbf{W} \cdot [H_{t-1}, X_t])$$



GRU:

$$H_t = (1 - z_t) \times H_{t-1} + z_t \times \tilde{H}_t$$

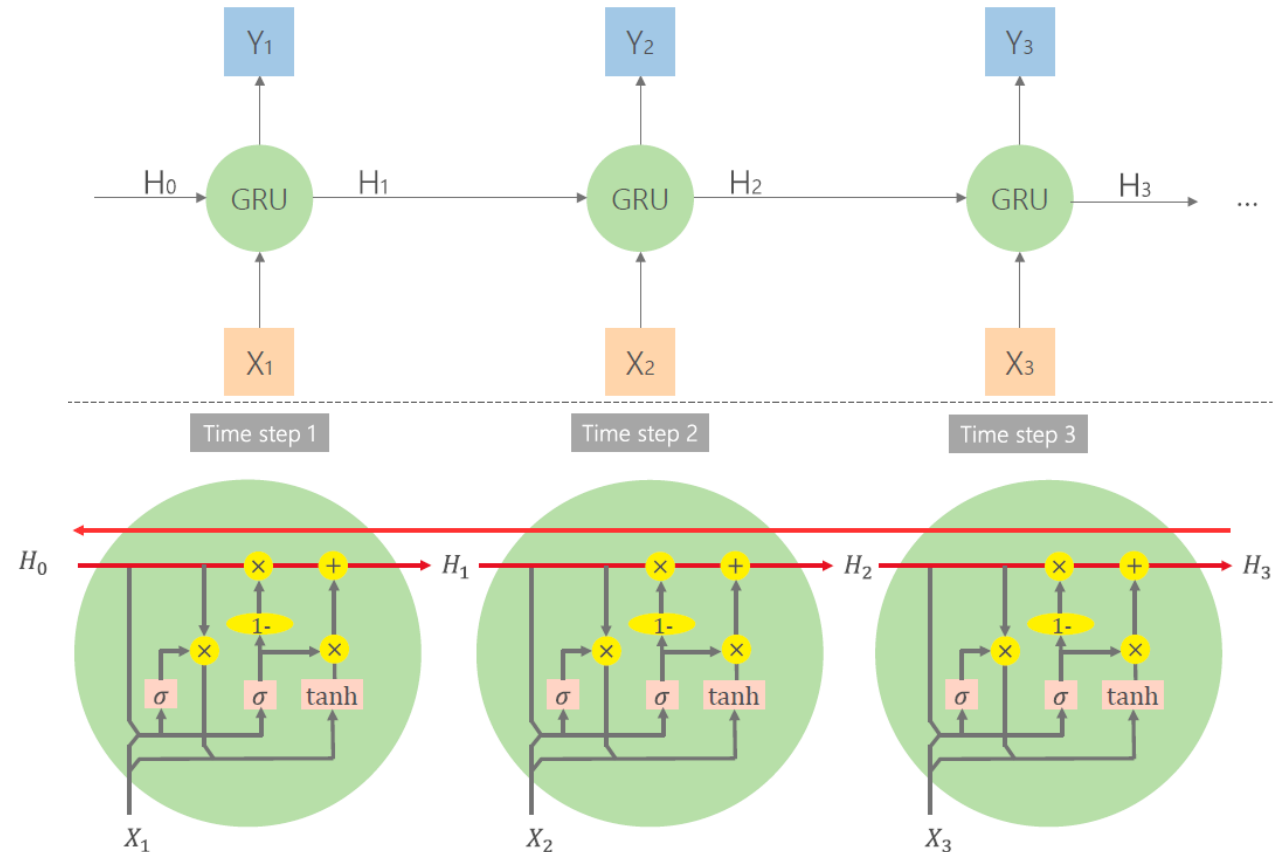


Uninterrupted gradient flow

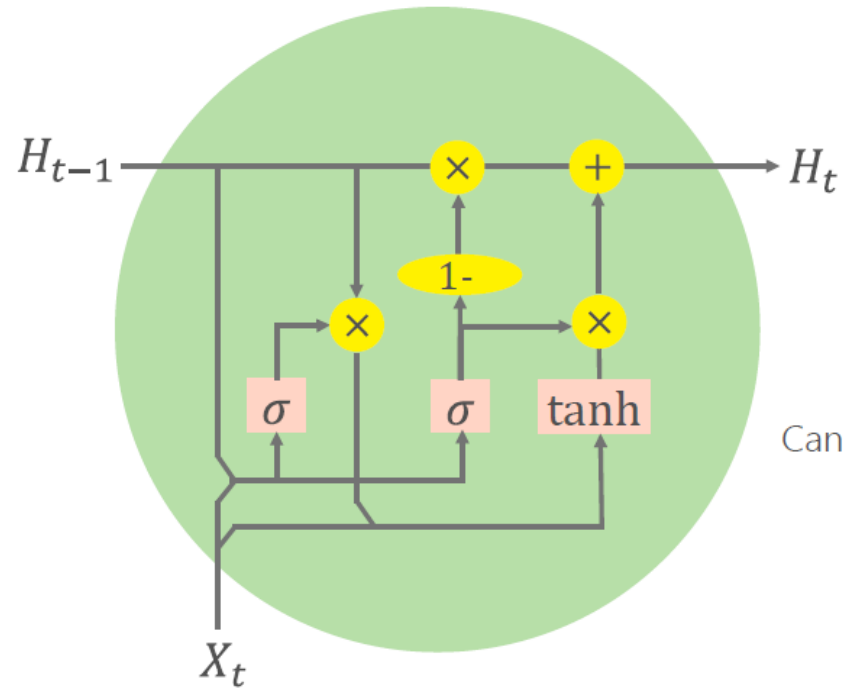
State runs straight through the entire chain with minor linear interactions which makes information very easy to pass.

GRU:

$$H_t = \underline{(1 - z_t) \times H_{t-1} + z_t \times \tilde{H}_t}$$



Gated Recurrent Unit (GRU)

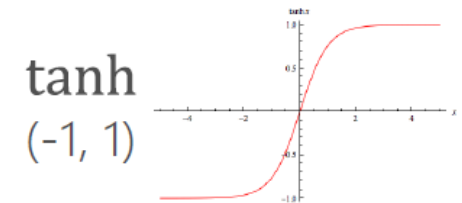
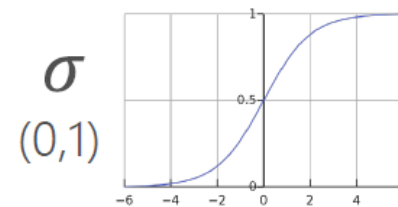


Hidden state: $H_t = (1 - z_t) \times H_{t-1} + z_t \times \tilde{H}_t$

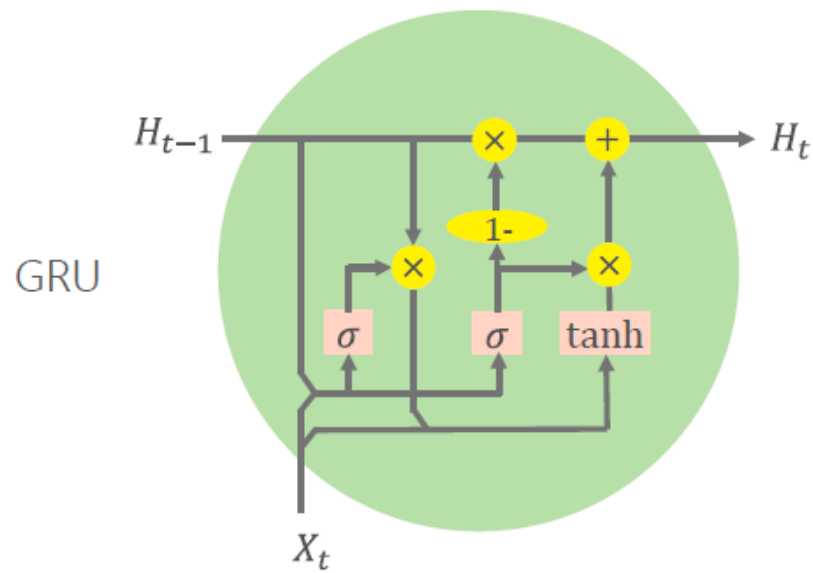
Update gates: $z_t = \sigma(W_z \cdot [H_{t-1}, X_t])$

