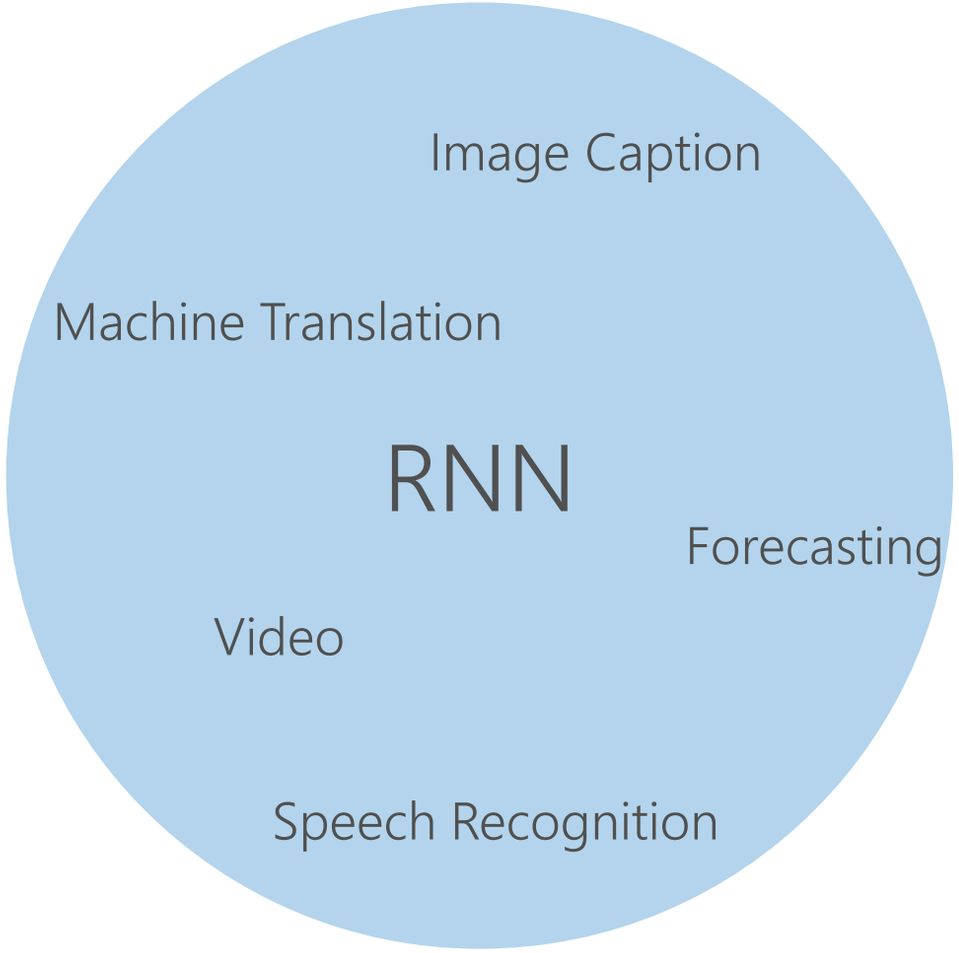
Microsoft

# Recurrent Neural Networks for Time Series Forecasting

Yijing Chen, Dmitry Pechyoni, Angus Taylor

# Tutorial Introduction

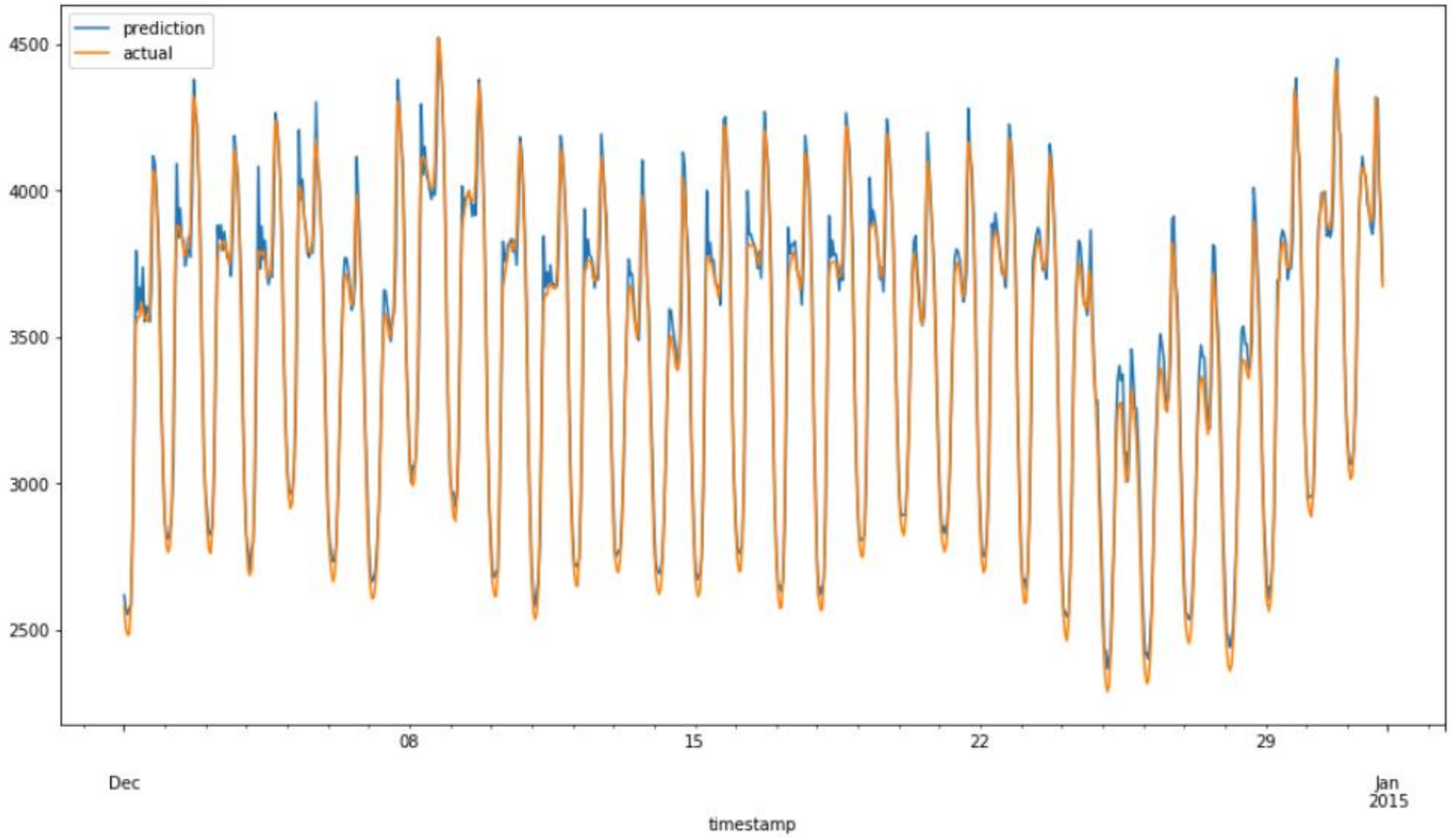



# Tutorial Goals

- Understand the basics of recurrent neural networks (RNN) and advanced RNN architectures including LSTM and GRU
- Know how to use Keras to build RNN model for time series forecasting
- Know a number of techniques and tricks that are important for building successful RNN-based time series forecasting models

# Tutorial Goals

By the end of this tutorial, you will have the knowledge to build RNN model to forecast New England energy consumption.



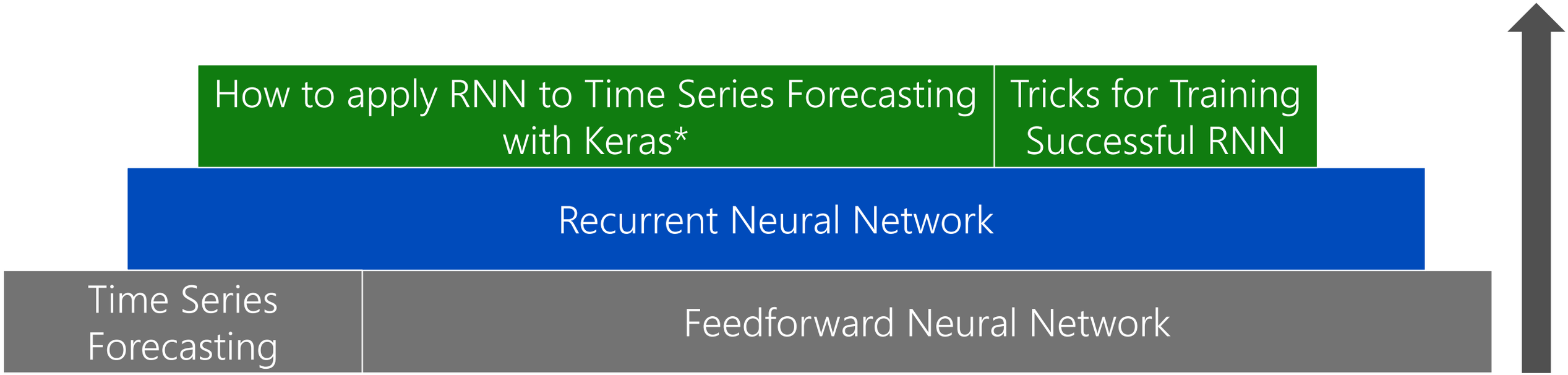
# Target Audience

- 200 level
- Know the basics of Python
- You need to understand the basic concepts of machine learning and have some experience of building machine learning models
- No prior experience with time series or neural networks is required

*Please provide feedbacks to help us improve!*

**tinyurl.com/rnnfeedback**

# Learning Path




- No hands-on lab but will have quizzes
- Q&A during break or in the end

* Keras: high-level neural networks library running on top of TensorFlow, CNTK, Theano, etc.

# 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 | ? |

# Why is Time Series Forecasting important?

- Every verticals has time series data (retail, energy, etc.)
- Forecasts guide future decisions
- Forecasts are needed in every industry
- Forecasts are usually the first step of other problems (e.g. optimization)

# 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?
- Can forecasts be *updated frequently* over time or must they be made once and remain static?

……

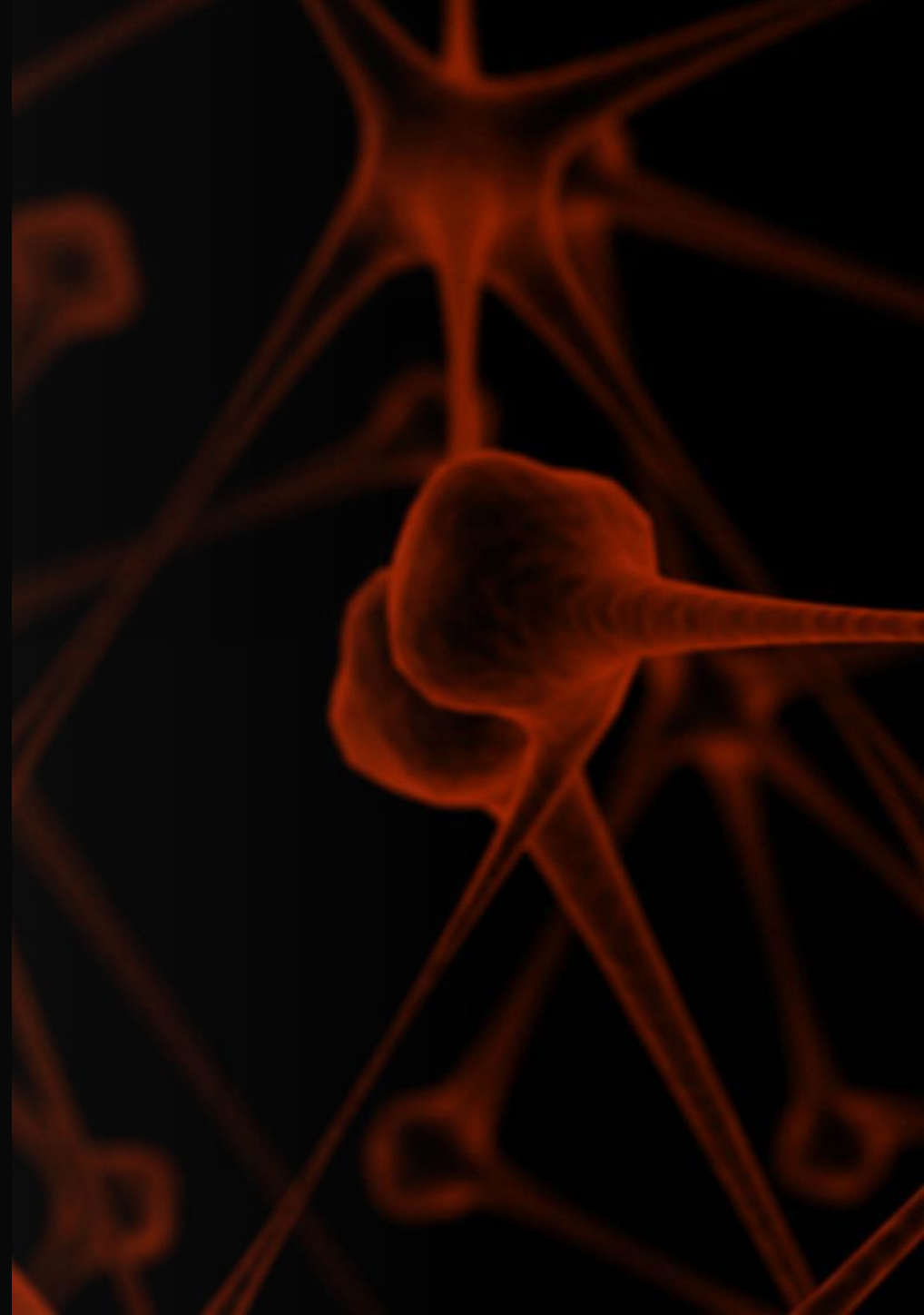
……

- What technique/model should I use?

# Why RNN?

- RNN 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)
- RNN model is very flexible
- RNN can learn from big data

# Introduction to Feedforward Neural Networks

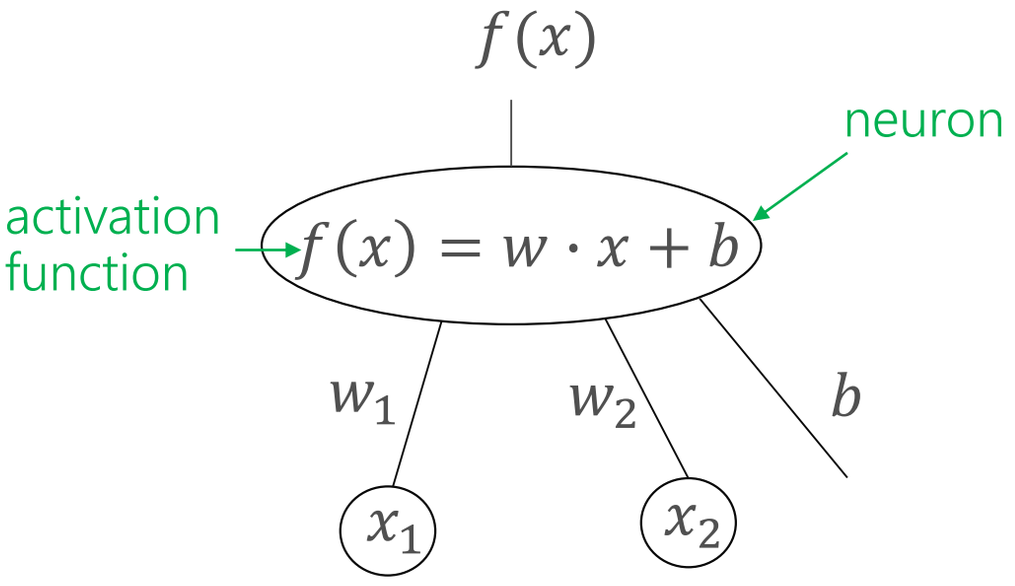


# Agenda

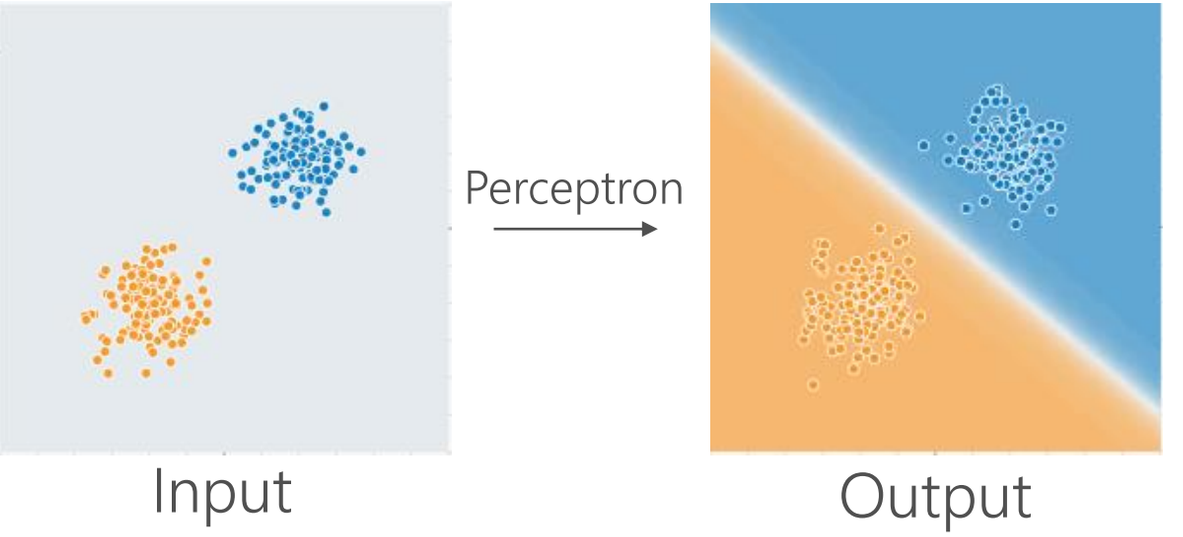
- Perceptron & Multilayer Perceptron
- Activation Functions
- Neural Network Training
  - Loss function
  - Gradient descent (stochastic, batch, mini-batch gradient descent)
  - Backpropagation

# Perceptron (Rosenblatt, 1957)

training example: $x = x_1 x_2$ , $y$



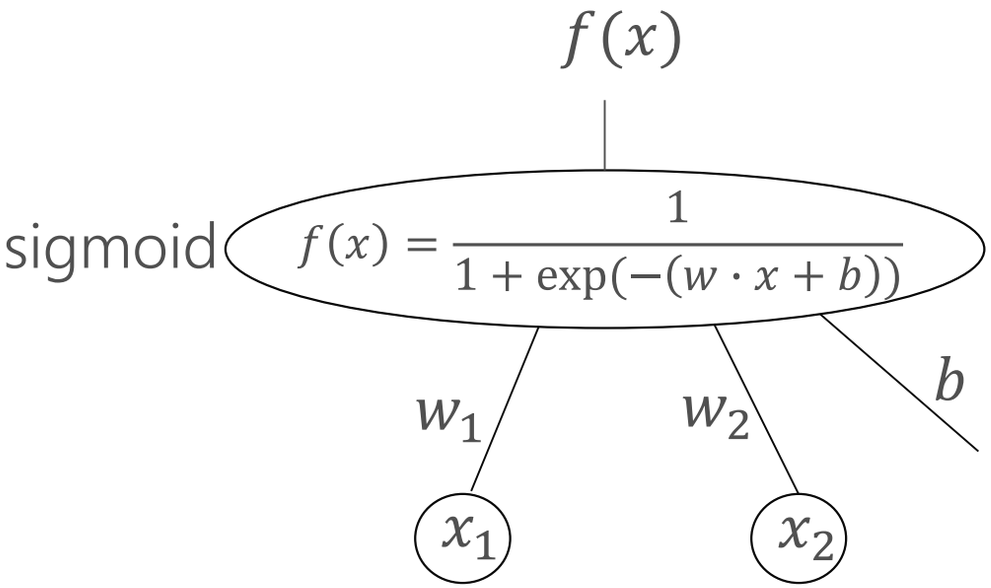

$$\hat{y} = \begin{cases} 1 & \text{if } f(x) > \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$



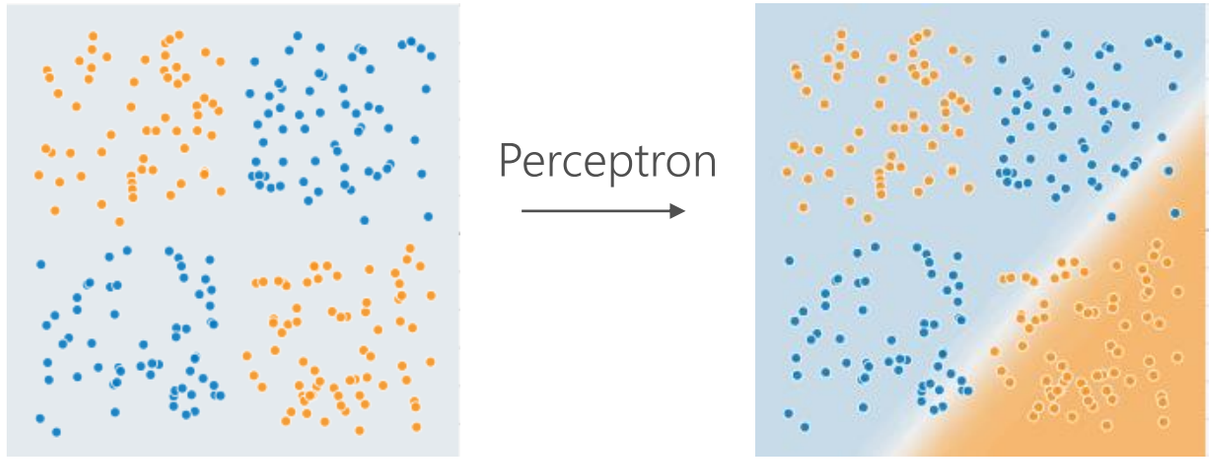

weights $w = w_1 w_2$ and bias $b$ are learned from training examples

# Perceptron – Nonlinear classification

Input example: $x = x_1 x_2, y$  Weights: $w = w_1 w_2$  Bias: $b$




$$\hat{y} = \begin{cases} 1 & \text{if } f(x) > \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$



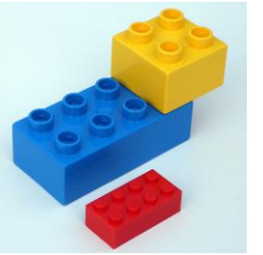

Input  Output

Q.: How can we get non-linear boundary?

