Microsoft

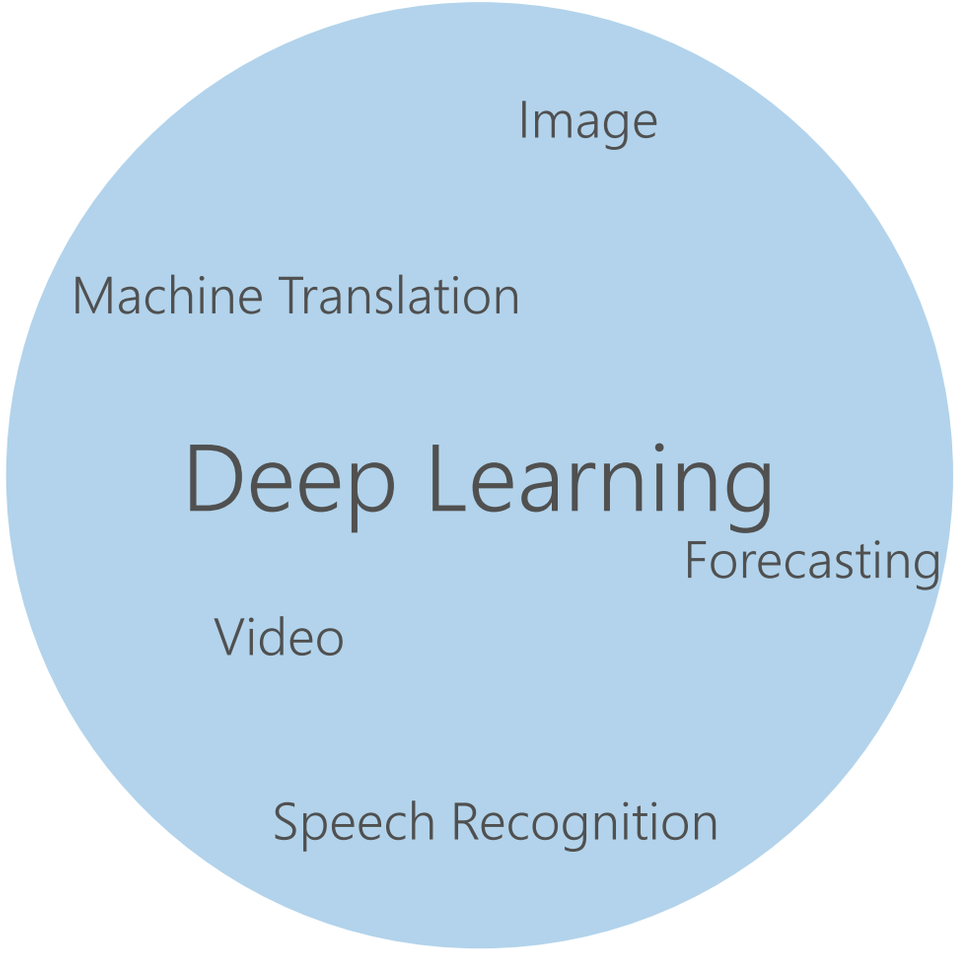
Deep Learning for Time Series Forecasting

Yijing Chen, Angus Taylor,
Vanja Paunic, Dmitry Pechyoni

Complete the pre-requisites: aka.ms/dlts/pr

Tutorial Introduction

This tutorial is about the intersection of DL and Time Series



Time Series & Time Series Forecasting

- *Time series* are observations taken sequentially in time

- *Time series forecasting* predicts future 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
Future	2018-06-13	?
	2018-06-14	?
	2018-06-15	?

Questions to ask before building forecast model

- Can it be forecast?
  - How well we understand the factors that contribute to it?
  - How much data are available and are we able to gather it?
- How far in future (*horizon*) we want to forecast?
- At what temporal **frequency** are forecasts required?

.....

.....

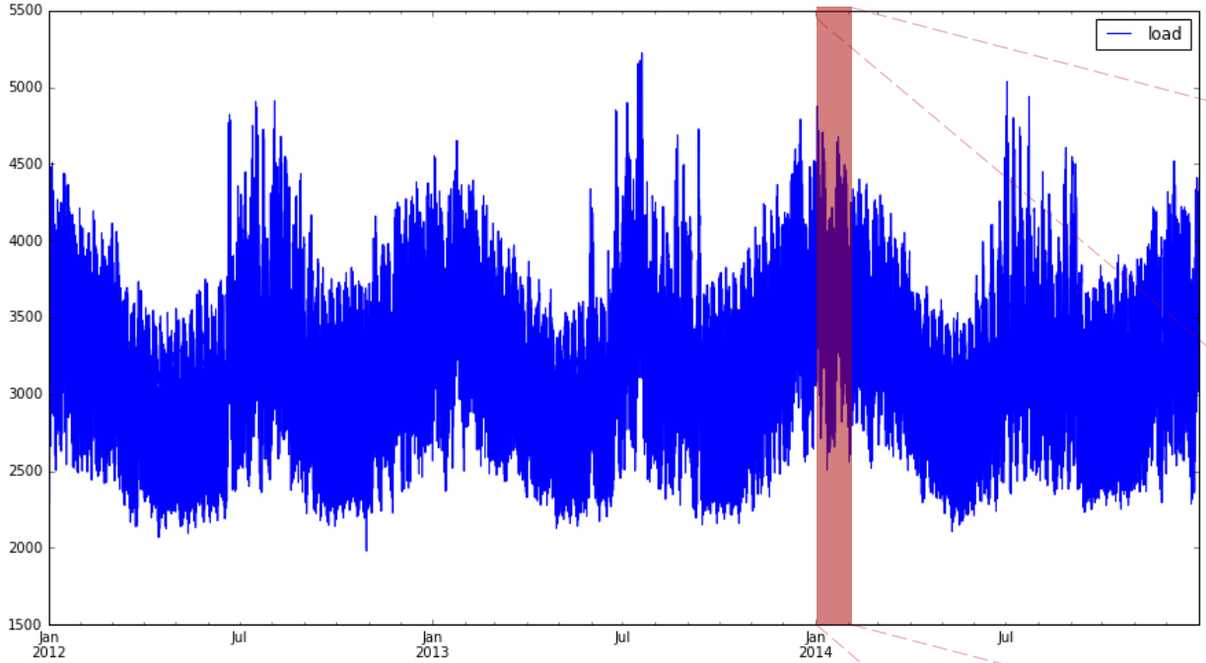
- What technique/model should I use?

Scenario: energy load forecasting

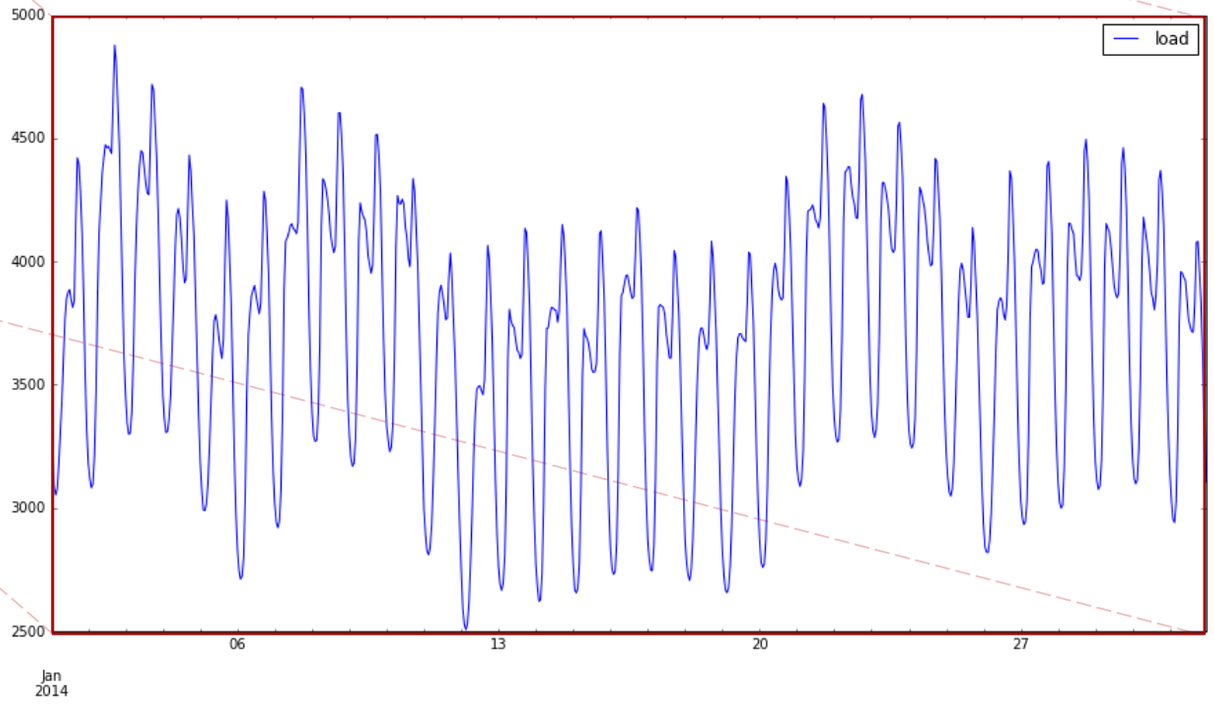
- Energy grid operators keep the supply and demand of electricity on power grids in balance

- Data in this example was used in the Global Energy Forecasting Competition in 2014 - GEFCom2014

New England ISO data



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



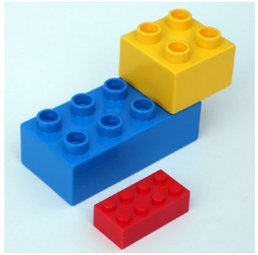
Tao Hong, Pierre Pinson, Shu Fan, Hamidreza Zareipour, Alberto Troccoli and Rob J. Hyndman, "Probabilistic energy forecasting: Global Energy Forecasting Competition 2014 and beyond", International Journal of Forecasting, vol.32, no.3, pp 896-913, July-September, 2016.

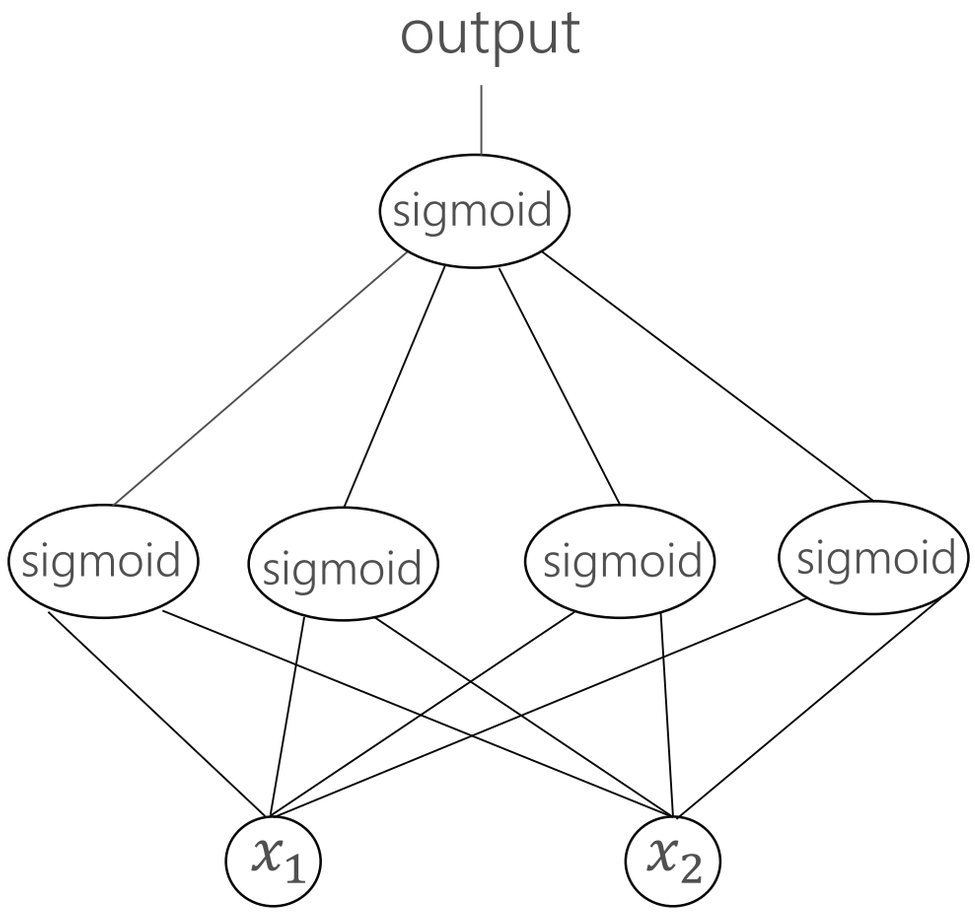
Why deep learning models?

- Deep learning model has been shown perform well in many scenarios
  - 2014 Global Energy Forecasting Competition (link)
  - 2016 CIF International Time Series Competition (link)
  - 2017 Web Traffic Time Series Forecasting (link)
  - 2018 Corporación Favorita Grocery Sales Forecasting (link)
  - 2018 M4-Competition (link)

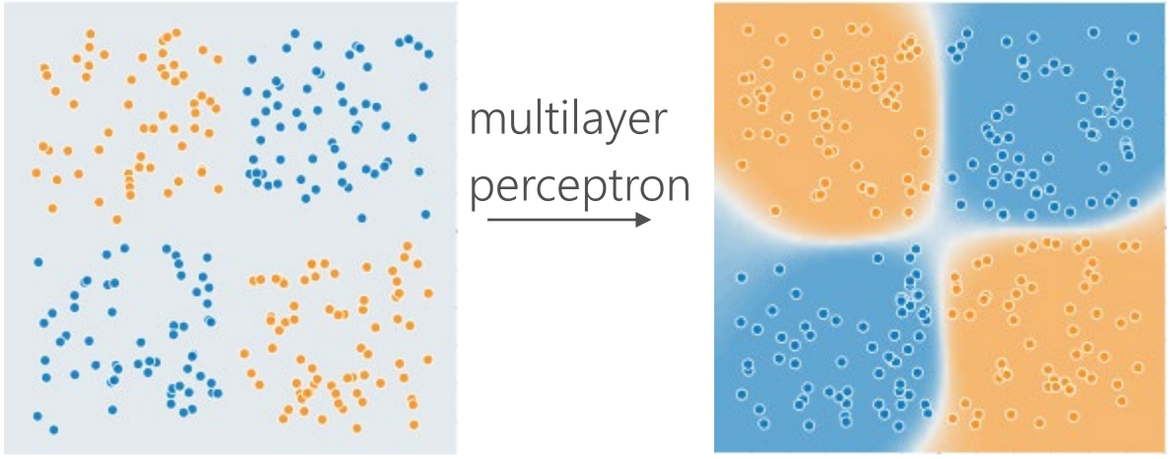
- Non-parametric
- Flexible and expressive
- Easily inject exogenous features into the model
- Learn from large time series datasets

Multilayer Perceptron





$$\hat{y} = \begin{cases} 1 & \text{if output} > \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$



Input

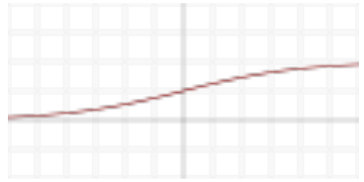
Output

also known as Feed-Forward Neural Network and Deep Neural Network

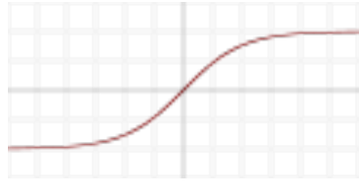
Activation Functions

Applied over $x' = w \cdot x + b$

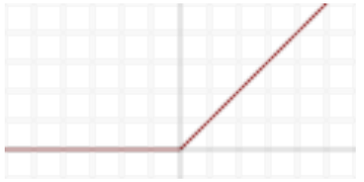
Sigmoid $f(x') = \sigma(x') = \frac{1}{1+e^{-x'}}$



tanh (hyperbolic tangent) $f(x') = \frac{e^{x'} - e^{-x'}}{e^{x'} + e^{-x'}}$



Rectifier linear unit (ReLU) $f(x') = \begin{cases} 0 \text{ for } x' < 0 \\ x' \text{ for } x' \geq 0 \end{cases}$

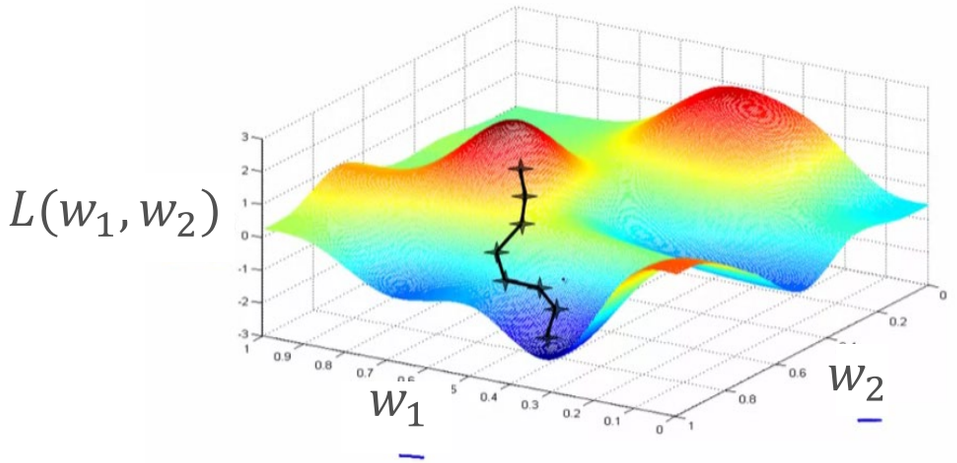


Training: overview

Use training examples to find good weights and biases of the network.