Candidate gates/states: $\tilde{H}_t = \tanh(W \cdot [r_t \times H_{t-1}, X_t])$

Reset gates: $r_t = \sigma(W_r \cdot [H_{t-1}, X_t])$



GRU vs LSTM (Long short-term memory) cell

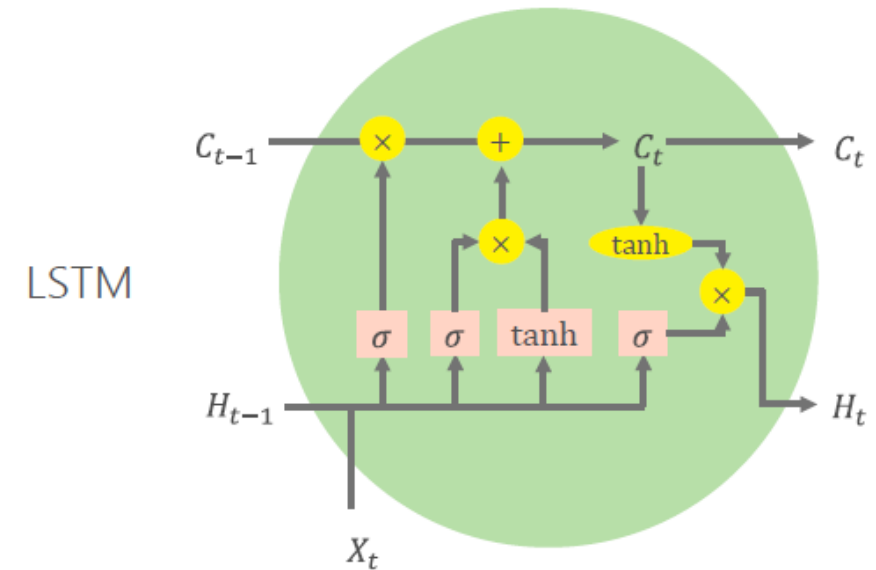


Hidden state: $H_t = (1 - z_t) \times H_{t-1} + z_t \times \tilde{H}_t$

Update gates: $z_t = \sigma(W_z \cdot [H_{t-1}, X_t])$

Reset gates: $r_t = \sigma(W_r \cdot [H_{t-1}, X_t])$

Candidate gates/states: $\tilde{H}_t = \tanh(W \cdot [r_t \times H_{t-1}, X_t])$



Hidden cell state: $C_t = f_t \times C_{t-1} + i_t \times \tilde{C}_t$

Hidden state: $H_t = o_t \times \tanh(C_t)$

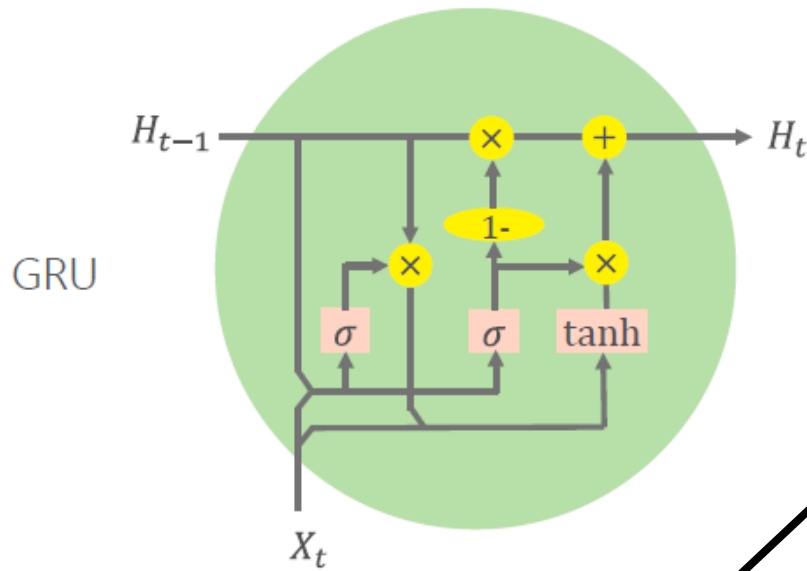
Forget gates: $f_t = \sigma(W_f \cdot [H_{t-1}, X_t])$

Input gates: $i_t = \sigma(W_i \cdot [H_{t-1}, X_t])$

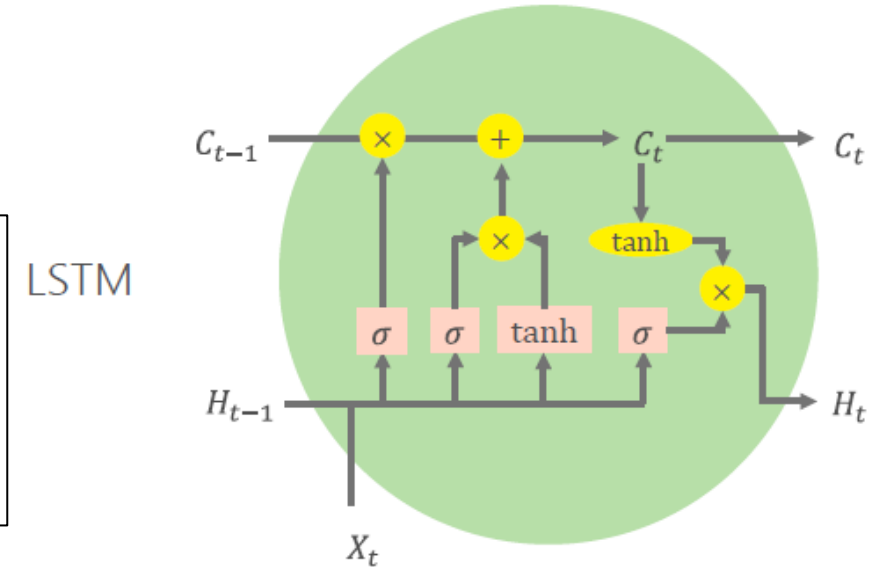
Candidate gates/states: $\tilde{C}_t = \tanh(W_g \cdot [H_{t-1}, X_t])$

Output gates: $o_t = \sigma(W_o \cdot [H_{t-1}, X_t])$

GRU vs LSTM (Long short-term memory) cell



How much of the candidate activation vector to incorporate into the new hidden state



Hidden state: $H_t = (1 - z_t) \times H_{t-1} + z_t \times \tilde{H}_t$

Update gates: $z_t = \sigma(W_z \cdot [H_{t-1}, X_t])$

Reset gates: $r_t = \sigma(W_r \cdot [H_{t-1}, X_t])$

Candidate gates/states: $\tilde{H}_t = \tanh(W \cdot [r_t \times H_{t-1}, X_t])$

Hidden cell state: $C_t = f_t \times C_{t-1} + i_t \times \tilde{C}_t$

Hidden state: $H_t = o_t \times \tanh(C_t)$

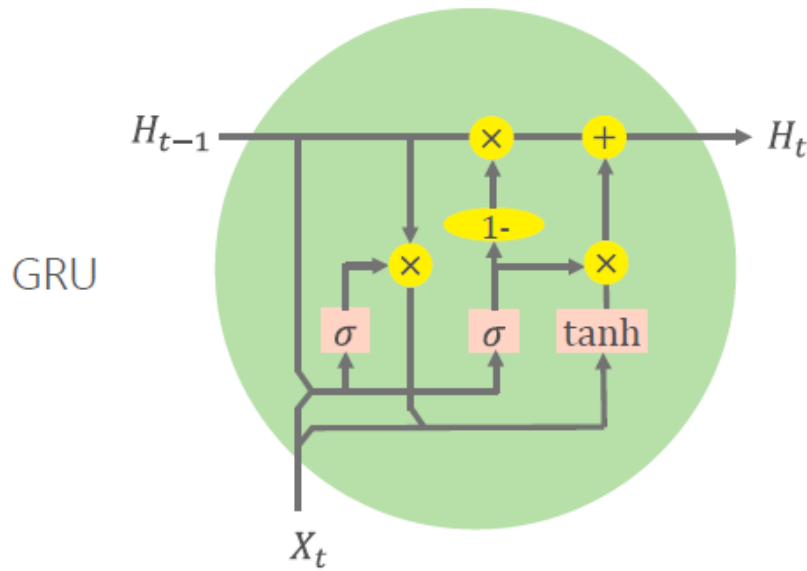
Forget gates: $f_t = \sigma(W_f \cdot [H_{t-1}, X_t])$

Input gates: $i_t = \sigma(W_i \cdot [H_{t-1}, X_t])$

Candidate gates/states: $\tilde{C}_t = \tanh(W_g \cdot [H_{t-1}, X_t])$

Output gates: $o_t = \sigma(W_o \cdot [H_{t-1}, X_t])$

GRU vs LSTM (Long short-term memory) cell



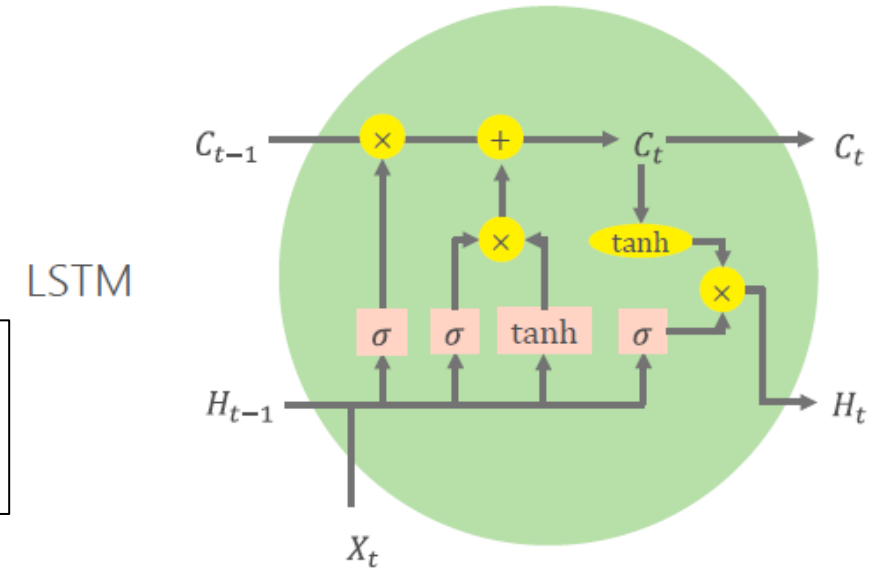
It determines how much of the previous hidden state to forget