A1: Use non-linear features (e.g. $x_1 \rightarrow x_1^2$)

A2: Multi-layer perceptron

# Multilayer Perceptron

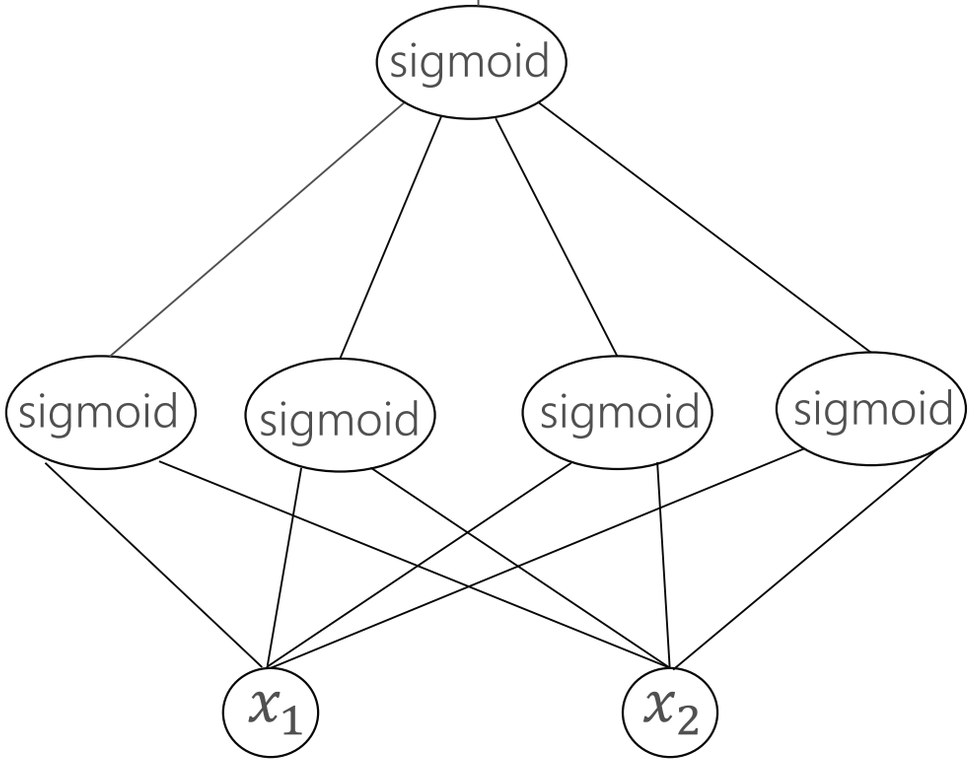


output

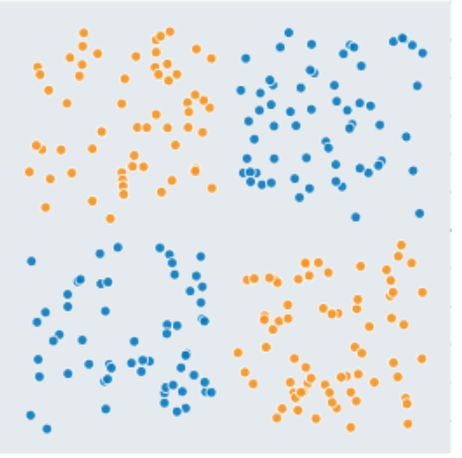
output layer

hidden layer

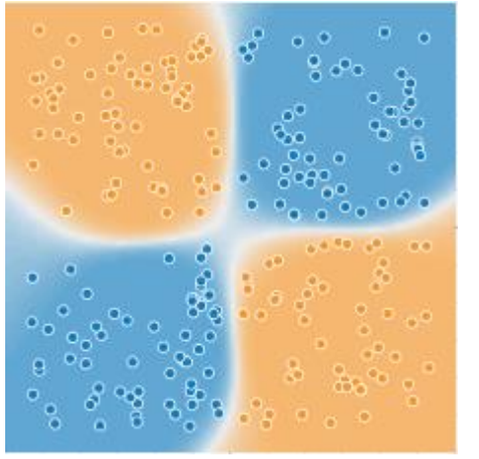
input layer




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



multilayer perceptron



Input

Output

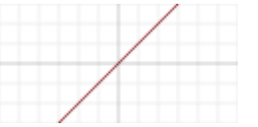
also known as Feed-Forward Neural Network and Deep Neural Network

# Activation Functions

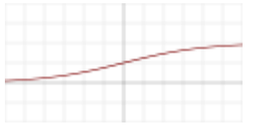
https://en.wikipedia.org/wiki/Activation_function

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

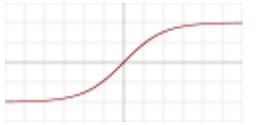
Identity $f(x') = x'$



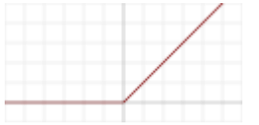
Sigmoid $f(x') = \sigma(x') = \frac{1}{1+e^{-x'}}$



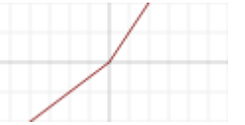
tanh (hyperbolic tangent) $f(x') = \frac{e^{x'}-e^{-x'}}{e^{x'}+e^{-x'}}$



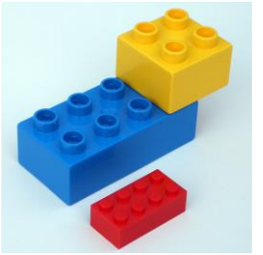
Rectifier linear unit (ReLU) $f(x') = \begin{cases} 0 \text{ for } x' < 0 \\ x' \text{ for } x' \geq 0 \end{cases}$

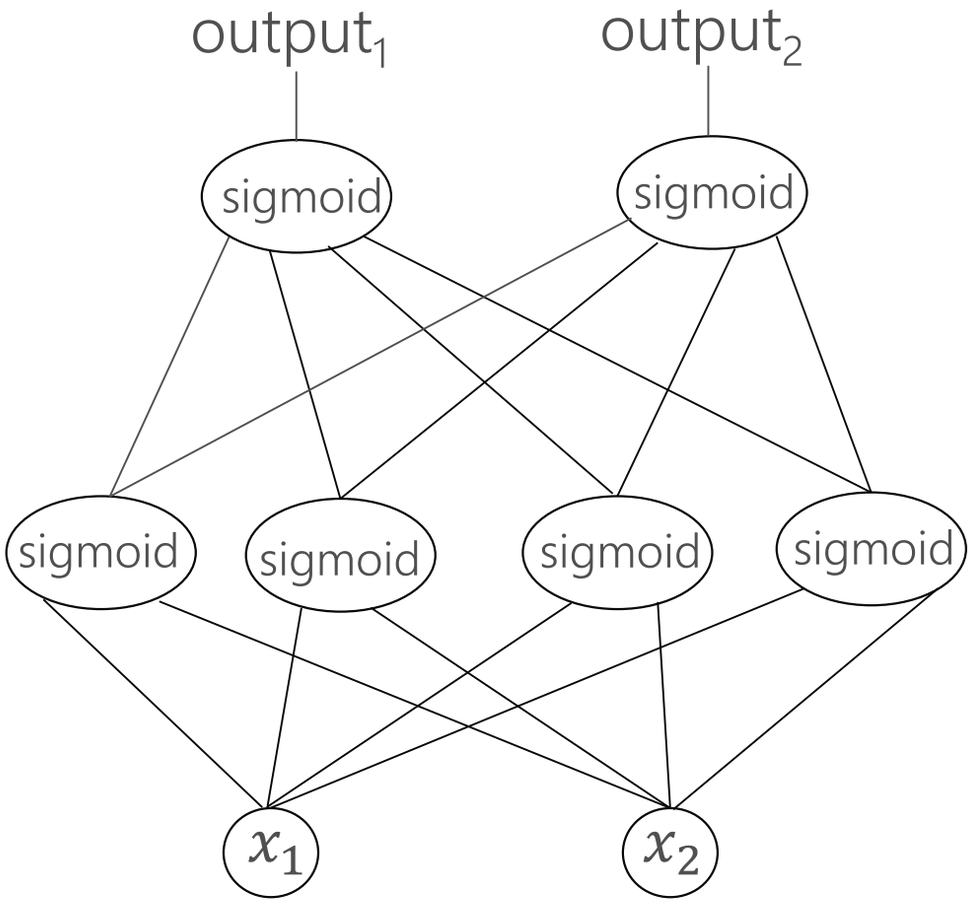


Parametric rectifier linear unit (PReLU) $f(x') = \begin{cases} \alpha x' \text{ for } x' < 0 \\ x' \text{ for } x' \geq 0 \end{cases}$

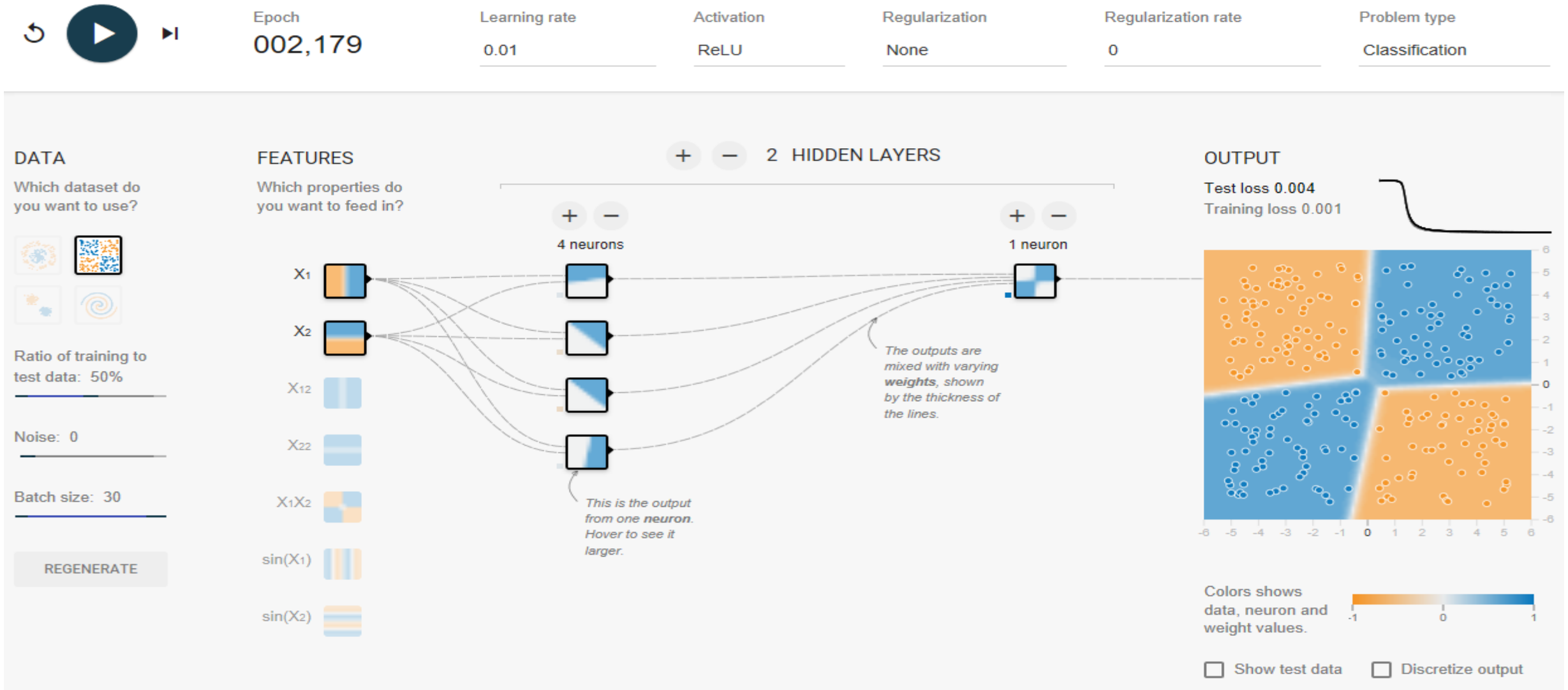


# Multiple Outputs





# Visualization of Feed-Forward Neural Net



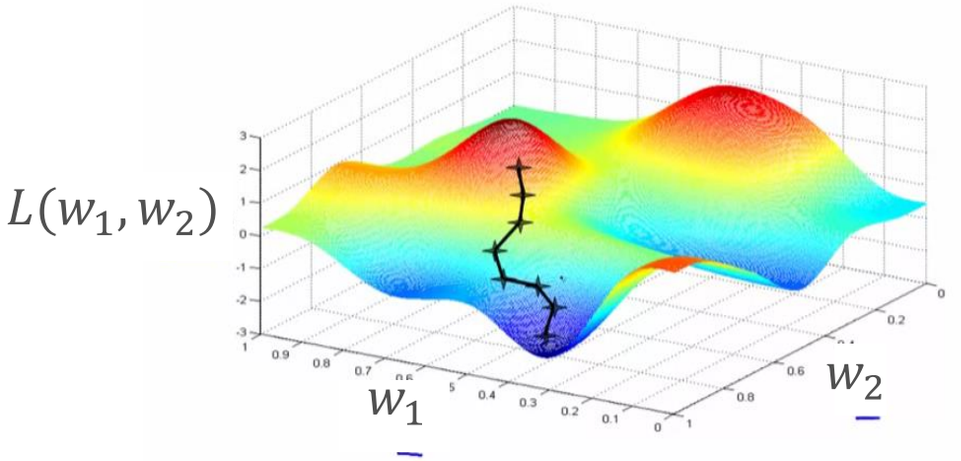

playground.tensorflow.org

# 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



# 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): $\frac{100}{N}\sum_{i=1}^{N}\frac{|y_i - \hat{y}_i|}{(|y_i| + |\hat{y}_i|)/2}$

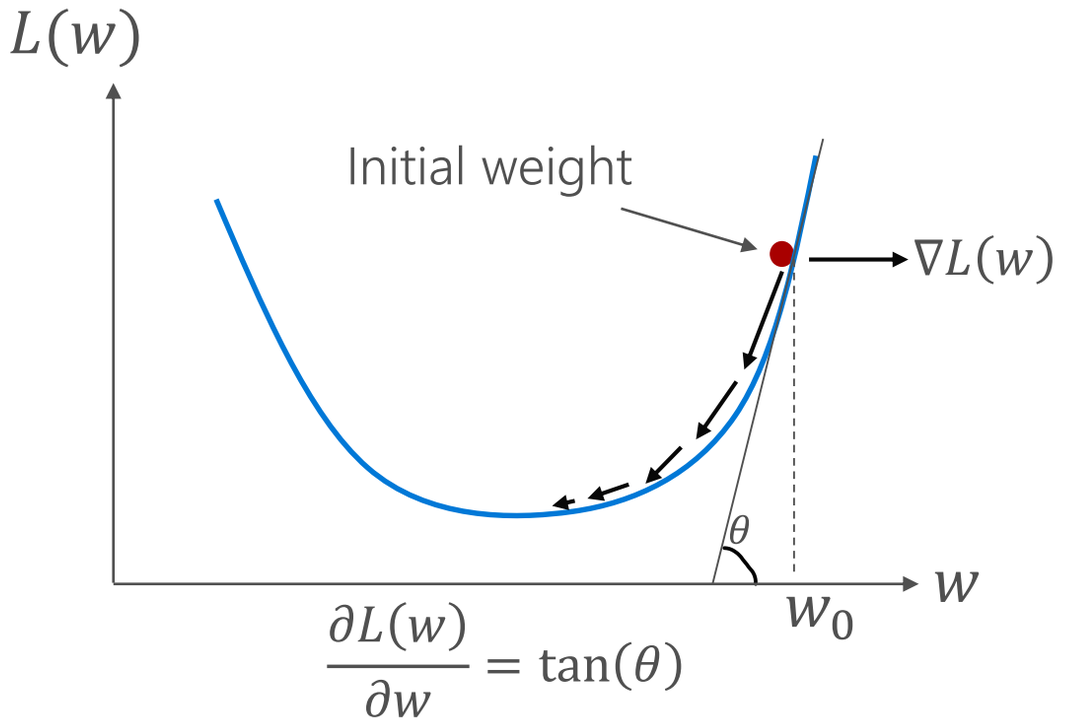
$\hat{y}_i$ is a function of weights $w$ and biases $b$.

Loss function decomposition $L(w, b) = \frac{1}{N}\sum_{i=1}^{N} L_i(w, b)$

# 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

# Batch gradient descent in machine learning

Loss function decomposition $L(w) = \frac{1}{N}\sum_{i=1}^{N} L_i(w)$

$$\nabla L(w) = \frac{1}{N}\sum_{i=1}^{N} \nabla L_i(w)$$

Initialization: $w = w_0$

While stopping criterion not met:

$$w = w - \frac{\alpha}{N}\sum_{i=1}^{N} \nabla L_i(w)$$

Need to go over all examples to complete a single optimization step.

With large $N$, gradient descent has a very slow convergence.

# Minibatch Stochastic Gradient Descent

Initialization: $w = w_0$

While stopping criterion not met:

Shuffle examples randomly

Partition examples into batches of size m

For $\mathbf{minibatch=1...N/m}$ examples

$$w = w - \frac{\alpha}{m}\sum_{i=1}^{m}\nabla L_i(w)$$

} epoch

$m$ – mini-batch size (default value 32 in Keras)

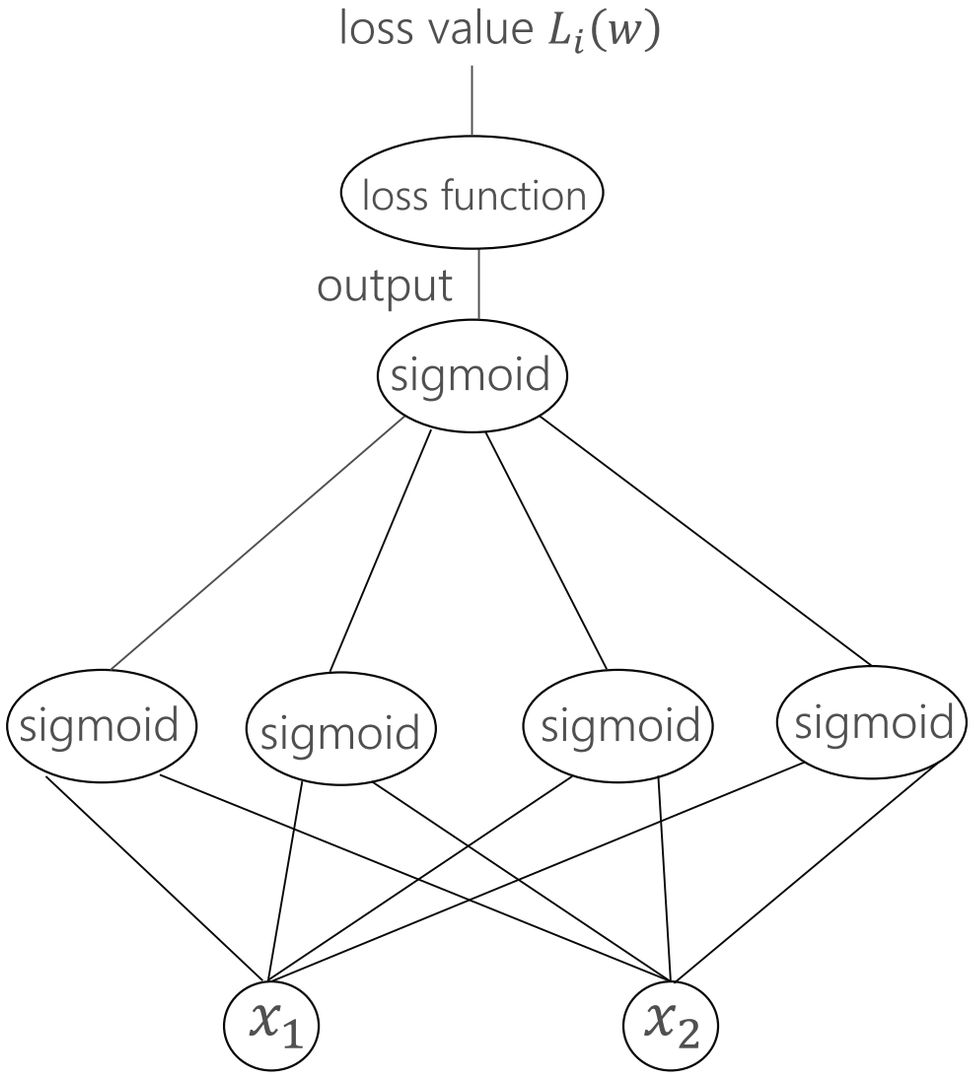
# Minibatch size

**Yann LeCun** @ylecun · Apr 26

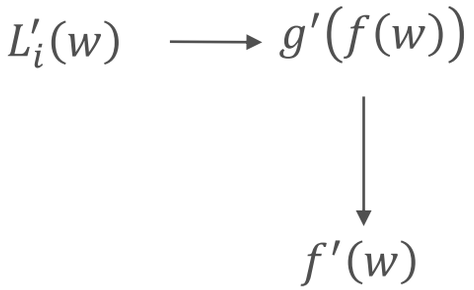
Training with large minibatches is bad for your health.
More importantly, it's bad for your test error.
Friends dont let friends use minibatches larger than 32. arxiv.org/abs/1804.07612

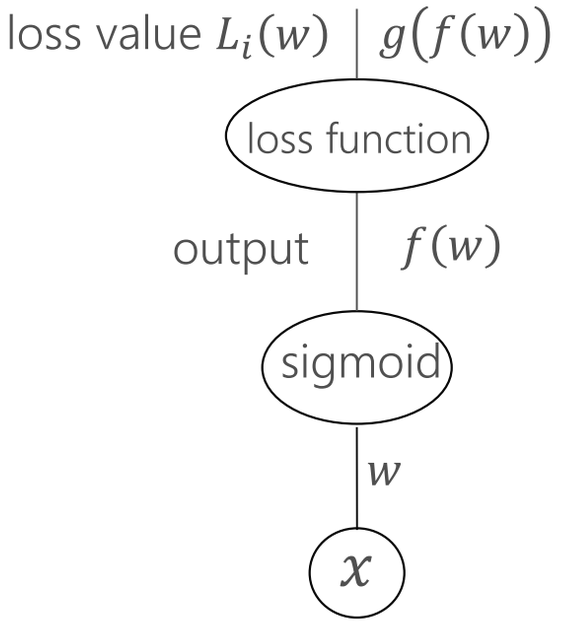
21 440 1.2K

# Computation of gradients in NN: Backpropagation



# Backpropagation – single branch



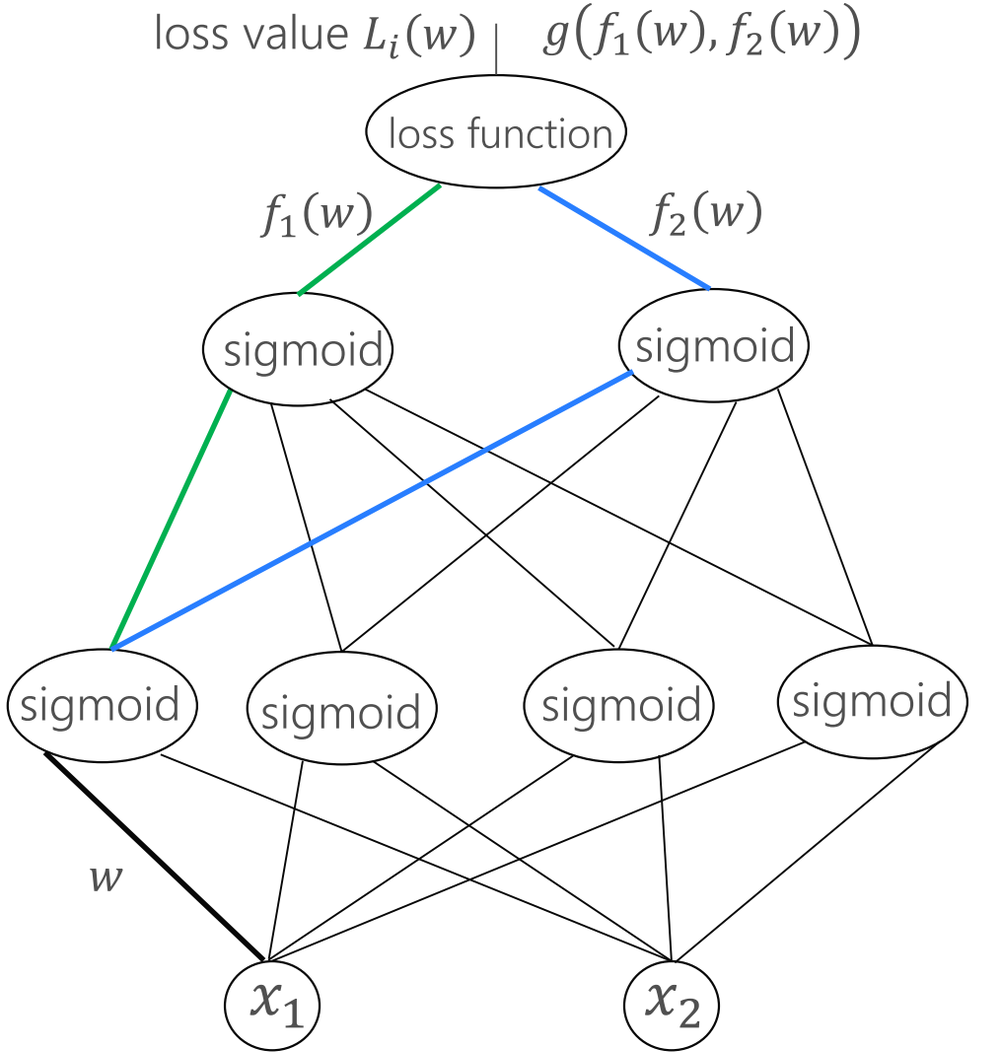




Chain rule

$$L_i(w) = g(f(w))$$

$$L_i'(w) = g'(f(w)) \cdot f'(w)$$

# Backpropagation – multiple branches




Chain rule

$$L_i(w) = g\big(f(w)\big) \quad L_i'(w) = g'\big(f(w)\big) \cdot f'(w)$$

Total derivative

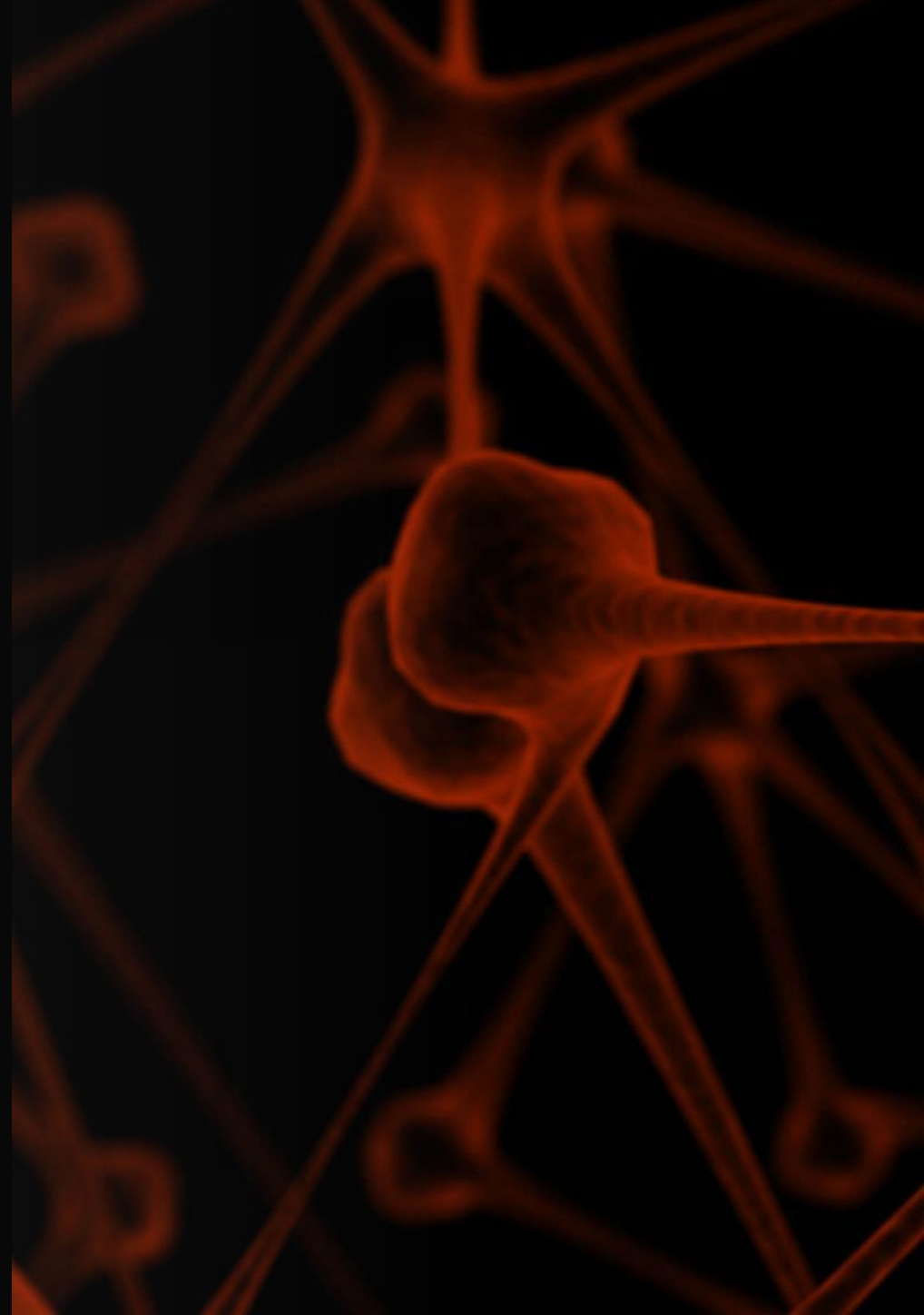
$$L_i(w) = g\big(f_1(w), f_2(w)\big)$$

$$L_i'(w) = \sum_{j=1}^{2} \frac{\partial g}{\partial f_j}\Big(f_j(w)\Big) \cdot \frac{\partial f_j}{\partial w}(w)$$