Main components:

- Loss function $L(weights, biases)$ that measures discrepancy between predicted and true values
- Optimization algorithm that finds weights and biases minimizing the loss function



Picture credit: Andrew Ng, Coursera ML class

Examples of loss function

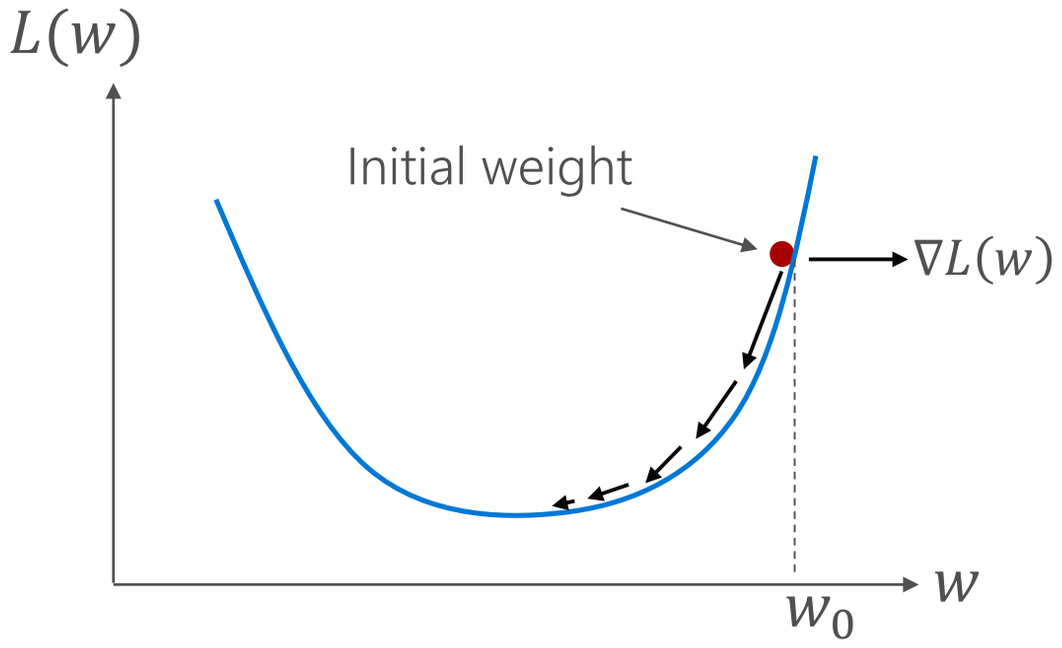
Commonly used loss functions in time series forecasting:

- Mean-squared-error (MSE): $\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$
- Mean Absolute Percentage Error (MAPE): $\frac{100}{N}\sum_{i=1}^{N}\left|\frac{y_i - \hat{y}_i}{y_i}\right|$
- Symmetric MAPE (sMAPE): $\frac{100}{N}\sum_{i=1}^{N}\frac{|y_i - \hat{y}_i|}{(|y_i| + |\hat{y}_i|)/2}$

Optimization algorithm: (Batch) gradient descent

Gradient $\nabla L(w) = \left(\frac{\partial L(w)}{\partial w_1}, \frac{\partial L(w)}{\partial w_2}, \ldots, \frac{\partial L(w)}{\partial w_d}\right)$ - direction of the maximal increase of $L(w)$

1D example: $\nabla L(w) = \left(\frac{\partial L(w)}{\partial w}\right)$



Optimization algorithm

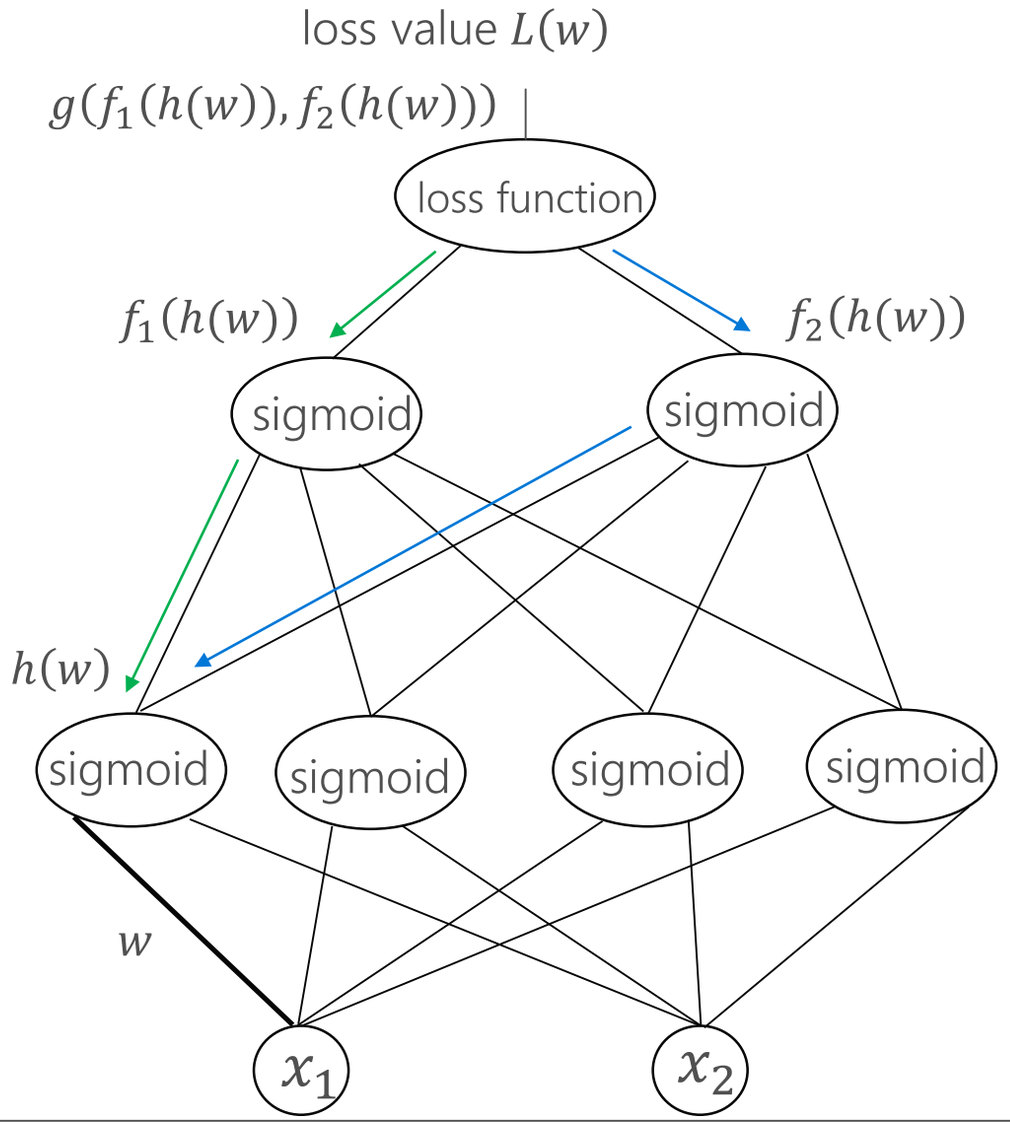
Initialization: $w = w_0$

While stopping criterion not met:

$$w = w - \alpha \cdot \nabla L(w)$$

learning rate

Computation of gradients in NN: Backpropagation



Computation of $L'(w)$

Chain rule

$$L(w) = g\big(f_1(w)\big)$$

$$L'(w) = g'\big(f_1(h(w))\big) \cdot f_1'(h(w)) \cdot h'(w)$$

$$L'(w) = g'\big(f_2(h(w))\big) \cdot f_2'(h(w)) \cdot h'(w)$$

Total derivative

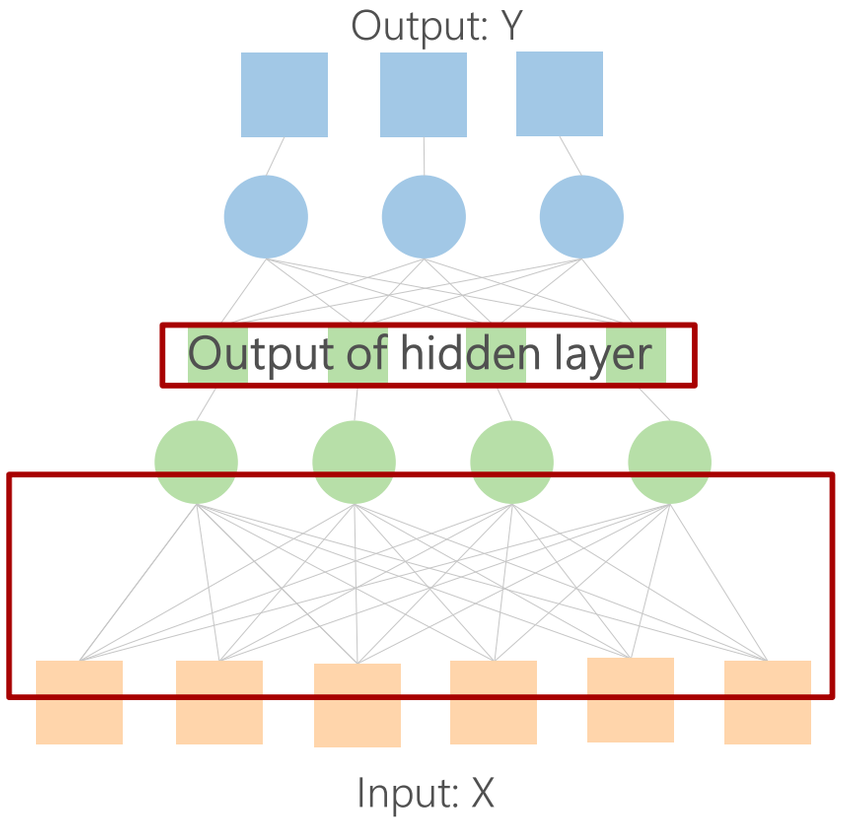
$$L(w) = g\big(f_1(w), f_2(w)\big)$$

$$L'(w) = L'(w) + L'(w)$$

Training tricks

- Early stopping
- Tuning hyperparameters
- Initialization Tutorial from deeplearning.ai
- Weight regularization Regularization for deep learning
- Dropout Dropout: A simple way to prevent neural networks from overfitting, Zoneout: Regularizing RNNs by randomly preserving hidden activations
- Batch normalization Batch normalization: Accelerating deep network training by reducing internal covariate shift, Recurrent batch normalization
- Learning rate decay Learning rate schedules and adaptive learning rate methods for deep learning
- Advanced optimization algorithms An overview of gradient descent optimization algorithms

Fully connected vs convolution layer



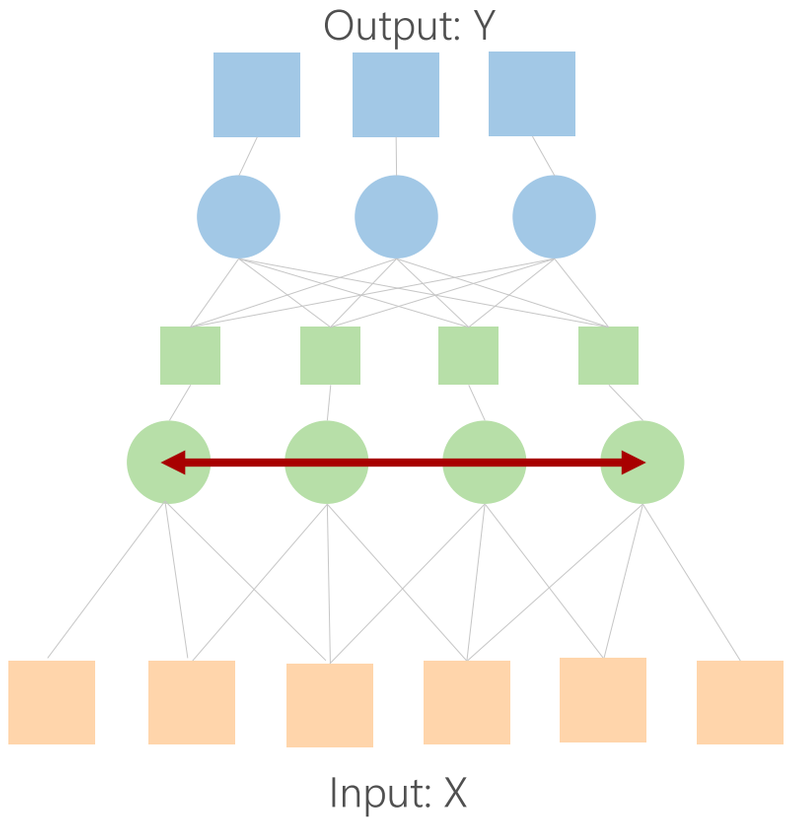
Fully connected:

- units in hidden layer are connected to every unit in previous layer

Fully connected vs convolution layer

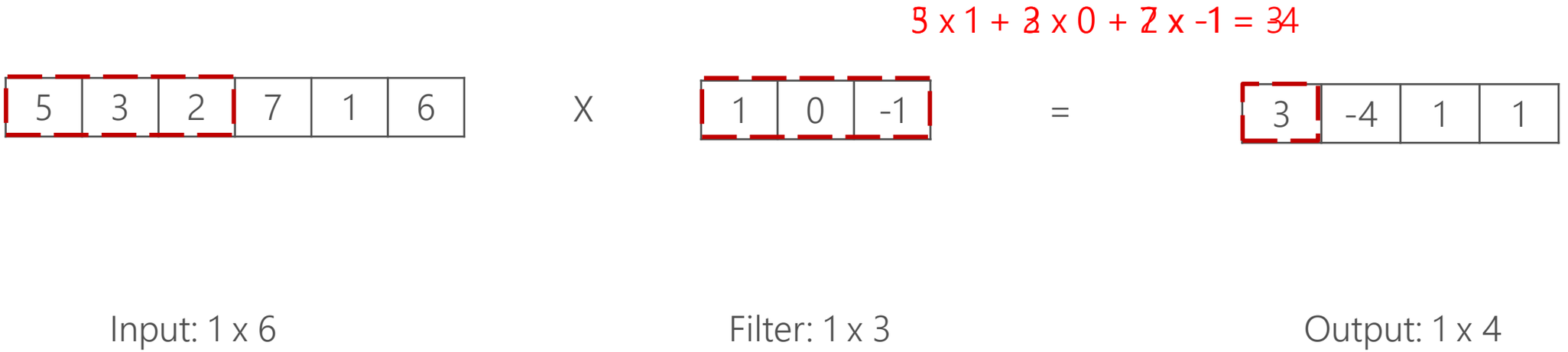
Convolution layer:

- units in hidden layer operate on a field of the input
- weights are shared across input



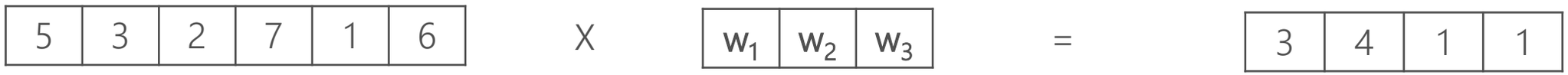
1D Convolutions

- Apply a 1D filter to all elements of a 1D input vector
- Result is the sum of the element-wise product



1D Convolutions

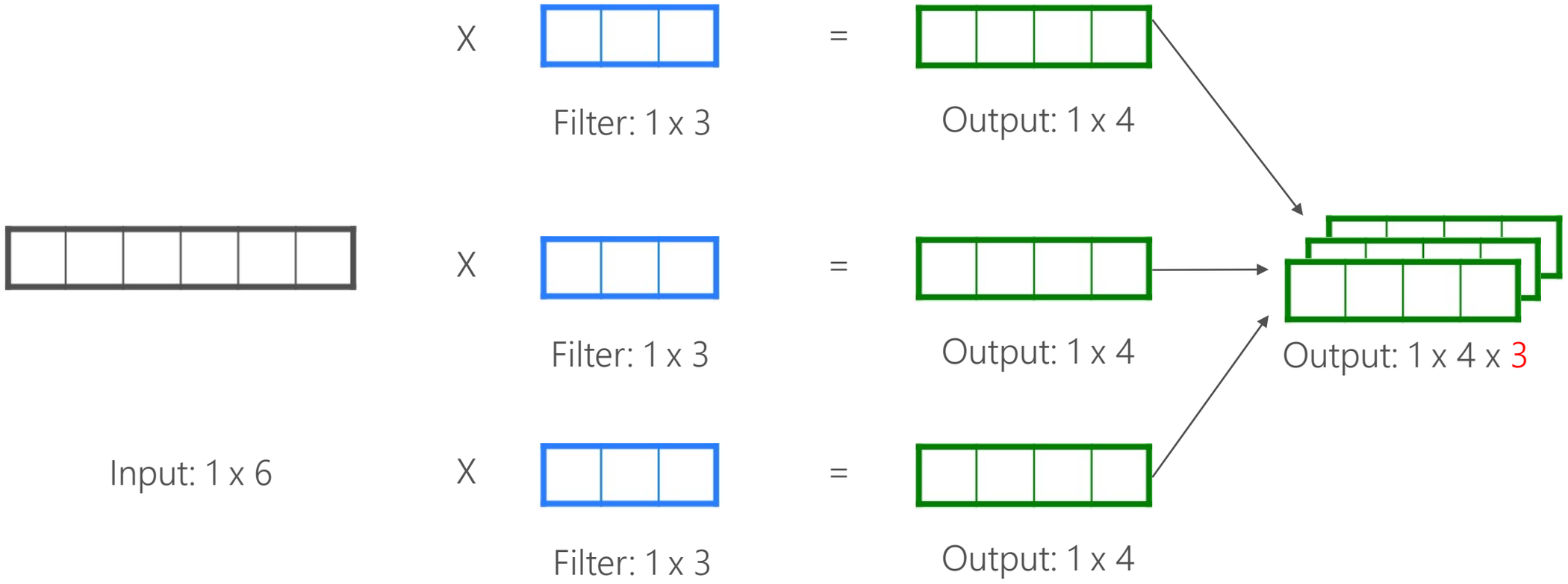
- Filters are trained to detect features in the input sequence
- Features can be detected regardless of where they appear in the input sequence