Hidden state: $H_t = (1 - z_t) \times H_{t-1} + z_t \times \tilde{H}_t$

Update gates: $z_t = \sigma(W_z \cdot [H_{t-1}, X_t])$

Reset gates: $r_t = \sigma(W_r \cdot [H_{t-1}, X_t])$

Candidate gates/states: $\tilde{H}_t = \tanh(W \cdot [r_t \times H_{t-1}, X_t])$



Hidden cell state: $C_t = f_t \times C_{t-1} + i_t \times \tilde{C}_t$

Hidden state: $H_t = o_t \times \tanh(C_t)$

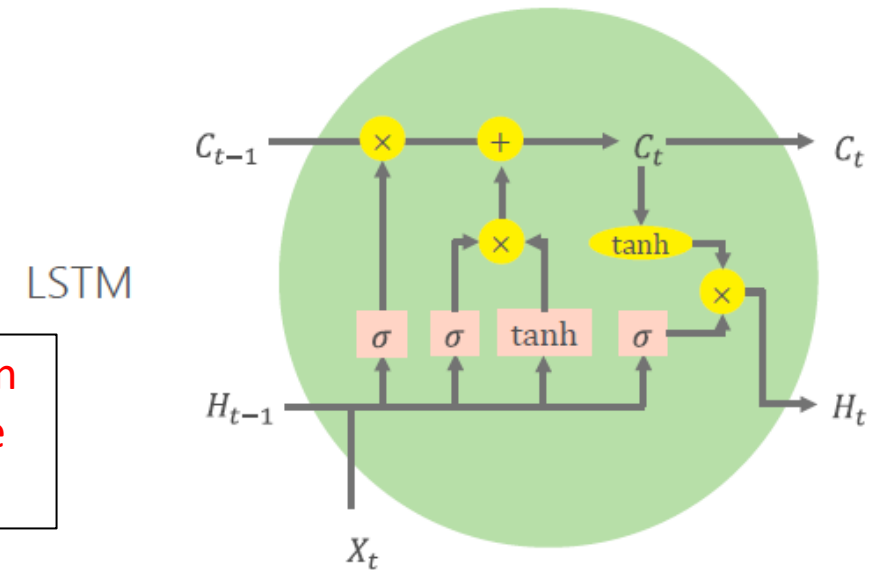
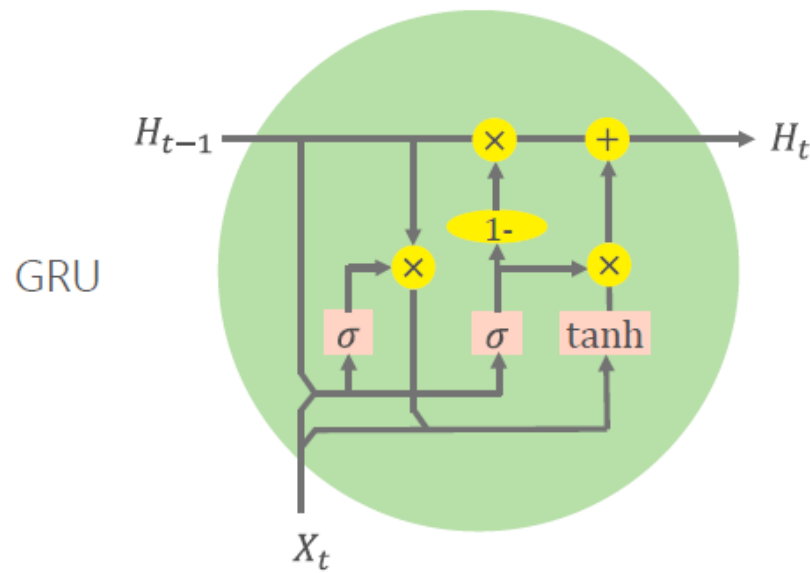
Forget gates: $f_t = \sigma(W_f \cdot [H_{t-1}, X_t])$

Input gates: $i_t = \sigma(W_i \cdot [H_{t-1}, X_t])$

Candidate gates/states: $\tilde{C}_t = \tanh(W_g \cdot [H_{t-1}, X_t])$

Output gates: $o_t = \sigma(W_o \cdot [H_{t-1}, X_t])$

GRU vs LSTM (Long short-term memory) cell



It combines information from the input and the previous hidden state

Hidden state: $H_t = (1 - z_t) \times H_{t-1} + z_t \times \tilde{H}_t$

Update gates: $z_t = \sigma(W_z \cdot [H_{t-1}, X_t])$

Reset gates: $r_t = \sigma(W_r \cdot [H_{t-1}, X_t])$

Candidate gates/states: $\tilde{H}_t = \tanh(W \cdot [r_t \times H_{t-1}, X_t])$

Hidden cell state: $C_t = f_t \times C_{t-1} + i_t \times \tilde{C}_t$

Hidden state: $H_t = o_t \times \tanh(C_t)$

Forget gates: $f_t = \sigma(W_f \cdot [H_{t-1}, X_t])$

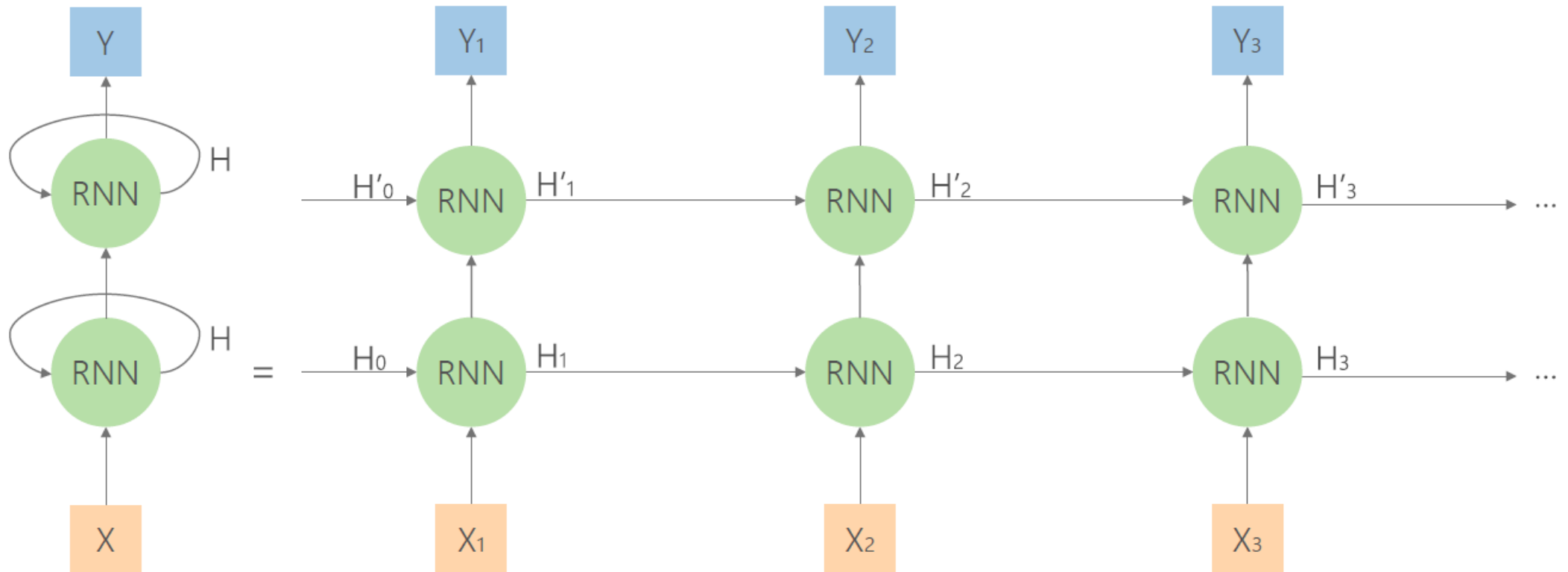
Input gates: $i_t = \sigma(W_i \cdot [H_{t-1}, X_t])$

Candidate gates/states: $\tilde{C}_t = \tanh(W_g \cdot [H_{t-1}, X_t])$

Output gates: $o_t = \sigma(W_o \cdot [H_{t-1}, X_t])$

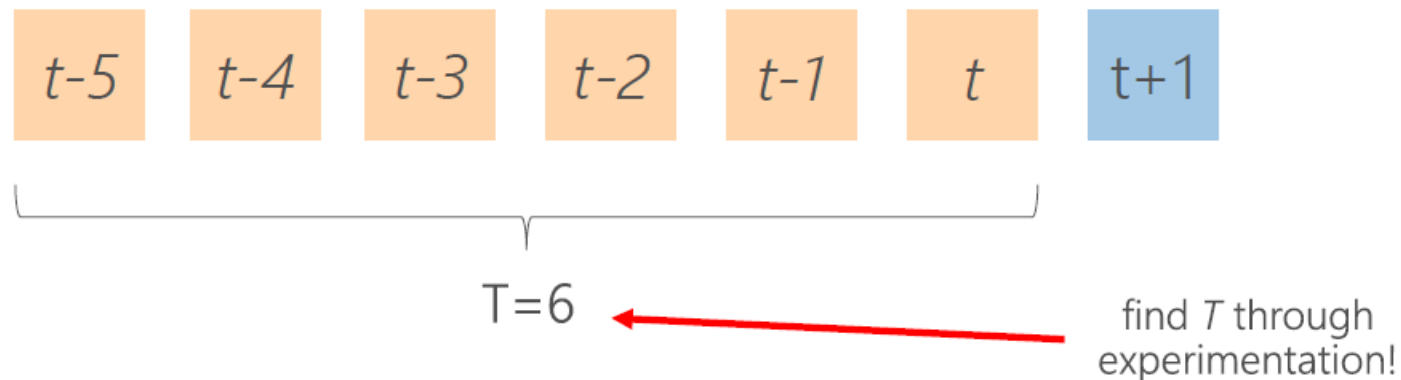
Cells Stacking

To learn more complex relationships, we can stack multiple cells



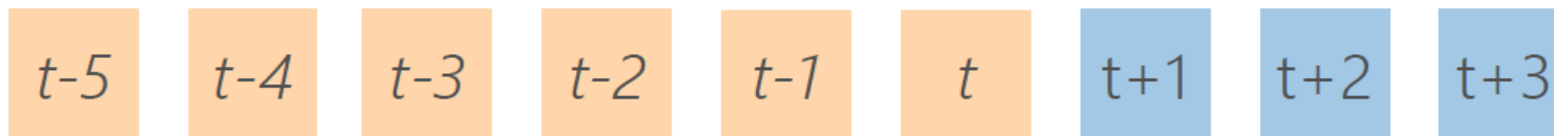
One step forecasting (HORIZON = 1)

- Assuming we are at time t ...
- ... predict the value at time $t+1$...
- ... conditional on the previous T values of the time series