# Training tricks

- Initialization
- Weight regularization
- Tuning hyperparameters
- Early stopping
- 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

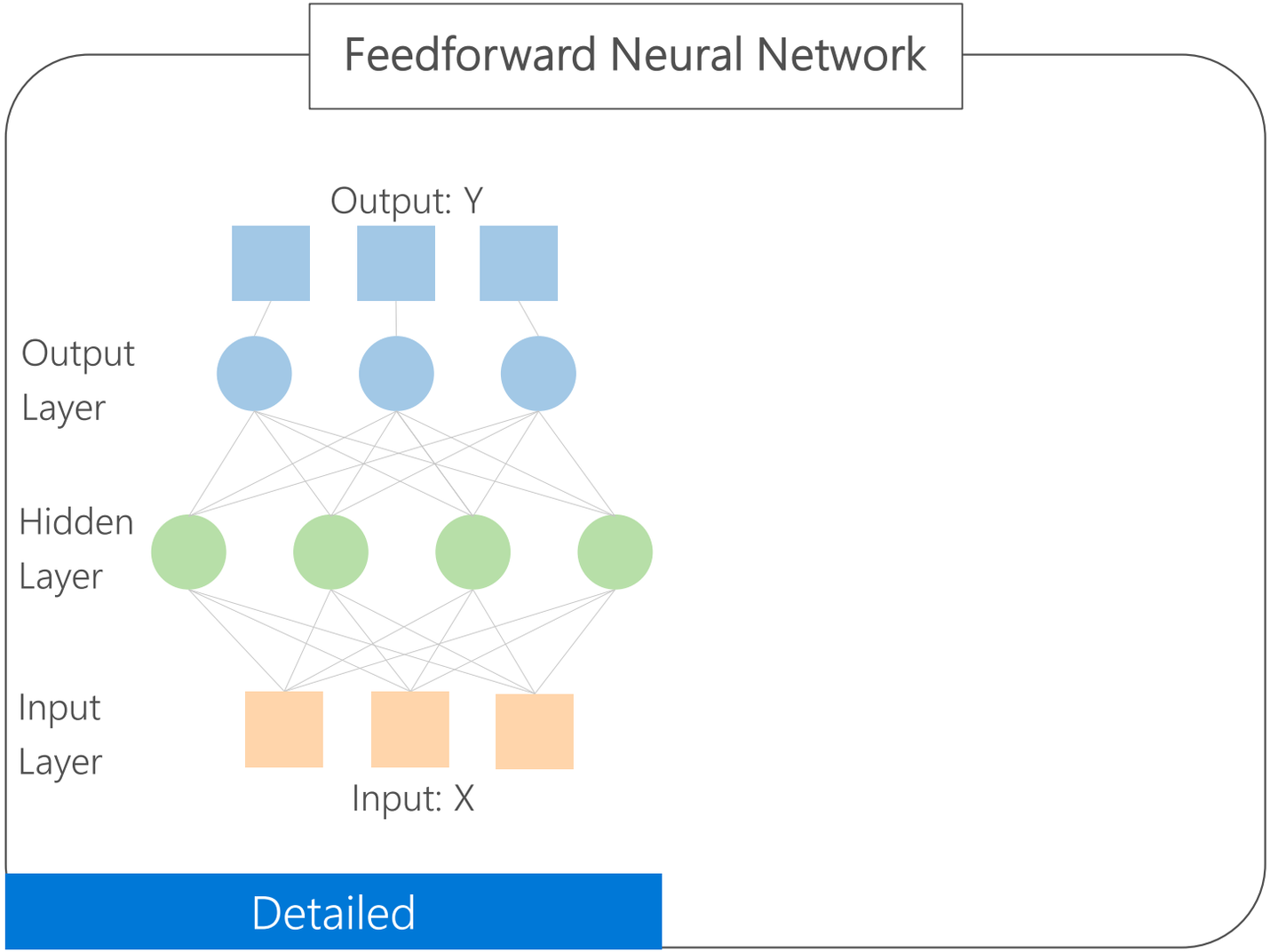
# Introduction to Recurrent Neural Networks



# Agenda

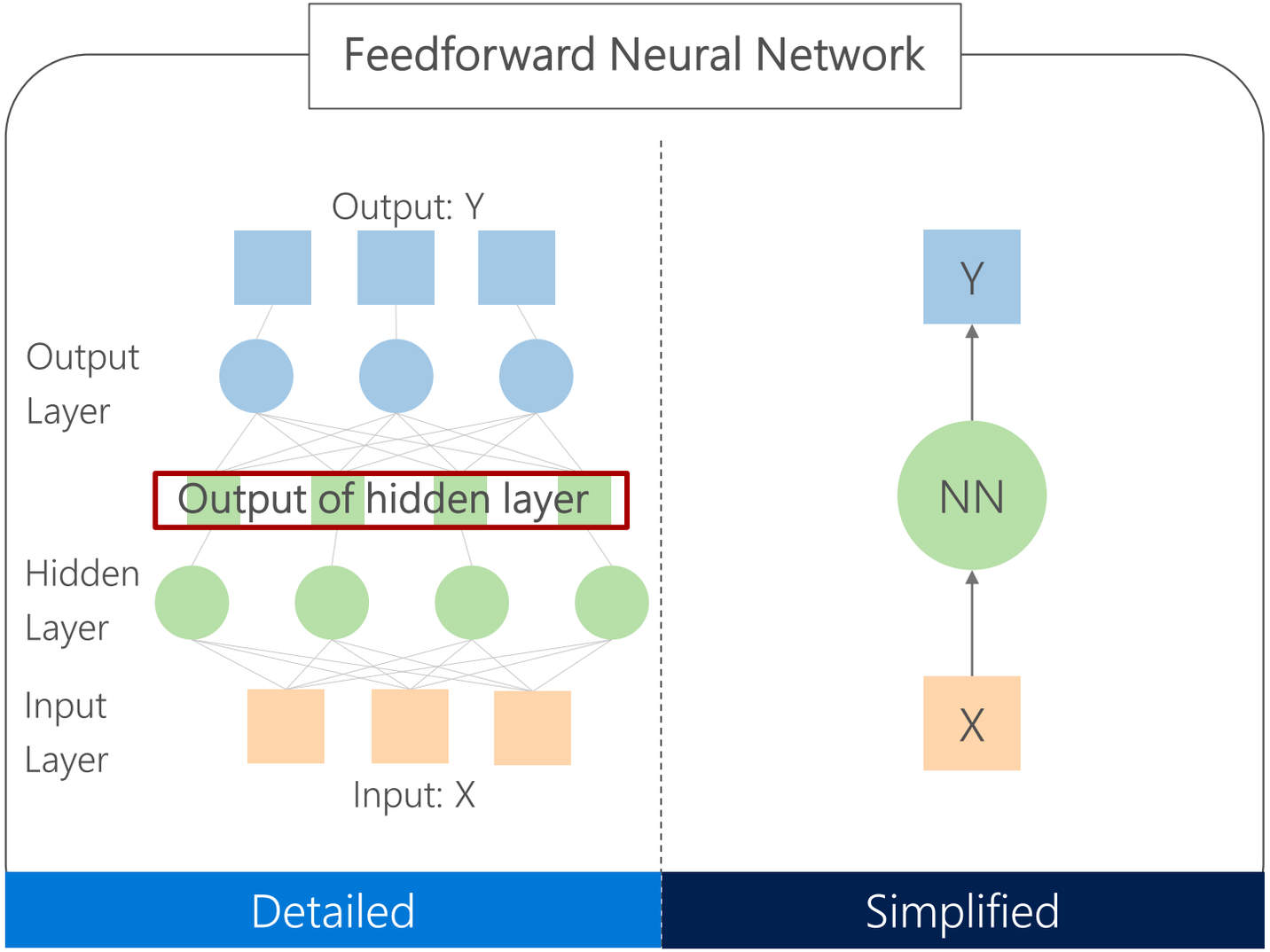
- What are RNNs?
- How RNNs are trained: Backpropagation through time (BPTT)
- Vanilla RNN and its gradient problems
- Other RNN units
  - GRU
  - LSTM
- RNN stacking
- Quiz & Break & Q&A

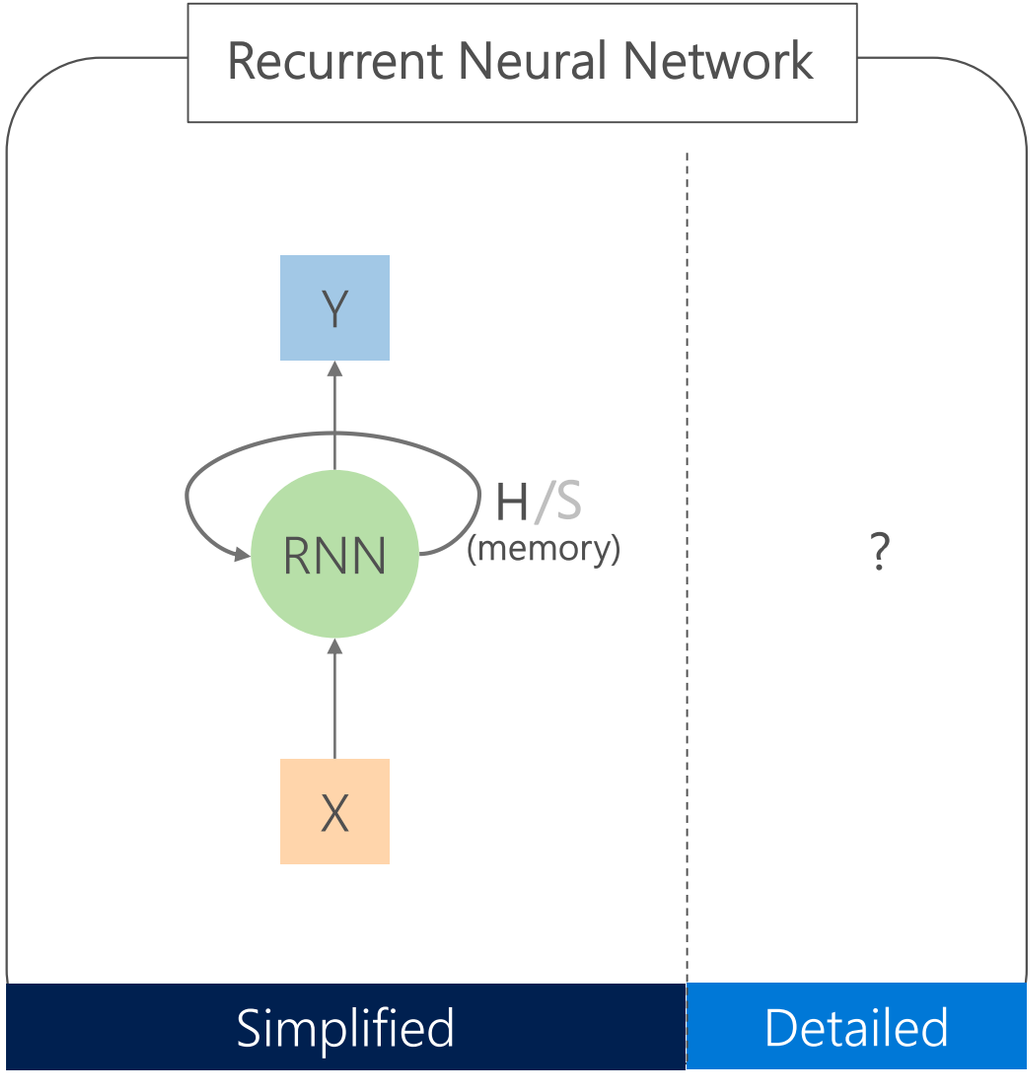
# What are RNNs?



# What are RNNs?

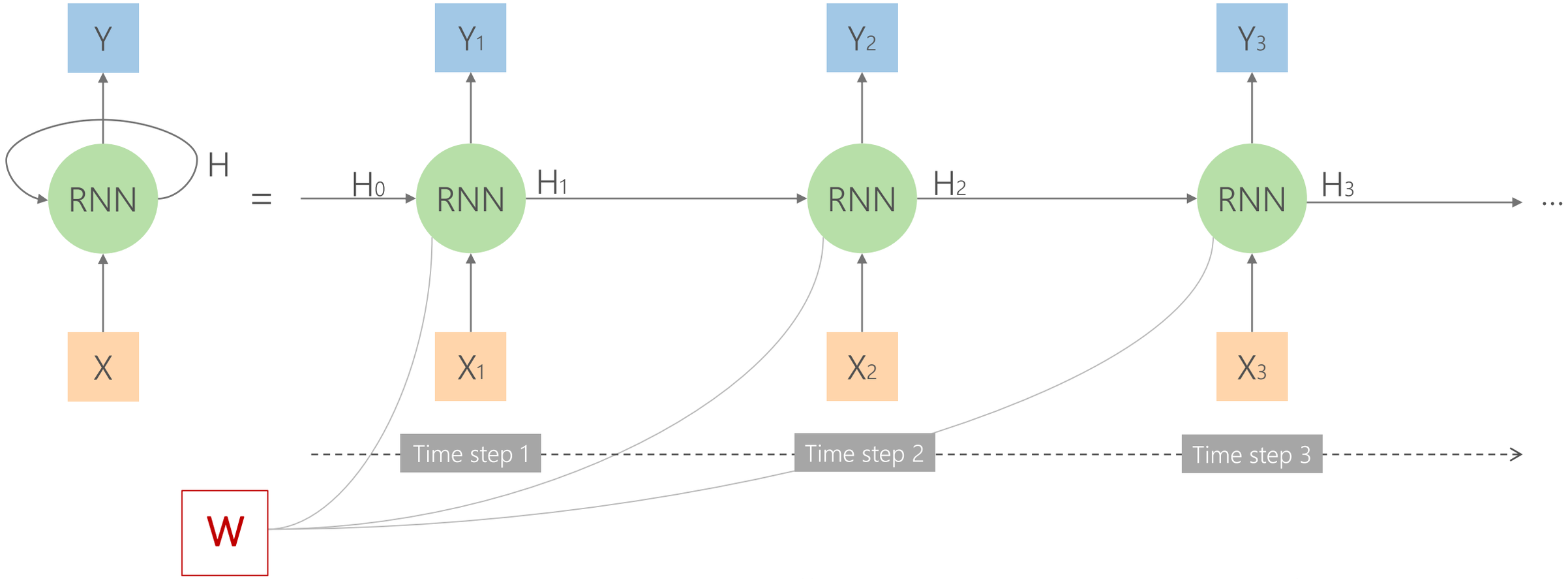
## RNN has internal hidden state which can be fed back to network