Input: 1 x 6 Filter: 1 x 3 Output: 1 x 4

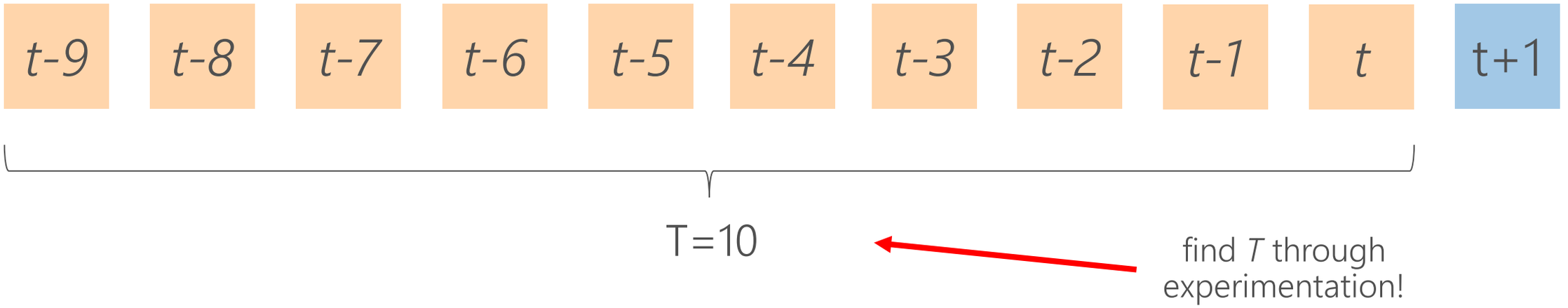
1D Convolutions

- Apply multiple filters to the input data to detect multiple features



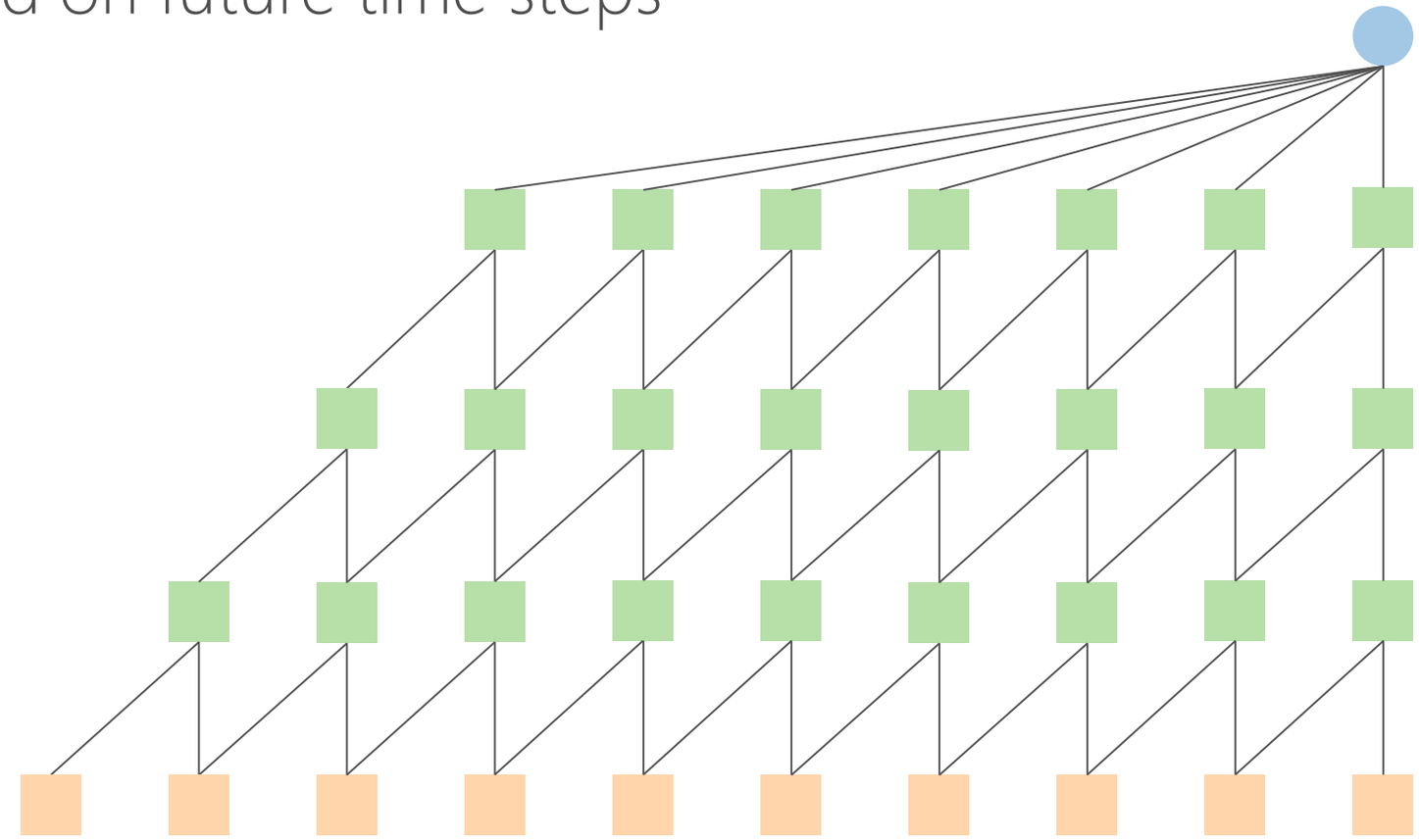
Application to forecasting

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



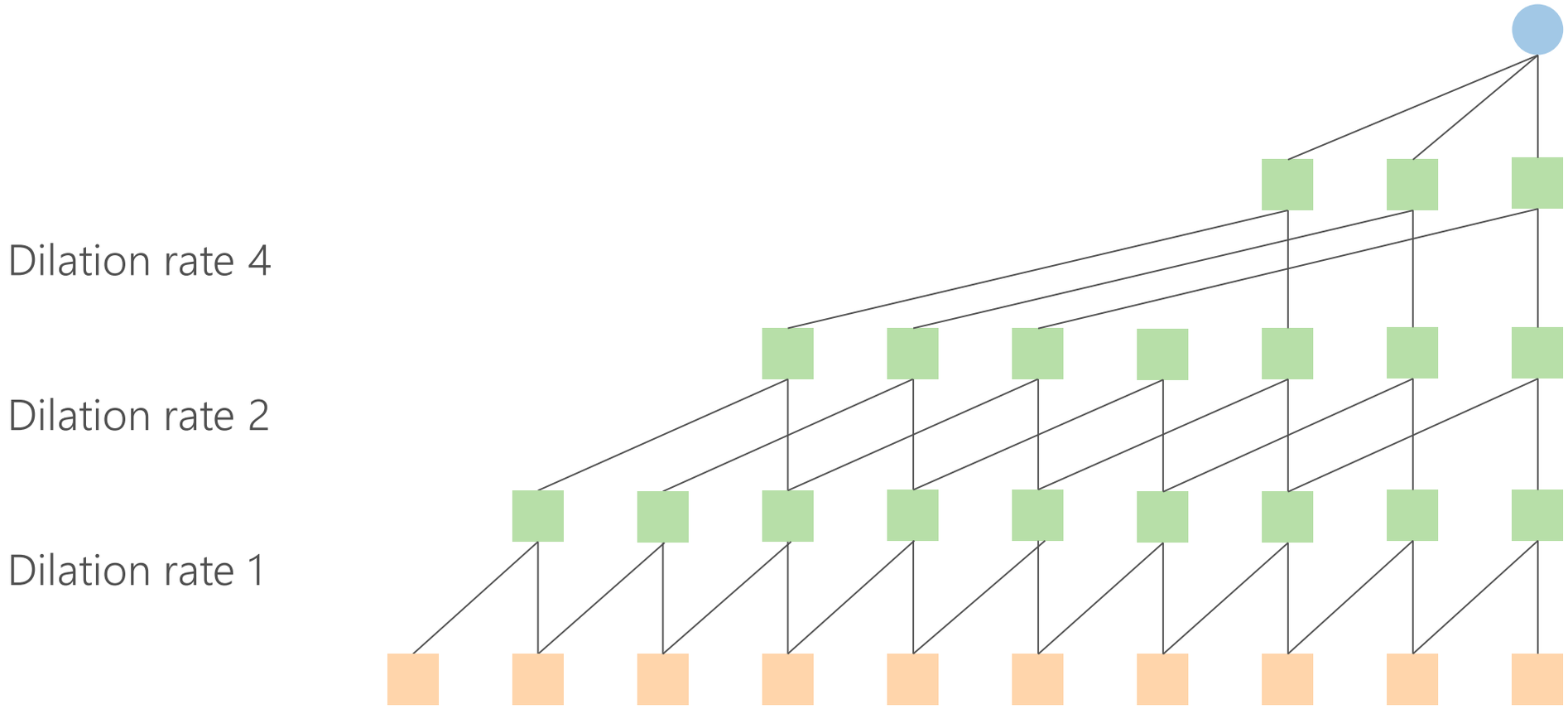
CNNs for forecasting

- Causal convolutions: the output at each time step do not depend on future time steps



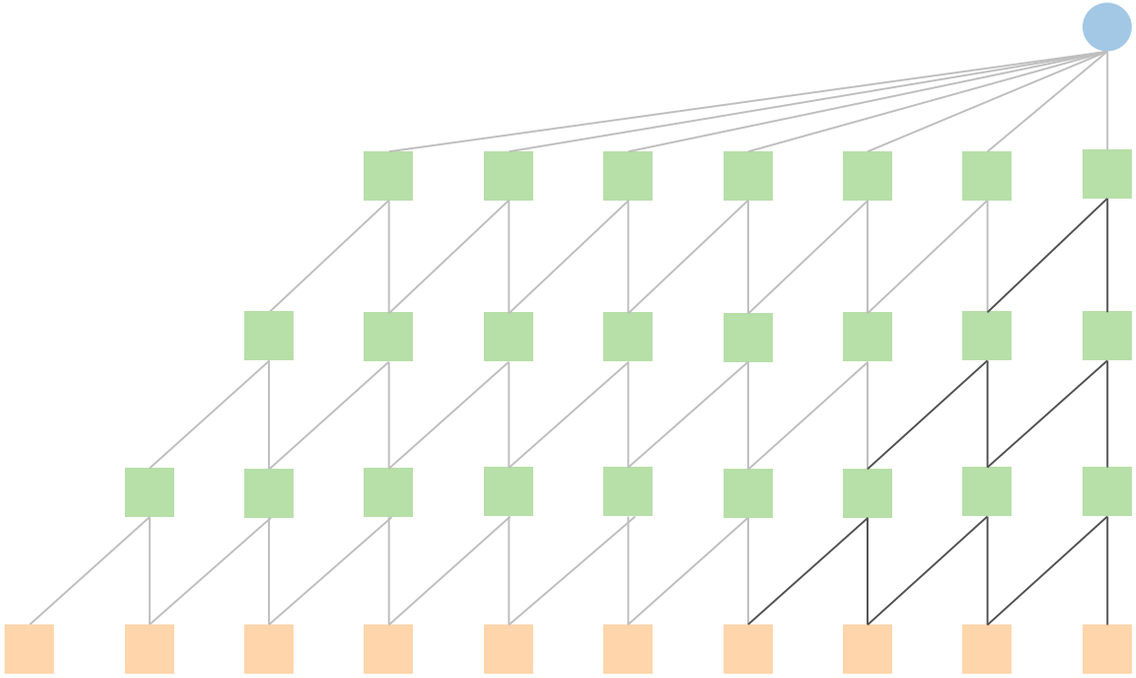
Dilated convolutions

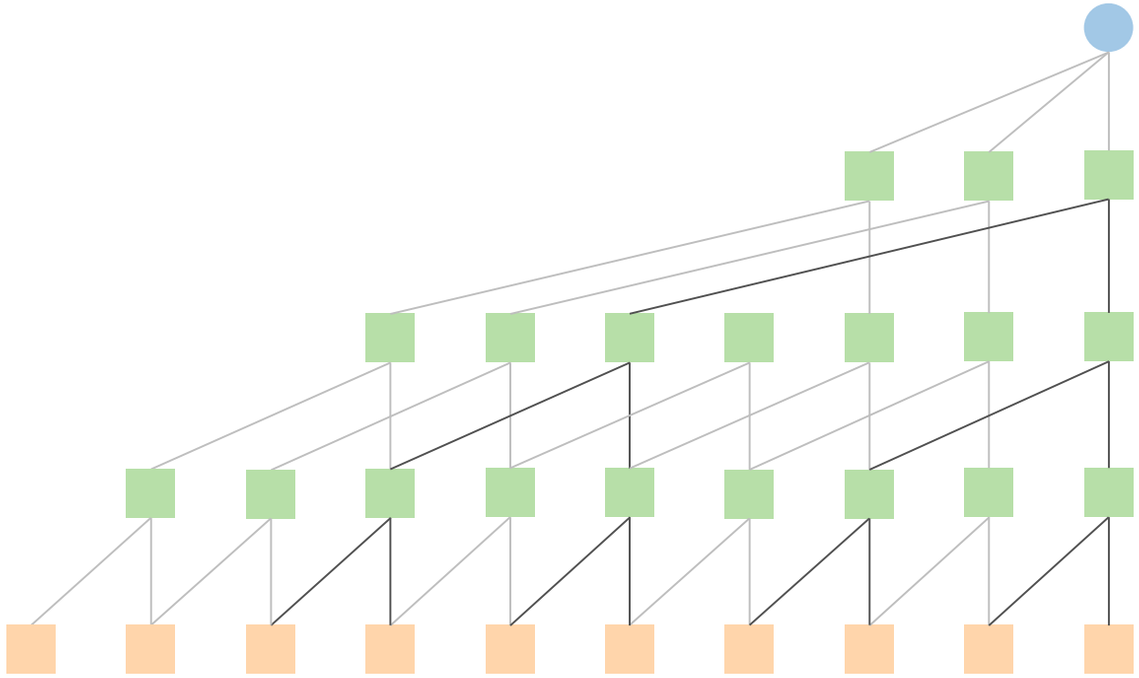
- Skip outputs from previous layers



Why dilated convolutions?

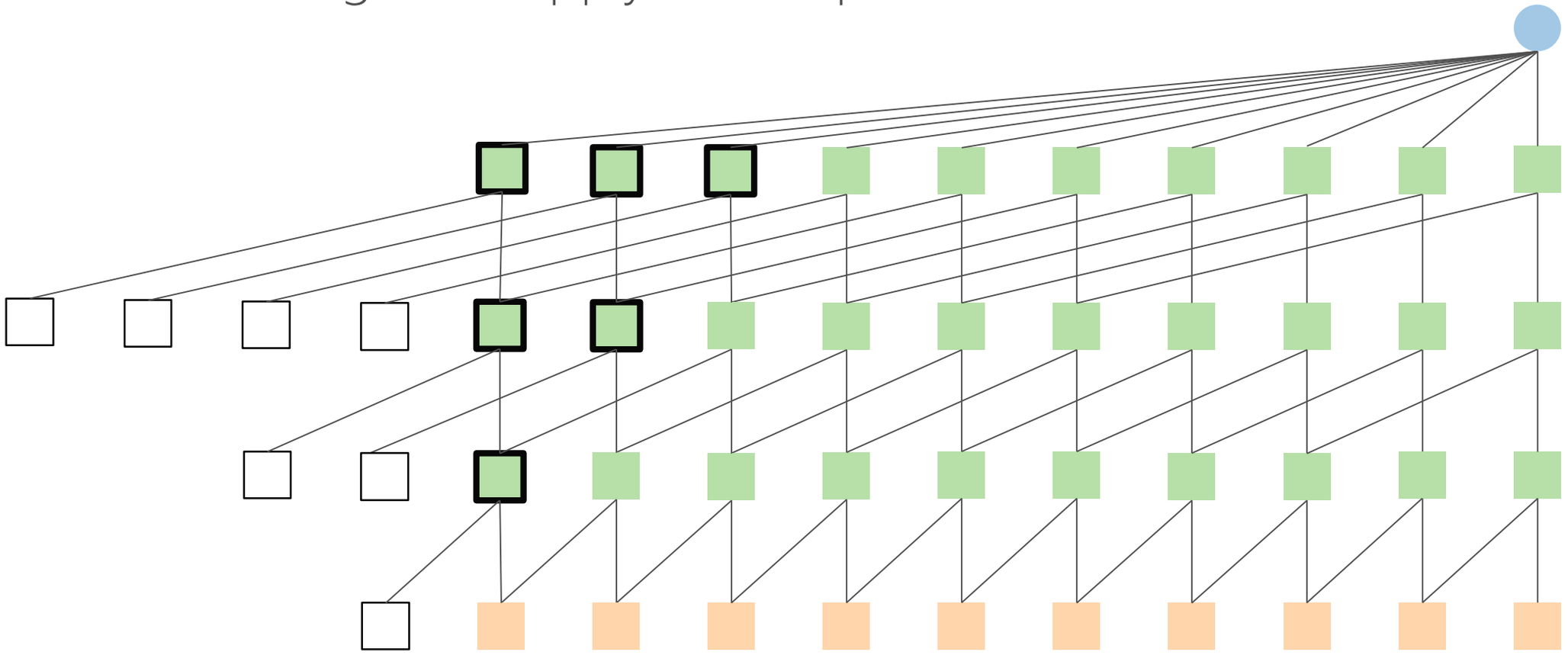
- Normal convolutions = more connections = more weights to train
- Reach information from more distant values in the time series





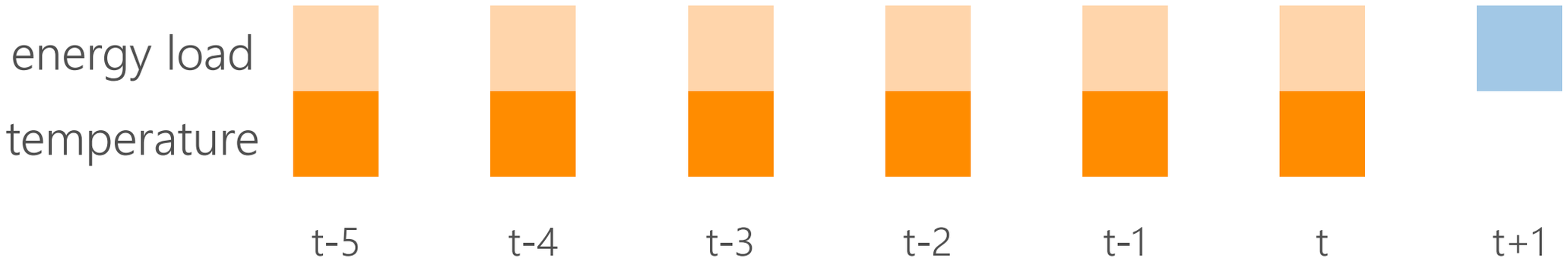
Causal padding

- Padding is necessary to preserve output dimension
- Allows all filter weights to apply to all inputs



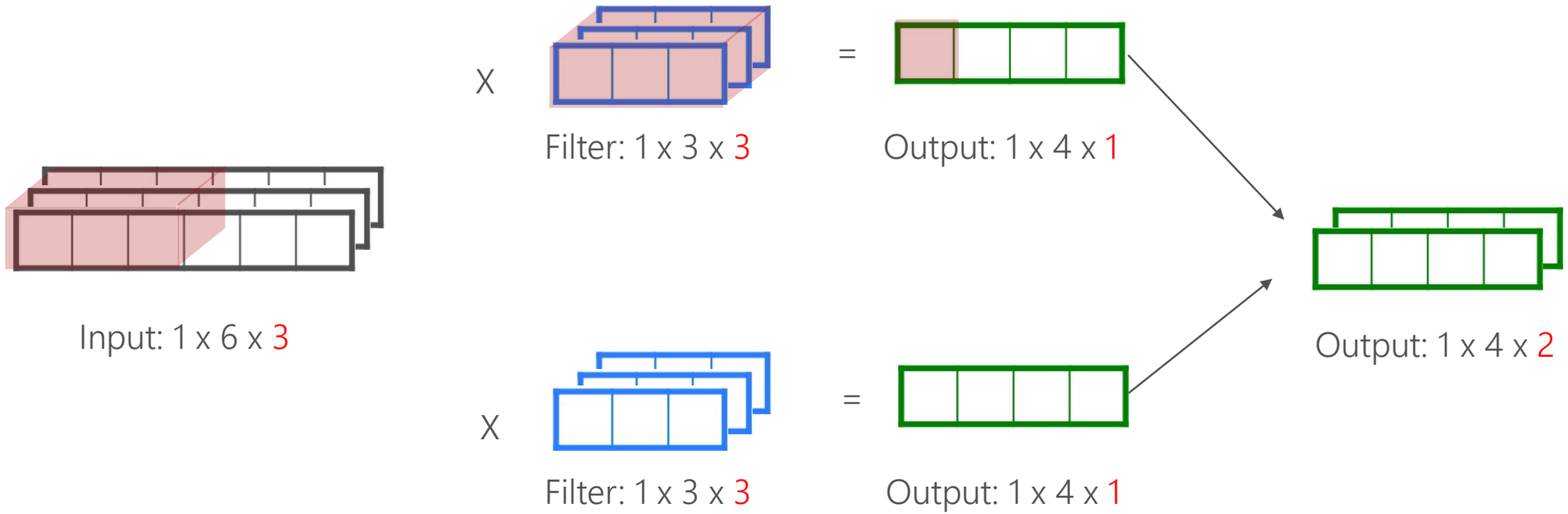
Multivariate time series

- Multiple time series in the input sequence



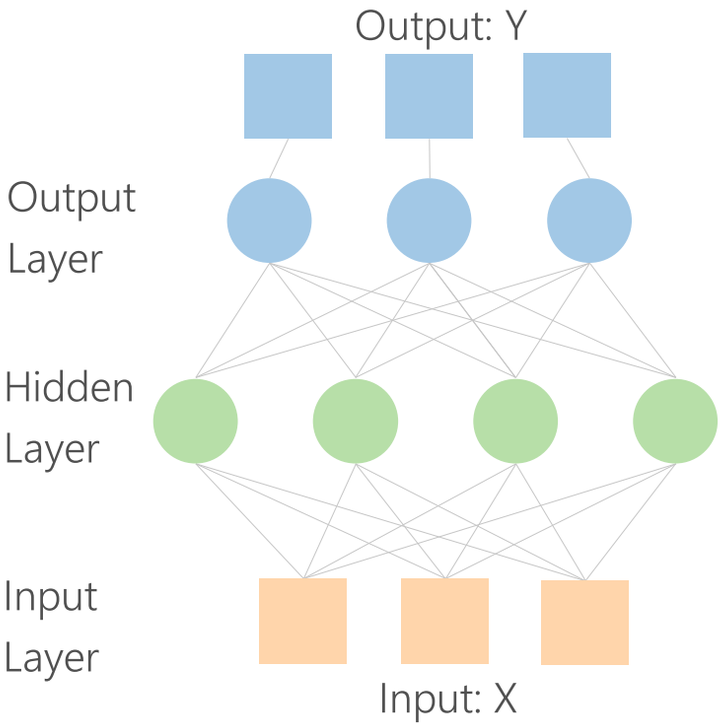
Multivariate time series

- Example: for 3 input time series use 3 channels



What are RNNs?