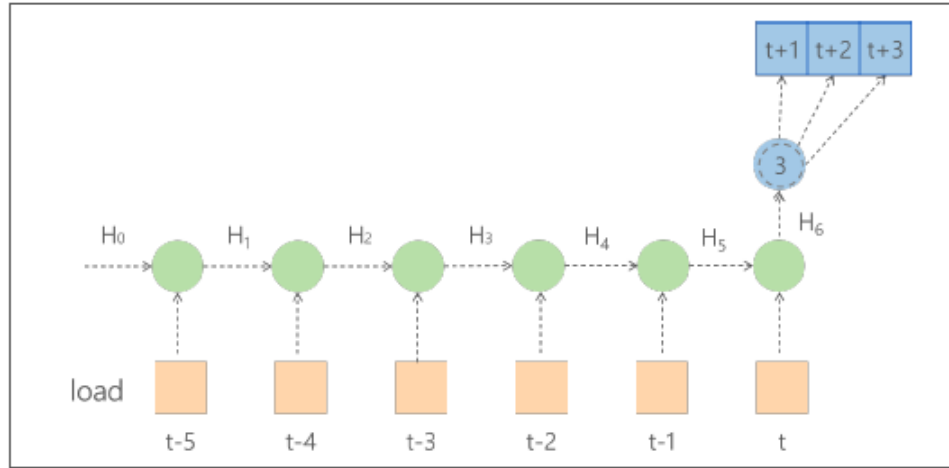


Multi-step forecast

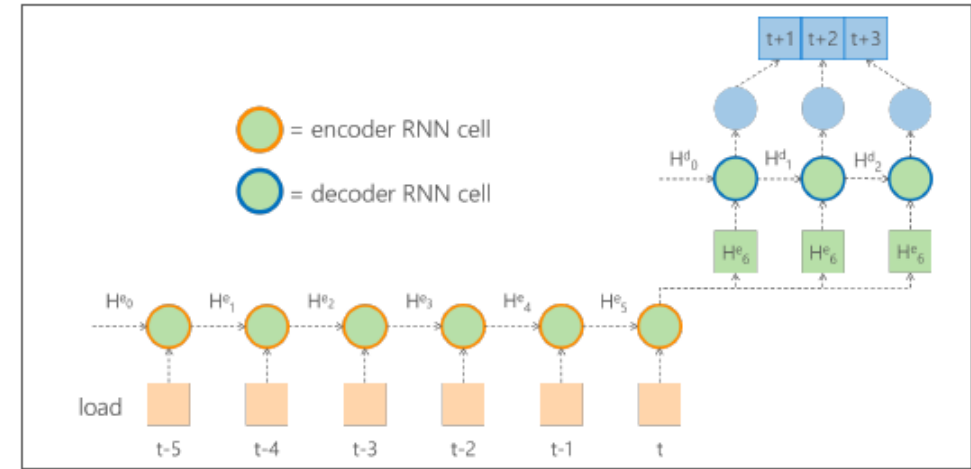
- Assuming we are at time t ...
- ... predict the values at times $(t+1, \dots, t+HORIZON)$...
- ... conditional on the previous T values of the time series



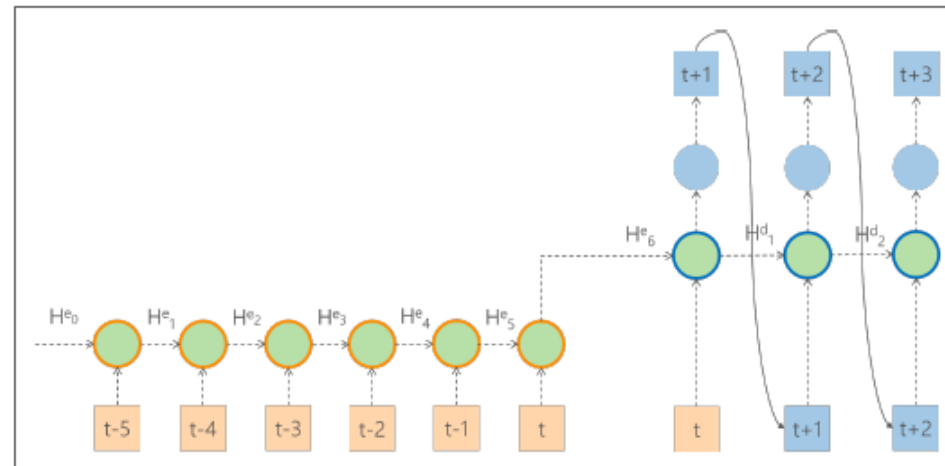
Multi-step forecast



Vector output



Simple encoder-decoder



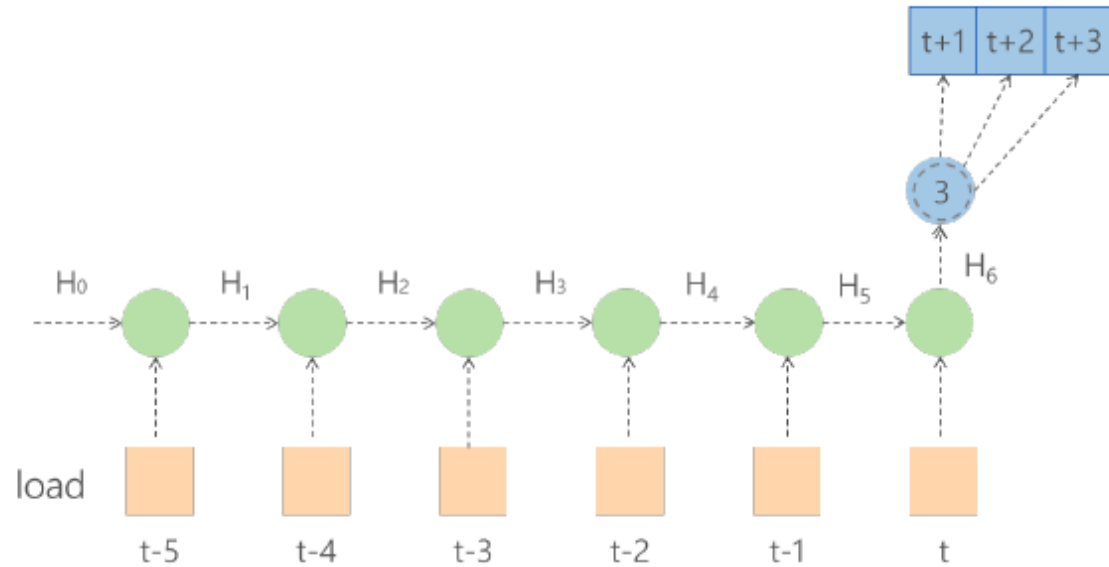
Recursive encoder-decoder

Vector Output

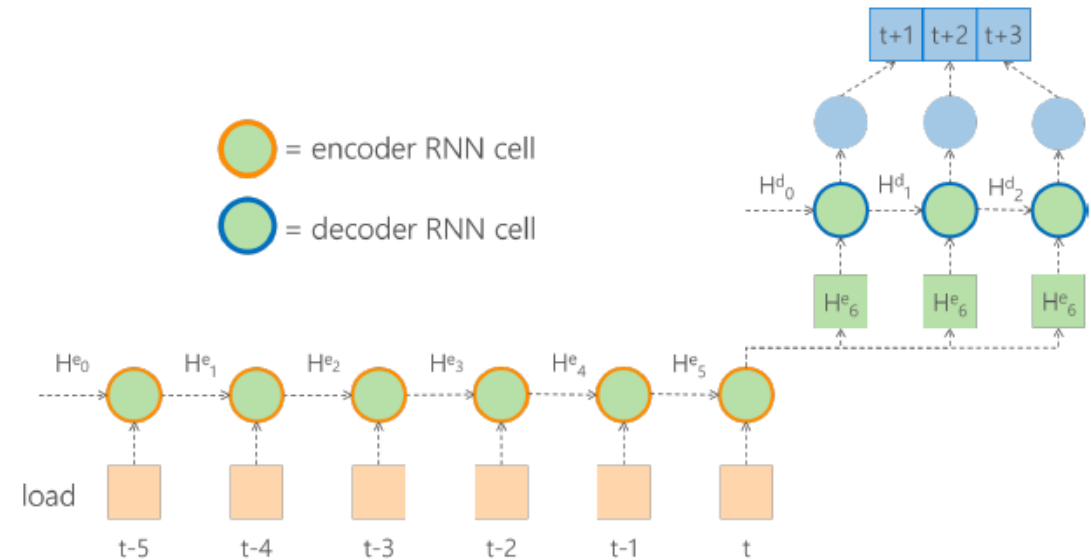
👍 Simplest to implement

👍 Fastest to train

👎 Does not model dependencies between predicted outputs

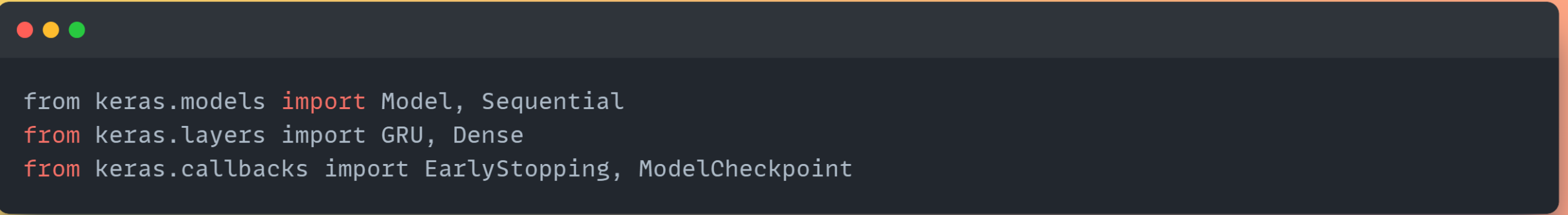


Simple Encoder - Decoder



- 👍 Fairly simple to implement
- 👍 Tries to capture dependencies between forecasted time steps through decoder hidden state
- 👎 Slower to train with stacked RNN layers

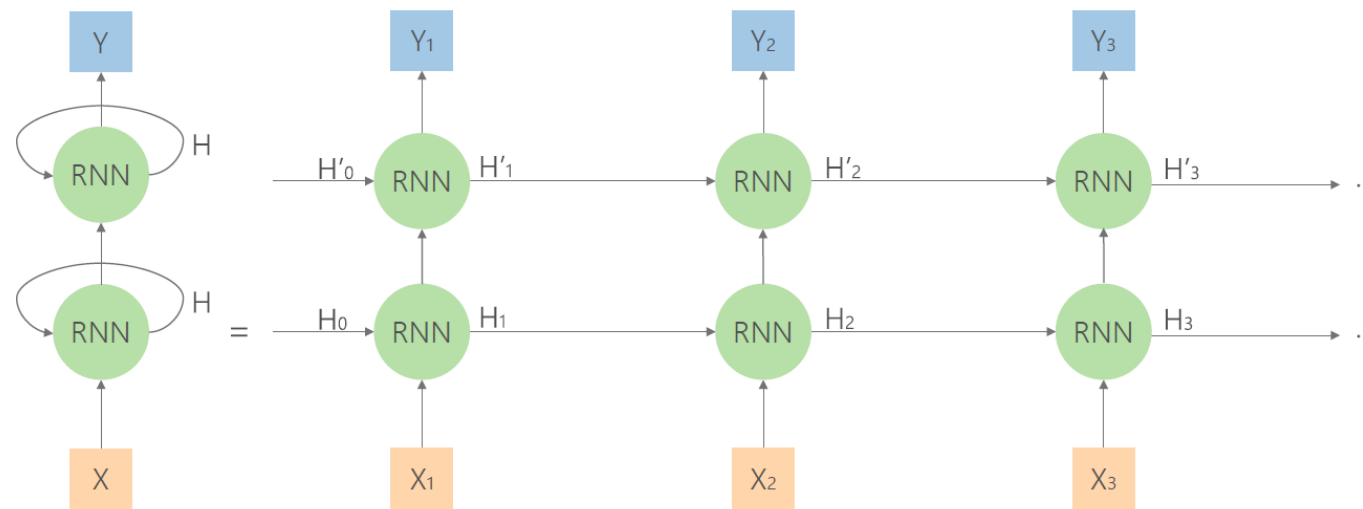
Import in Keras



```
from keras.models import Model, Sequential
from keras.layers import GRU, Dense
from keras.callbacks import EarlyStopping, ModelCheckpoint
```