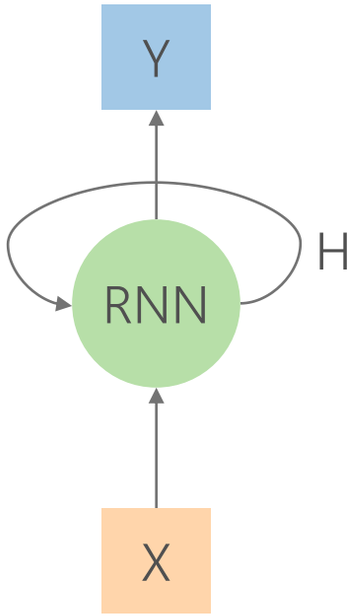
# Unrolled RNN

## The same weight and bias shared across all the steps



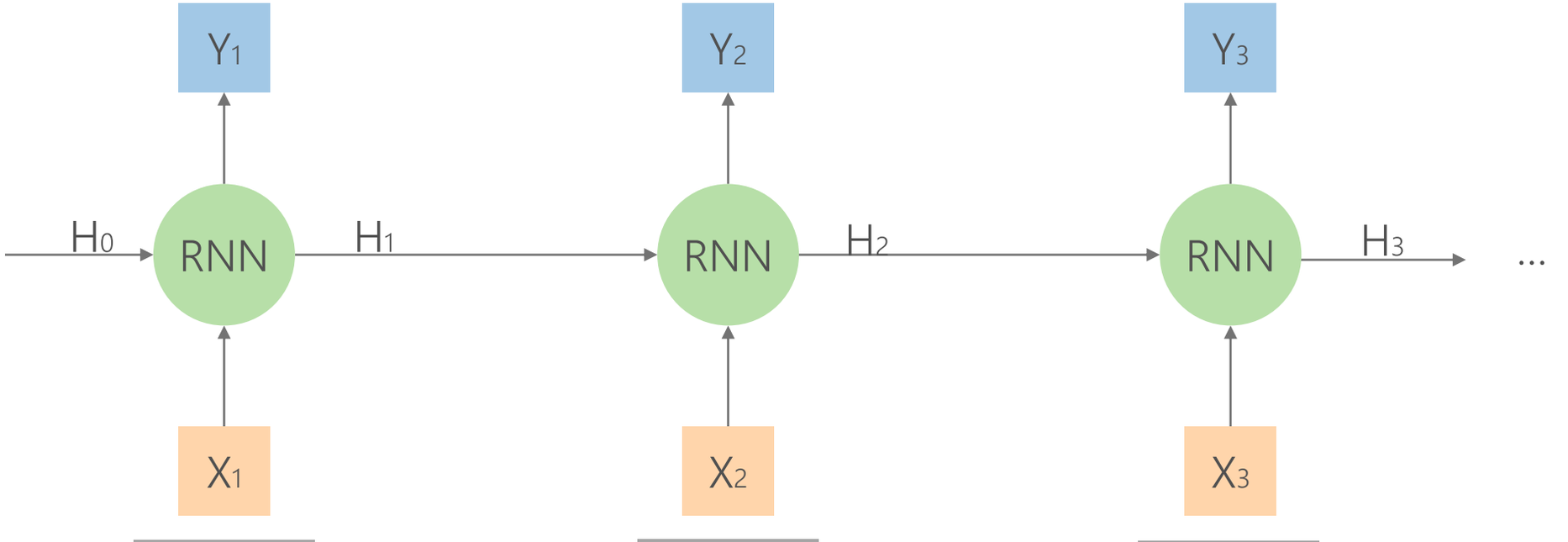

*In Keras you will see "timesteps" when set data input shape*

# Unrolled RNN

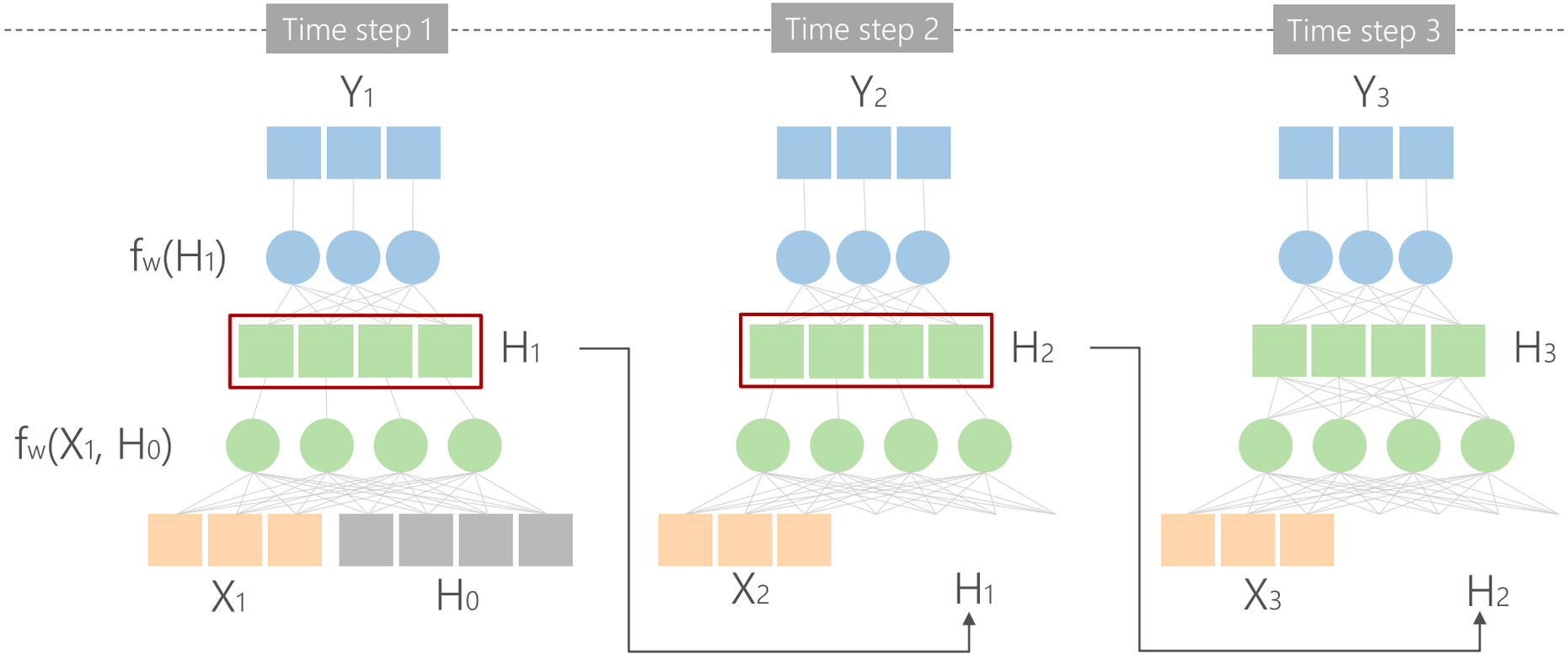



=

Simplified



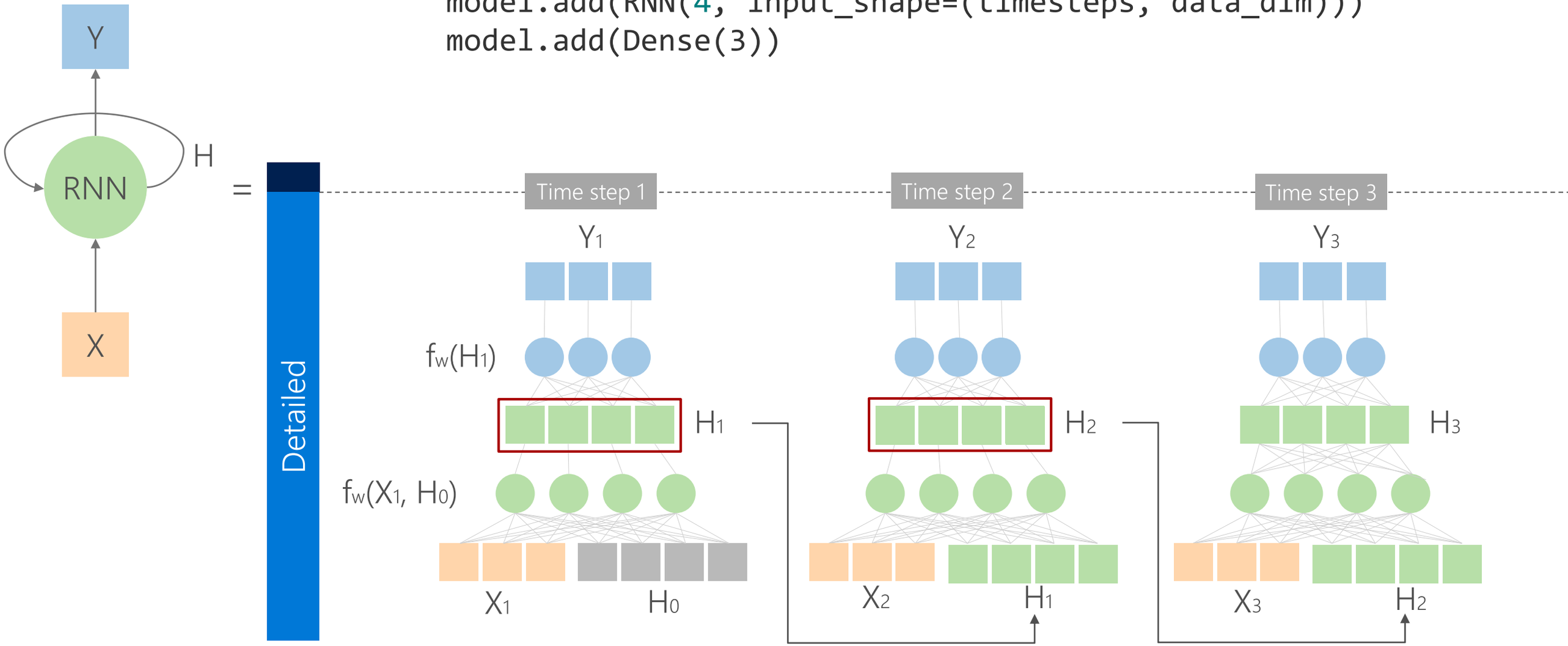

Detailed



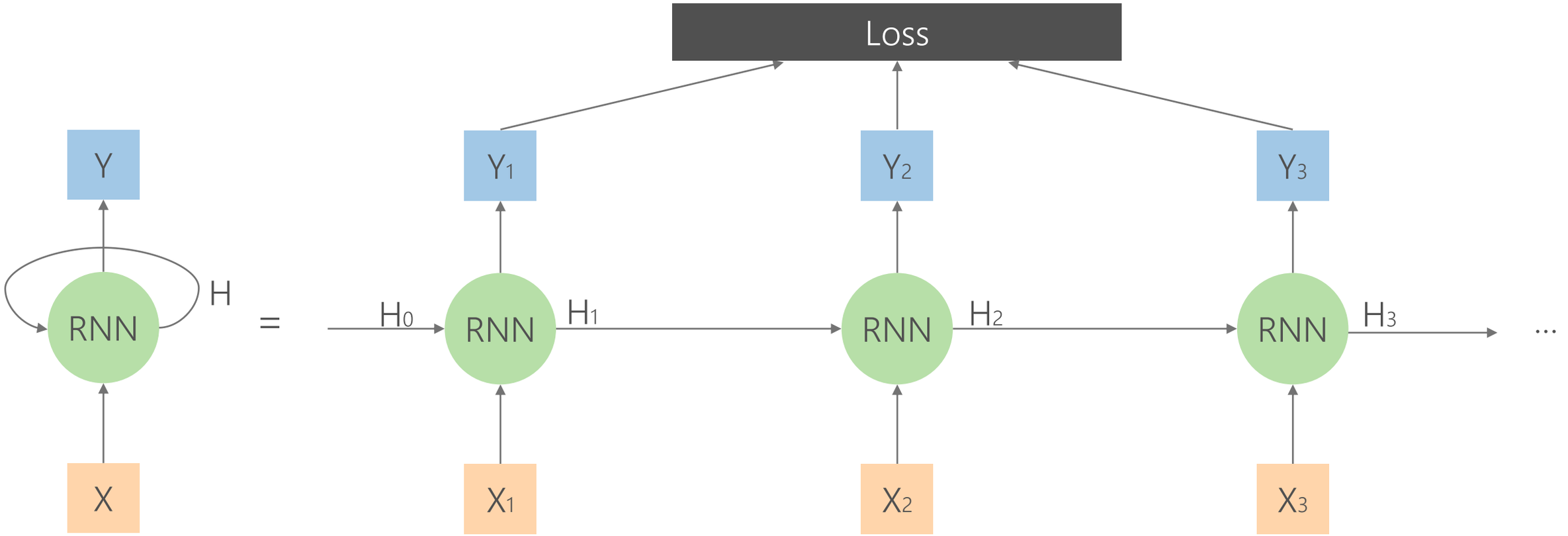
# Unrolled RNN

*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))
```



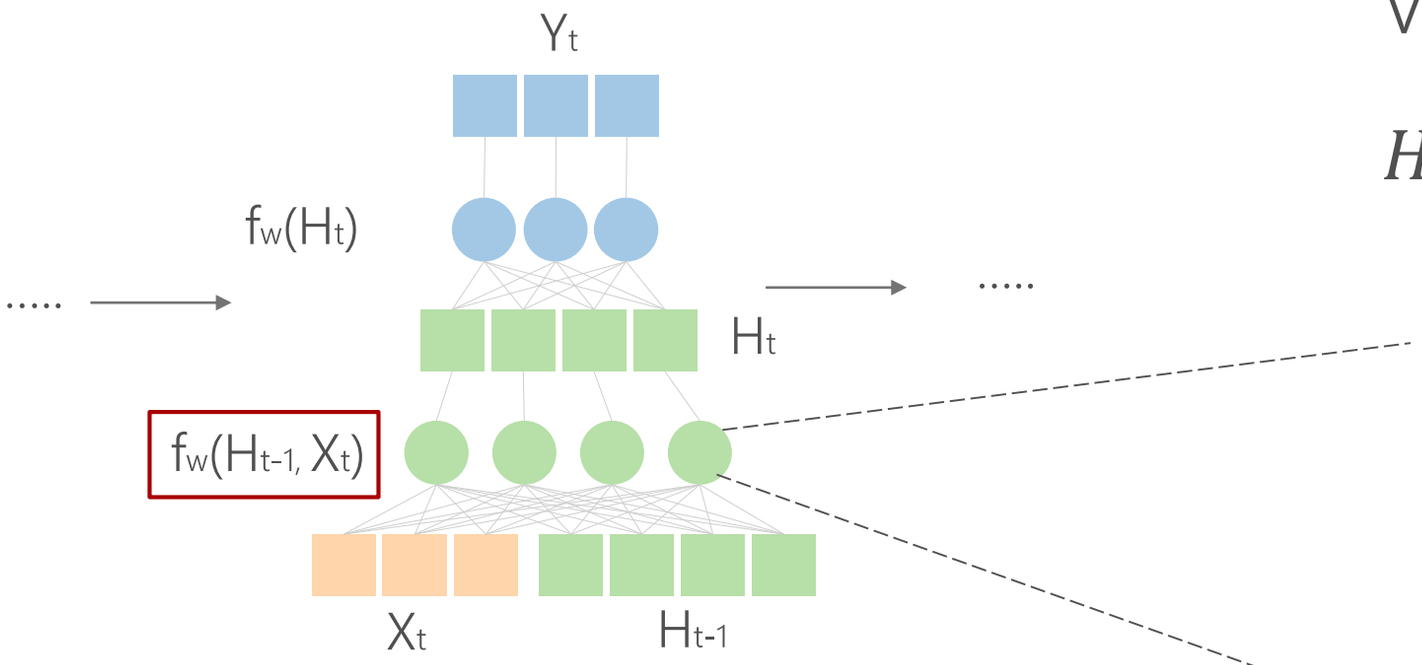
# Backpropagation through time (BPTT)




Forward through entire sequence to compute loss

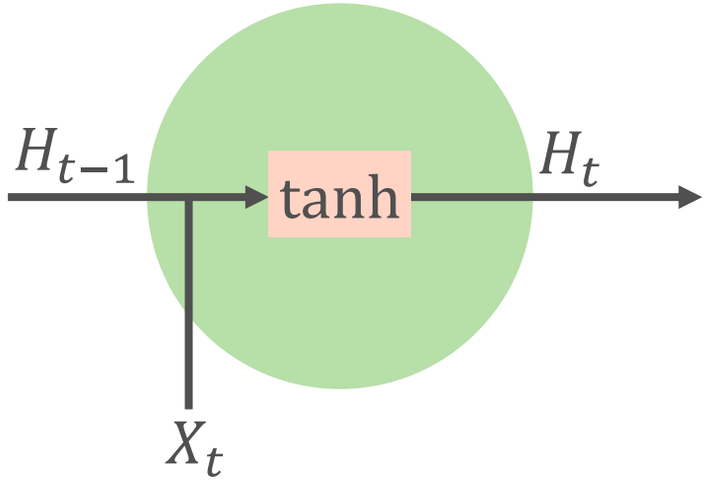
then backward through entire sequence to compute gradient

# Vanilla RNN




Vanilla RNN:

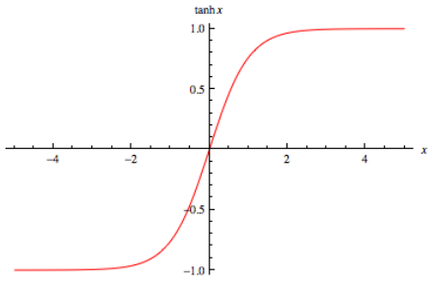
$$H_t = \tanh(W_h H_{t-1} + W_x X_t)$$
$$= \tanh(\mathbf{W} \cdot [H_{t-1}, X_t])$$



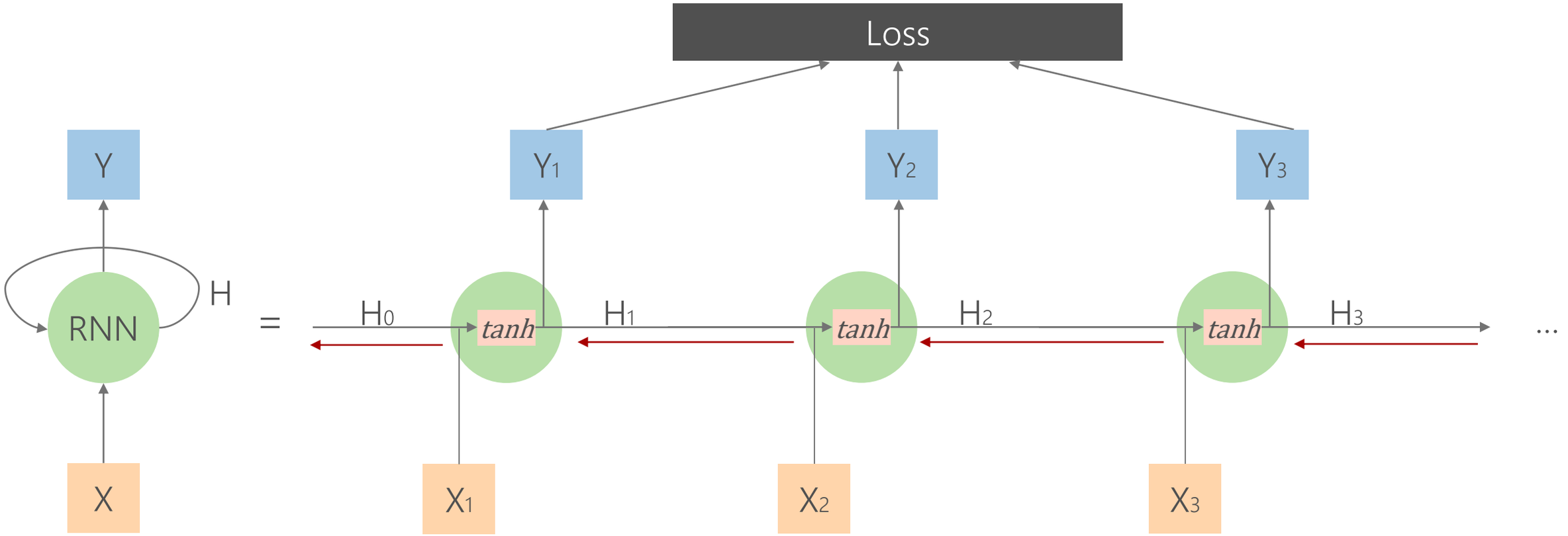

some function with parameter W

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

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



# Vanilla RNN BPTT

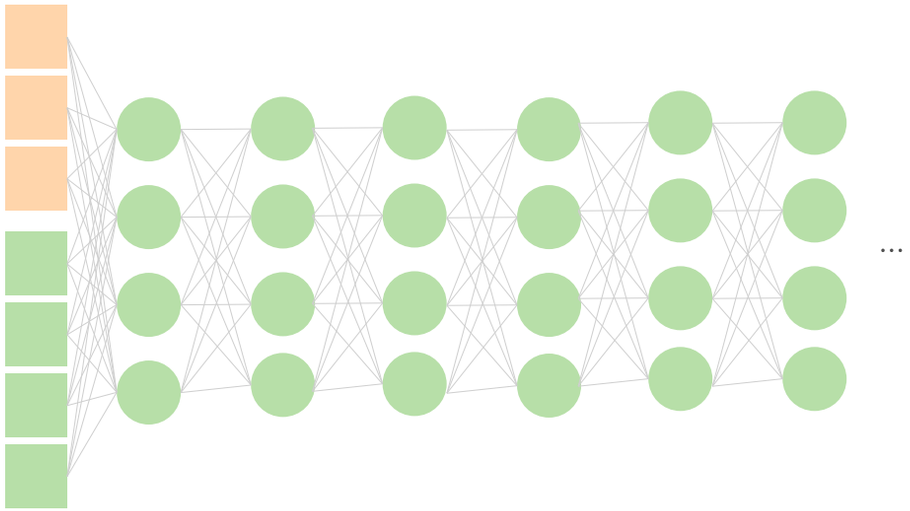



Computing gradient of $h_0$ involves **repeated tanh and many factors of W**

# 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

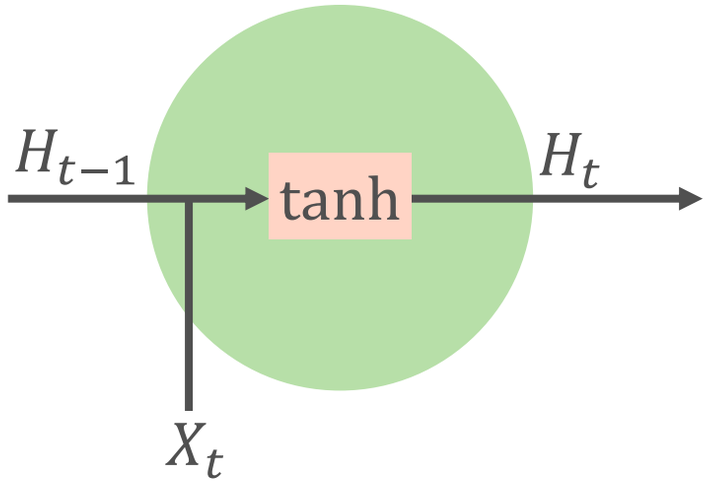
# Gated Recurrent Unit (GRU)

Vanilla RNN:

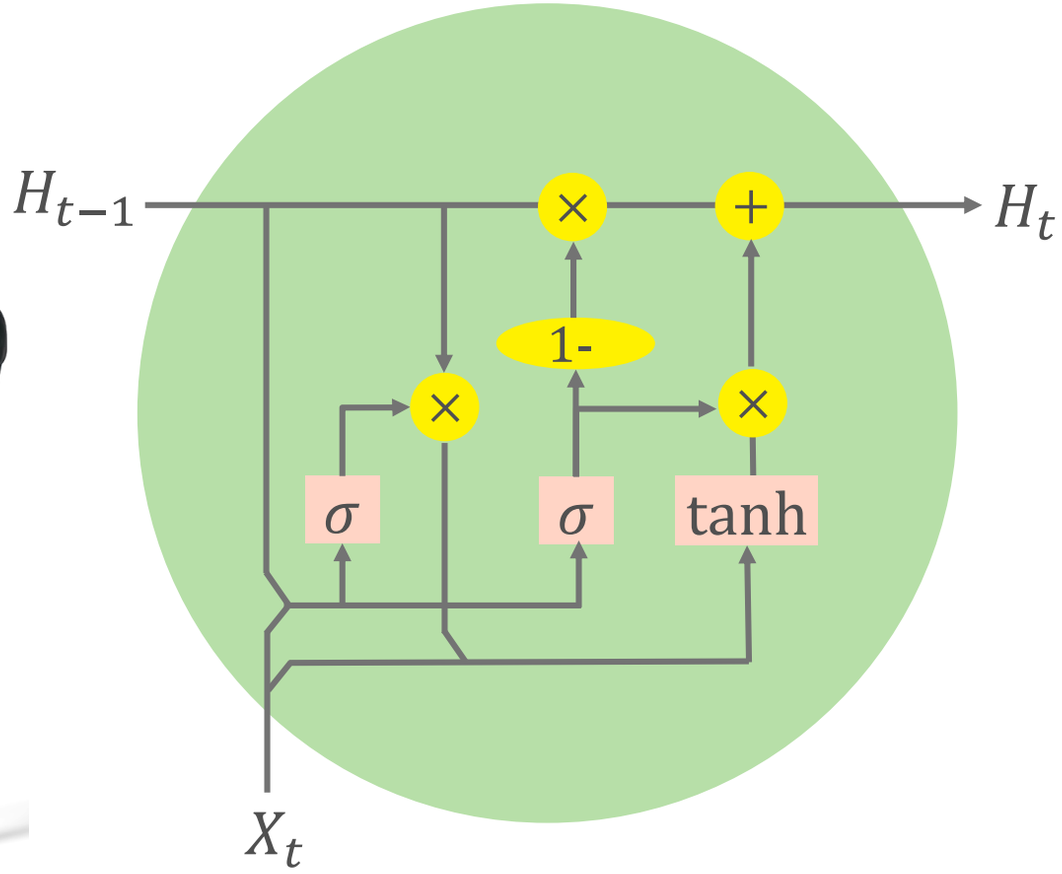
$$H_t = \tanh(\mathbf{W} \cdot [H_{t-1}, X_t])$$

GRU:

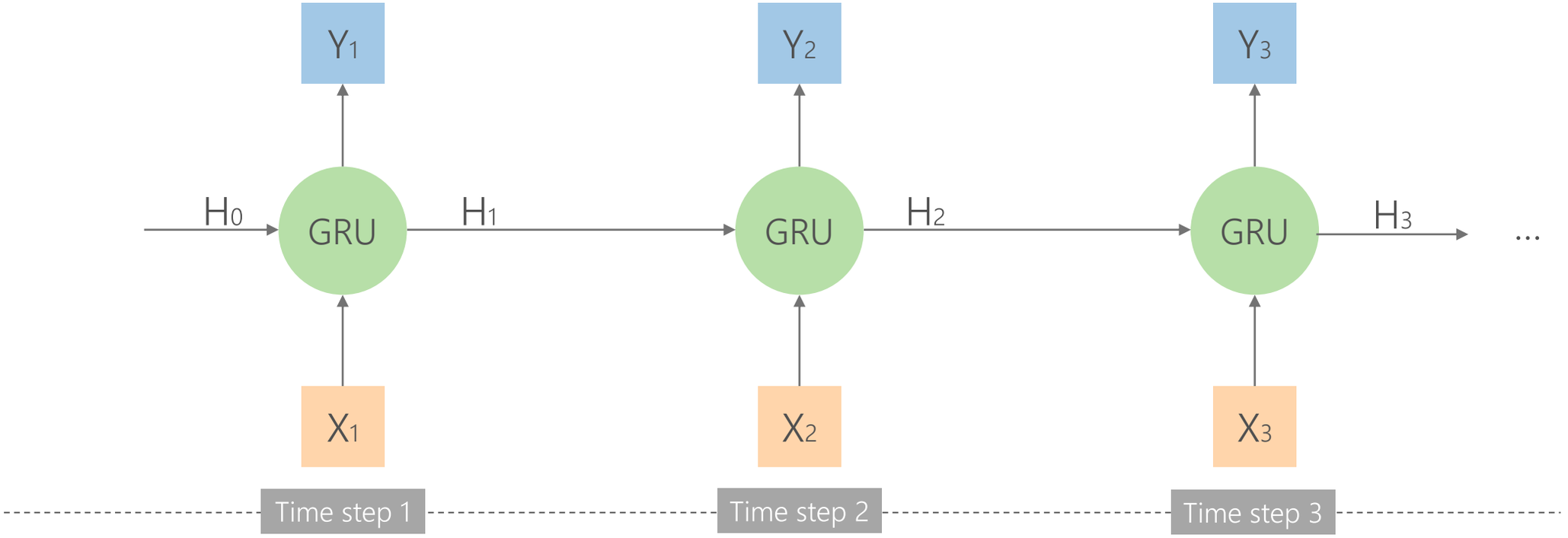
$$H_t = (1 - z_t) \times H_{t-1} + z_t \times \tilde{H}_t$$





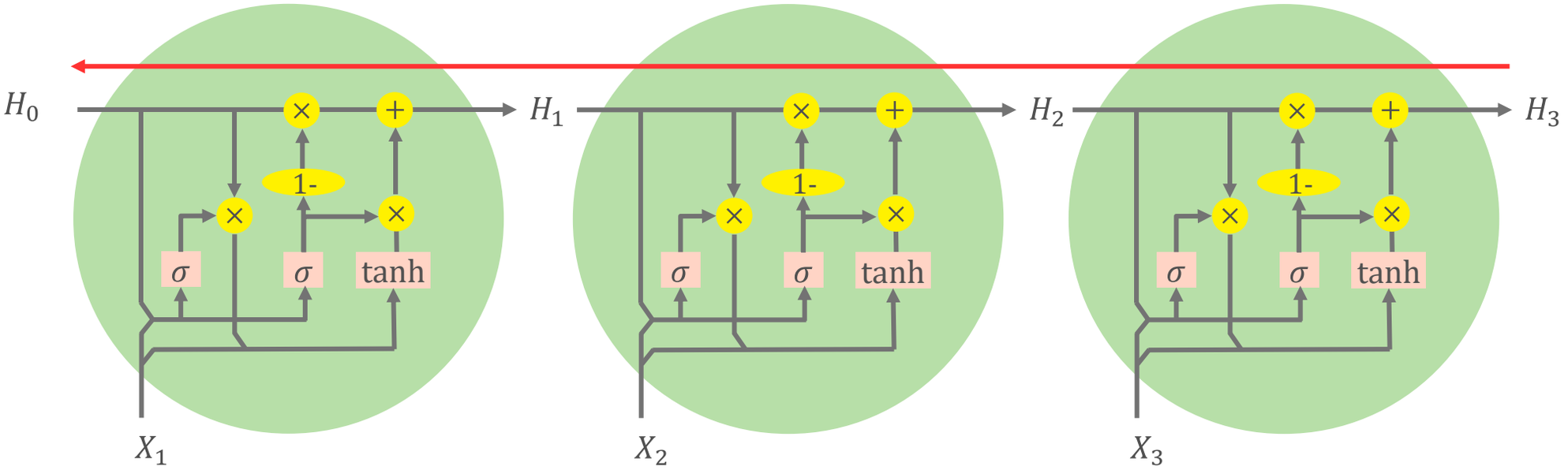


# GRU

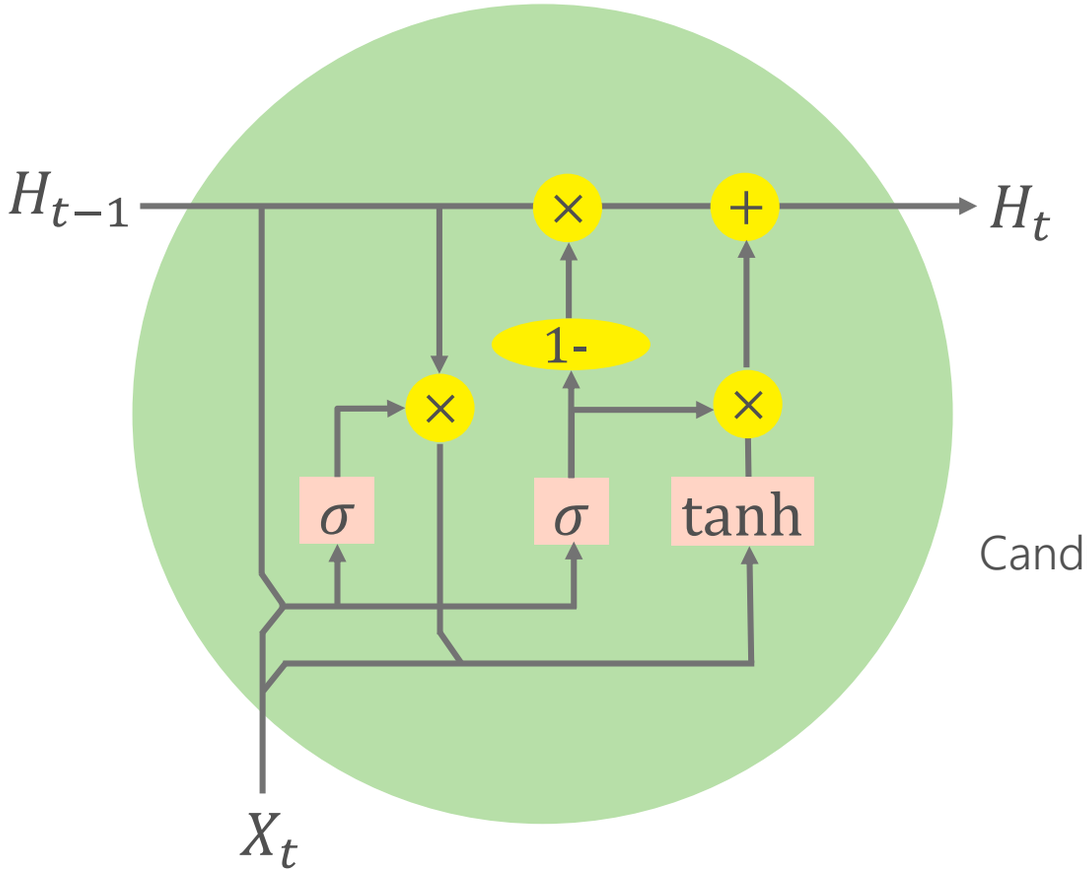



## Uninterrupted gradient flow

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



# Gated Recurrent Unit (GRU)




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

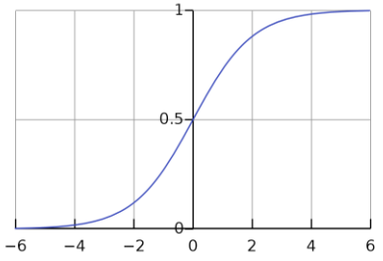
Update gates: $z_t = \sigma( W_z \cdot [H_{t-1}, X_t] )$

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

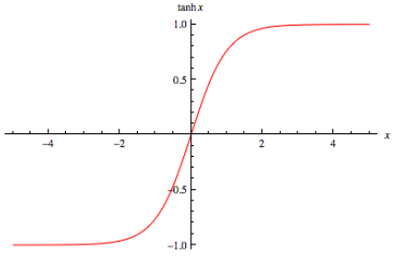
Reset gates: $r_t = \sigma( W_r \cdot [H_{t-1}, X_t] )$

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

$\sigma$
(0,1)

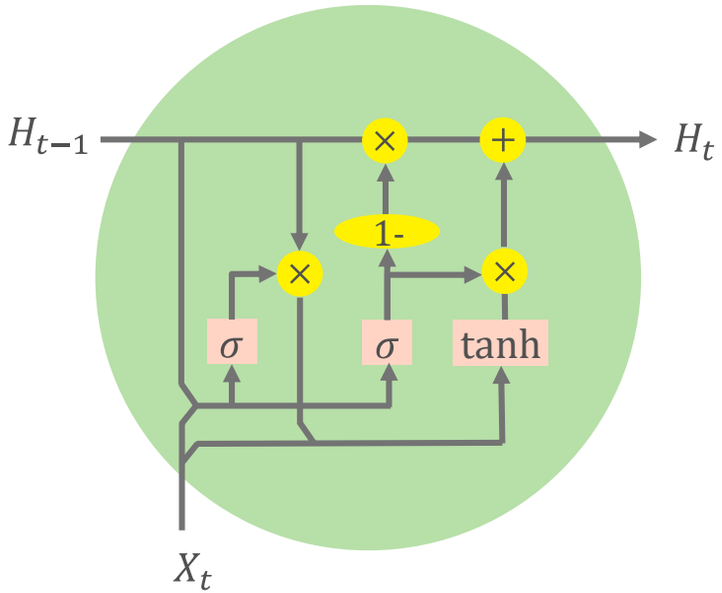



tanh
(-1, 1)

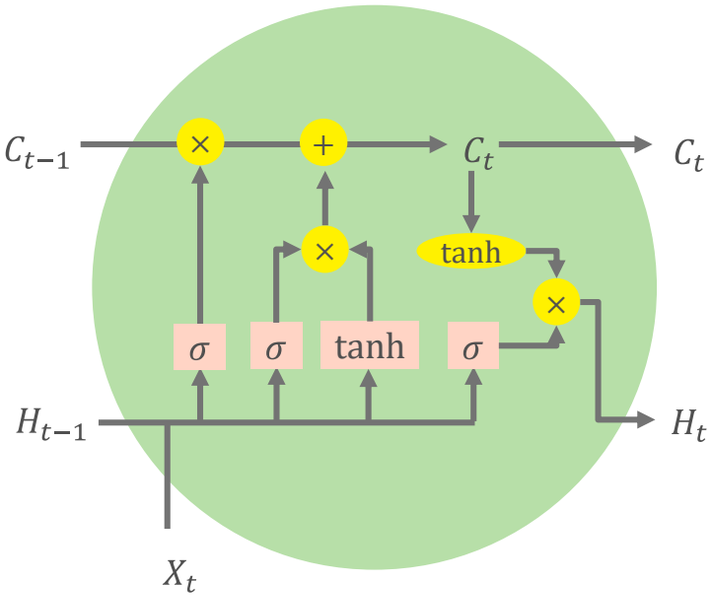


# GRU vs LSTM (Long Short Term Memory)

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

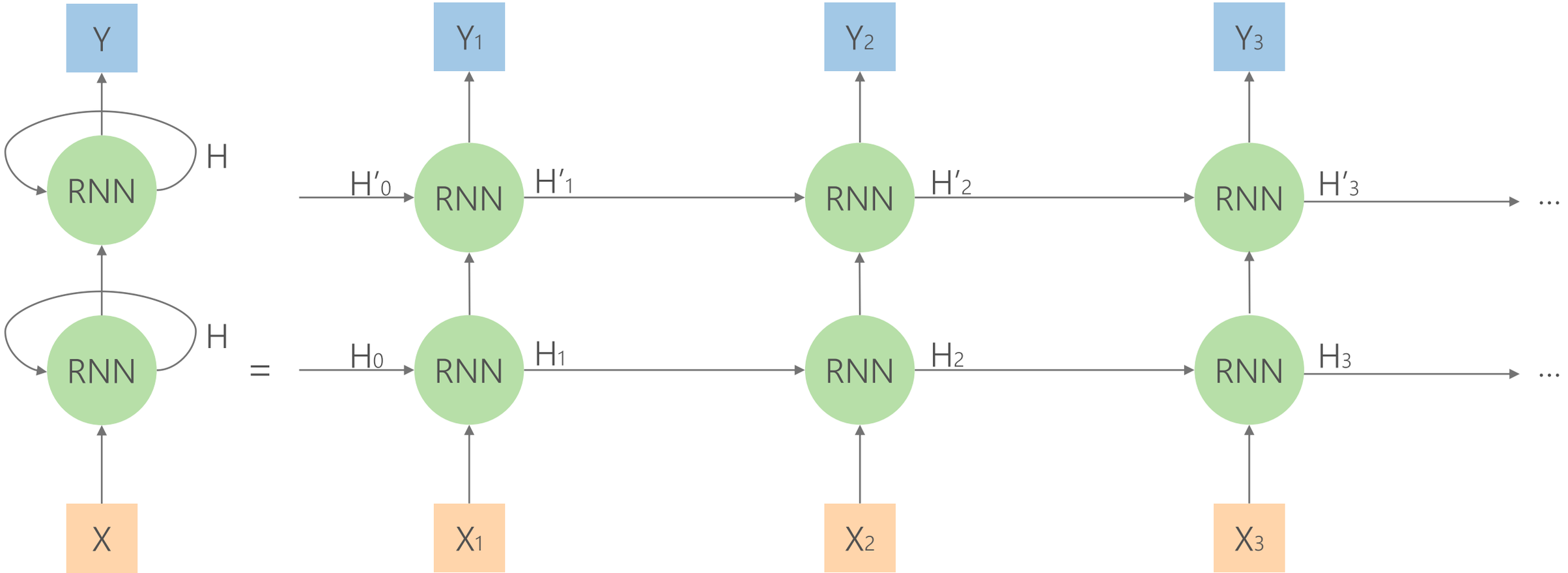
Output gates: $o_t = \sigma(\ W_o \cdot [H_{t-1}, X_t]\ )$

*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))
```