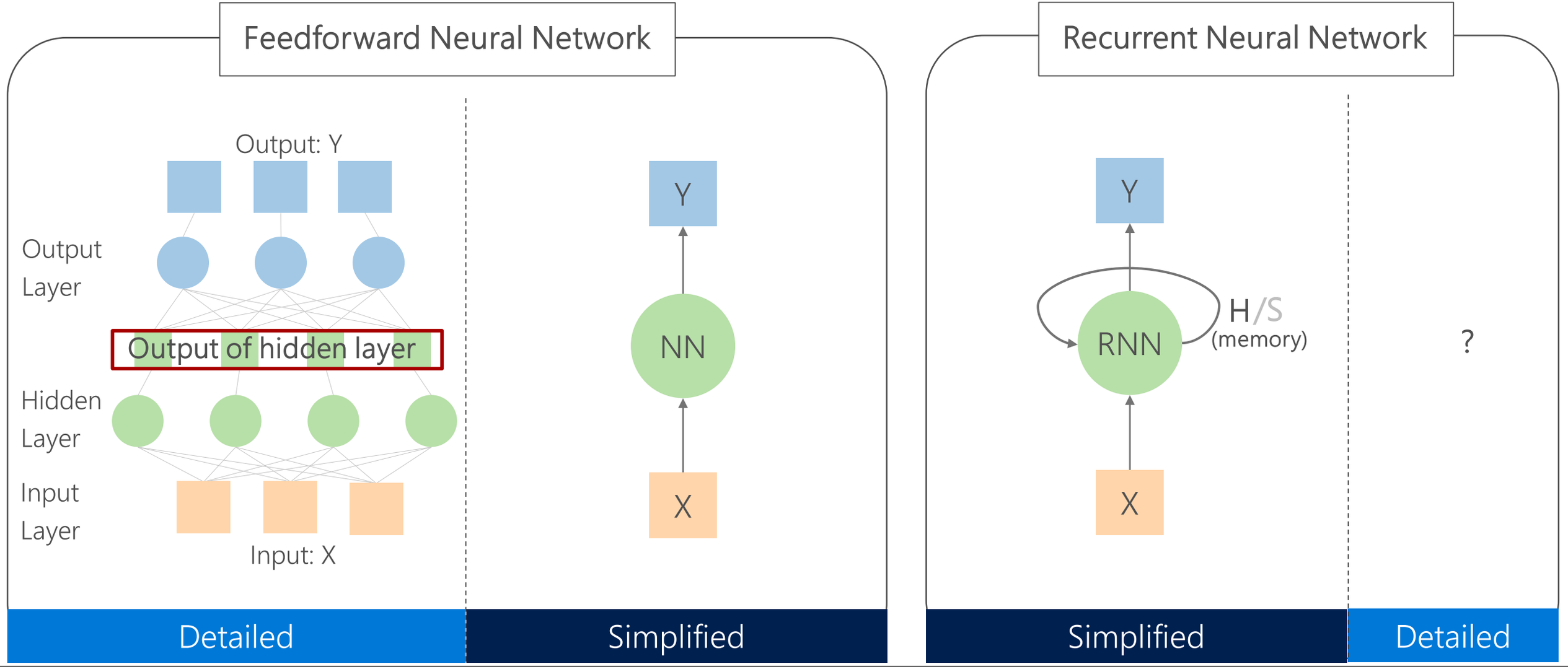
Feedforward Neural Network



Detailed

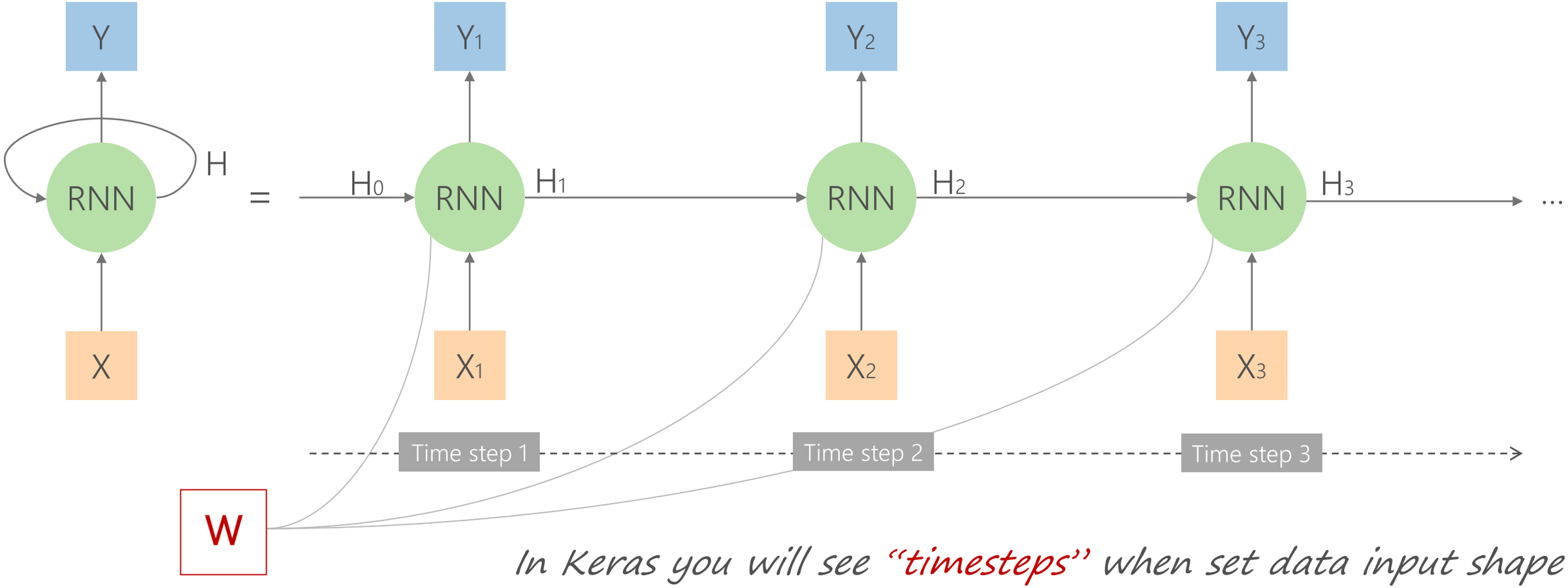
What are RNNs?

RNN has internal hidden state which can be fed back to network



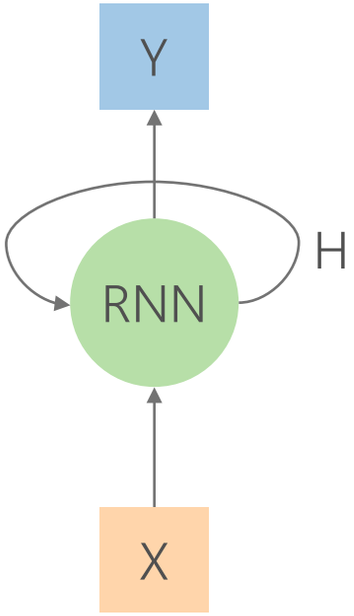
Unrolled RNN

The same weight and bias shared across all the steps



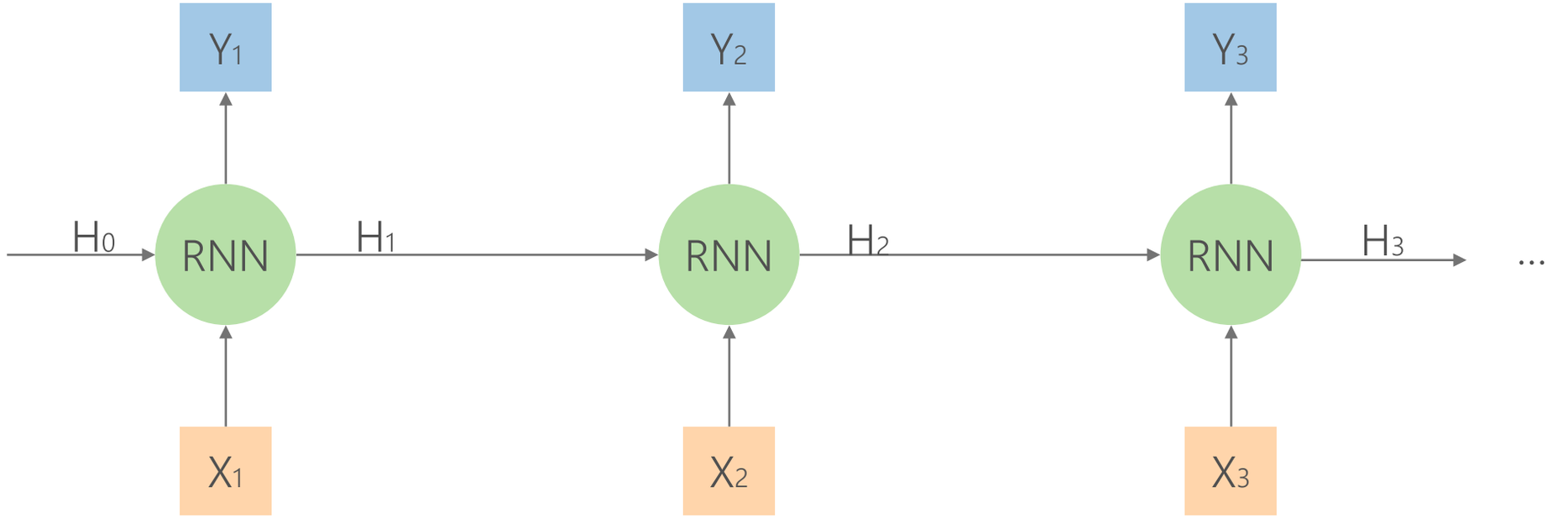
In Keras you will see "timesteps" when set data input shape

Unrolled RNN

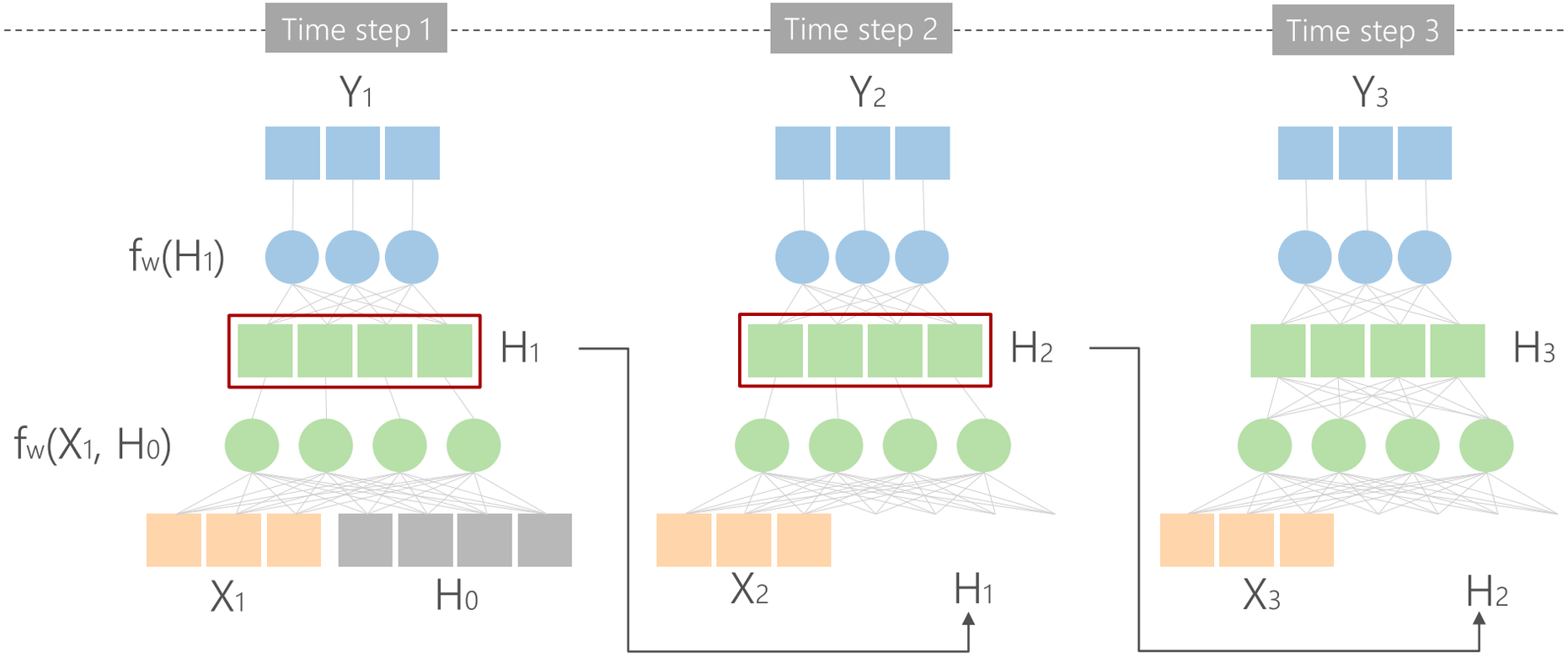


=

Simplified



Detailed

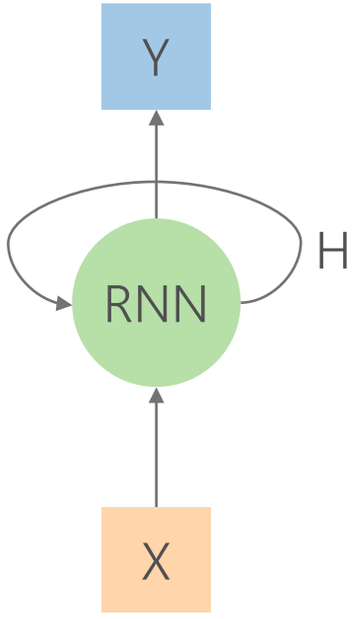


Unrolled RNN

In Keras, the parameter "units" is dimensions of hidden state.
(Think of it as feedforward neural network number of units in hidden layer.)

```
model = Sequential()
model.add(RNN(4, input_shape=(timesteps, data_dim)))
model.add(Dense(3))
```

(timesteps, features)



H

=

Detailed

Time step 1 | Time step 2 | Time step 3

Y_1 Y_2 Y_3

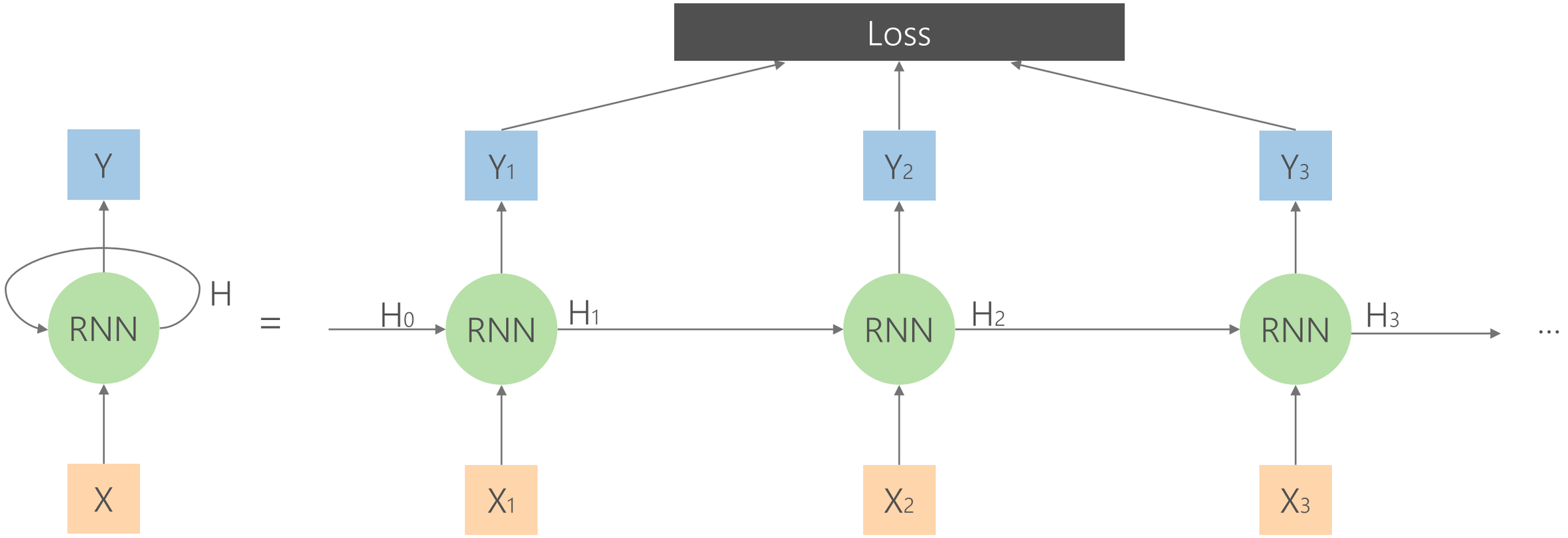
$f_w(H_1)$

H_1 H_2 H_3

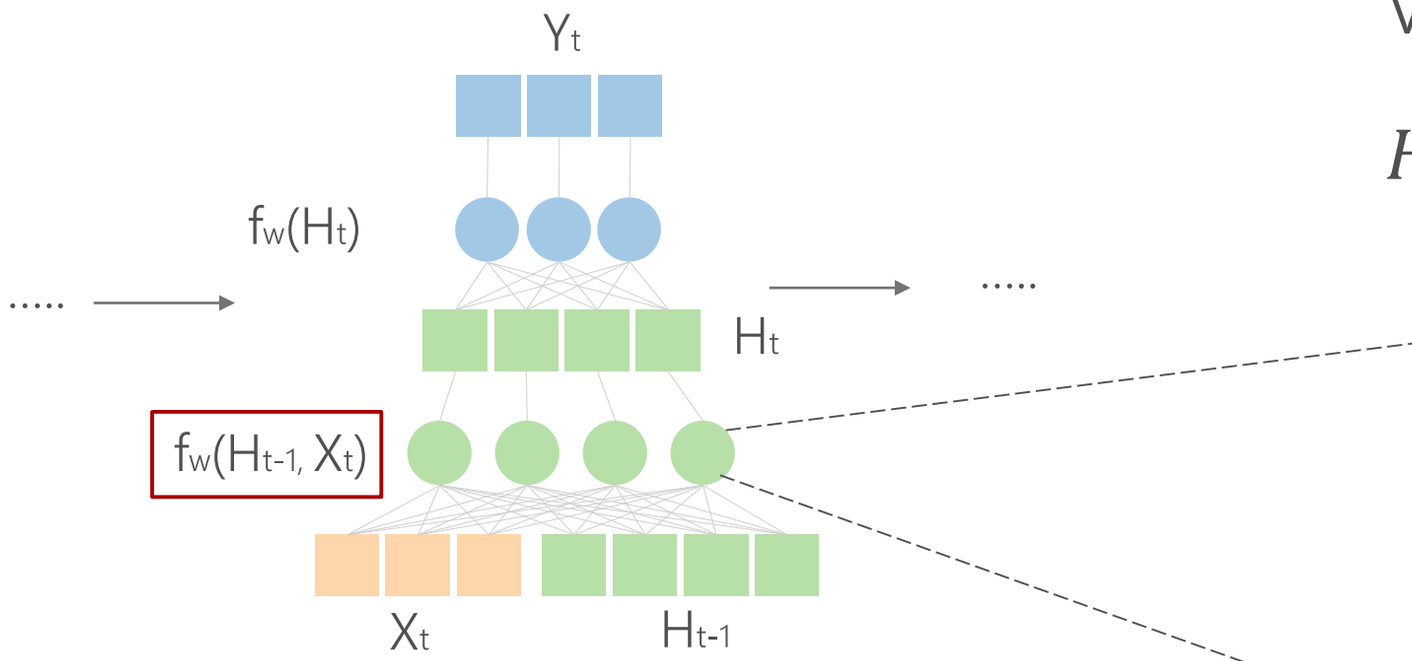
$f_w(X_1, H_0)$

X_1 H_0 X_2 H_1 X_3 H_2

Backpropagation through time (BPTT)