Cells Stacking

To learn more complex relationships, we can stack multiple cells



```
model = Sequential()
model.add(tf.keras.layers.SimpleRNN(4, (return_sequences = True), (input_shape = (timesteps, data_dim))))
model.add(tf.keras.layers.SimpleRNN(4))
```

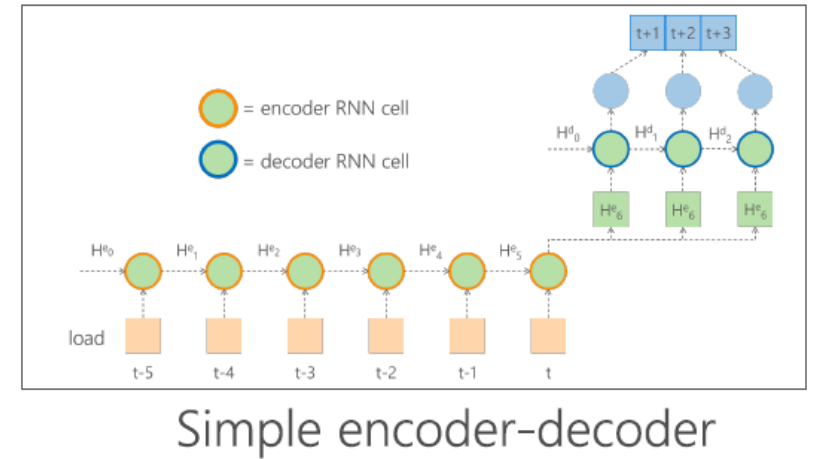
The GRU layer – 1D time series

```
model = Sequential()  
model.add(GRU(LATENT_DIM, input_shape=(T, 1)))  
model.add(Dense(HORIZON))
```

snappify.com

https://keras.io/api/layers/recurrent_layers/gru/

Multistep forecast architecture 1D time series

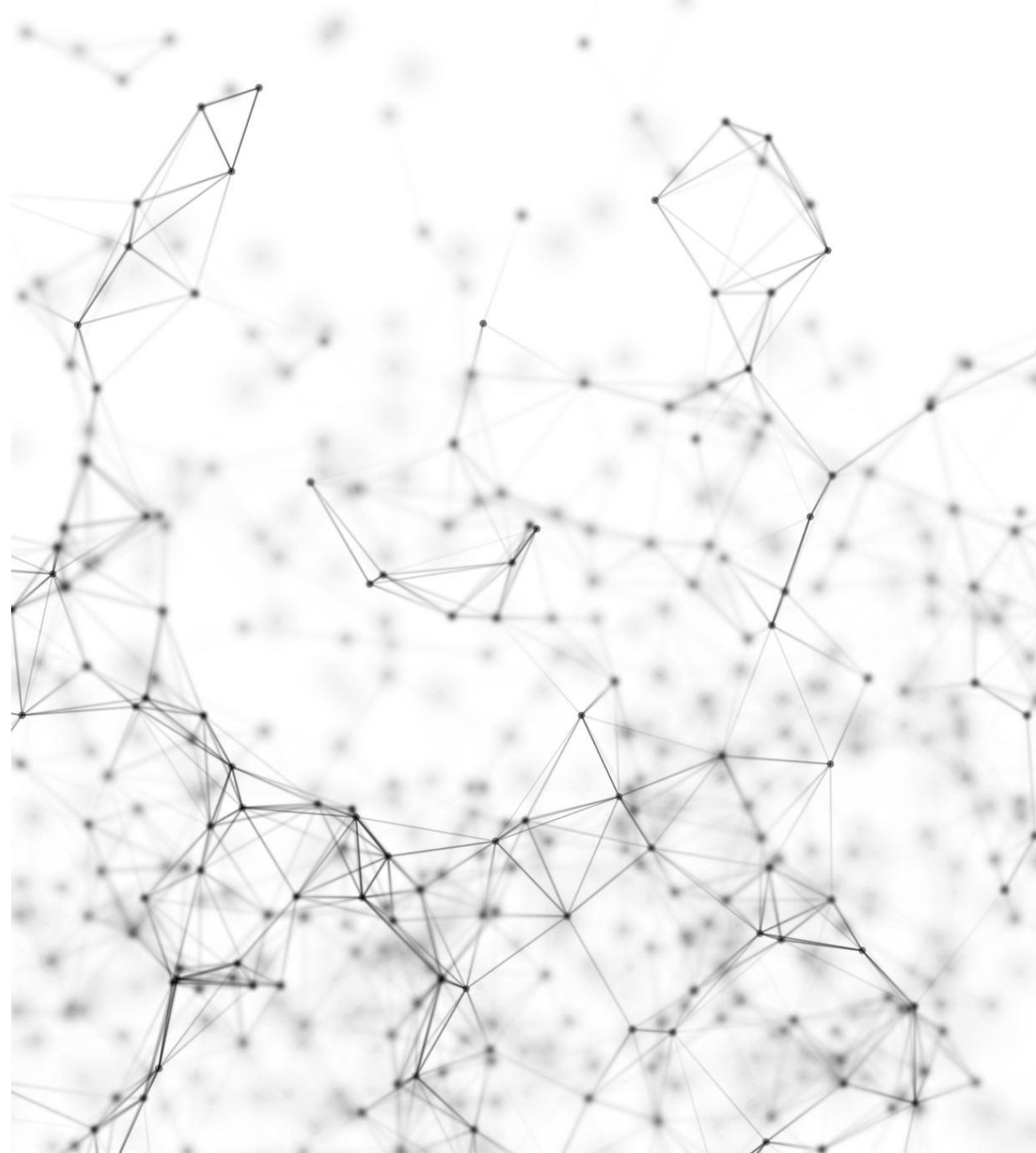


```
NEW_HORIZON = ...  
model = Sequential()  
model.add(GRU(LATENT_DIM, input_shape=(T, 1)))  
model.add(RepeatVector(NEW_HORIZON))  
model.add(GRU(LATENT_DIM, return_sequences=True))  
model.add(TimeDistributed(Dense(1)))  
model.add(Flatten()))N))
```

snappify.com

https://keras.io/api/layers/reshaping_layers/repeat_vector/
https://keras.io/api/layers/recurrent_layers/time_distributed/

Attention-based Transformer for Time series Forecasting



Attention-based Transformer

- Attention-based Transformer models are designed around the attention mechanism, which allows the model to dynamically focus on different parts of input sequences, effectively capturing important dependencies regardless of their distance in time.
- The intuition behind the attention mechanism, is that it allows the model to dynamically focus on the most relevant components of an input sequence when generating a representation or output. Instead of processing each element locally or sequentially (as in RNNs or CNNs)

Attention-based Transformer

- **Self-Attention:** The mechanism by which each element of an input sequence considers every other element, computing weighted relationships. This enables the model to learn long-range temporal dependencies crucial for time series and sequence modeling.
- **Multi-Head Attention:** Multiple attention heads run in parallel, each capturing different aspects or patterns within the input, enhancing representation richness

Self-attention Mechanism

The self-attention operation in Transformers starts with building three different linearly-weighted vectors from the input $\{X_i\}_{i=1}^n$, referred to as query $q \in \mathbb{R}^{s \times 1}$, key $k \in \mathbb{R}^{s \times 1}$ and value $v \in \mathbb{R}^s$

- For an input x_i , the query q_i , key k_i and value v_i vectors can be found using: $q_i = W_q x_i$, $k_i = W_k x_i$, and $v_i = W_v x_i$, (3) where W_q and $W_k \in \mathbb{R}^{s \times 1 \times d}$, $W_v \in \mathbb{R}^{s \times d}$, represent learnable weight matrices. The output vectors $\{z_i\}_{i=1}^n$ are given by, $z_i = \sum_j X_j \text{softmax}(q_i^T k_j) v_j$

Attention-based transformer

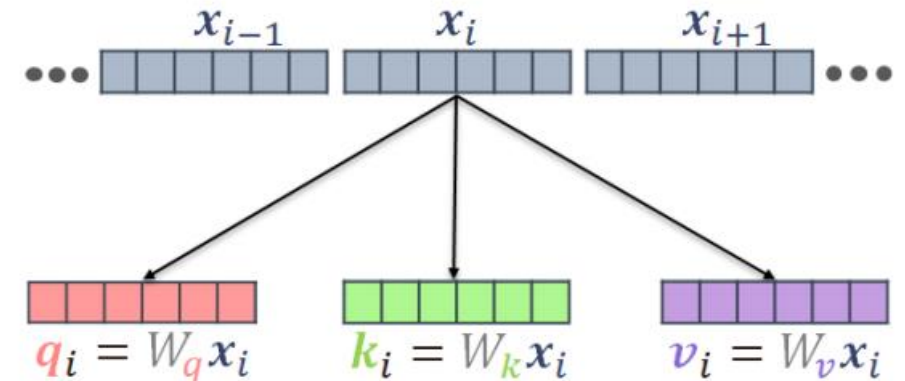
- For an input x_i , the query q_i , key k_i and value v_i :

$$\mathbf{q}_i = W_q \mathbf{x}_i, \mathbf{k}_i = W_k \mathbf{x}_i, \text{ and } \mathbf{v}_i = W_v \mathbf{x}_i$$

where W_q and $W_k \in \mathbb{R}^{s_1 \times d}$, $W_v \in \mathbb{R}^{s \times d}$, represent learnable weight matrices.