# Quiz

1. Find the error in the following code:

```
model = Sequential()
model.add(GRU(32, return_sequences=True, input_shape=(8, 16))) # input_shape=(timesteps, data_dim)
model.add(GRU(20))
model.add(GRU(12))
model.add(Dense(10))
```

2. How many layers of GRU we are stacking?
3. During BPTT, how many timesteps we need to go through?
4. What's the input data dimension?
5. What's the output data dimension?
6. What's the number of unit in the first RNN layer?

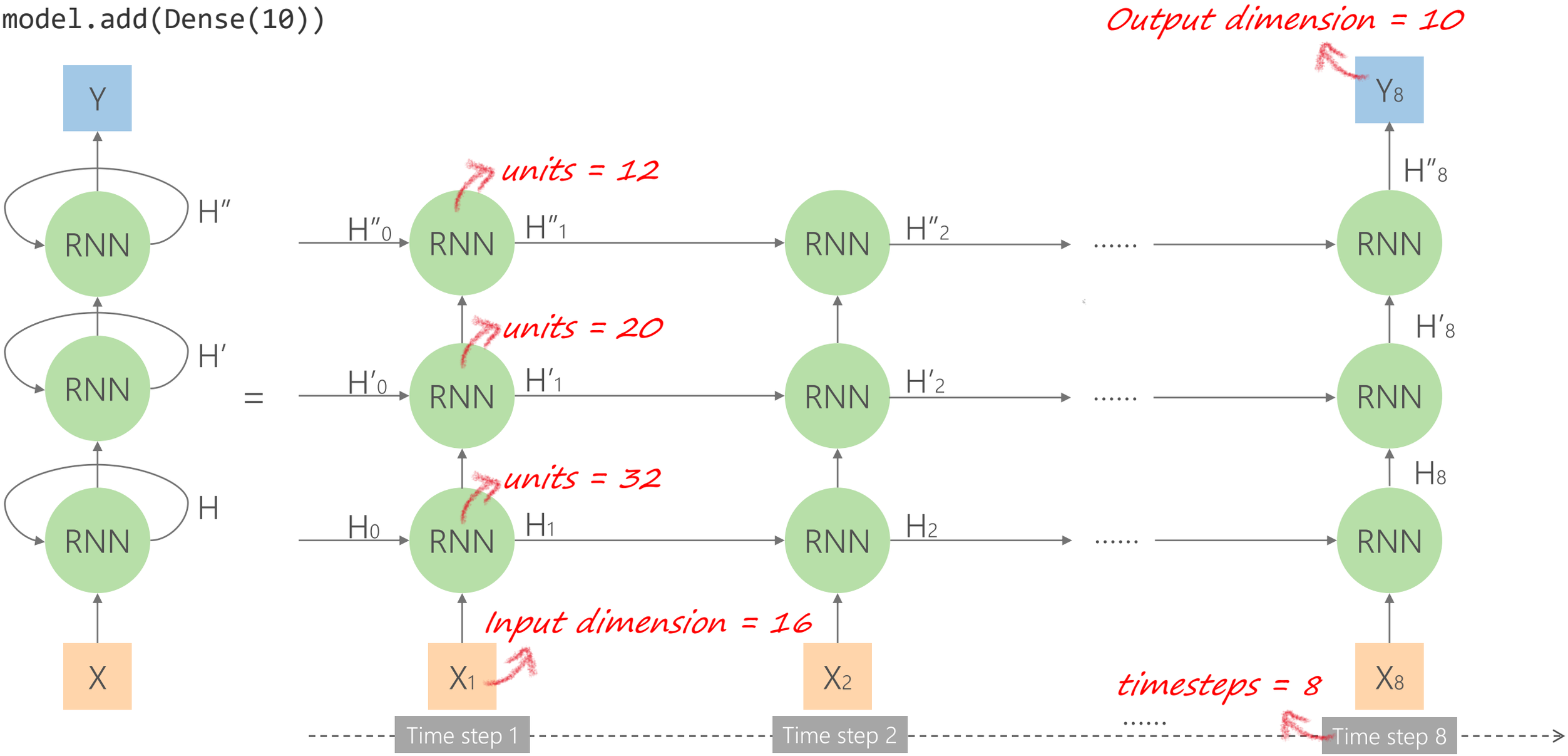
# Answer

1. Find the error in the following code:

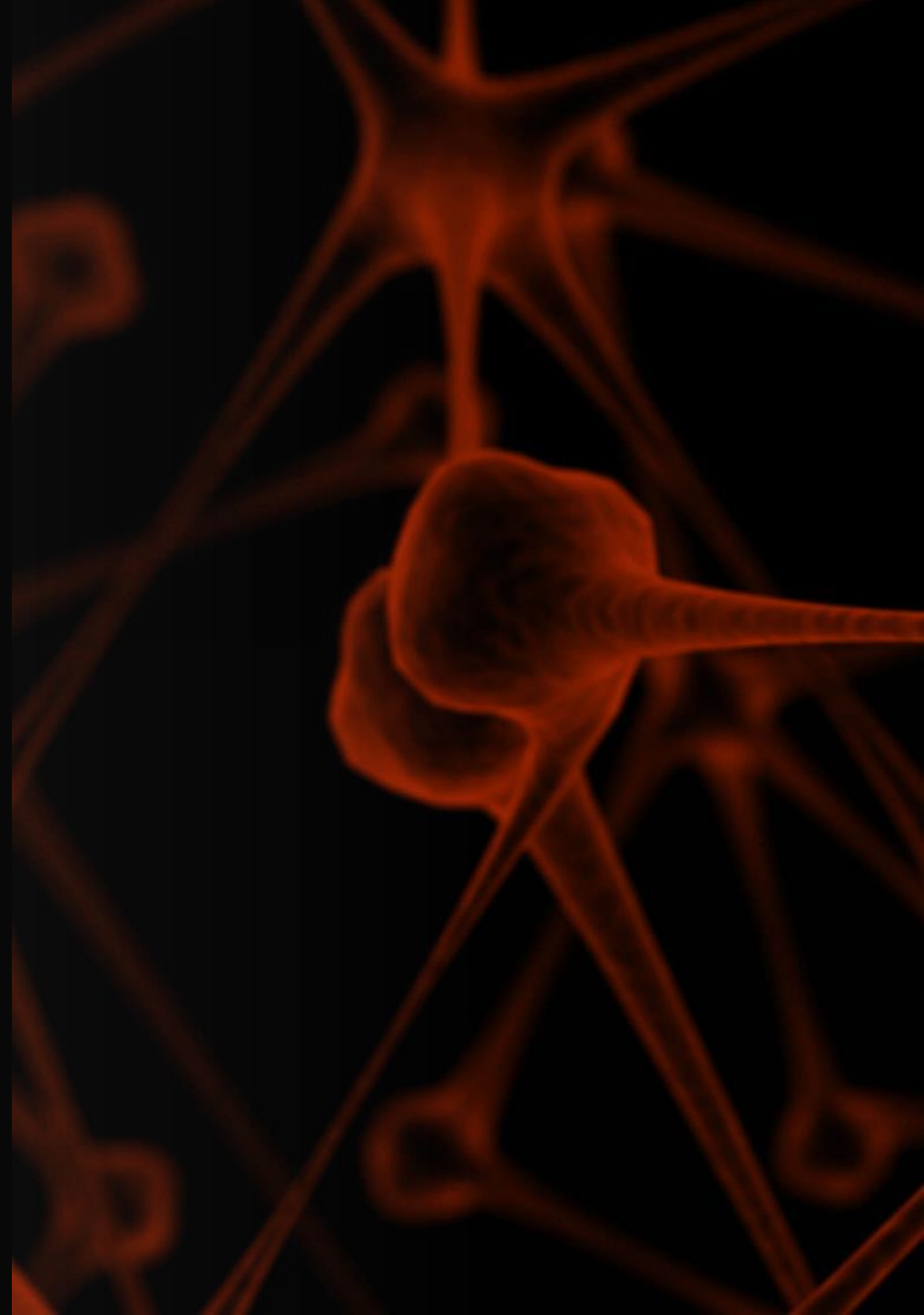
```
model = Sequential()
model.add(GRU(32, return_sequences=True, input_shape=(8, 16))) # input_shape=(timesteps, data_dim)
model.add(GRU(20, return_sequences=True))
model.add(GRU(12))
model.add(Dense(10))
```

2. How many layers of LSTM we are stacking? 3
3. During BPTT, how many timesteps we need to go through? 8
4. What's the input data dimension? 16
5. What's the output data dimension? 10
6. What's the number of unit in the first RNN layer? 32

```
model = Sequential()
model.add(GRU(32, return_sequences=True, input_shape=(8, 16))) # input_shape=(timesteps, data_dim)
model.add(GRU(20, return_sequences=True))
model.add(GRU(12))
model.add(Dense(10))
```



# How to apply RNNs to time series forecasting



# Agenda

- Scenario: Energy load forecasting
- Environment for experimentation
- Experimental setup
- Data preparation
- One-step forecast
- Multi-step forecast

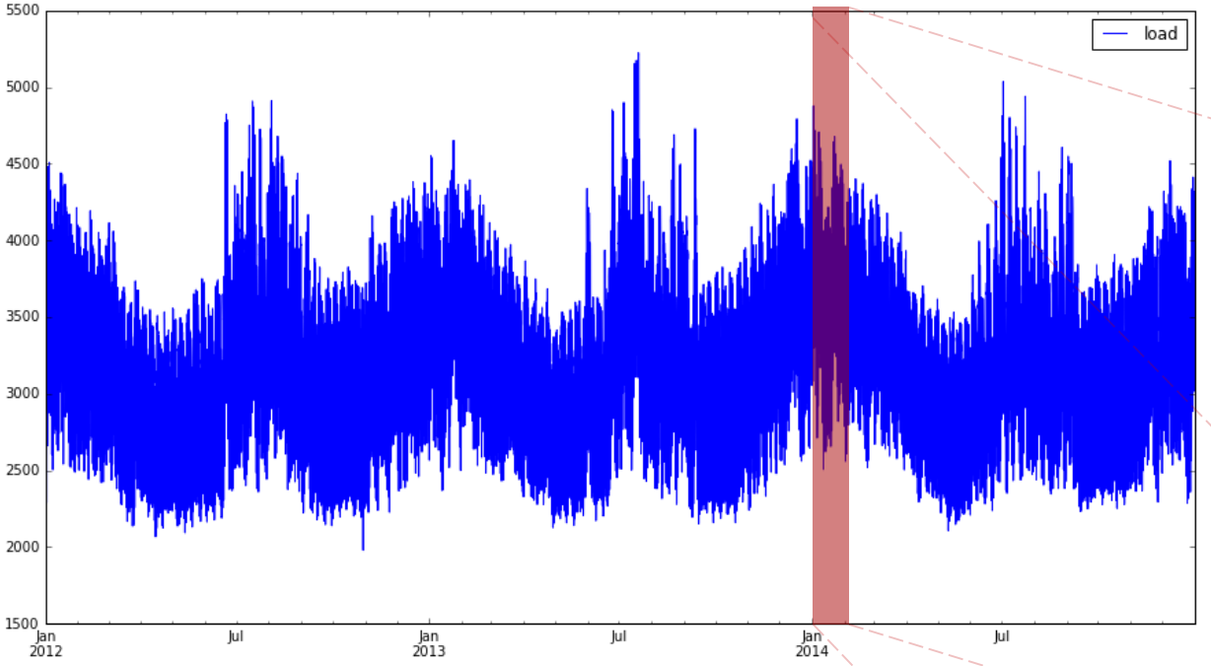
Code examples can be found at
https://github.com/yijingchen/RNNForTimeSeriesForecastTutorial

# Scenario:
# Energy load forecasting

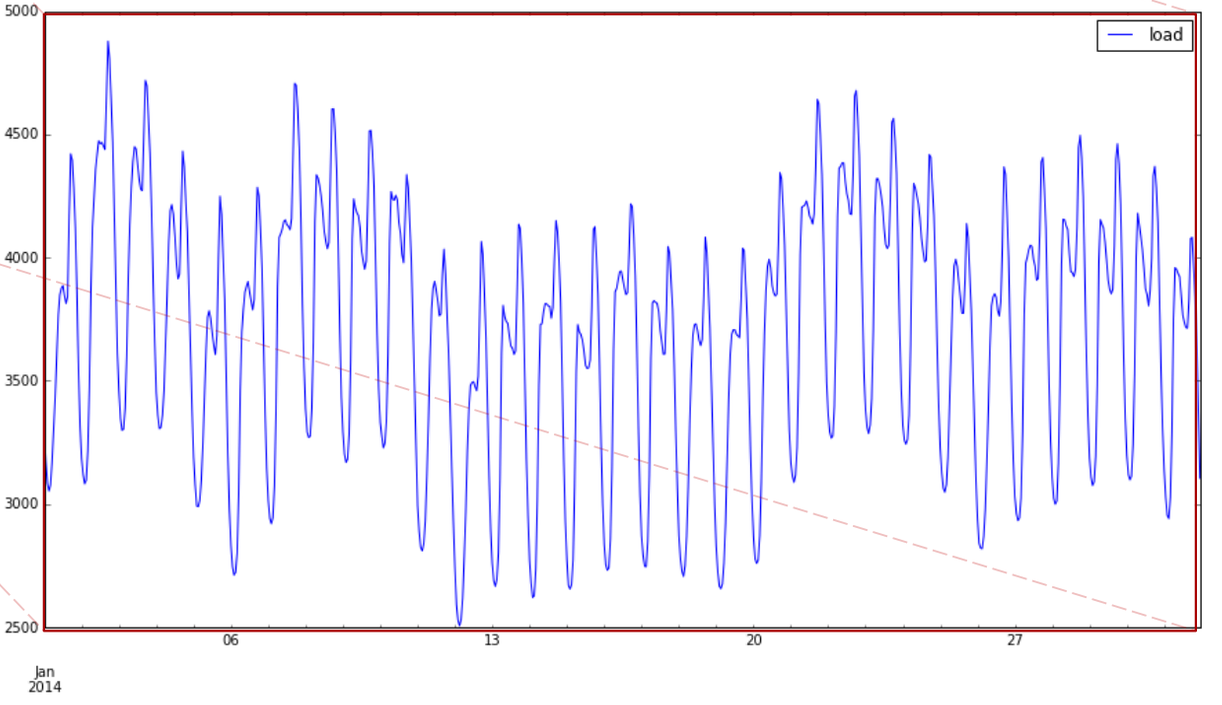
# Scenario: energy load forecasting

- Energy grid operators keep the supply and demand of electricity on power grids in balance
- They need short term demand forecasts to manage the supply of electricity on the grid
- Data in this example was used in the GEFCom2014 energy load forecasting competition

# New England ISO data

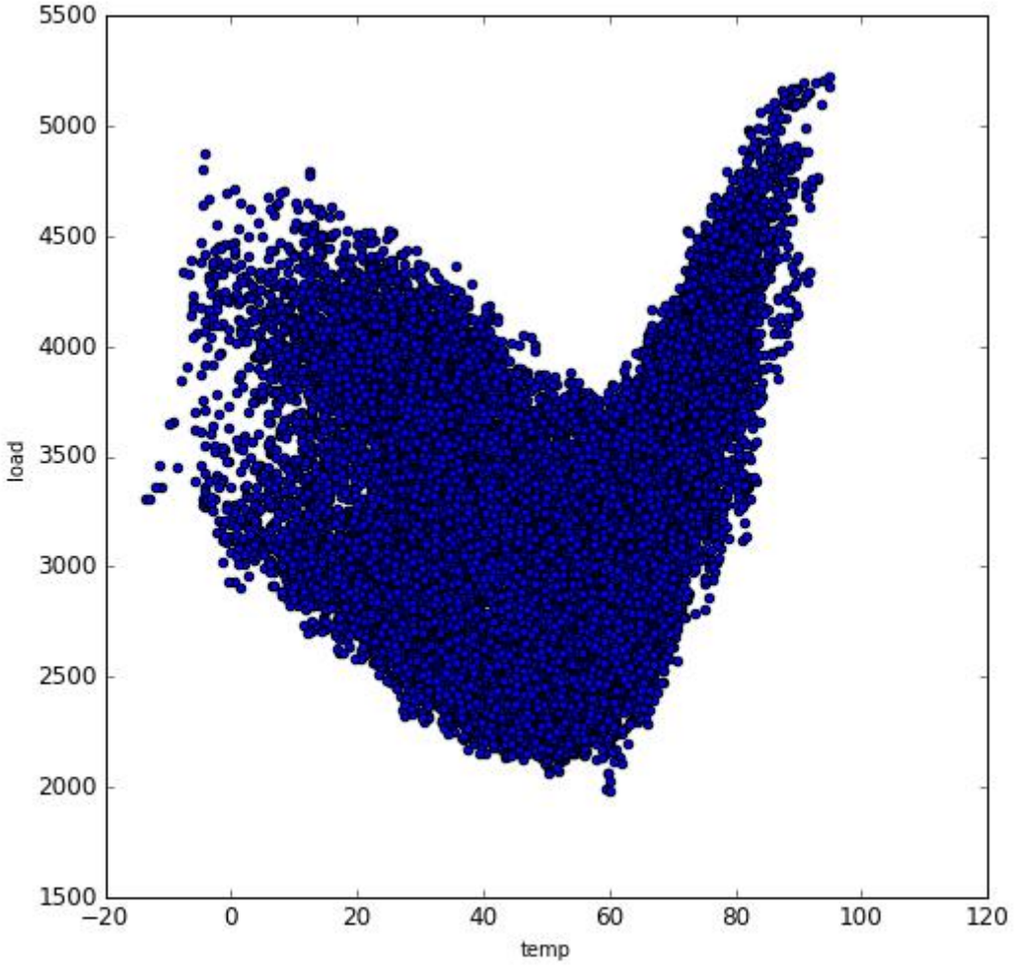



- 26,000 hourly load values
- Annual, weekly and daily seasonality




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.