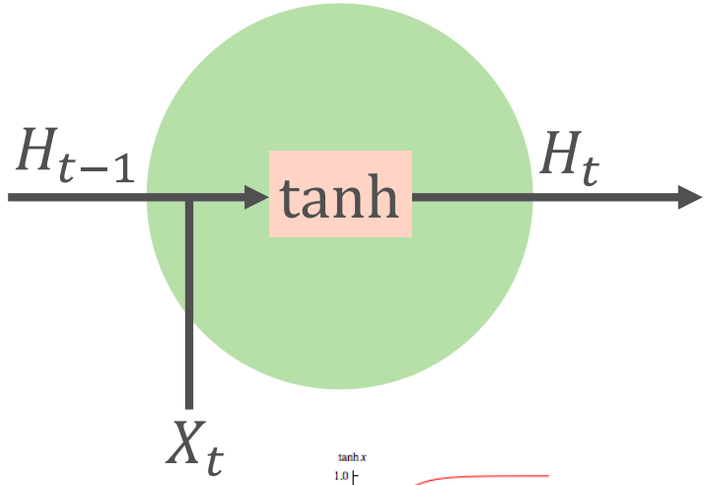


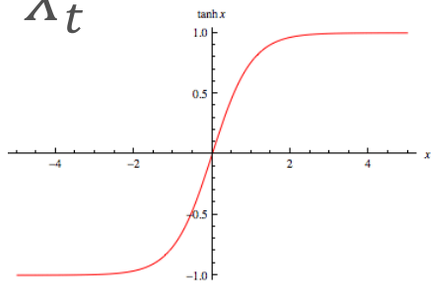
Vanilla RNN



Vanilla RNN:

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



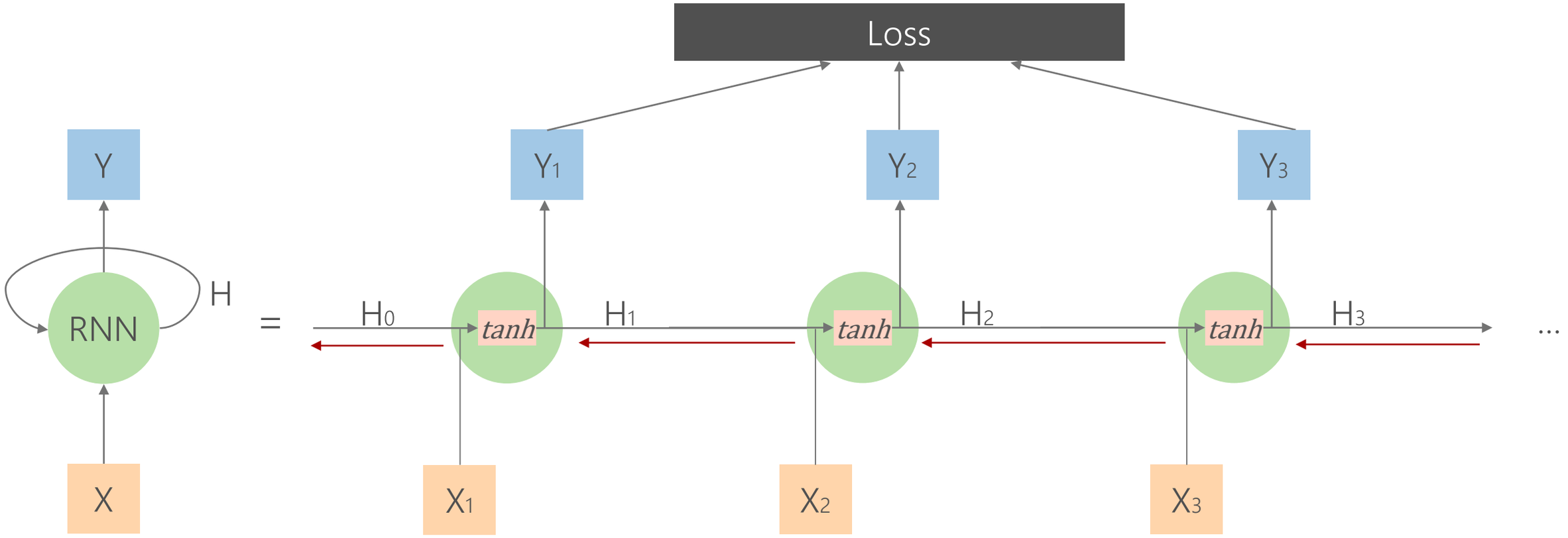


some function with parameter W

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

new state — old state — Input vector at some time step

Vanilla RNN BPTT

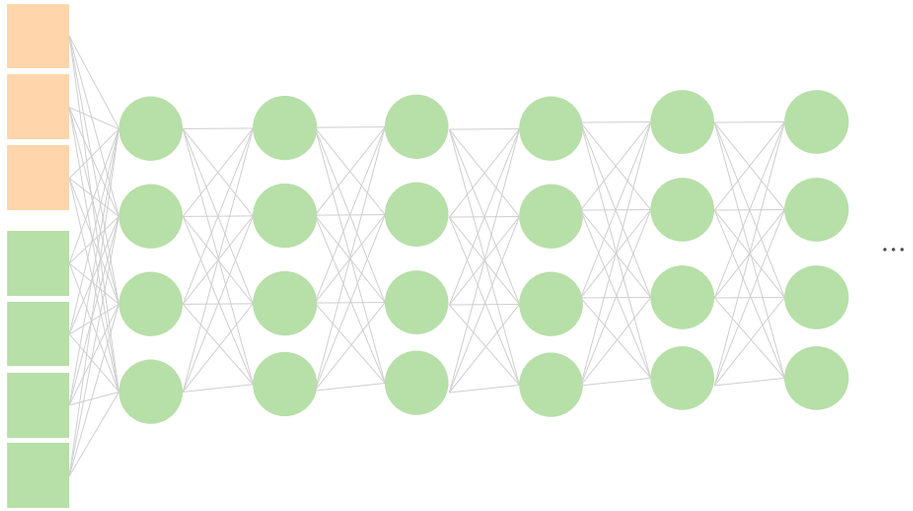


Computing gradient of h_0 involves repeated tanh and many factors of W

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*……)



100 time steps is similar to 100 layers feedforward neural net

Exploding Gradient

- Exploding gradients are obvious. Your gradients will become NaN (not a number) and your program will crash
- Solution: Gradient clipping

  Clip the gradient when it goes higher than a threshold

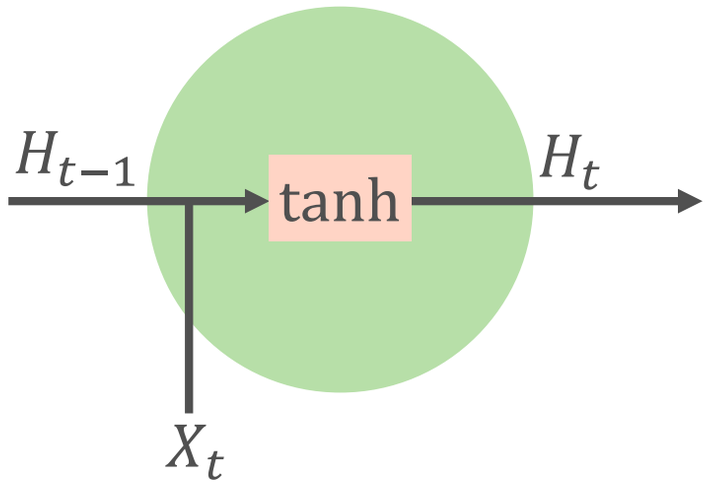
Vanishing Gradient

- 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
  - Change architecture to LSTM or GRU

Gated Recurrent Unit (GRU)

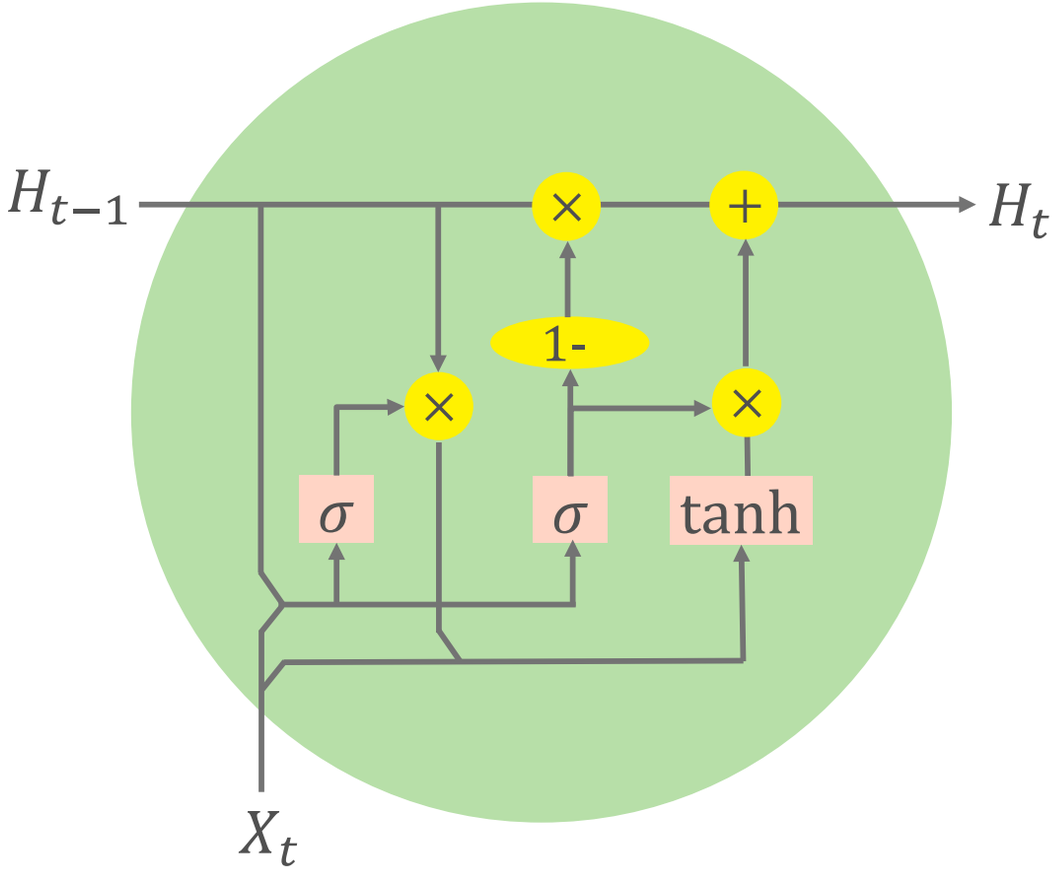
Vanilla RNN:

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

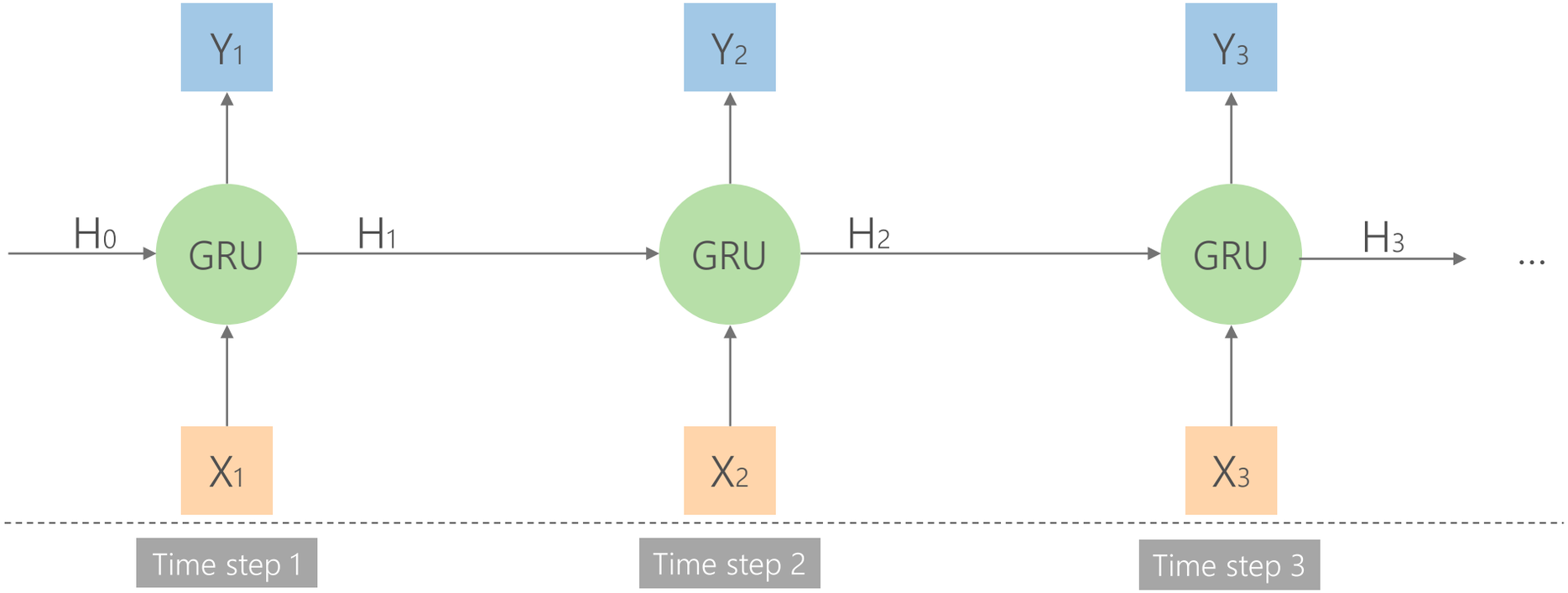


GRU:

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

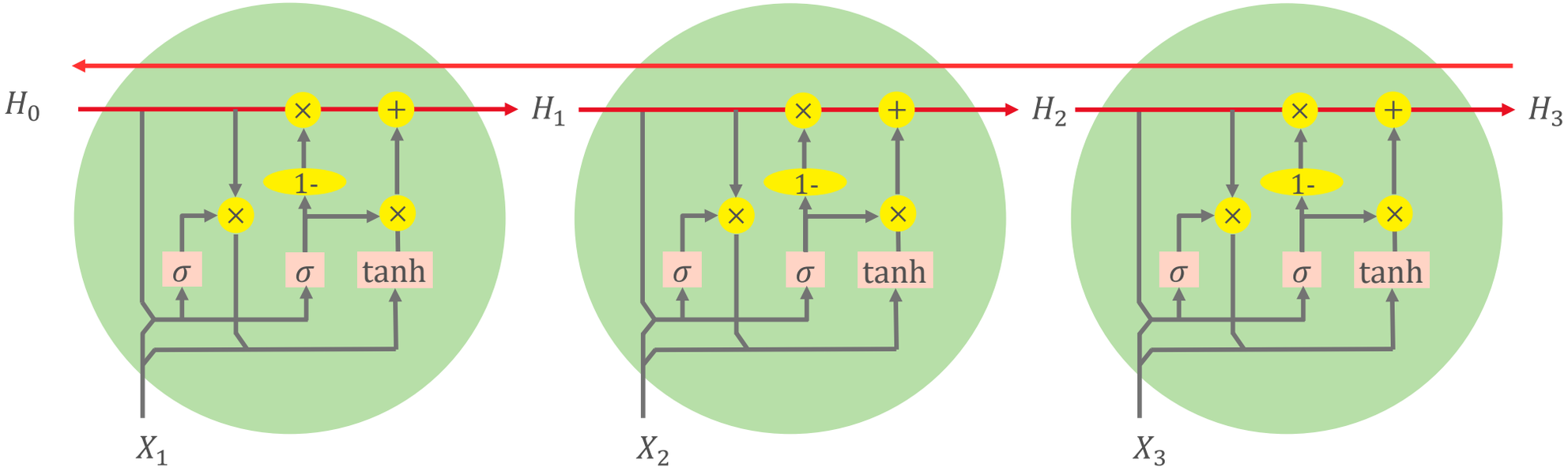


GRU

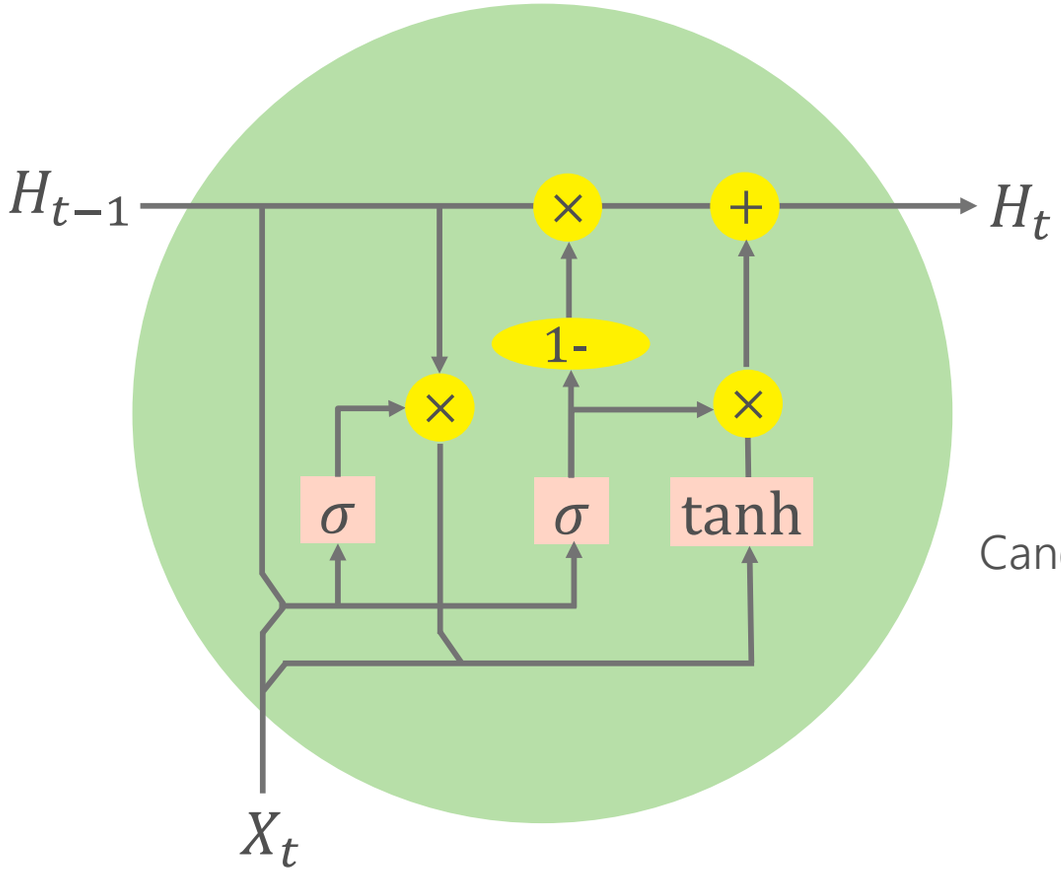


Uninterrupted gradient flow

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



Gated Recurrent Unit (GRU)



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

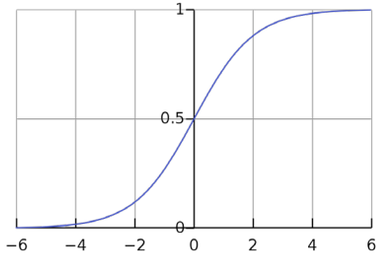
Update gates: $z_t = \sigma(W_z \cdot [H_{t-1}, X_t])$