- The output vectors of attention $\{z_i\}_{i=1}^n$ are given by

$$z_i = \sum_j \text{softmax}(q_i^T k_j) v_j$$

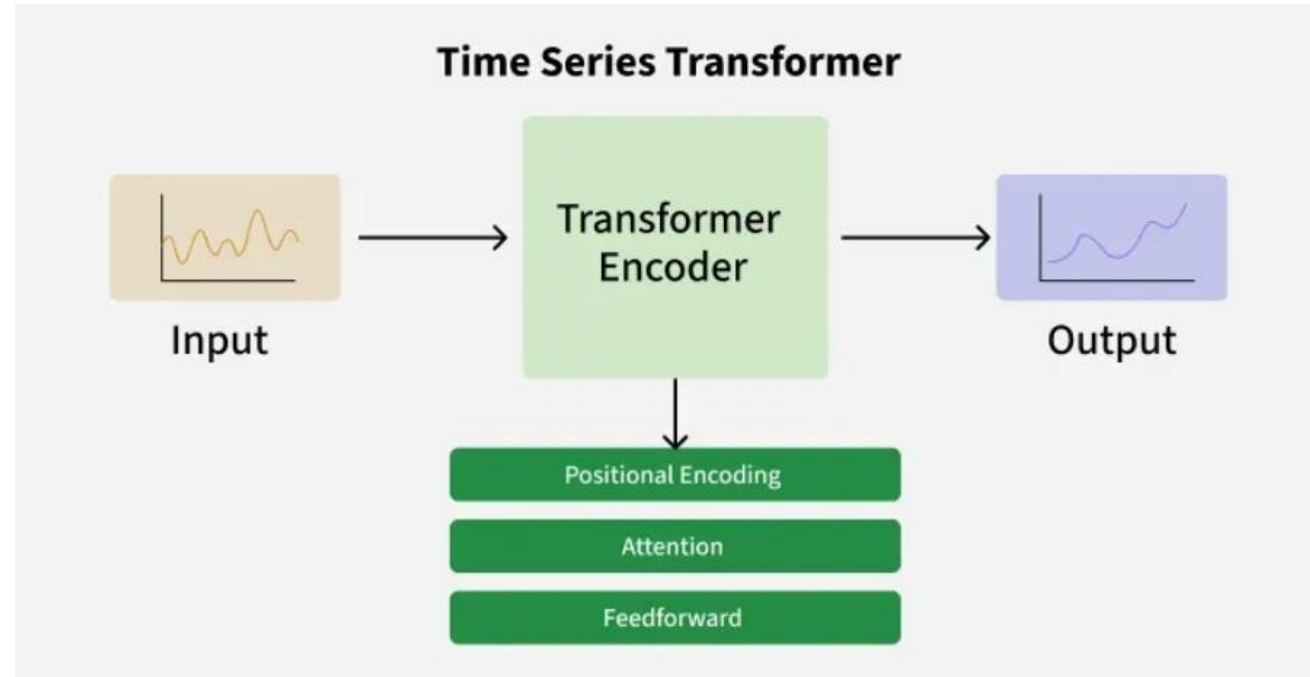


Scaled dot-product attention

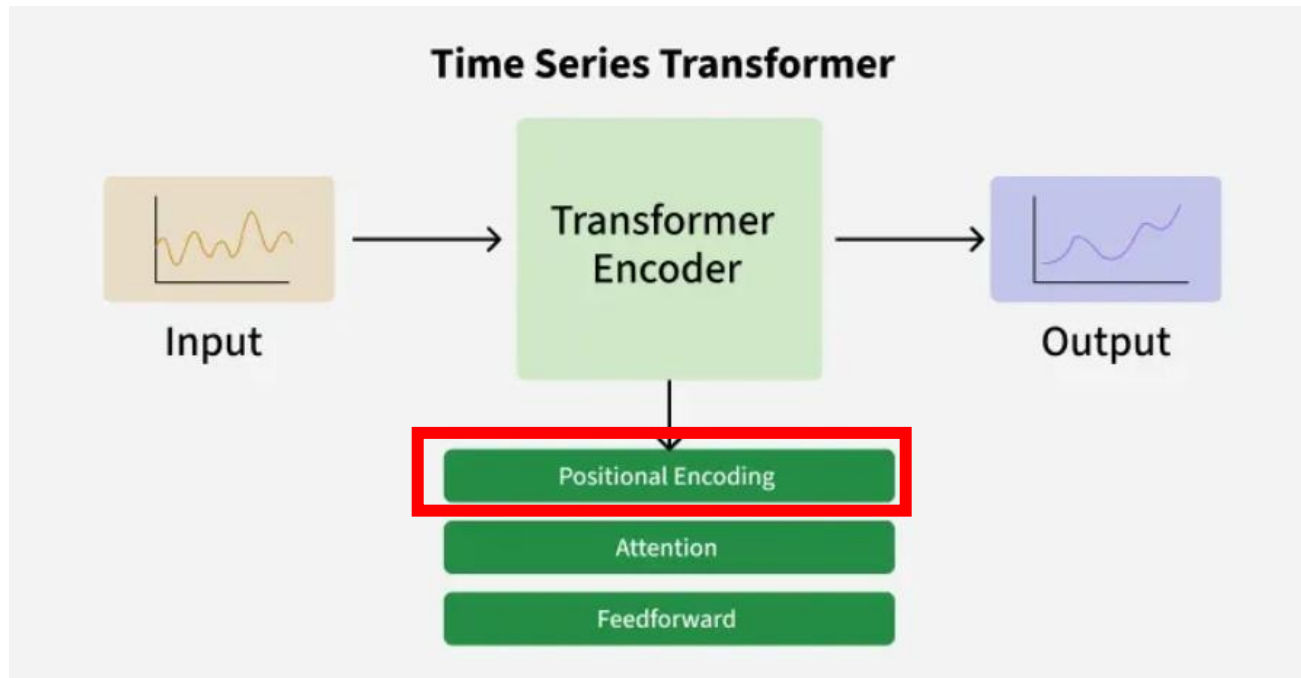
- As the softmax function is sensitive to large values, the attention weights are scaled by the square root of the size of the query and key vectors d_q as given by

$$z_i = \sum_j \text{softmax} \left(\frac{q_i^T k_j}{\sqrt{d_q}} \right) v_j$$

Time Series Transformer Architecture



Time Series Transformer Architecture



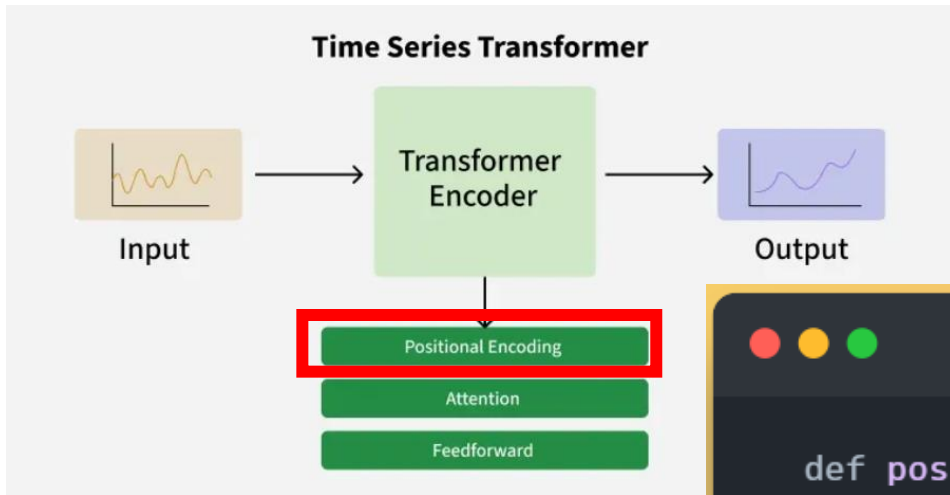
- Positional encoding injects information about the position (or order) of time series values directly into their vector representations.
- Possible solution Vector using sine and cosine functions of varying frequencies :

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE(pos, 2i + 1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