# New England ISO data




- Load is also highly dependent on temperature

# Environment for experimentation

# Environment for experimentation

- RNNs can be extremely computationally complex
  - Multiple layers of RNN cells operating on multiple time steps can make RNN models very “deep”
  - Many hyperparameters and network structures to experiment with
- Possibly intensive data preparation and feature engineering
- Frequent model retraining, scoring and evaluation common in time series forecasting

# Hardware

- GPU required for serious work
- Deep Learning Virtual Machine
  - GPUs attached
  - scale as necessary
  - libraries preinstalled

# Software

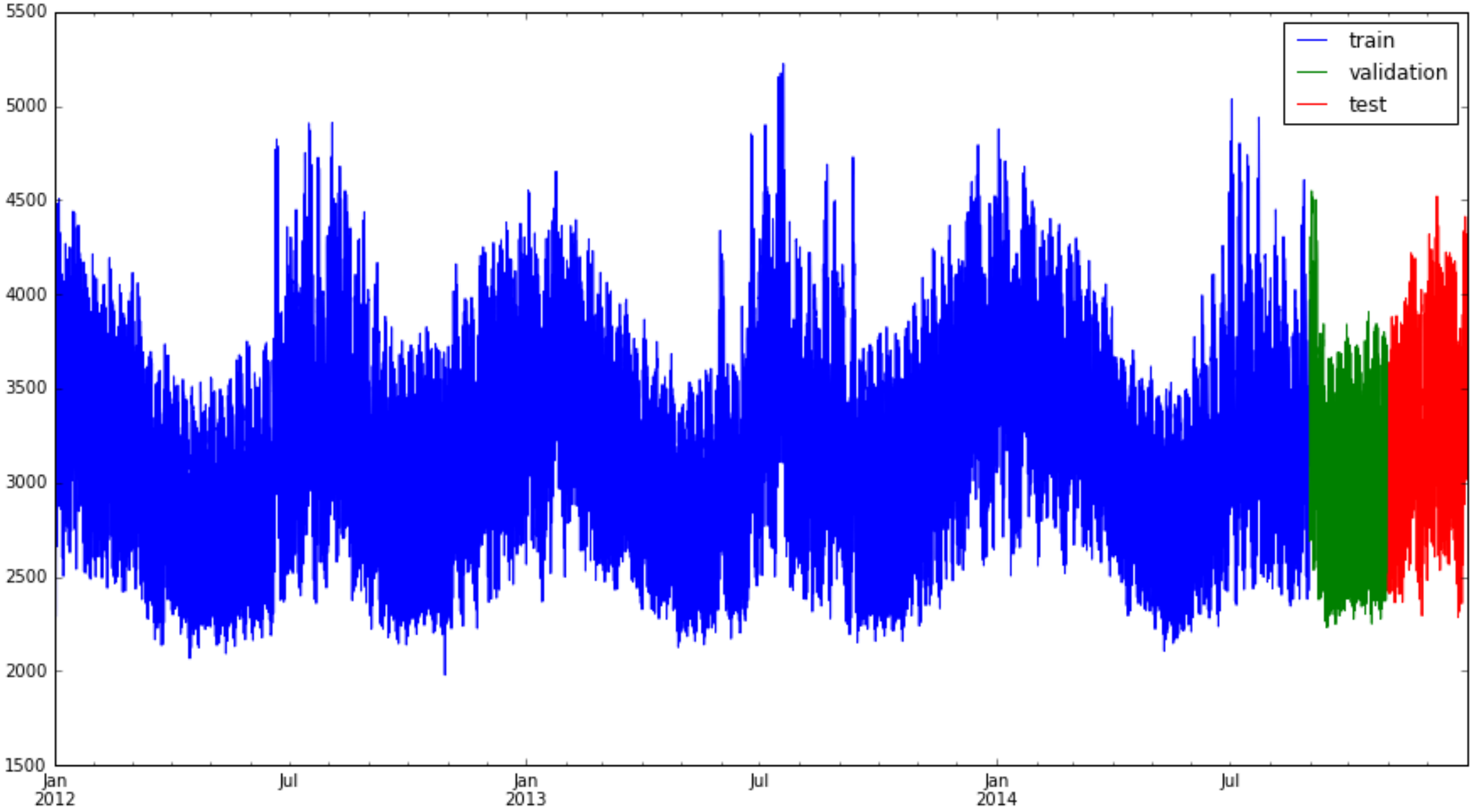
- Deep learning framework (e.g. Keras, Tensorflow, PyTorch)
- Python (recommended) or R
- CUDA
  - Nvidia GPU drivers
- cuDNN
  - Nvidia library for deep learning computation on GPU

# Why Keras?

- 👍 High-level API for deep learning
- 👍 Simplified syntax for common DL tasks
- 👍 Operates over efficient backends (e.g. TensorFlow, CNTK)
- 👎 Less control over low-level operations
- 👎 Sometimes not as efficient as other frameworks

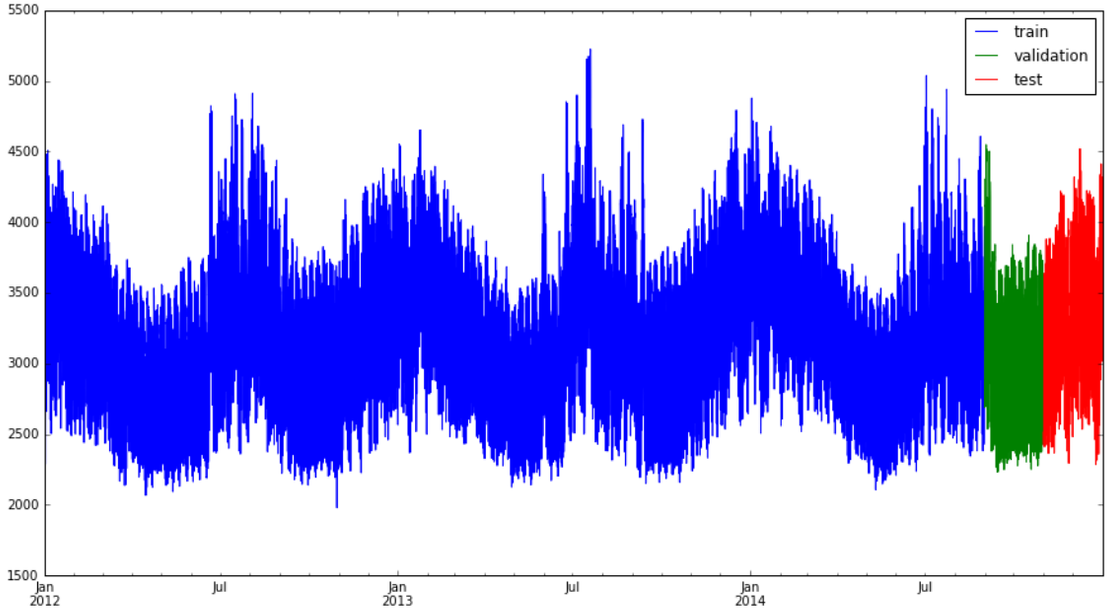
# Experimental setup

# Training, validation and test sets



# Training, validation and test sets

- **Train** – used for model training. Contains no information from the other datasets
- **Validation** – used for testing different model hyperparameters
- **Test** – used to evaluate the model and compare to others



# Data preparation

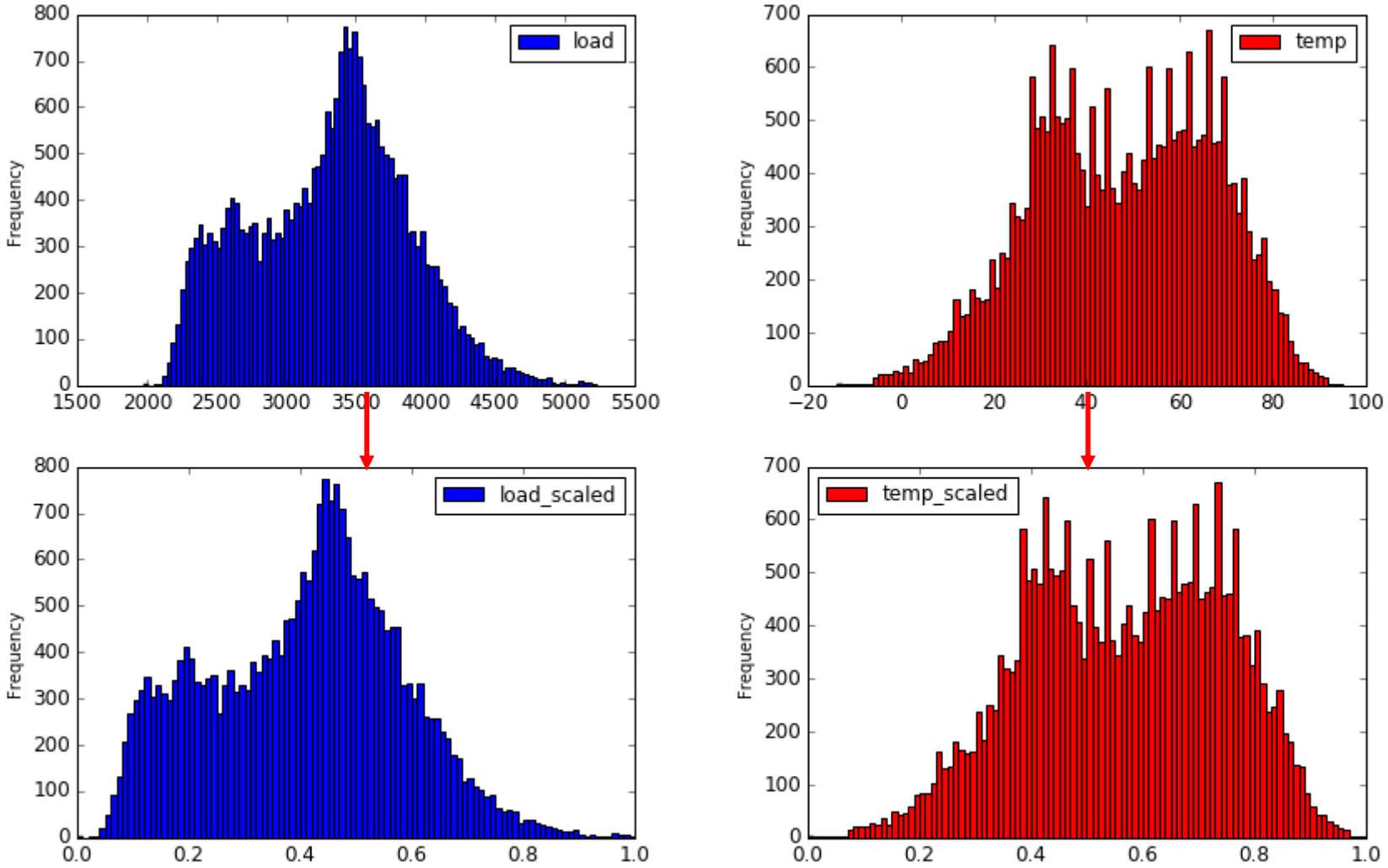
# Load data into Pandas dataframe

With data loaded in a Pandas dataframe you can:

- Index the data on timestamp for time-based filtering
- Visualize your data in tables with named columns
- Identify missing periods and missing values
- Create time-shifted variables

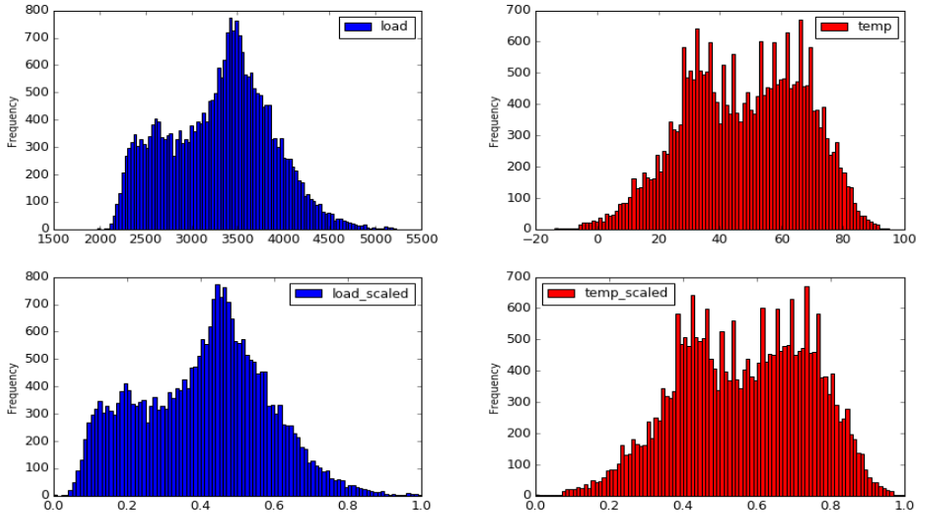
| | load | temp |
|---|---|---|
| 2012-01-01 00:00:00 | 2,698.00 | 32.00 |
| 2012-01-01 01:00:00 | 2,558.00 | 32.67 |
| 2012-01-01 02:00:00 | 2,444.00 | 30.00 |
| 2012-01-01 03:00:00 | 2,402.00 | 31.00 |
| 2012-01-01 04:00:00 | 2,403.00 | 32.00 |
| 2012-01-01 05:00:00 | 2,453.00 | 31.33 |
| 2012-01-01 06:00:00 | 2,560.00 | 30.00 |
| 2012-01-01 07:00:00 | 2,719.00 | 29.00 |
| 2012-01-01 08:00:00 | 2,916.00 | 29.00 |
| 2012-01-01 09:00:00 | 3,105.00 | 33.33 |

# Feature scaling



# Feature scaling

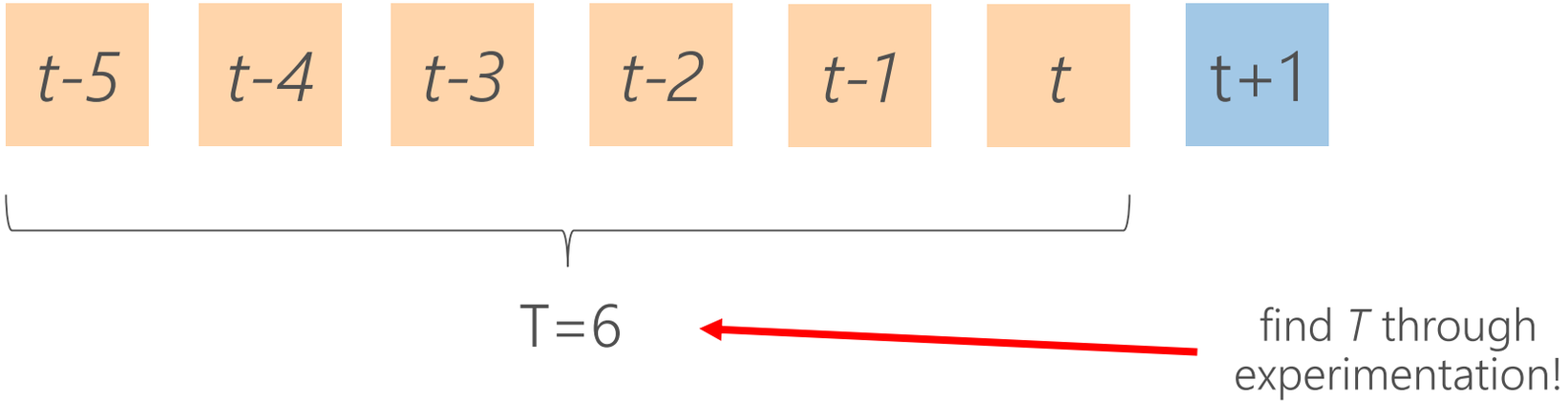
- Large values cause problems for optimization during training
- Optimizers are not scale invariant



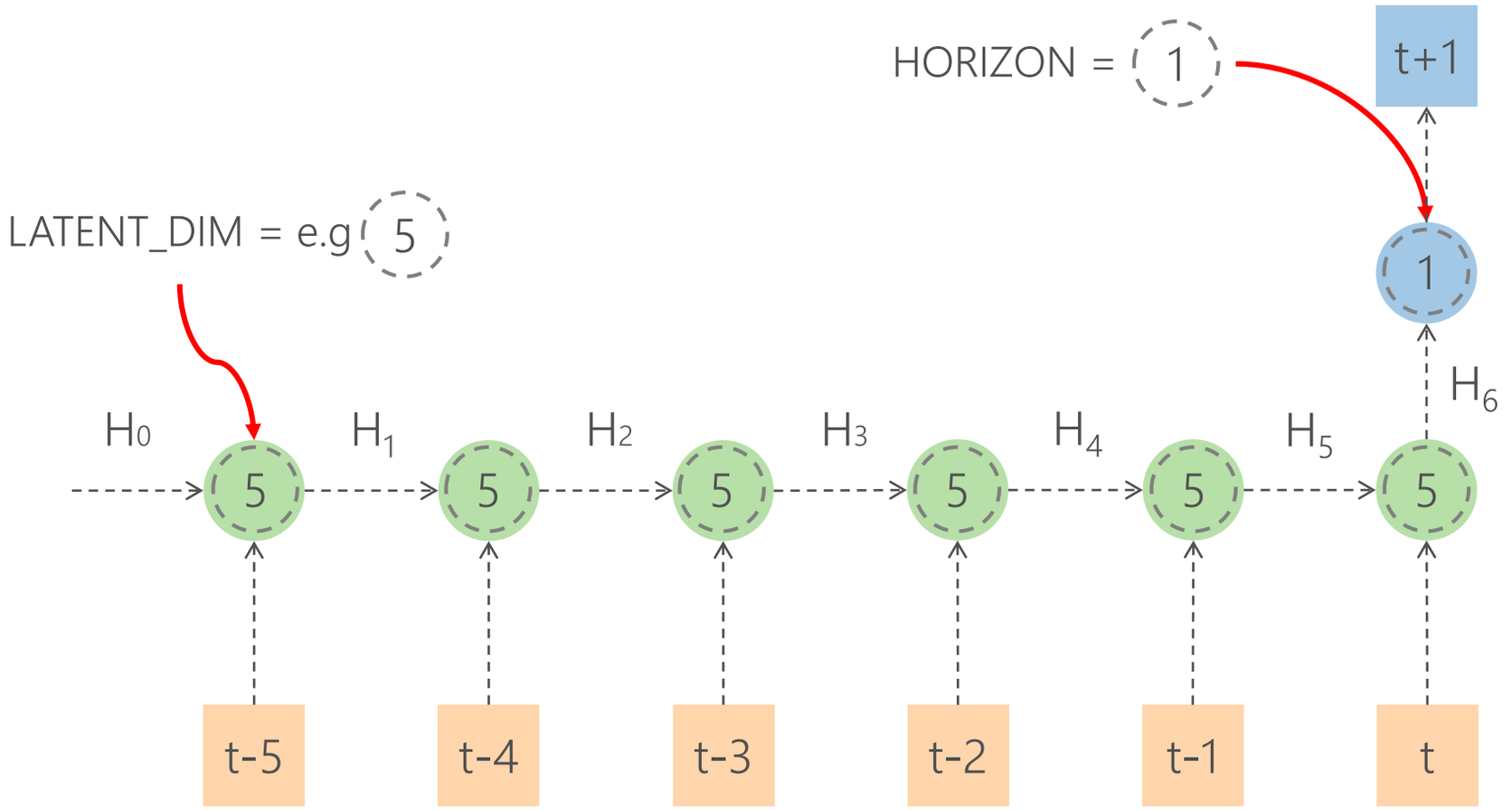
# One-step forecast

# 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



# Network structure




= dense layer

= recurrent layer

5 = number of units in the layer

# Format the data

- Time series needs to be transformed into two tensors:

| X | y |
| --- | --- |
| (samples, time steps, features) | (samples, horizon) |

# Format the data

- Time series needs to be transformed into two tensors:

| X | y |
| --- | --- |
| (samples, time steps, features) | (samples, horizon) |
| ( 23370 , 6 , 1 ) | ( 23370 , 1 ) |

# Format the data

- Step 1: shift *load* backwards to create target variable ($y_{t+1}$):

assume dataframe is indexed on time *t*, the time the prediction is made

| | load | y_t+1 |
|---|---|---|
| 2012-01-01 00:00:00 | 0.22 | 0.18 |
| 2012-01-01 01:00:00 | 0.18 | 0.14 |
| 2012-01-01 02:00:00 | 0.14 | 0.13 |
| 2012-01-01 03:00:00 | 0.13 | 0.13 |
| 2012-01-01 04:00:00 | 0.13 | 0.15 |
| 2012-01-01 05:00:00 | 0.15 | 0.18 |
| 2012-01-01 06:00:00 | 0.18 | 0.23 |
| 2012-01-01 07:00:00 | 0.23 | 0.29 |
| 2012-01-01 08:00:00 | 0.29 | 0.35 |
| 2012-01-01 09:00:00 | 0.35 | 0.37 |

# Format the data

- Step 2: shift *load* forwards *T* times to create feature ($load_{t-5}$, … , $load_t$)

| | load_original | y_t+1 | load_t-5 | load_t-4 | load_t-3 | load_t-2 | load_t-1 | load_t |
|---|---|---|---|---|---|---|---|---|
| **2012-01-01 00:00:00** | 0.22 | 0.18 | nan | nan | nan | nan | nan | 0.22 |
| **2012-01-01 01:00:00** | 0.18 | 0.14 | nan | nan | nan | nan | 0.22 | 0.18 |
| **2012-01-01 02:00:00** | 0.14 | 0.13 | nan | nan | nan | 0.22 | 0.18 | 0.14 |
| **2012-01-01 03:00:00** | 0.13 | 0.13 | nan | nan | 0.22 | 0.18 | 0.14 | 0.13 |
| **2012-01-01 04:00:00** | 0.13 | 0.15 | nan | 0.22 | 0.18 | 0.14 | 0.13 | 0.13 |
| 2012-01-01 05:00:00 | 0.15 | 0.18 | 0.22 | 0.18 | 0.14 | 0.13 | 0.13 | 0.15 |
| **2012-01-01 06:00:00** | 0.18 | 0.23 | 0.18 | 0.14 | 0.13 | 0.13 | 0.15 | 0.18 |
| **2012-01-01 07:00:00** | 0.23 | 0.29 | 0.14 | 0.13 | 0.13 | 0.15 | 0.18 | 0.23 |
| **2012-01-01 08:00:00** | 0.29 | 0.35 | 0.13 | 0.13 | 0.15 | 0.18 | 0.23 | 0.29 |
| **2012-01-01 09:00:00** | 0.35 | 0.37 | 0.13 | 0.15 | 0.18 | 0.23 | 0.29 | 0.35 |

# Format the data

- Step 3: remove incomplete samples

| | load_original | y_t+1 | load_t-5 | load_t-4 | load_t-3 | load_t-2 | load_t-1 | load_t |
|---|---|---|---|---|---|---|---|---|
| **2012-01-01 00:00:00** | 0.22 | 0.18 | nan | nan | nan | nan | nan | 0.22 |
| **2012-01-01 01:00:00** | 0.18 | 0.14 | nan | nan | nan | nan | 0.22 | 0.18 |
| **2012-01-01 02:00:00** | 0.14 | 0.13 | nan | nan | nan | 0.22 | 0.18 | 0.14 |
| **2012-01-01 03:00:00** | 0.13 | 0.13 | nan | nan | 0.22 | 0.18 | 0.14 | 0.13 |
| **2012-01-01 04:00:00** | 0.13 | 0.15 | nan | 0.22 | 0.18 | 0.14 | 0.13 | 0.13 |
| **2012-01-01 05:00:00** | 0.15 | 0.18 | 0.22 | 0.18 | 0.14 | 0.13 | 0.13 | 0.15 |
| **2012-01-01 06:00:00** | 0.18 | 0.23 | 0.18 | 0.14 | 0.13 | 0.13 | 0.15 | 0.18 |
| **2012-01-01 07:00:00** | 0.23 | 0.29 | 0.14 | 0.13 | 0.13 | 0.15 | 0.18 | 0.23 |
| **2012-01-01 08:00:00** | 0.29 | 0.35 | 0.13 | 0.13 | 0.15 | 0.18 | 0.23 | 0.29 |
| **2012-01-01 09:00:00** | 0.35 | 0.37 | 0.13 | 0.15 | 0.18 | 0.23 | 0.29 | 0.35 |

# Format the data

- Step 4: transpose into numpy arrays

| | load_original | y_t+1 | load_t-5 | load_t-4 | load_t-3 | load_t-2 | load_t-1 | load_t |
|---|---|---|---|---|---|---|---|---|
| 2012-01-01 05:00:00 | 0.15 | 0.18 | 0.22 | 0.18 | 0.14 | 0.13 | 0.13 | 0.15 |
| 2012-01-01 06:00:00 | 0.18 | 0.23 | 0.18 | 0.14 | 0.13 | 0.13 | 0.15 | 0.18 |
| 2012-01-01 07:00:00 | 0.23 | 0.29 | 0.14 | 0.13 | 0.13 | 0.15 | 0.18 | 0.23 |

y

```
array([[ 0.18],
       [ 0.23],
       [ 0.29]])
```

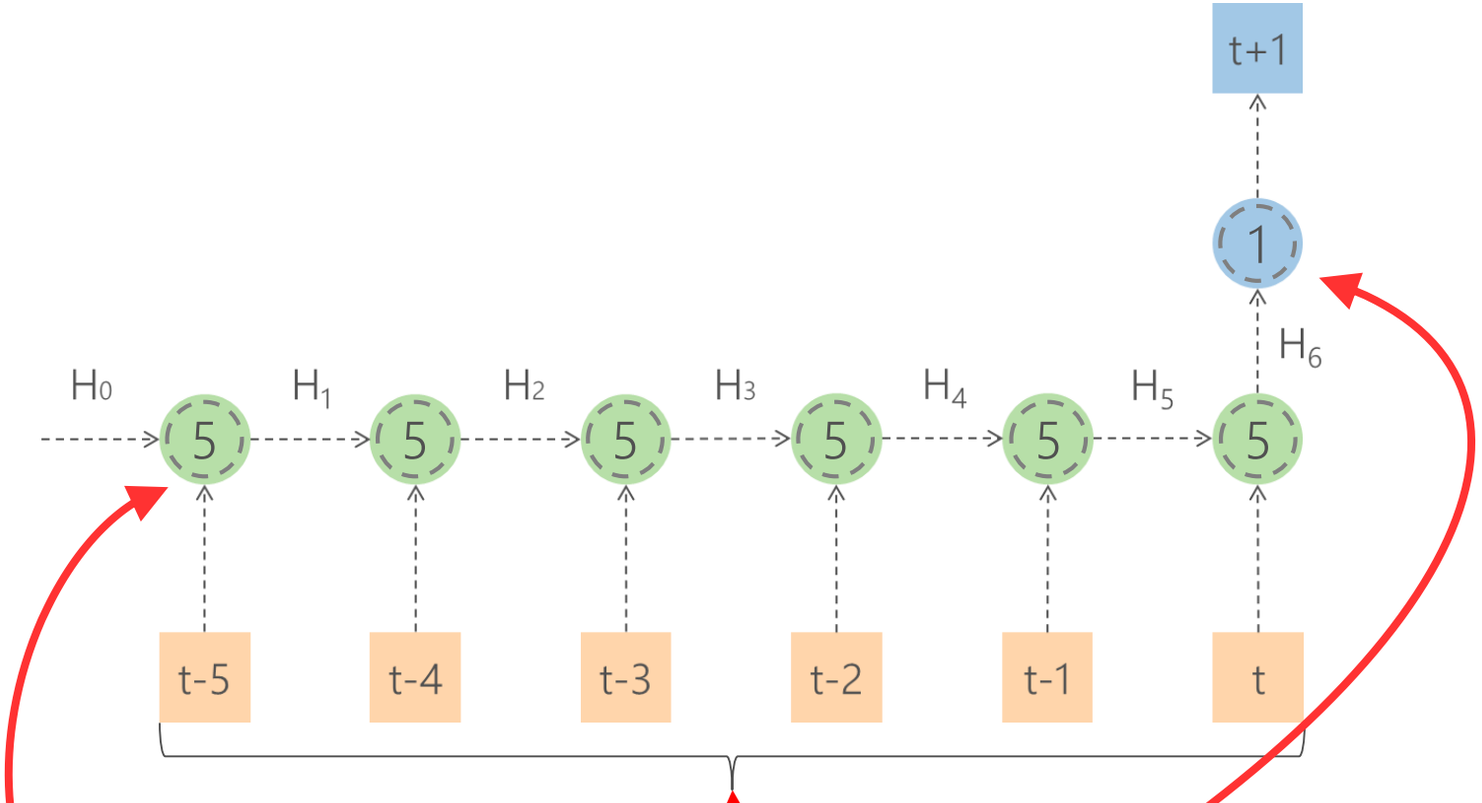
X

```
array([[[ 0.22],
        [ 0.18],
        [ 0.14],
        [ 0.13],
        [ 0.13],
        [ 0.15]],

       [[ 0.18],
        [ 0.14],
        [ 0.13],
        [ 0.13],
        [ 0.15],
        [ 0.18]],

       [[ 0.14],
        [ 0.13],
        [ 0.13],
        [ 0.15],
        [ 0.18],
        [ 0.23]]])
```

# Network implementation




choose RNN cell through experimentation!