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

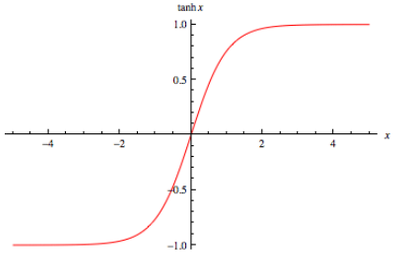
Reset gates: $r_t = \sigma(W_r \cdot [H_{t-1}, X_t])$

GRU [Learning phrase representations using rnn encoderdecoder for statistical machine translation, Cho et al. 2014]

σ (0,1)

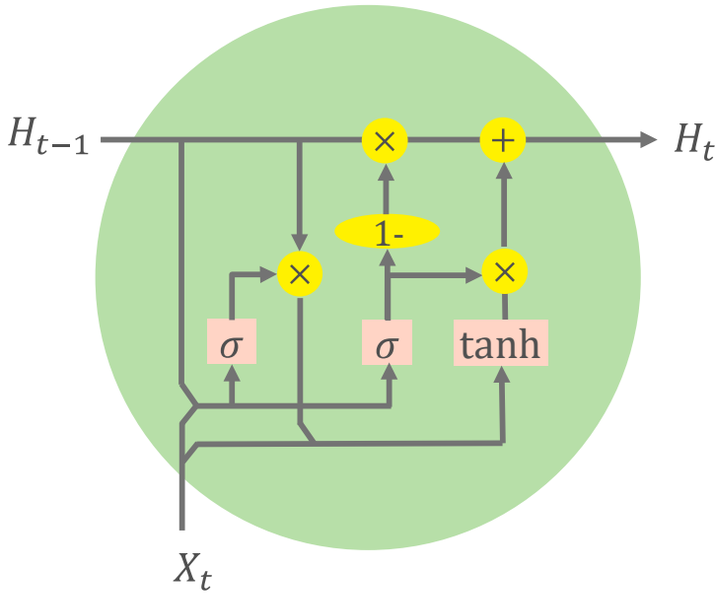


tanh (-1, 1)

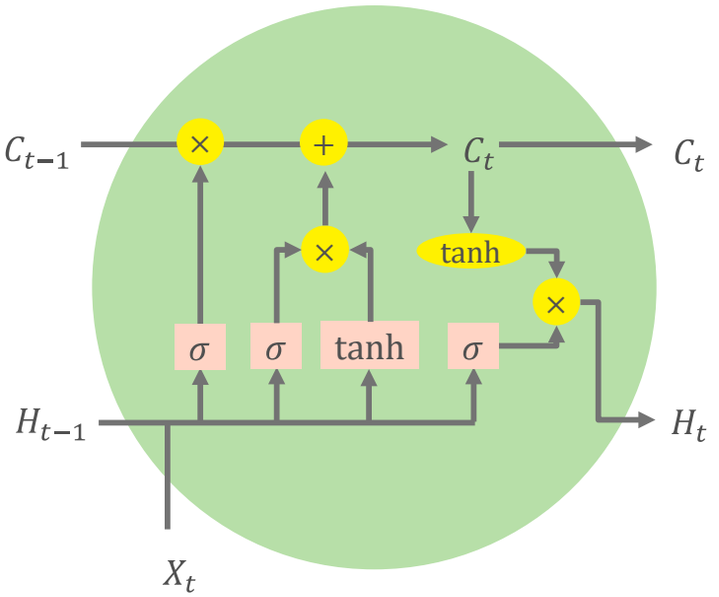


GRU vs LSTM (Long Short Term Memory)

GRU



LSTM



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

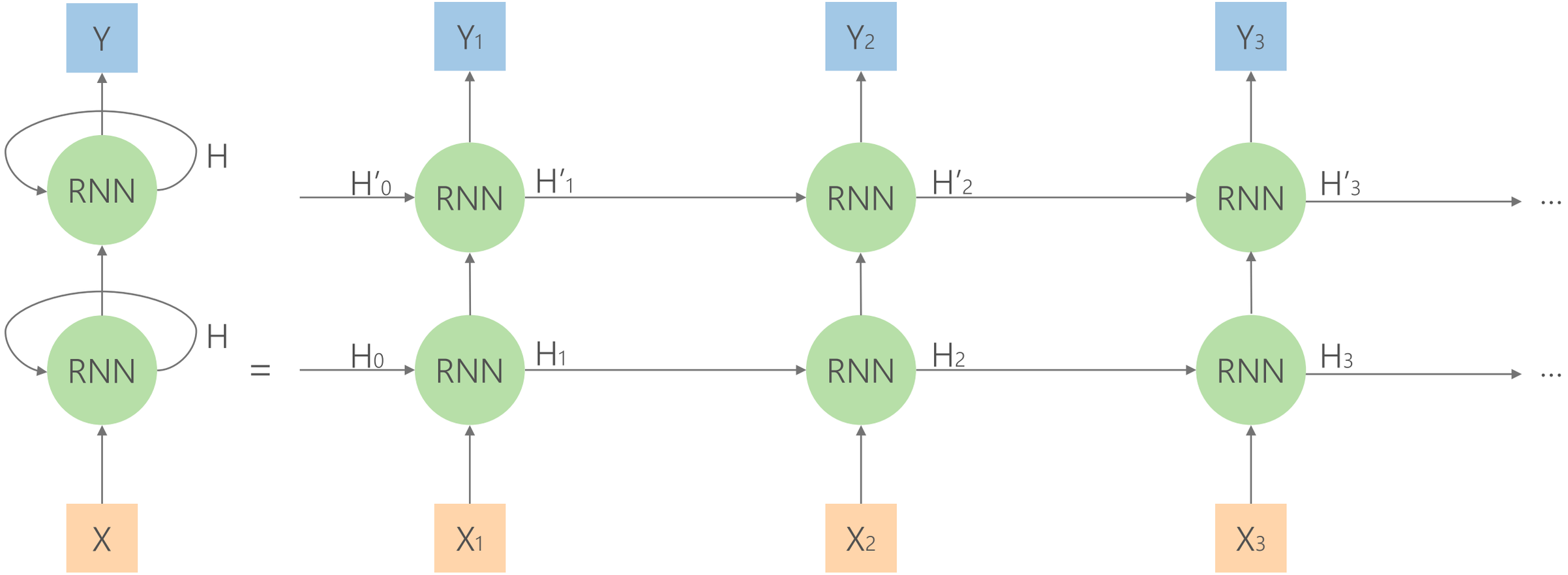
Output gates: $o_t = \sigma(\ W_o \cdot [H_{t-1}, X_t]\ )$

[LSTM: A Search Space Odyssey, Greff et al., 2015]

[Long Short-Term Memory, Hochreiter & Schmidhuber (1997)]

RNN Stacking

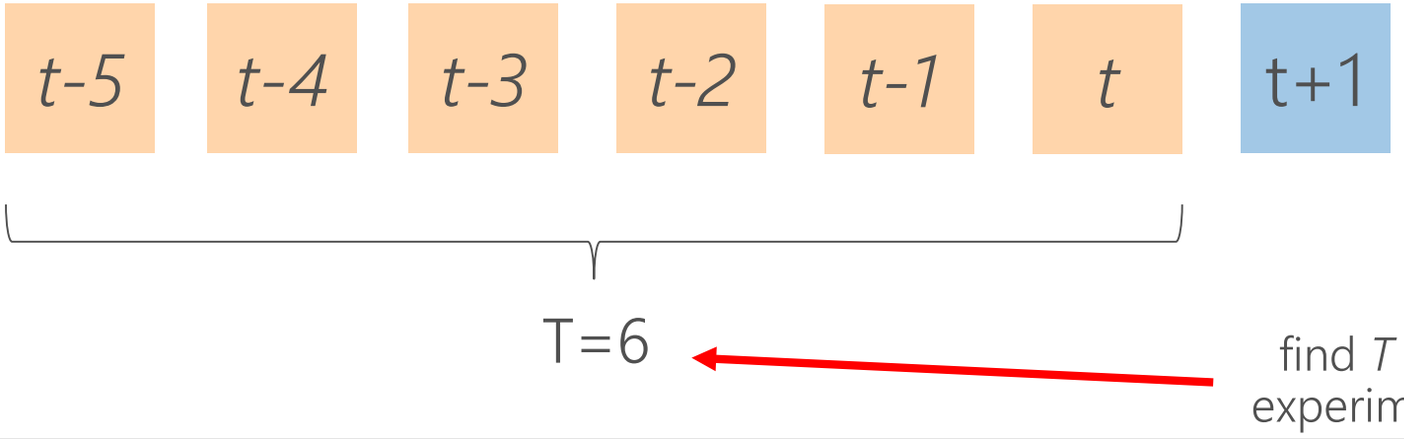
To learn more complex relationships, we can go deep by stacking the cells.




```
model = Sequential()
model.add(RNN(4, return_sequences=True, input_shape=(timesteps, data_dim)))
model.add(RNN(4))
```

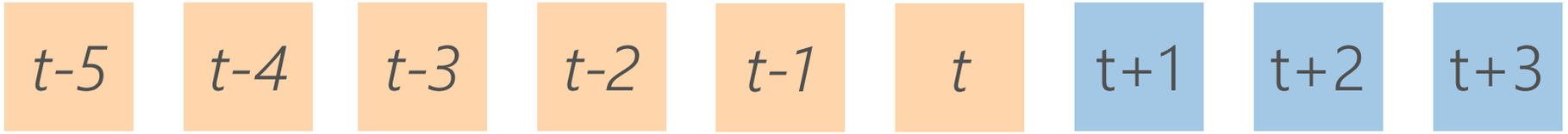
One-step forecast

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

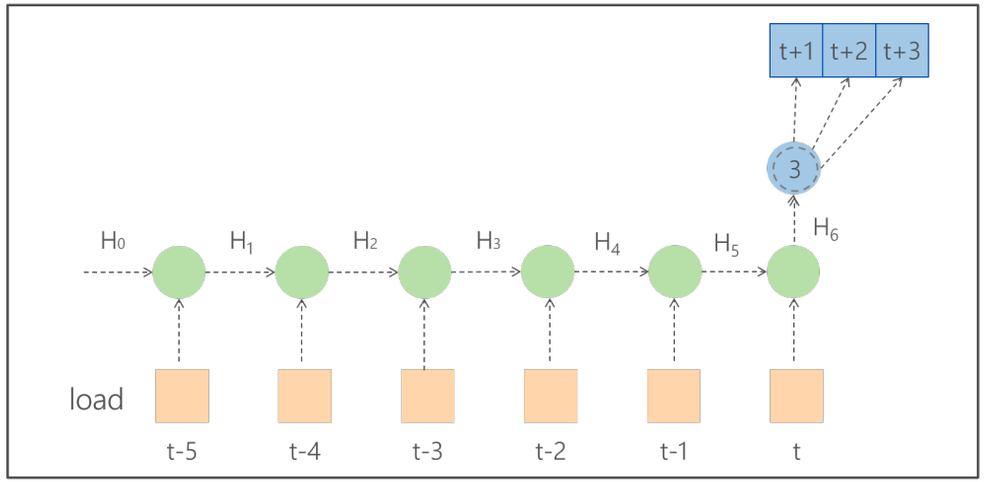


Multi-step forecast

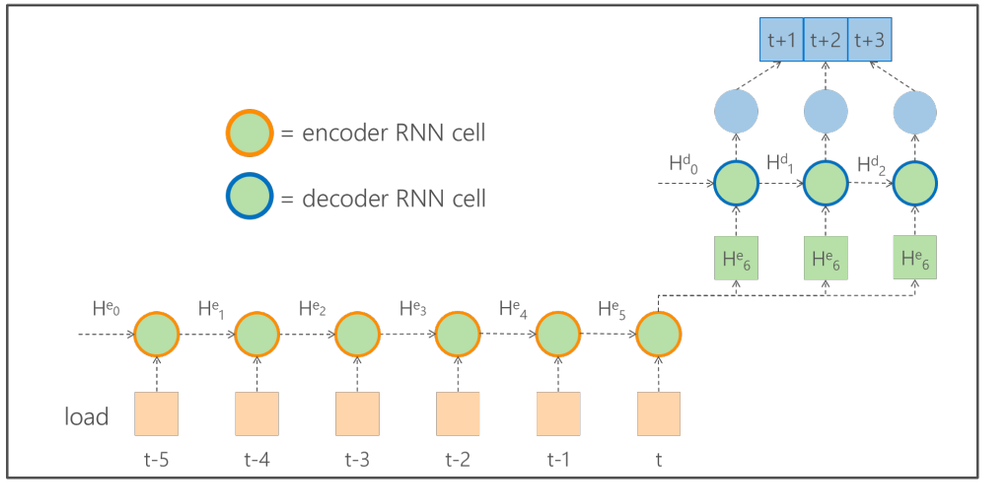
- Assuming we are at time *t* …
- … predict the values at times (*t+1*, … , *t+HORIZON*) …
- … conditional on the previous *T* values of the time series