- i : index of the dimension
- d_{model} : embedding size

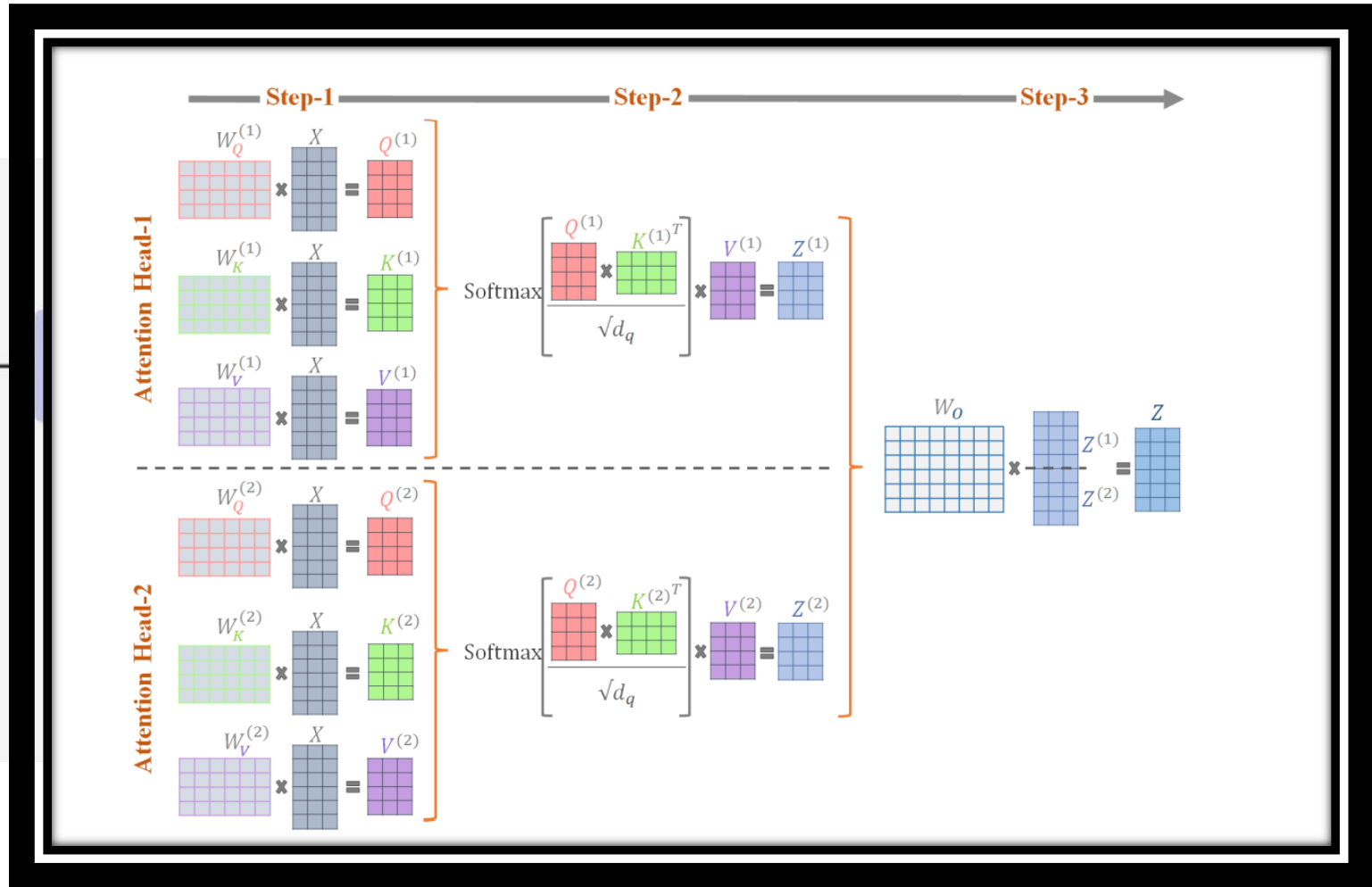
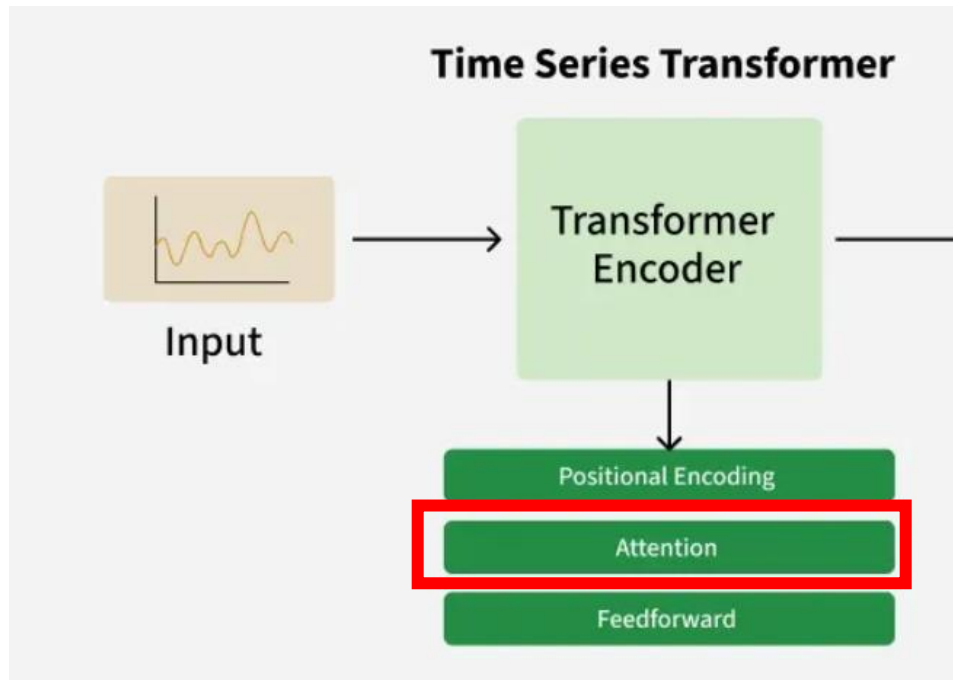
Time Series Transformer Architecture



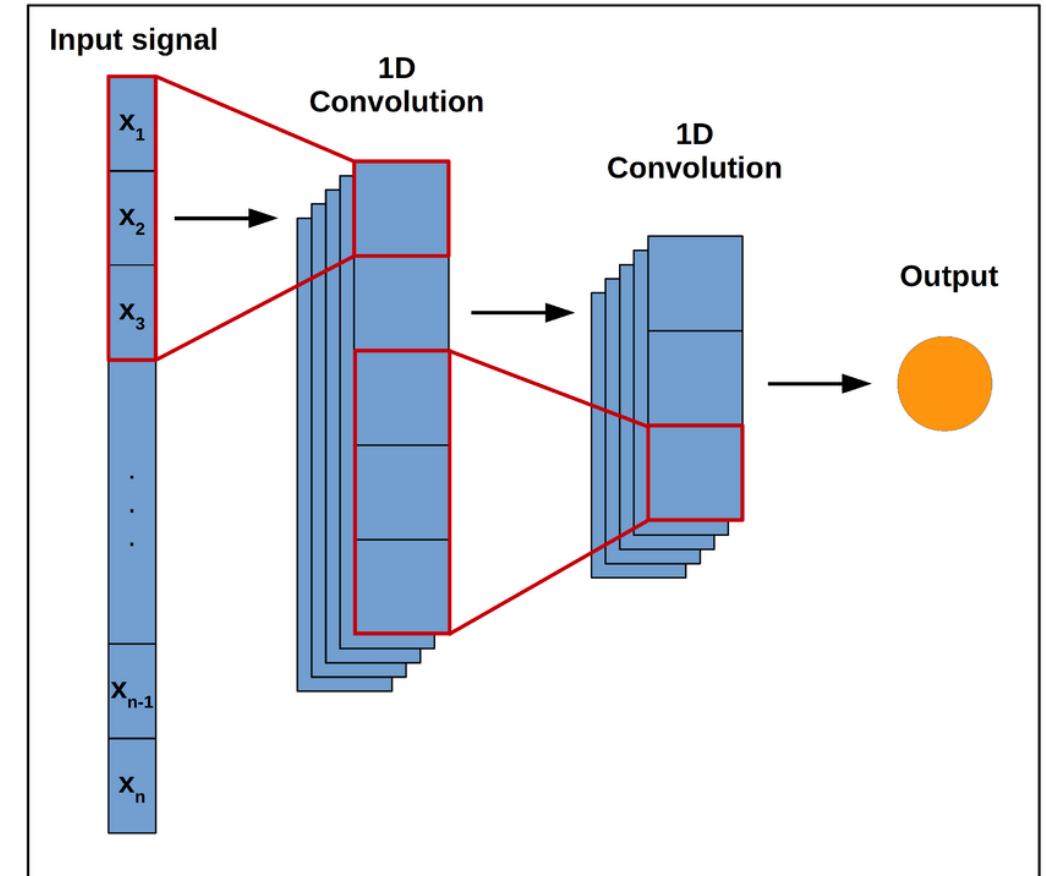
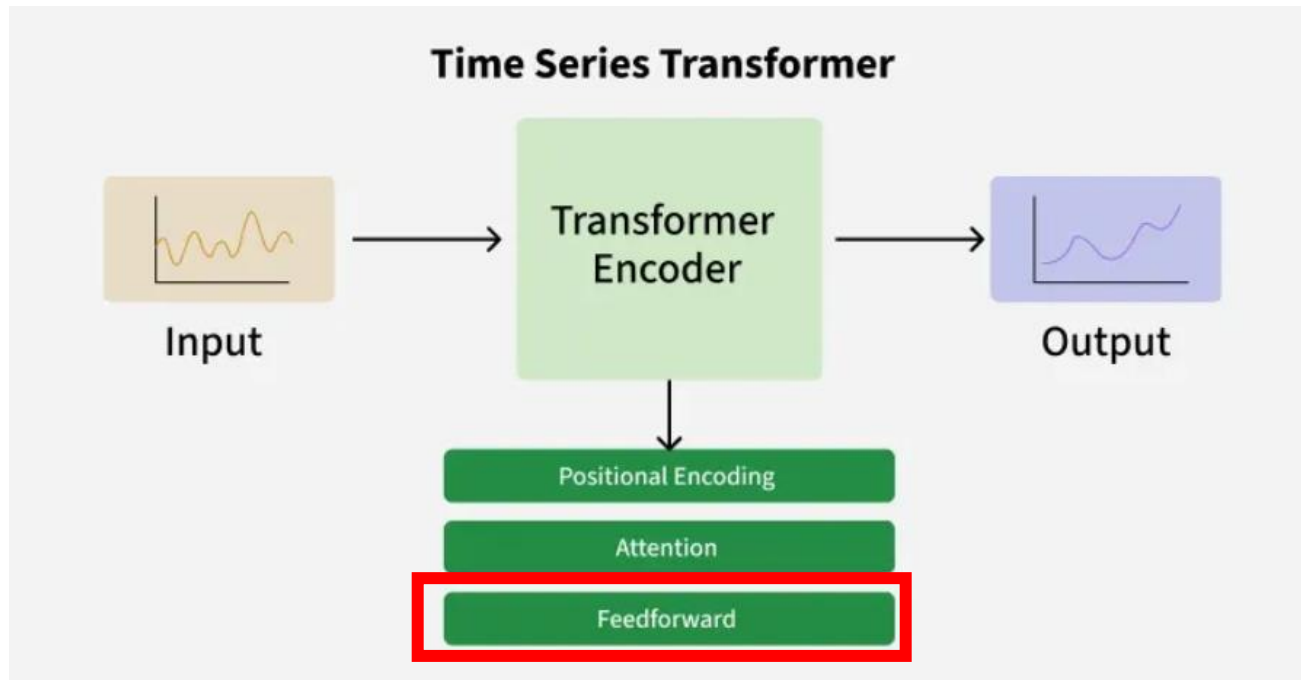
```
def positional_encoding(self, seq_len, dm):
```

```
    pos = tf.range(seq_len)[: , tf.newaxis]
    i = tf.range(dm)[tf.newaxis, :]
    angle_rates = 1 / tf.pow(10000., (2 * (i // 2)) / tf.cast(dm, tf.float32))
    angle_rads = tf.cast(pos, tf.float32) * angle_rates
    # apply sin to even indices in the array; 2i
    sines = tf.math.sin(angle_rads[:, 0::2])
    # apply cos to odd indices in the array; 2i+1
    cosines = tf.math.cos(angle_rads[:, 1::2])
    pos_encoding = tf.concat([sines, cosines], axis=-1)
    pos_encoding = pos_encoding[tf.newaxis, ...]
    return tf.cast(pos_encoding, tf.float32)
```