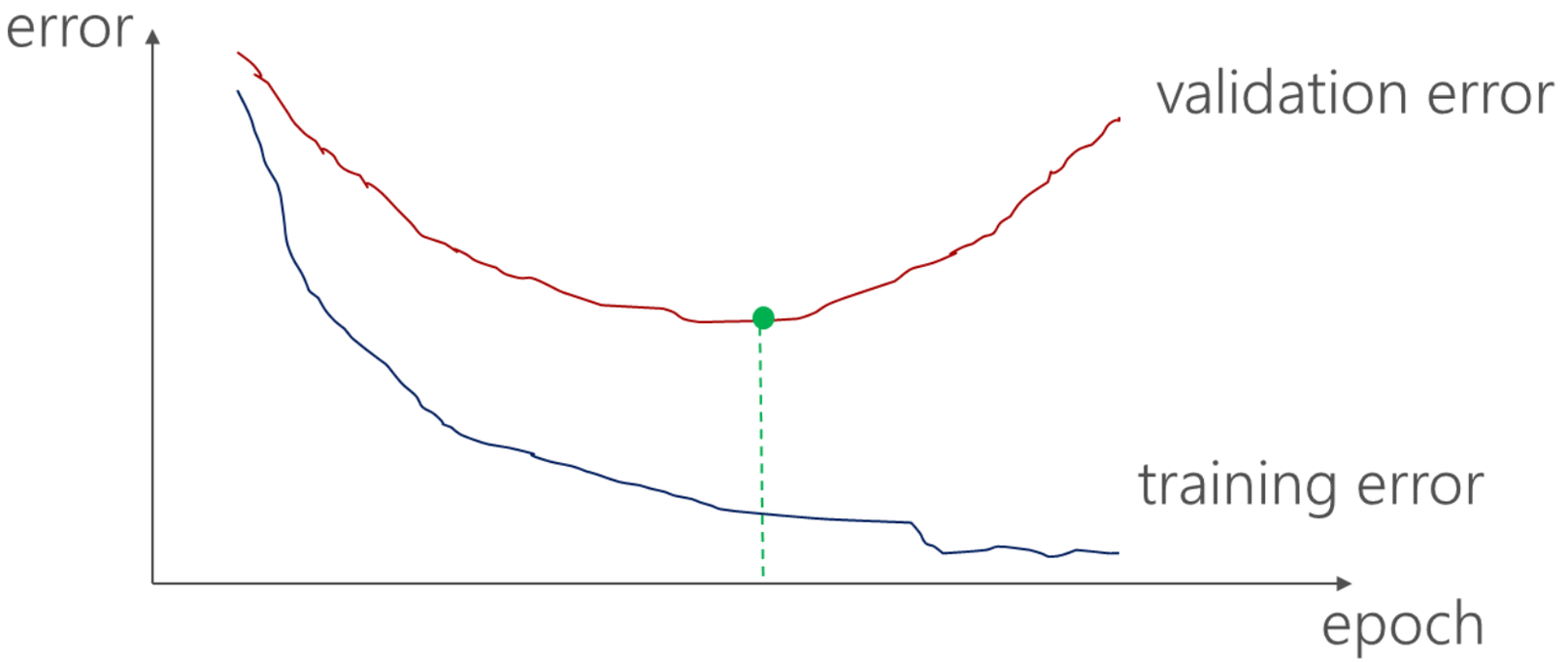
```
model = Sequential()
model.add(GRU(LATENT_DIM, input_shape=(6, 1)))
model.add(Dense(1, activation='linear'))
```

# Network compilation

- Compiling builds the network's static graph
- Ensures the input/output dimensions of consecutive layers are consistent
- Choose:
  - optimizer (e.g. stochastic gradient descent, Adam, RMSprop)
  - loss function (e.g. mean squared error)

```
model.compile(optimizer='sgd', loss='mse')
```

# Early stopping




```
earlystop = EarlyStopping(monitor='val_loss’,
                          min_delta=0,
                          patience=5)
```

# Fit the model

store metrics (e.g. training and validation loss after each epoch)

the maximum epochs, overridden by early stopping criteria

```
log = model.fit(X_train,
                y_train,
                batch_size=BATCH_SIZE,
                epochs=EPOCHS,
                validation_data=(X_valid, y_valid),
                callbacks=[earlystop],
                verbose=1)
```

check against early stopping criteria after each epoch

validation loss computed after each epoch

# Evaluation

1. Make predictions on the test set:

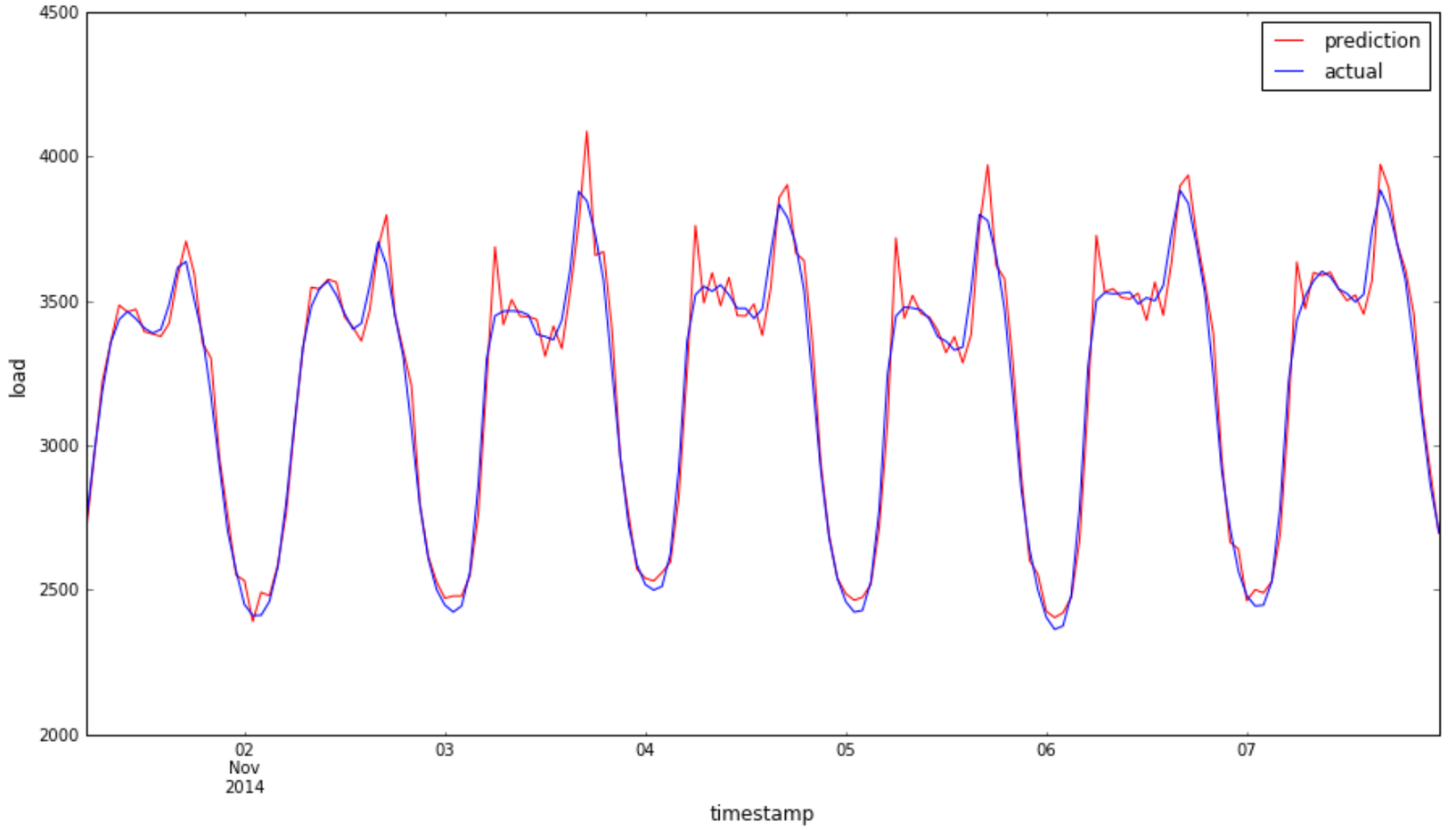
```
predictions = model.predict(X_test)
```

2. Compare to the actuals:

| | timestamp | h | prediction | actual |
|---|---|---|---|---|
| **0** | 2014-11-01 05:00:00 | t+1 | 2,693.81 | 2,714.00 |
| **1** | 2014-11-01 06:00:00 | t+1 | 2,951.02 | 2,970.00 |
| **2** | 2014-11-01 07:00:00 | t+1 | 3,184.23 | 3,189.00 |
| **3** | 2014-11-01 08:00:00 | t+1 | 3,308.91 | 3,356.00 |
| **4** | 2014-11-01 09:00:00 | t+1 | 3,452.54 | 3,436.00 |

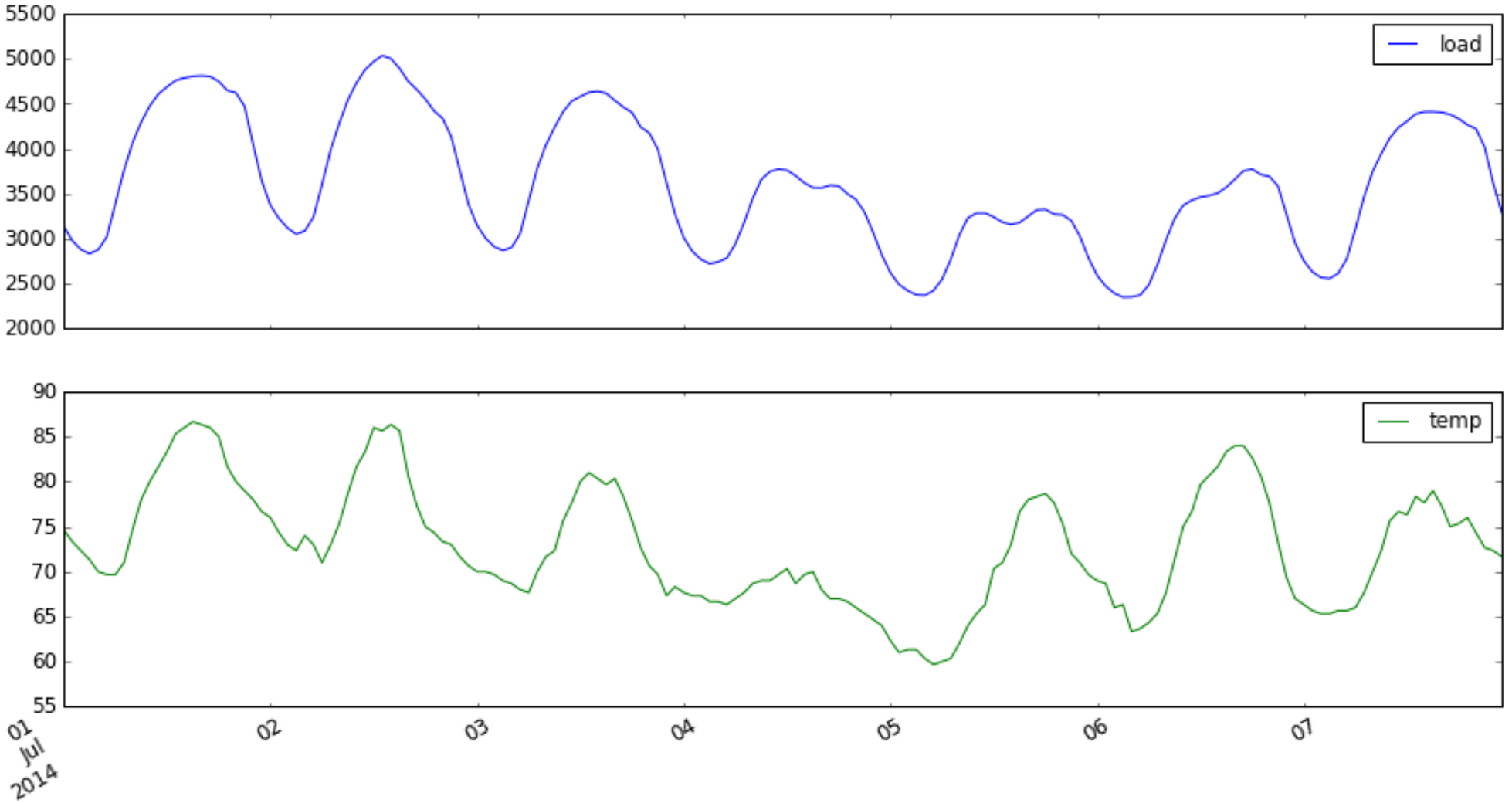
# Evaluation

- Perform “walk forward” evaluation
- Plot predictions vs actuals for first week of test set

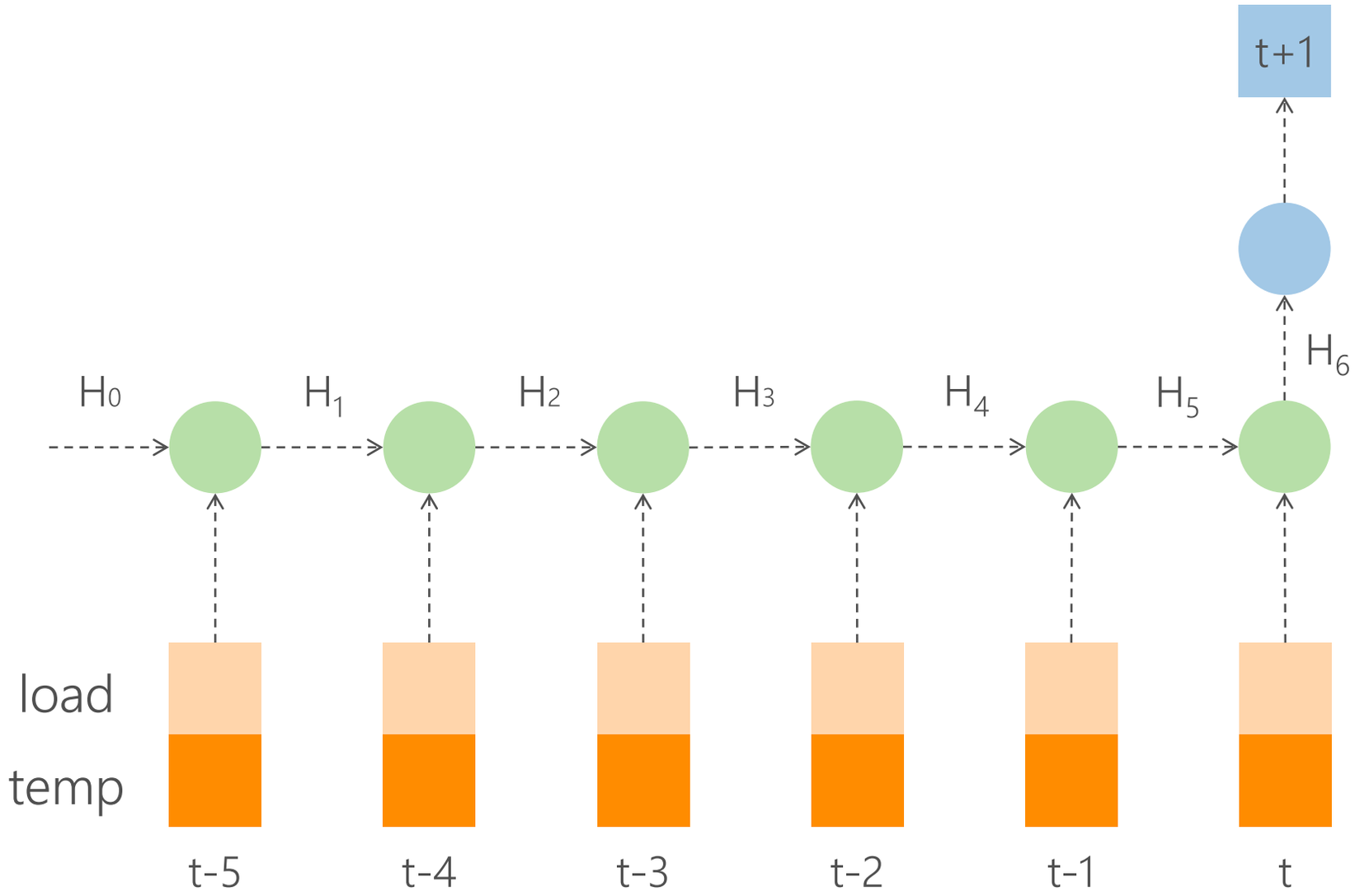



Overall MAPE
1.56%

# Multivariate input



# Multivariate input



# Multivariate input

| feature | y | load | | | | | | temp | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| time step | t+1 | t-5 | t-4 | t-3 | t-2 | t-1 | t-0 | t-5 | t-4 | t-3 | t-2 | t-1 | t-0 |
| 2012-01-01 05:00:00 | 0.18 | 0.22 | 0.18 | 0.14 | 0.13 | 0.13 | 0.15 | 0.42 | 0.43 | 0.40 | 0.41 | 0.42 | 0.41 |
| 2012-01-01 06:00:00 | 0.23 | 0.18 | 0.14 | 0.13 | 0.13 | 0.15 | 0.18 | 0.43 | 0.40 | 0.41 | 0.42 | 0.41 | 0.40 |
| 2012-01-01 07:00:00 | 0.29 | 0.14 | 0.13 | 0.13 | 0.15 | 0.18 | 0.23 | 0.40 | 0.41 | 0.42 | 0.41 | 0.40 | 0.39 |
| 2012-01-01 08:00:00 | 0.35 | 0.13 | 0.13 | 0.15 | 0.18 | 0.23 | 0.29 | 0.41 | 0.42 | 0.41 | 0.40 | 0.39 | 0.39 |
| 2012-01-01 09:00:00 | 0.37 | 0.13 | 0.15 | 0.18 | 0.23 | 0.29 | 0.35 | 0.42 | 0.41 | 0.40 | 0.39 | 0.39 | 0.43 |
| 2012-01-01 10:00:00 | 0.37 | 0.15 | 0.18 | 0.23 | 0.29 | 0.35 | 0.37 | 0.41 | 0.40 | 0.39 | 0.39 | 0.43 | 0.46 |
| 2012-01-01 11:00:00 | 0.37 | 0.18 | 0.23 | 0.29 | 0.35 | 0.37 | 0.37 | 0.40 | 0.39 | 0.39 | 0.43 | 0.46 | 0.50 |
| 2012-01-01 12:00:00 | 0.36 | 0.23 | 0.29 | 0.35 | 0.37 | 0.37 | 0.37 | 0.39 | 0.39 | 0.43 | 0.46 | 0.50 | 0.53 |
| 2012-01-01 13:00:00 | 0.35 | 0.29 | 0.35 | 0.37 | 0.37 | 0.37 | 0.36 | 0.39 | 0.43 | 0.46 | 0.50 | 0.53 | 0.52 |
| 2012-01-01 14:00:00 | 0.36 | 0.35 | 0.37 | 0.37 | 0.37 | 0.36 | 0.35 | 0.43 | 0.46 | 0.50 | 0.53 | 0.52 | 0.54 |

# Multivariate input

(23370, 6, 2)

| feature | y | | | | | | | load | | | | | temp |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| time step | t+1 | t-5 | t-4 | t-3 | t-2 | t-1 | t-0 | t-5 | t-4 | t-3 | t-2 | t-1 | t-0 |
| 2012-01-01 05:00:00 | 0.18 | 0.22 | 0.18 | 0.14 | 0.13 | 0.13 | 0.15 | 0.42 | 0.43 | 0.40 | 0.41 | 0.42 | 0.41 |
| 2012-01-01 06:00:00 | 0.23 | 0.18 | 0.14 | 0.13 | 0.13 | 0.15 | 0.18 | 0.43 | 0.40 | 0.41 | 0.42 | 0.41 | 0.40 |
| 2012-01-01 07:00:00 | 0.29 | 0.14 | 0.13 | 0.13 | 0.15 | 0.18 | 0.23 | 0.40 | 0.41 | 0.42 | 0.41 | 0.40 | 0.39 |

```
array([[[ 0.22,  0.42],
        [ 0.18,  0.43],
        [ 0.14,  0.4 ],
        [ 0.13,  0.41],
        [ 0.13,  0.42],
        [ 0.15,  0.41]],

       [[ 0.18,  0.43],
        [ 0.14,  0.4 ],
        [ 0.13,  0.41],
        [ 0.13,  0.42],
        [ 0.15,  0.41],
        [ 0.18,  0.4 ]],

       [[ 0.14,  0.4 ],
        [ 0.13,  0.41],
        [ 0.13,  0.42],
        [ 0.15,  0.41],
        [ 0.18,  0.4 ],
        [ 0.23,  0.39]]])
```

X

y

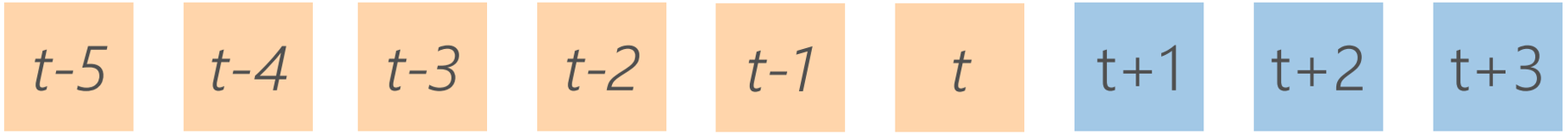
```
array([[ 0.18],
       [ 0.23],
       [ 0.29]])
```

(23370, 1)

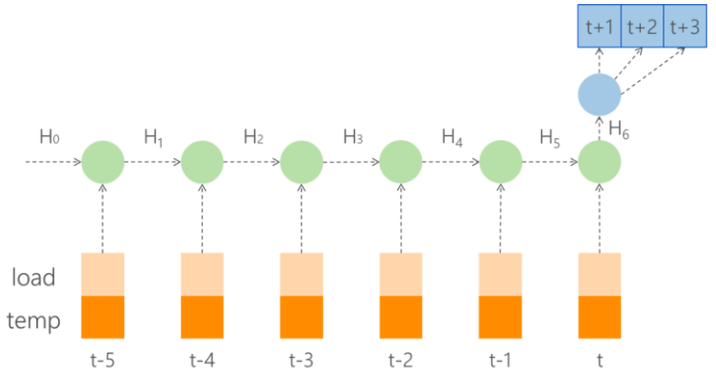
# Multi-step forecast

# Multi-step forecast

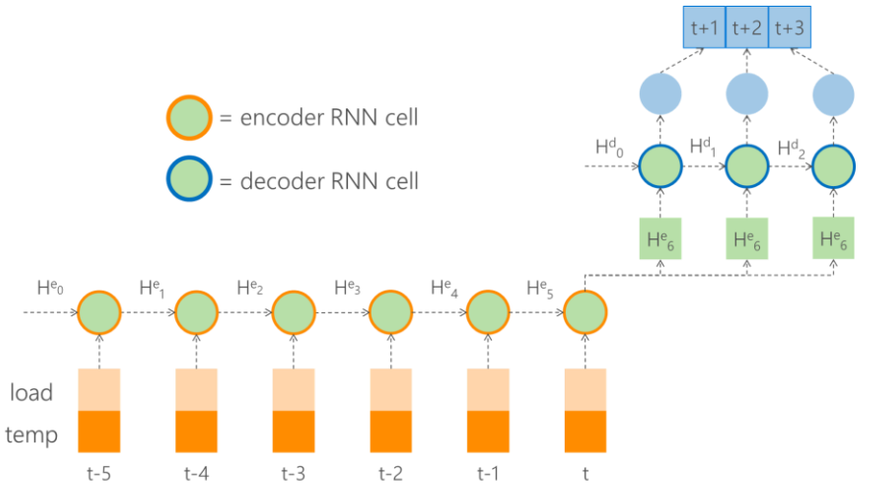
- Assuming we are at time *t* …
- … predict the values at times (*t+1*, … , *t+HORIZON*) …
- … conditional on the previous *T* values of the time series