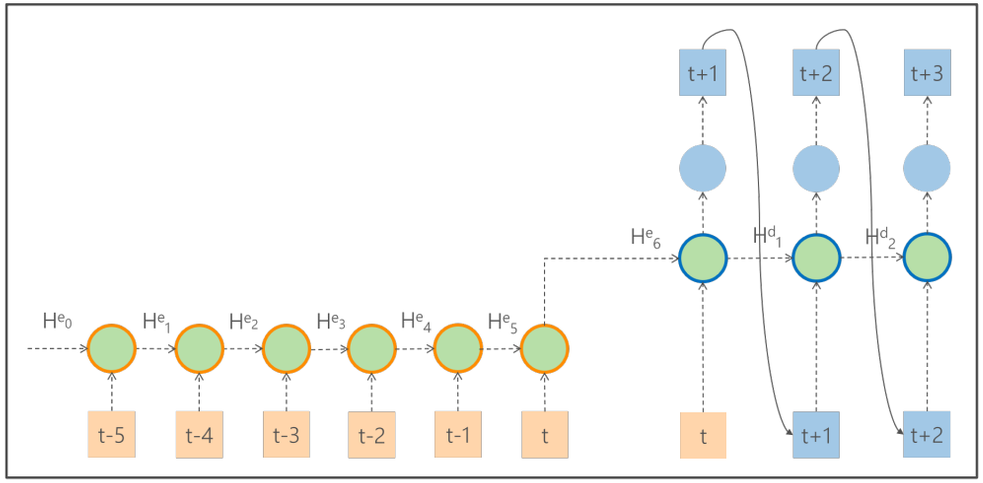
Multi-step forecast



Vector output

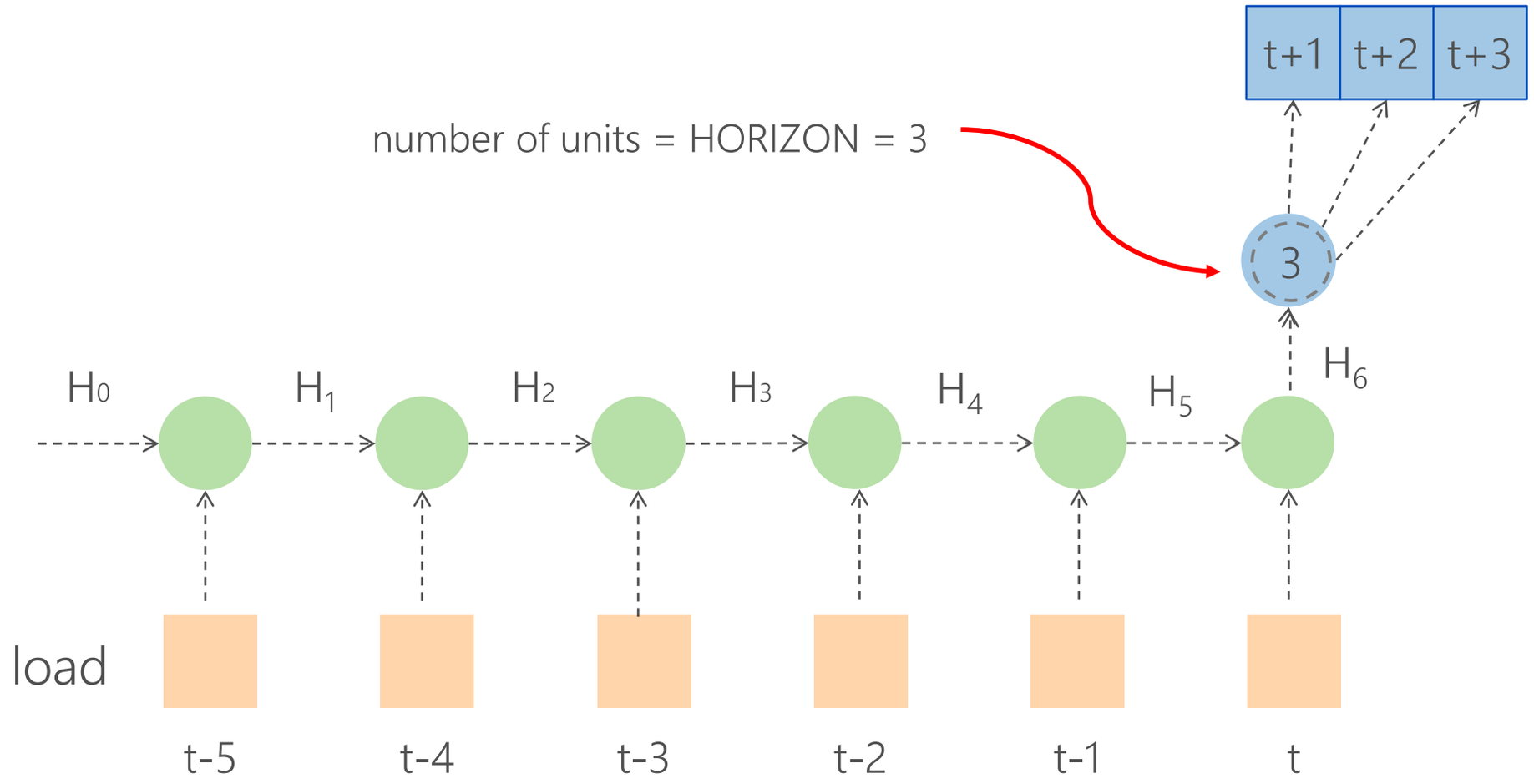


Simple encoder-decoder

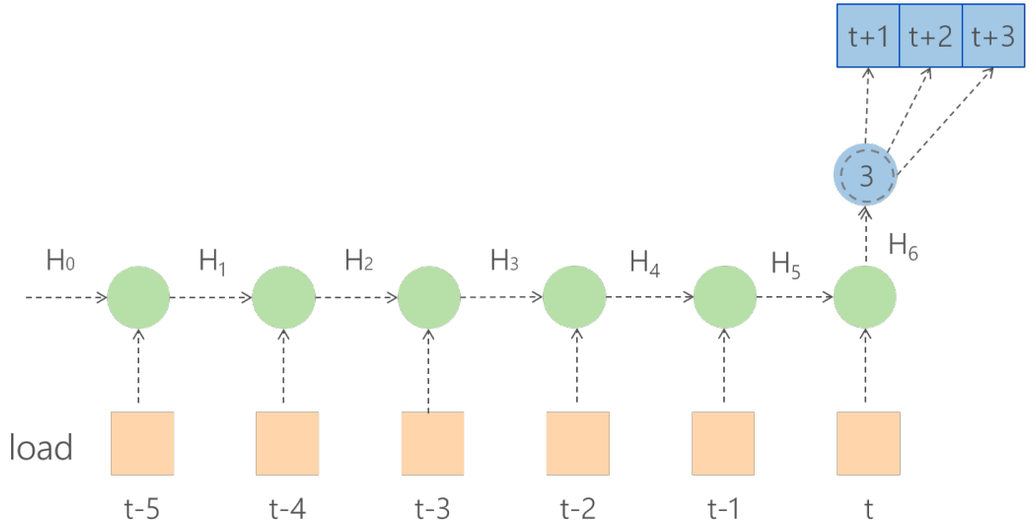


Recursive encoder-decoder

Vector output approach



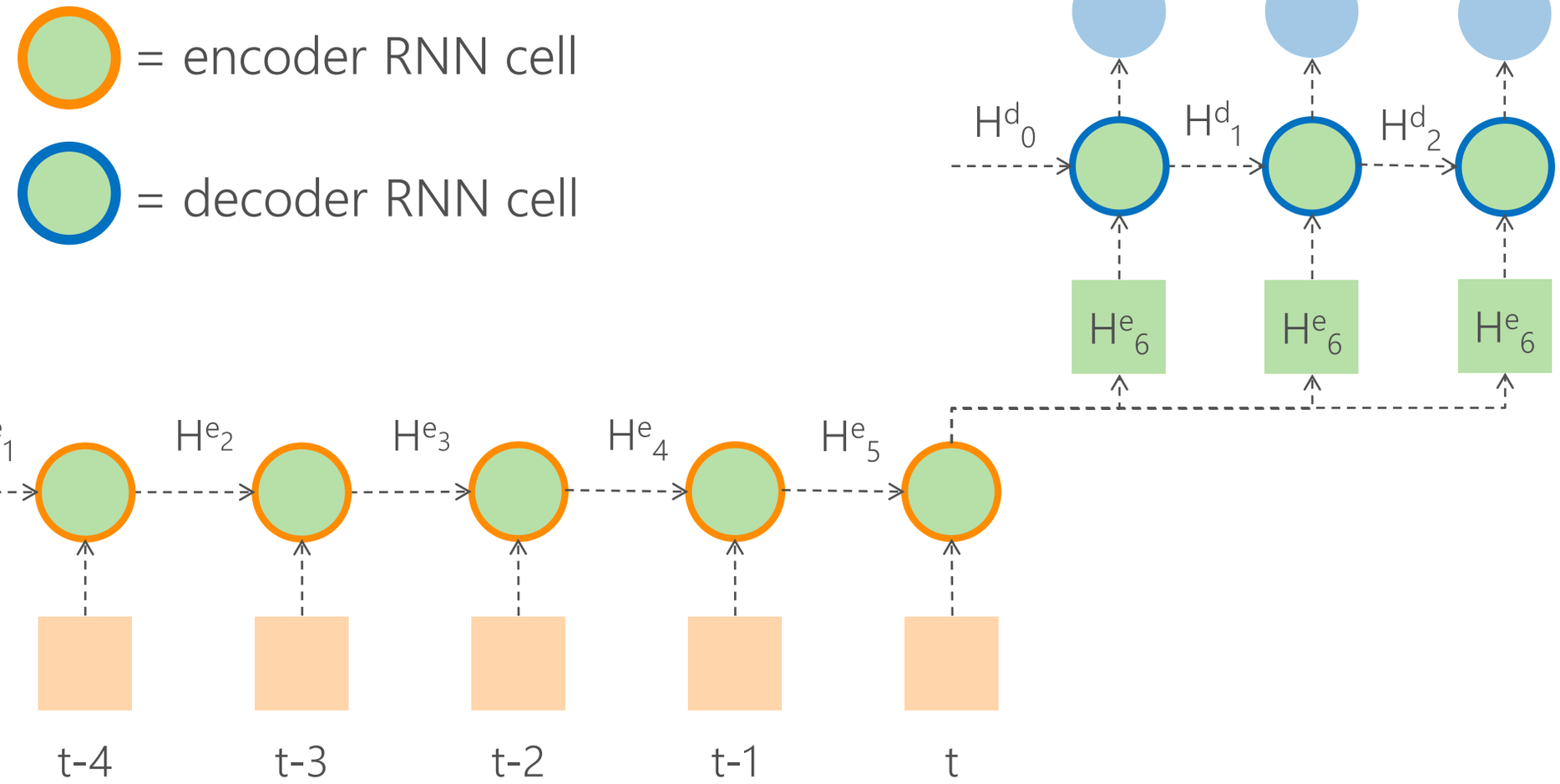
Vector output approach



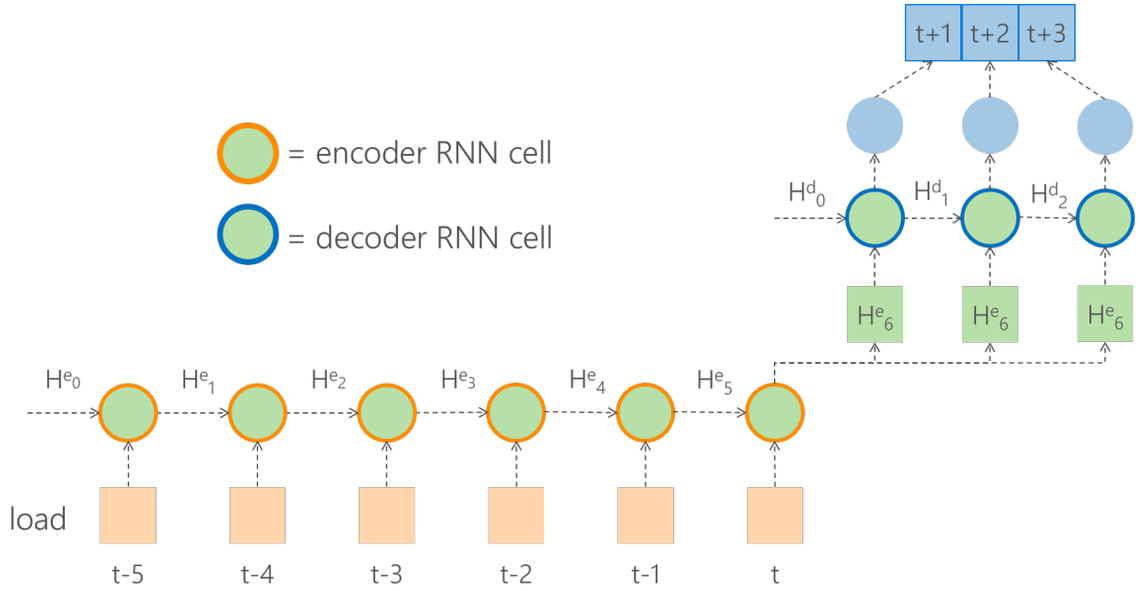
- Simplest to implement
- Fastest to train
- Does not model dependencies between predicted outputs

Reference notebook: multi_step_RNN_vector_output.ipynb

Simple encoder-decoder



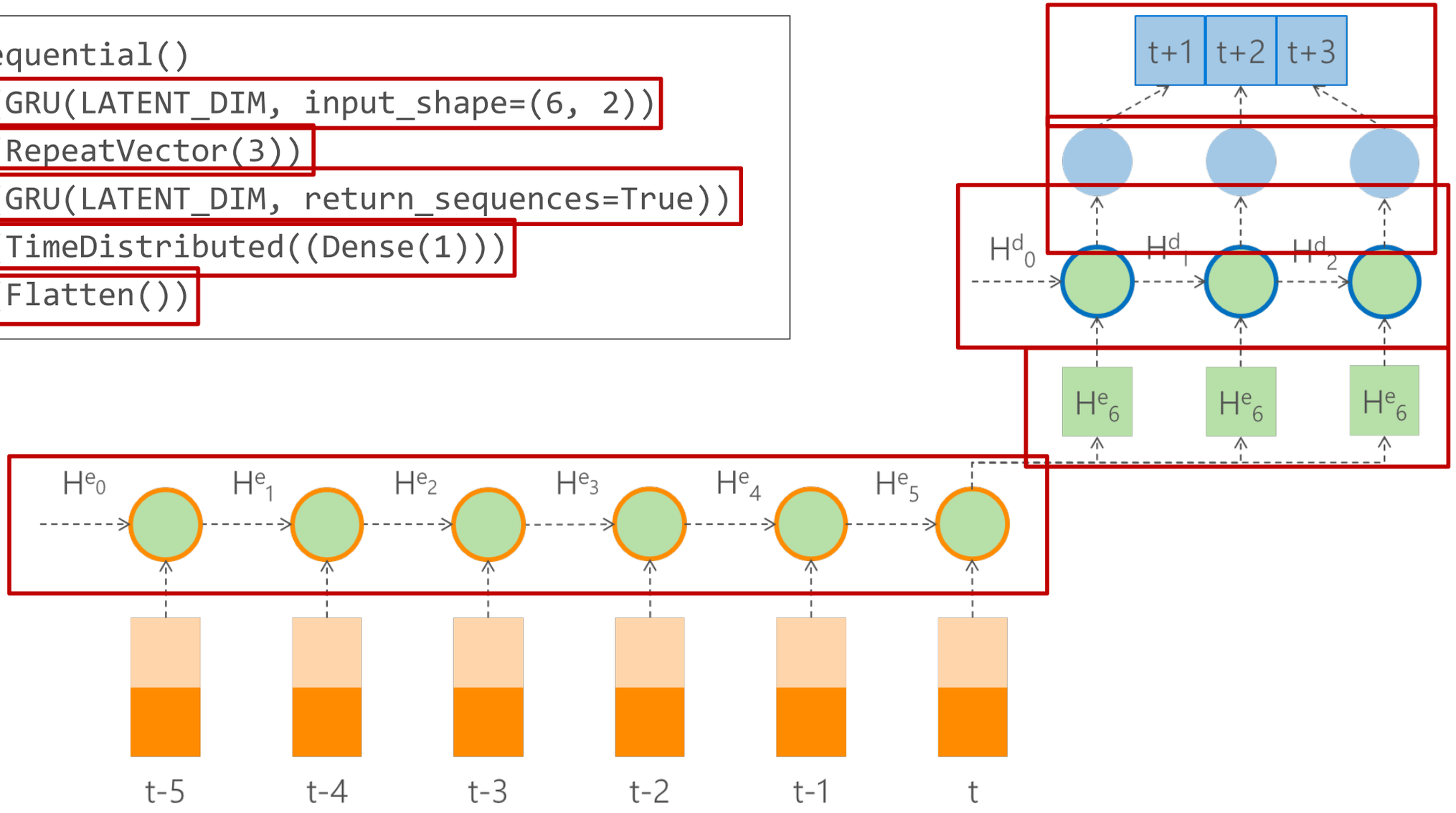
Simple encoder-decoder



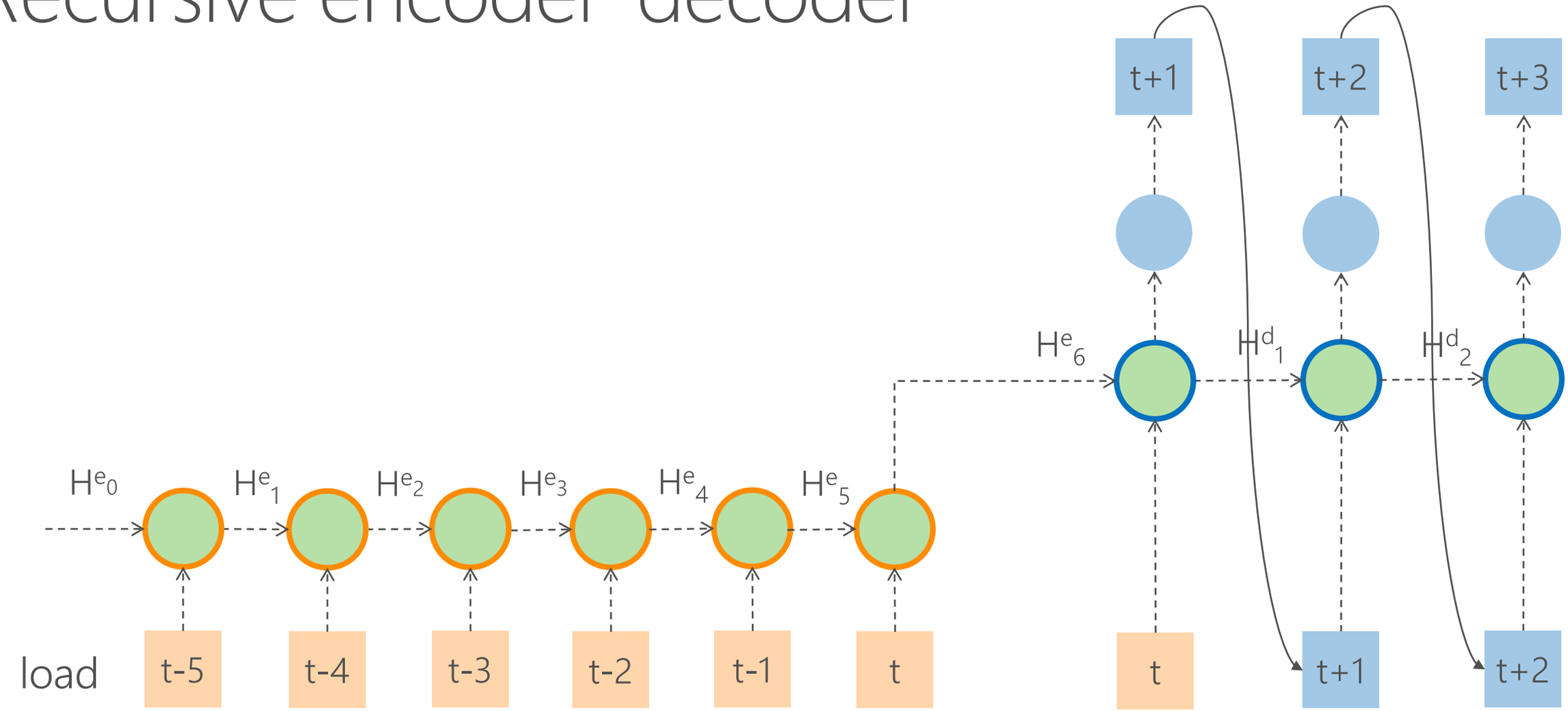
- Fairly simple to implement
- Tries to capture dependencies between forecasted time steps through decoder hidden state
- Slower to train with stacked RNN layers

Simple encoder-decoder

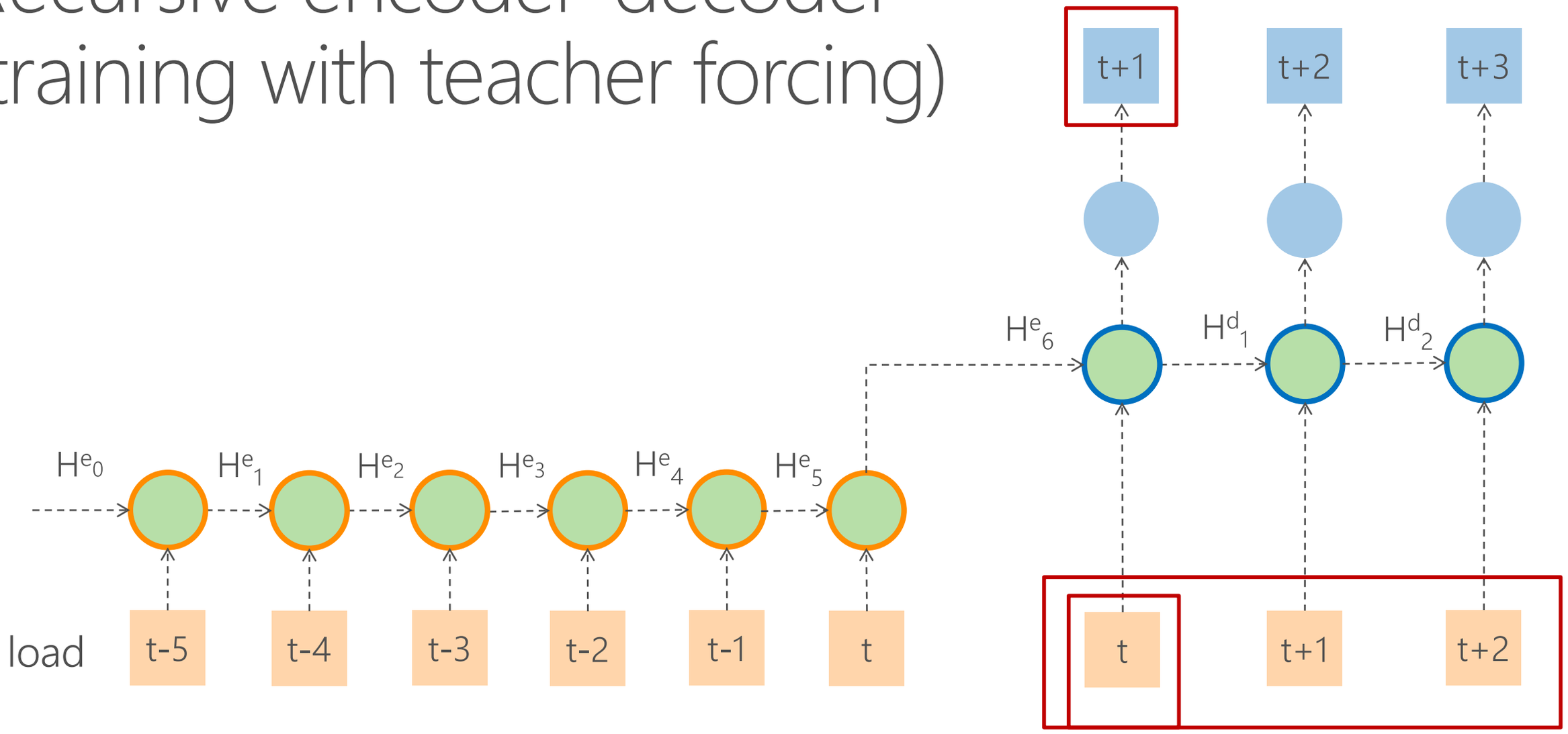
```
model = Sequential()
model.add(GRU(LATENT_DIM, input_shape=(6, 2))
model.add(RepeatVector(3))
model.add(GRU(LATENT_DIM, return_sequences=True))
model.add(TimeDistributed((Dense(1)))
model.add(Flatten())
```



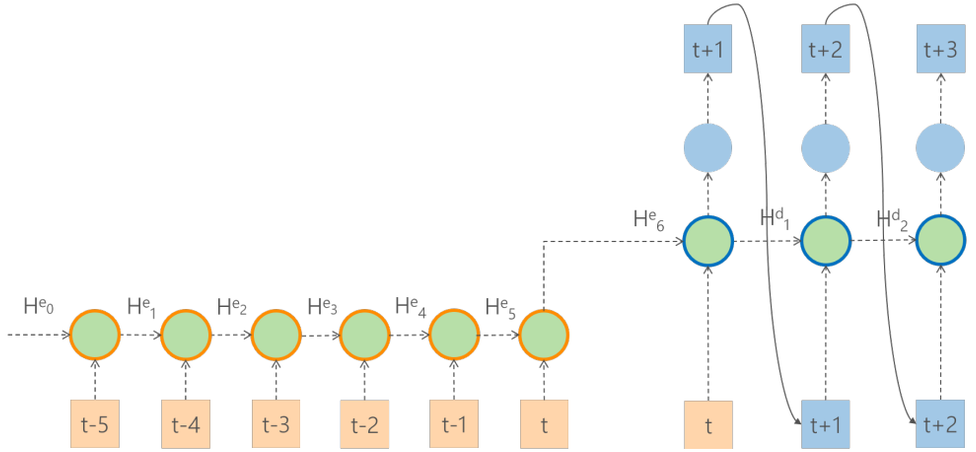
Recursive encoder-decoder



Recursive encoder-decoder (training with teacher forcing)