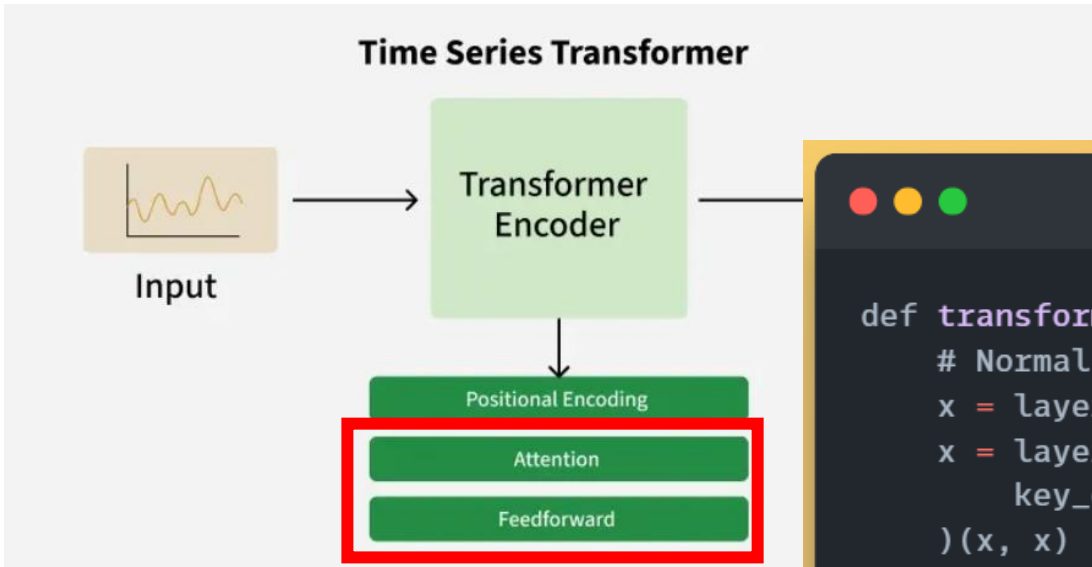
Time Series Transformer Architecture



Time Series Transformer Architecture

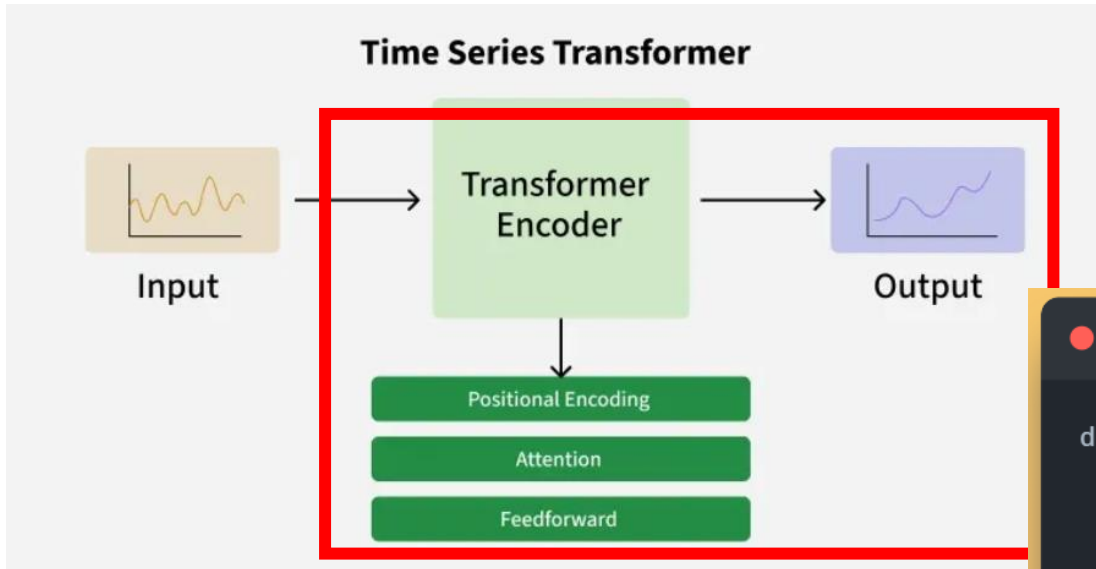


Time Series Transformer Architecture



```
def transformer_encoder(inputs, head_size, num_heads, ff_dim, dropout=0):  
    # Normalization and Attention  
    x = layers.LayerNormalization(epsilon=1e-6)(inputs)  
    x = layers.MultiHeadAttention(  
        key_dim=head_size, num_heads=num_heads, dropout=dropout  
    )(x, x)  
    x = layers.Dropout(dropout)(x)  
    res = x + inputs  
  
    # Feed Forward Part  
    x = layers.LayerNormalization(epsilon=1e-6)(res)  
    x = layers.Conv1D(filters=ff_dim, kernel_size=1, activation="relu")(x)  
    x = layers.Dropout(dropout)(x)  
    x = layers.Conv1D(filters=inputs.shape[-1], kernel_size=1)(x)  
    return x + res
```


Time Series Transformer Architecture



K Keras

```
def build_model(input_shape, head_size, num_heads, ff_dim, um_transformer_blocks,
                mlp_units, dropout=0, mlp_dropout=0):

    inputs = keras.Input(shape=input_shape)
    # Add positional encoding
    feature_dim = 64
    x = PositionalEncoding(T, feature_dim)(inputs)
    for _ in range(num_transformer_blocks):
        x = transformer_encoder(x, head_size, num_heads, ff_dim, dropout)

    x = layers.GlobalAveragePooling1D(data_format="channels_first")(x)

    for dim in mlp_units:
        x = layers.Dense(dim, activation="relu")(x)
        x = layers.Dropout(mlp_dropout)(x)
    outputs = layers.Dense(1)(x)

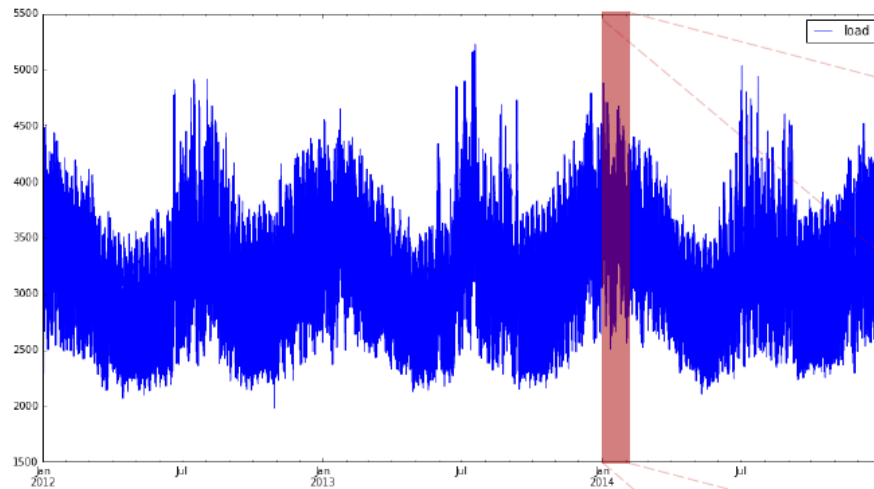
    return keras.Model(inputs, outputs)
```

A black and white photograph of a wooden floor with white chalk markings. The markings include the letters 'FF' and the number '0'. The text 'Presentation du TD' is overlaid in the center.

Presentation du TD

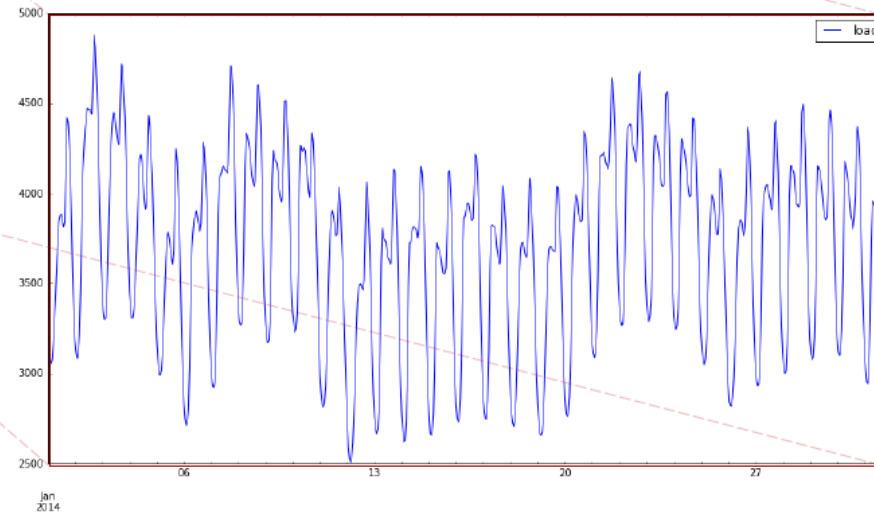
Data

New England ISO data



- 26,000 hourly load values
- Annual, weekly and daily seasonality

Tao Hong, Pierre Pinson, Shu Fan, Hamidreza Zareipour, Alberto Troccoli and Rob J. Hyndman, "Probabilistic energy forecasting: Global Energy Forecasting Competition 2014 and beyond", International Journal of Forecasting, vol.32, no.3, pp 896-913, July-September, 2016.



Notebook

- Open the file (Python Notebook):

TD_RNN_TRANSFORMER_TS_FORECASTING.ipynb

- Instruction are contained in the notebook

References

- <https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks>
- **DeepLearningForTimeSeriesForecasting (Microsoft)**
<https://github.com/Azure/DeepLearningForTimeSeriesForecasting>
- **Ben Auffarth** Machine Learning for Time-Series with Python
- <https://github.com/PacktPublishing/Machine-Learning-for-Time-Series-with-Python/tree/main/chapter10>
- TRANSFORMERS IN TIME-SERIES ANALYSIS: A TUTORIAL (<https://arxiv.org/pdf/2205.01138>)
- <https://www.geeksforgeeks.org/deep-learning/transformer-for-time-series-forecasting/>