Recursive encoder-decoder

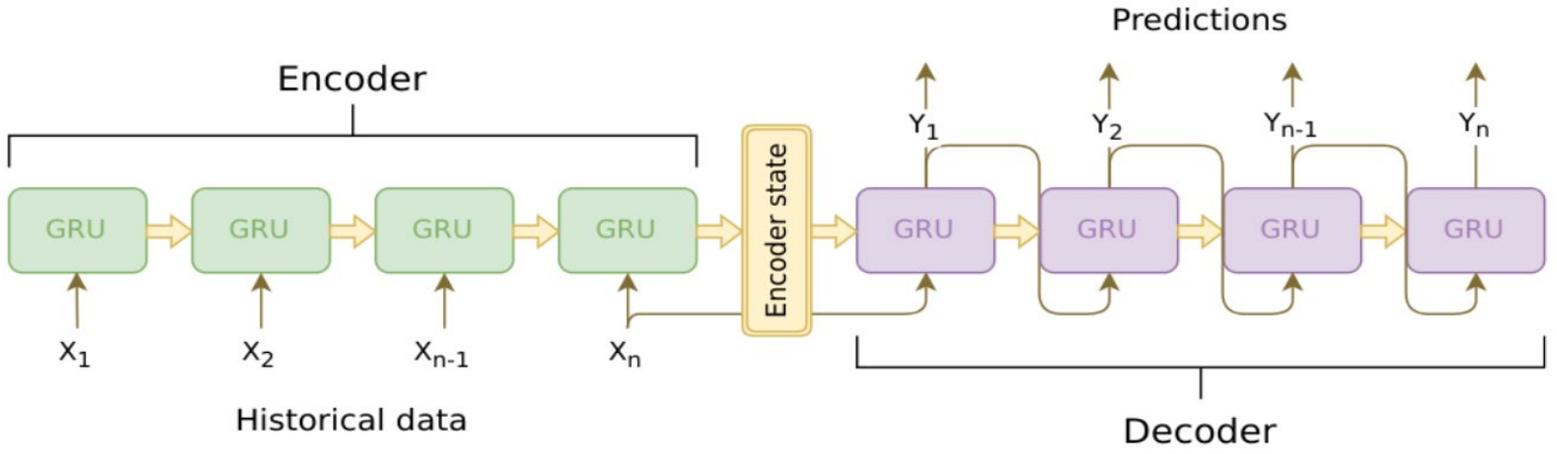


- Tries to capture dependencies between forecasted time steps through recursive decoder
- Variable length output (can generate forecasts of variable horizons)
- More difficult to implement
- Slower to train and slower to generate forecasts
- Error propagation due to recursive decoder

Web traffic forecasting competition

Forecast traffic of 145K Wikipedia pages

Winning solution: github.com/Arturus/kaggle-web-traffic



- Feature engineering: transform univariate to multivariate time series by adding seasonality features:

$$x_t \rightarrow (x_t, x_{t\text{-quarter}}, x_{t\text{-year}})$$

- COCOB optimizer that doesn't require learning rate tuning and converges considerably faster, custom implementation
- Sophisticated technique for building ensemble of RNN models, mainly for reducing model variance

Simple Exponential Smoothing (SES)

Time-series: $y_1, y_2, \ldots, y_t$

learned parameter

$$\hat{y}_{t+1|t} = \alpha y_t + \alpha(1-\alpha)y_{t-1} + \alpha(1-\alpha)^2 y_{t-2} + \ldots$$

Weighted average, where weight decrease exponentially as observations get older.

Weighted average form:

$$\hat{y}_{t+1|t} = \alpha y_t + (1-\alpha)\hat{y}_{t|t-1}$$

- $\hat{y}_{t+1|t}$: forecast of next value
- y_t: actual current value
- $\hat{y}_{t|t-1}$: forecast of current value

Component form:

Forecast: $\hat{y}_{t+1|t} = l_t$

Smoothing: $l_t = \alpha y_t + (1-\alpha) l_{t-1}$

l_t: level

In Simple Exponential Smoothing, level is the only component.

Exponential Smoothing (ES)

Forecasting: $\hat{y}_{t+1|t} = l_t \cdot s_{t+1}$ ← seasonality term (multiplicative method)

Smoothing: $l_t = \alpha \frac{y_t}{s_t} + (1 - \alpha) l_{t-1}$

Seasonal: $s_{t+m} = \gamma \frac{y_t}{l_t} + (1 - \gamma) s_t$

forecast of the next seasonality term

actual value of the current seasonality term

forecast of the current seasonality term

m - length of seasonality

ES-RNN

Forecasting: $\hat{y}_{t+1|t} = l_t \cdot s_{t+1} \cdot RNN(x_t)$

Smoothing: $l_t = \alpha \frac{y_t}{s_t} + (1 - \alpha) l_{t-1}$

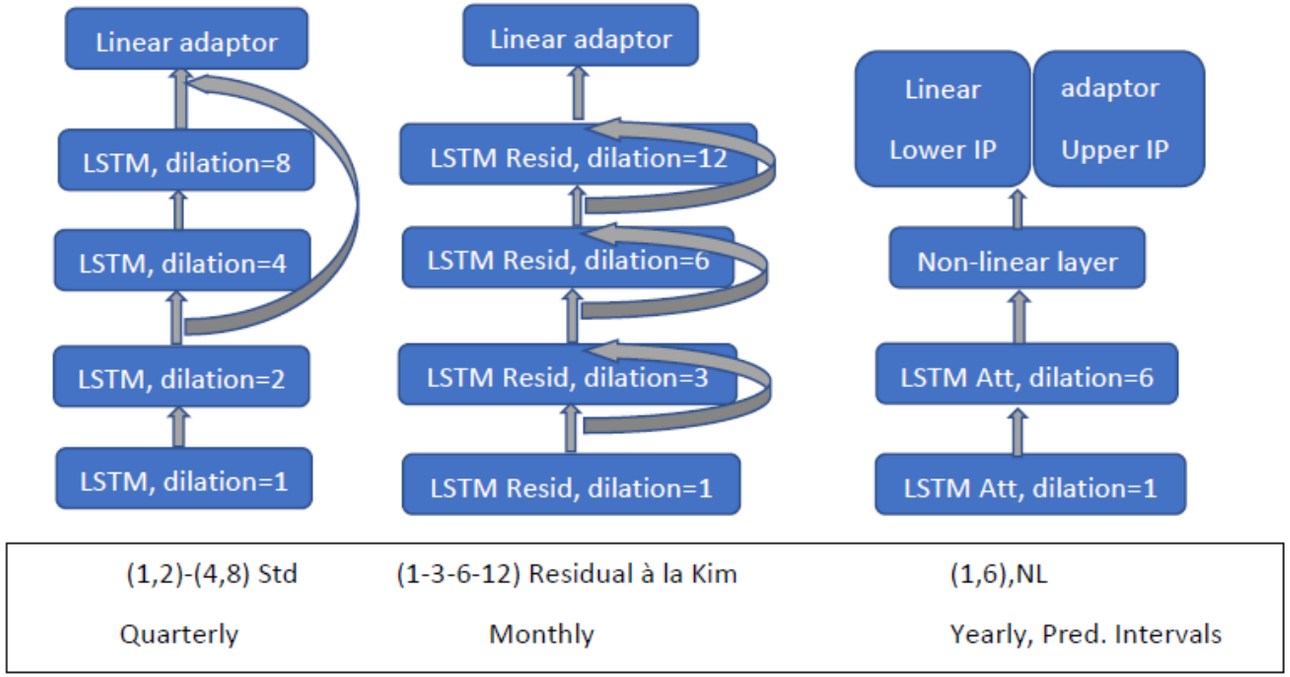
Seasonal: $s_{t+m} = \gamma \frac{y_t}{l_t} + (1 - \gamma) s_t$

De-seasonalized and normalized time-series: $x_t = \frac{y_t}{l_t s_t}$

ES-RNN Neural Architectures

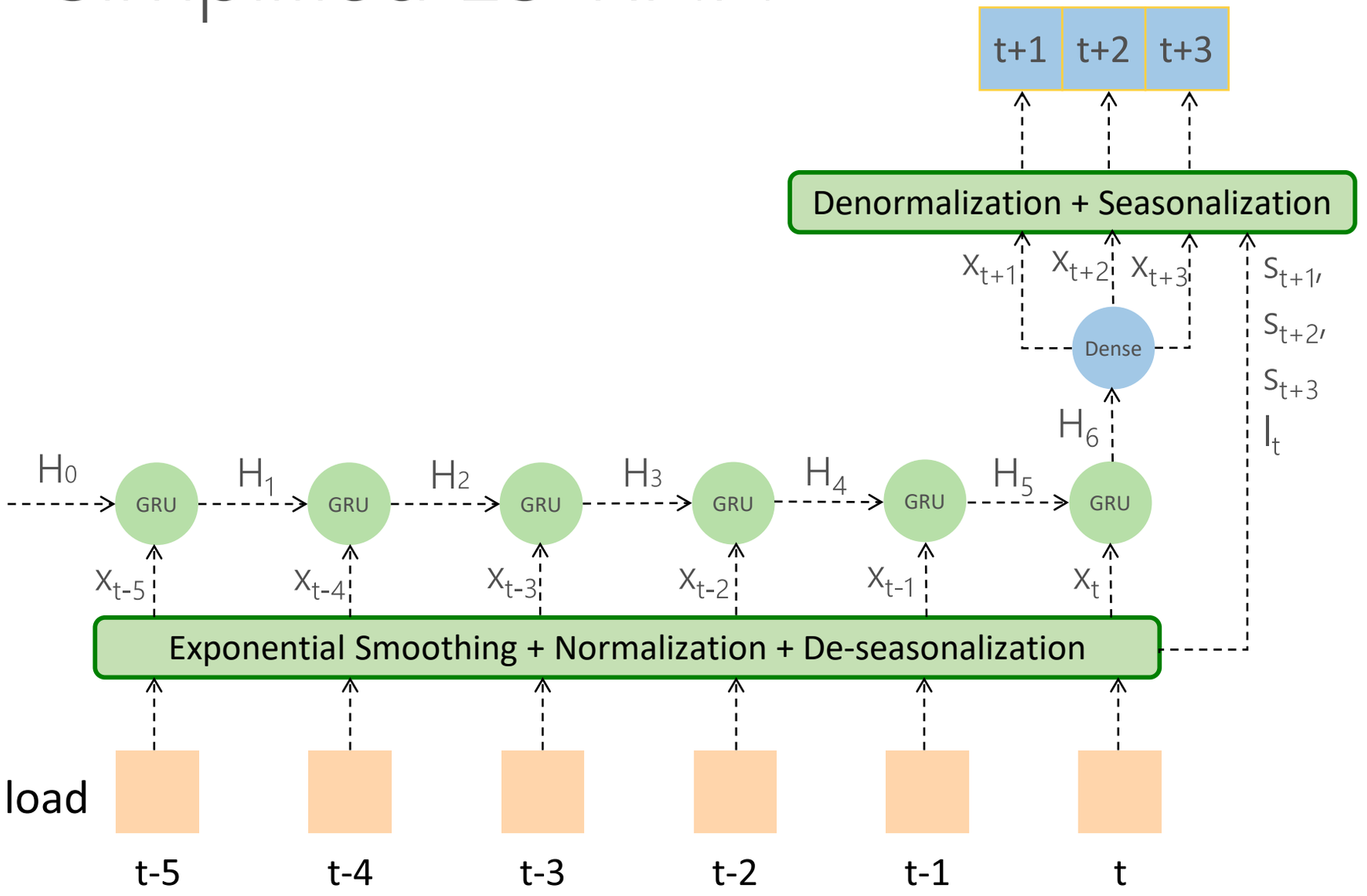
$$\hat{y}_{t+1|t} = l_t \cdot s_{t+1} \cdot \boxed{RNN(x_t)}$$

- Stack of dilated LSTM-based blocks connected with shortcuts.
- ES-RNN uses 3 types of dilated LSTMs:
  - Dilated Recurrent Neural Networks
  - Dual-Stage Attention-Based Recurrent Neural Network
  - Residual LSTM



In the example notebook of this tutorial, we simplify it to standard GRU.

Simplified ES-RNN



$$\hat{y}_{t+1|t} = l_t \cdot s_{t+1} \cdot RNN(x_t)$$

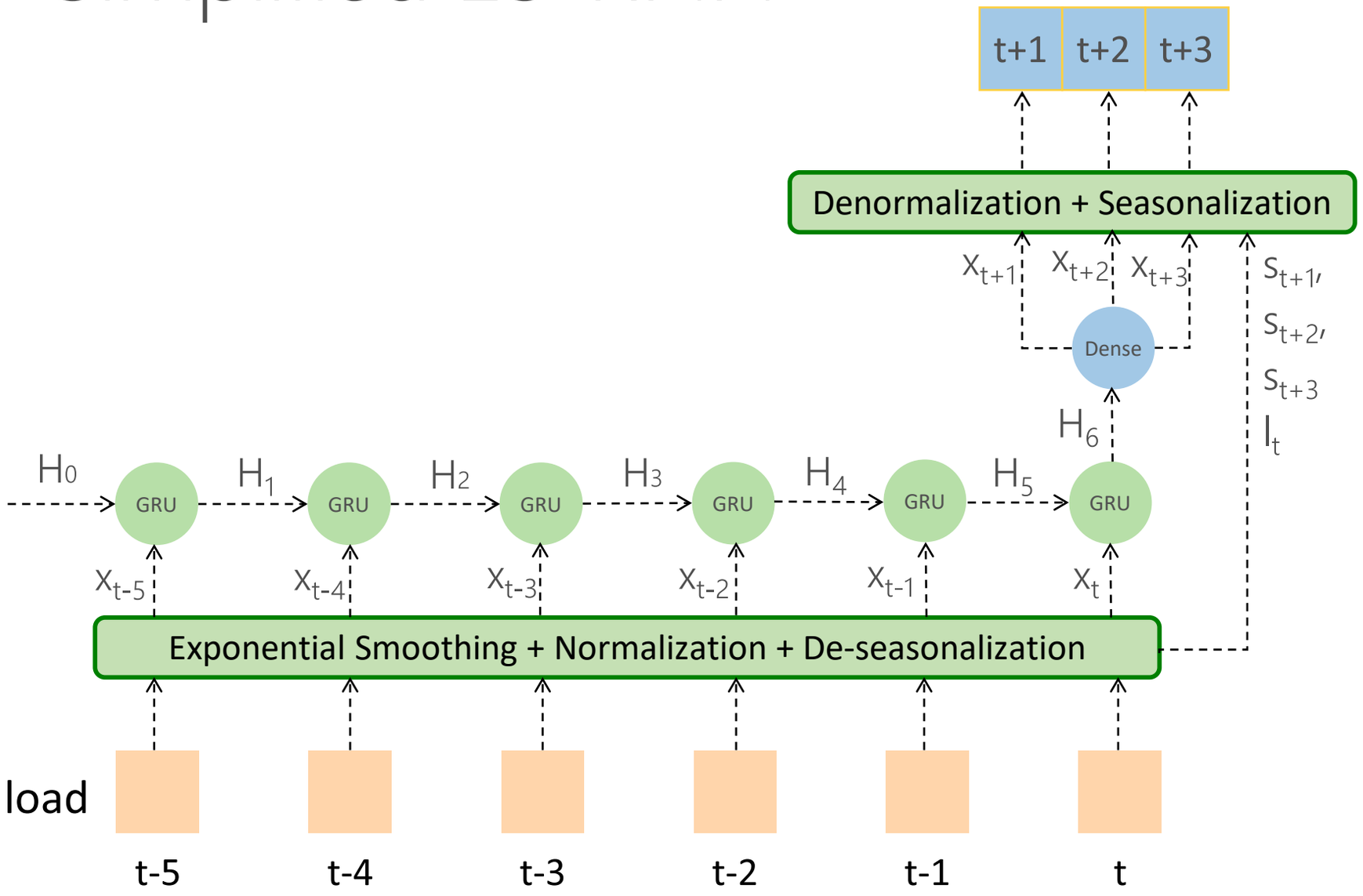
$$RNN(x_t)$$

Smoothing: $l_t = \alpha \frac{y_t}{s_t} + (1-\alpha) l_{t-1}$

Seasonal: $s_{t+m} = \gamma \frac{y_t}{l_t} + (1-\gamma) s_t$

De-seasonalized and normalized time-series: $x_t = \frac{y_t}{l_t s_t}$

Simplified ES-RNN

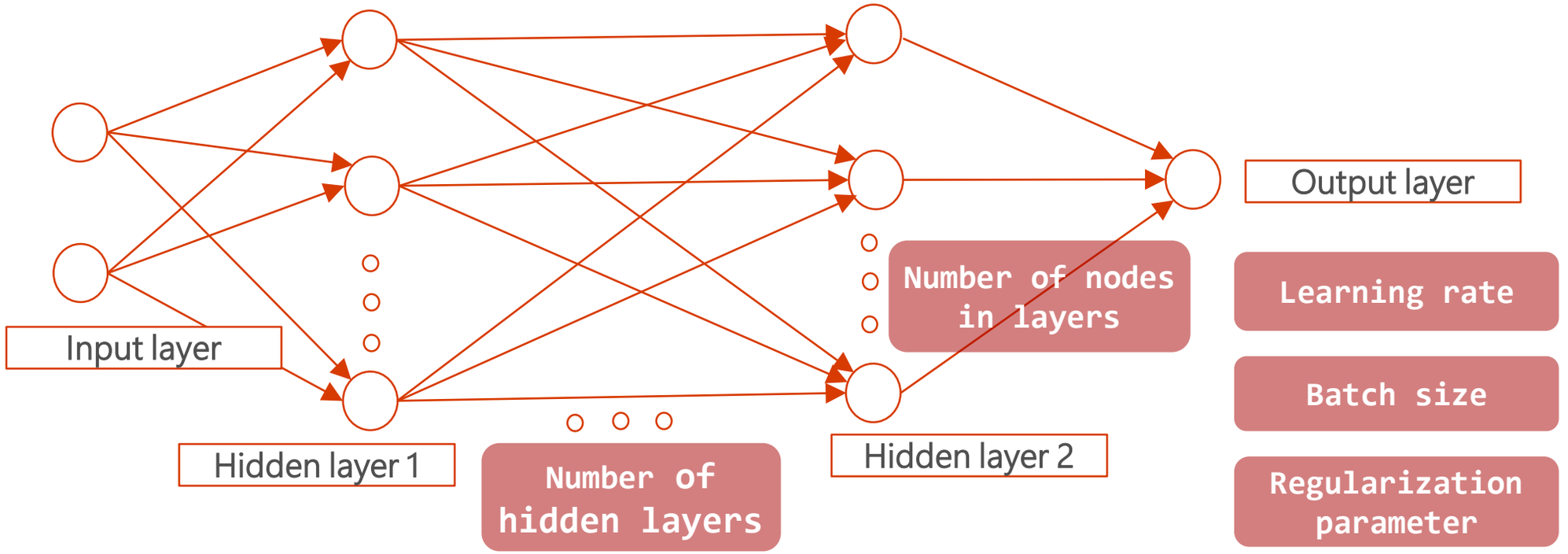


- ES-RNN can be used to forecast multiple time-series
  - Local parameters for each time series: $\alpha, \gamma, s_0, s_1, \ldots, s_{m-1}$
  - Global parameters, shared by all time series: weights of connections inside RNN

- Cost function depends on all local and global parameters which are learned by minimizing cost function (using BPTT)

What are hyperparameters?

- Adjustable parameters that govern model training
  - **Architecture**: number of layers, number of cells in each layer, connections
  - **Optimization**: learning rate, mini-batch size, stopping criterion, initialization
  - **Loss function**: weight of regularization term
- Chosen prior to training, stay constant during training, e.g.



Hyper-parameter tuning

- Search across various hyperparameter configurations
- Find the configuration that results in best performance

Challenges

- Huge search space to explore
- Sparsity of good configurations
- Expensive evaluation
- Limited time and resources

Key Takeaways

- Deep learning models are very powerful models, sometimes are the most accurate models.
- Neural network models share many techniques/tricks
- Tuning hyperparameters and other tricks are important for creating an accurate model