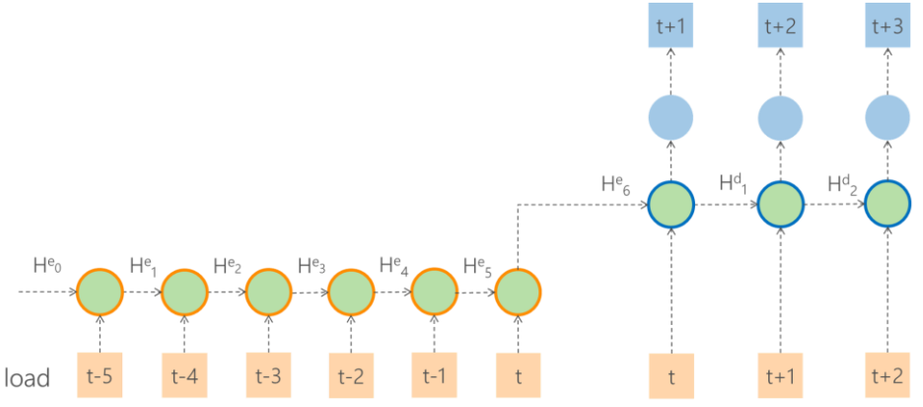
# Multi-step forecast




Vector output

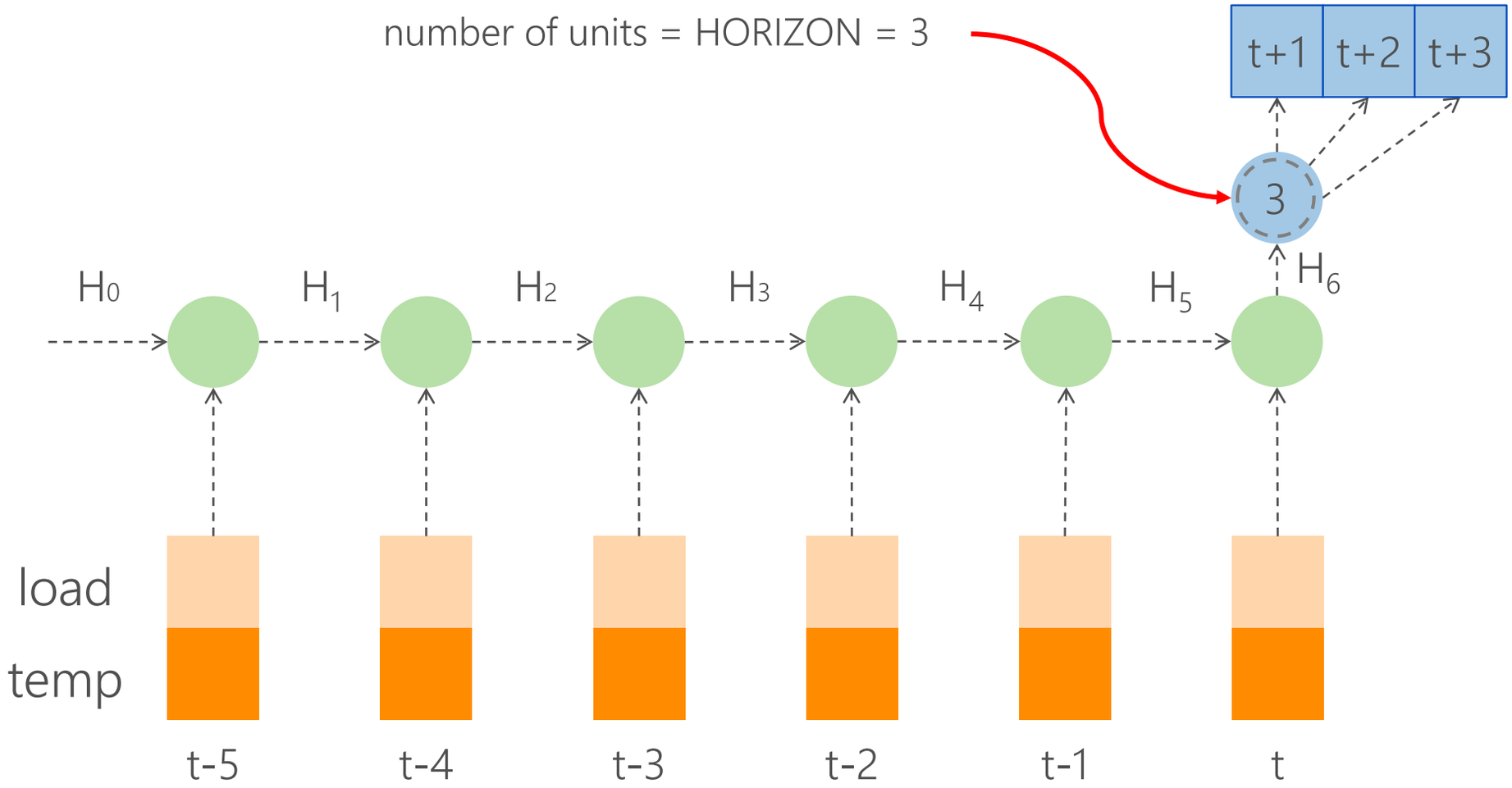



Simple encoder-decoder




Encoder-decoder with teacher forcing

# Vector output approach



# Vector output approach

| feature | y | | | load | | | | | | temp | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| time step | t+1 | t+2 | t+3 | t-5 | t-4 | t-3 | t-2 | t-1 | t-0 | t-5 | t-4 | t-3 | t-2 | t-1 | t-0 |
| 2012-01-01 05:00:00 | 0.18 | 0.23 | 0.29 | 0.22 | 0.18 | 0.14 | 0.13 | 0.13 | 0.15 | 0.42 | 0.43 | 0.40 | 0.41 | 0.42 | 0.41 |
| 2012-01-01 06:00:00 | 0.23 | 0.29 | 0.35 | 0.18 | 0.14 | 0.13 | 0.13 | 0.15 | 0.18 | 0.43 | 0.40 | 0.41 | 0.42 | 0.41 | 0.40 |
| 2012-01-01 07:00:00 | 0.29 | 0.35 | 0.37 | 0.14 | 0.13 | 0.13 | 0.15 | 0.18 | 0.23 | 0.40 | 0.41 | 0.42 | 0.41 | 0.40 | 0.39 |

```
array([[ 0.22,  0.42],
       [ 0.18,  0.43],
       [ 0.14,  0.4 ],
       [ 0.13,  0.41],
       [ 0.13,  0.42],
       [ 0.15,  0.41]],

      [[ 0.18,  0.43],
       [ 0.14,  0.4 ],
       [ 0.13,  0.41],
       [ 0.13,  0.42],
       [ 0.15,  0.41],
       [ 0.18,  0.4 ]],

      [[ 0.14,  0.4 ],
       [ 0.13,  0.41],
       [ 0.13,  0.42],
       [ 0.15,  0.41],
       [ 0.18,  0.4 ],
       [ 0.23,  0.39]]])
```

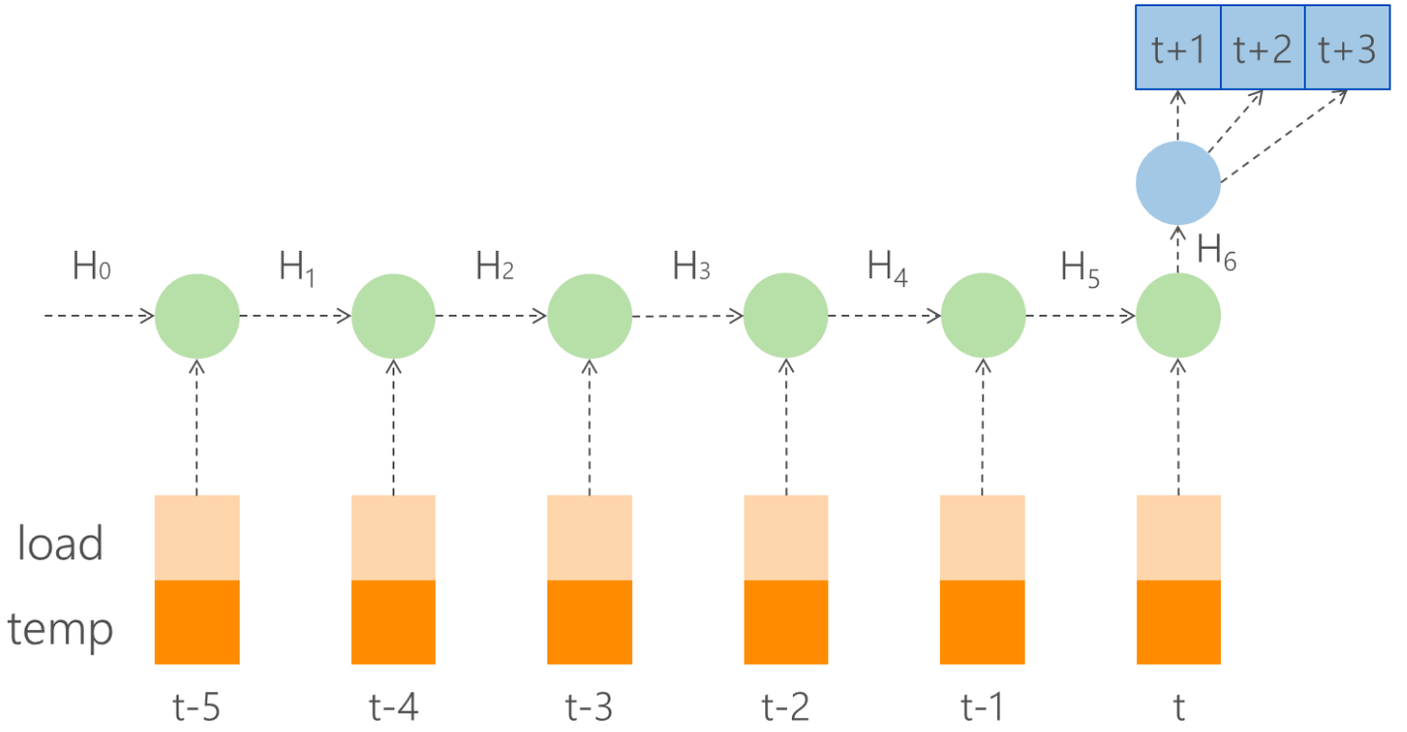
X

y

```
array([[ 0.18,  0.23,  0.29],
       [ 0.23,  0.29,  0.35],
       [ 0.29,  0.35,  0.37]])
```

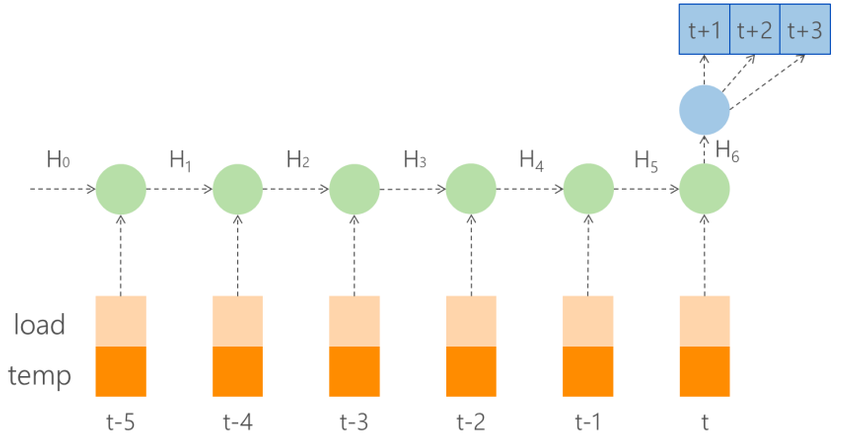
(23370, 3)

# Vector output approach



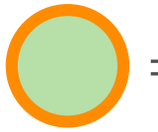

```
model = Sequential()
model.add(GRU(LATENT_DIM, input_shape=(6, 2)))
model.add(Dense(3, activation='linear'))
```

# Vector output approach



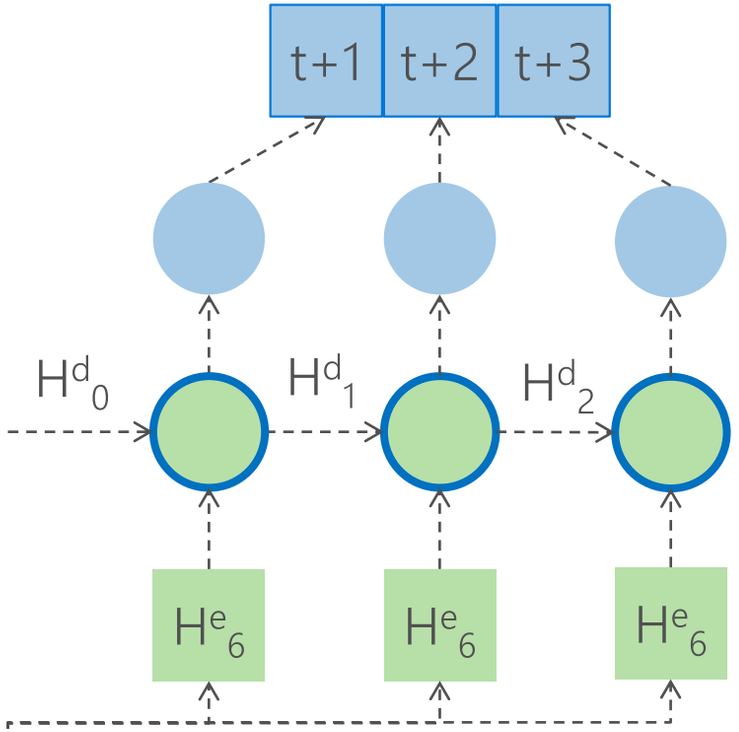

- Simplest to implement
- Fastest to train
- Fixed length output (can't produce forecasts with variable horizons)
- Does not model dependencies between predicted outputs

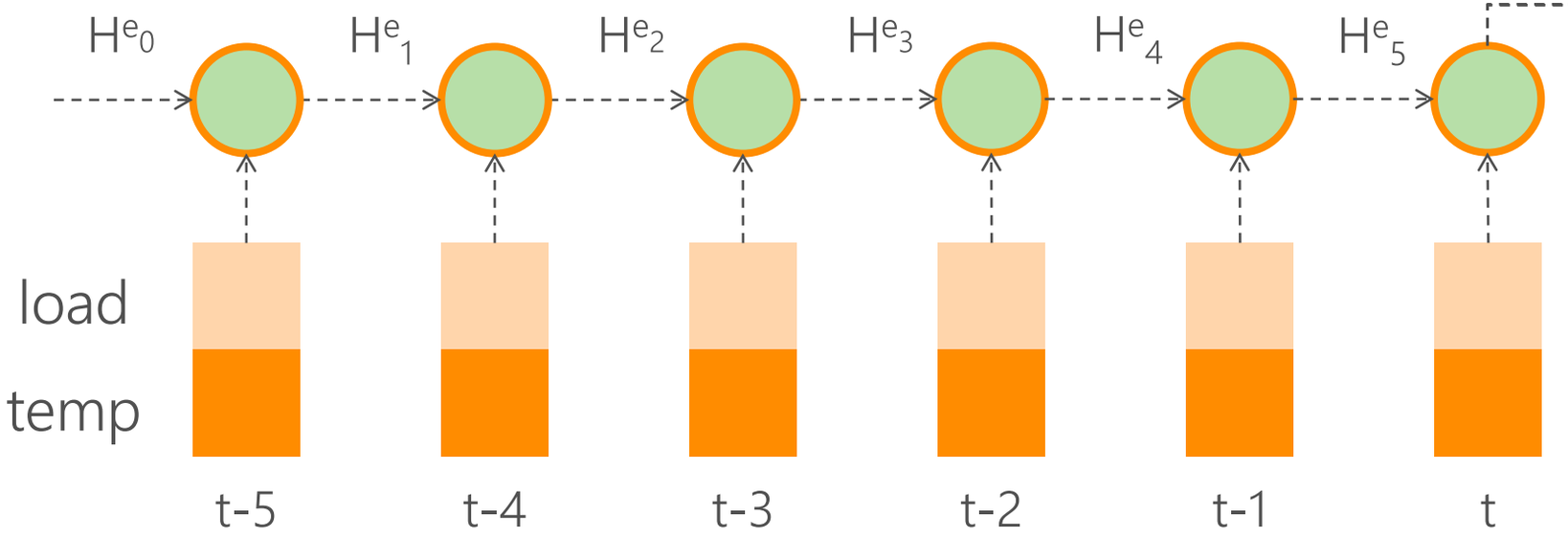
# Simple encoder-decoder



= encoder RNN cell

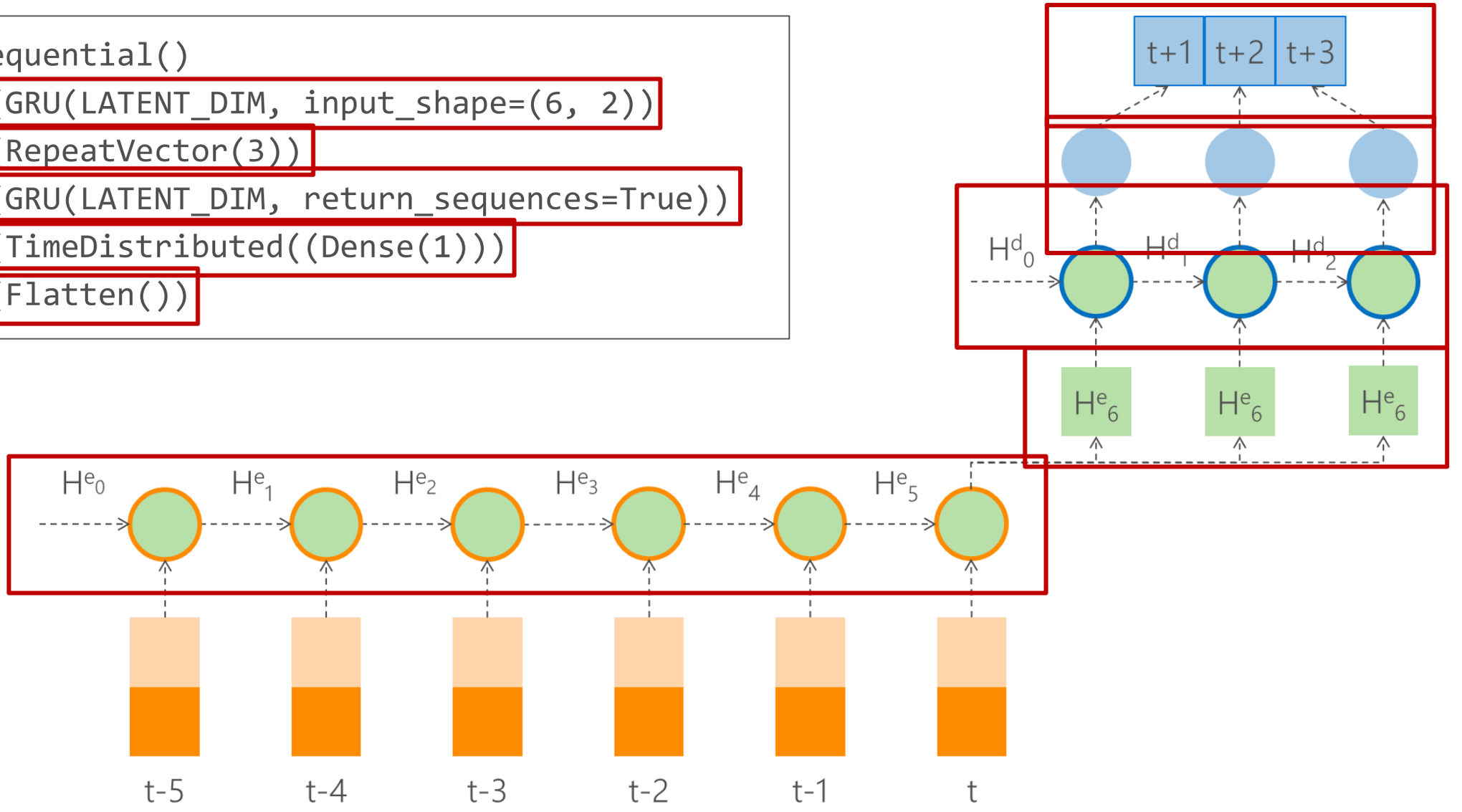
= decoder RNN cell



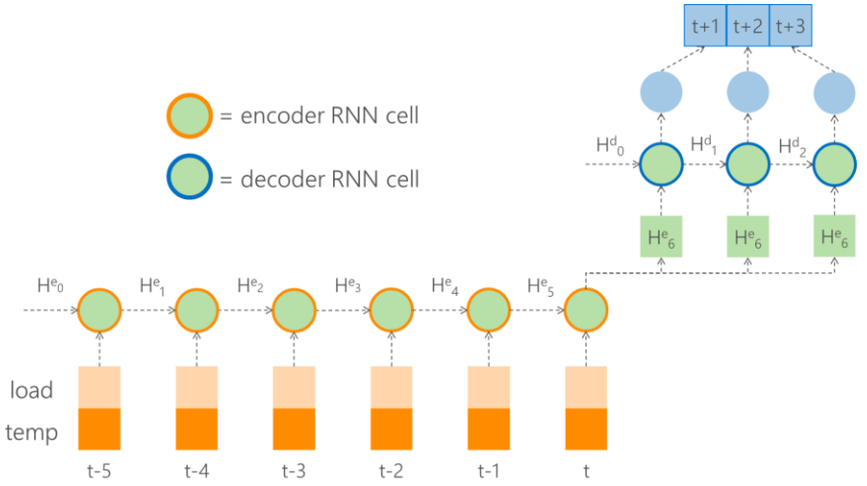



# Simple encoder-decoder

```
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())
```

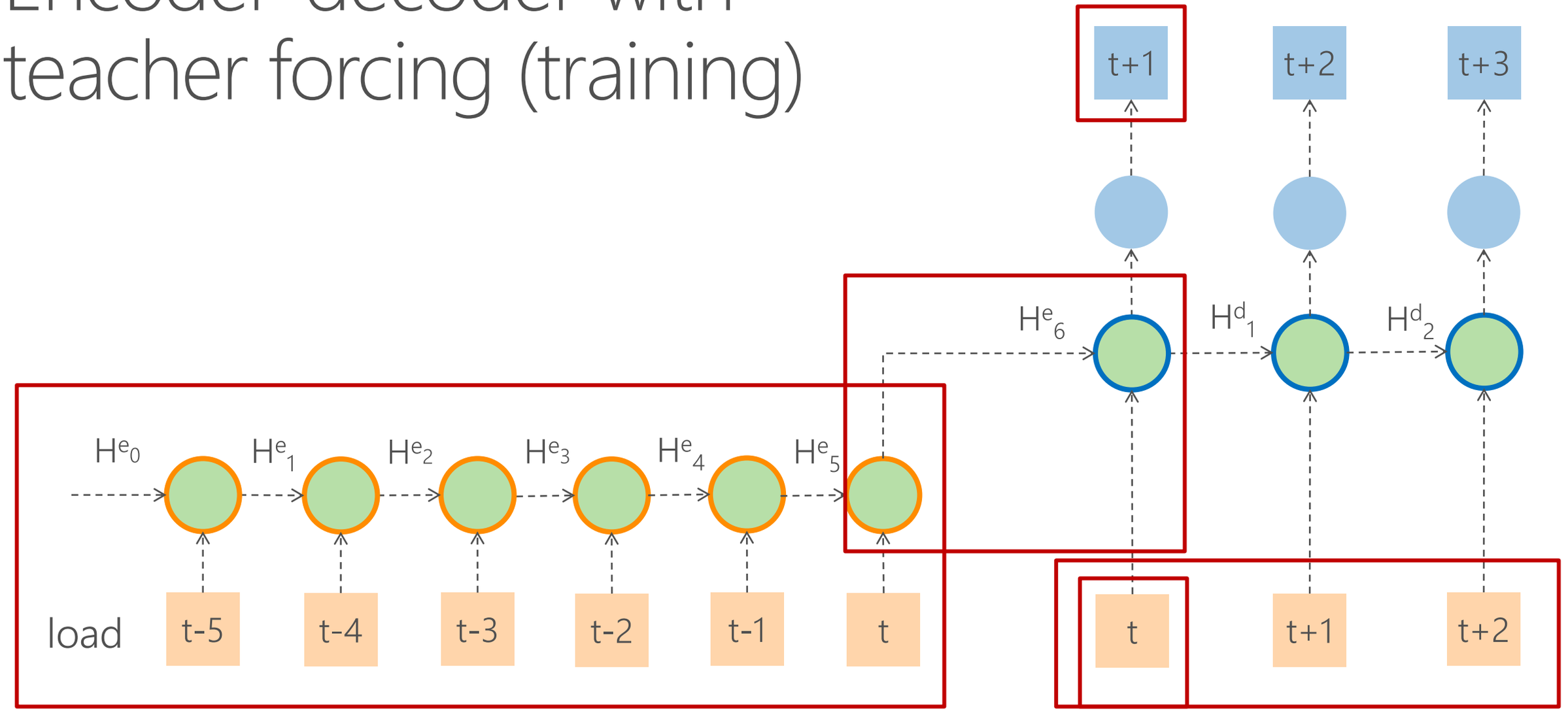


# Simple encoder-decoder

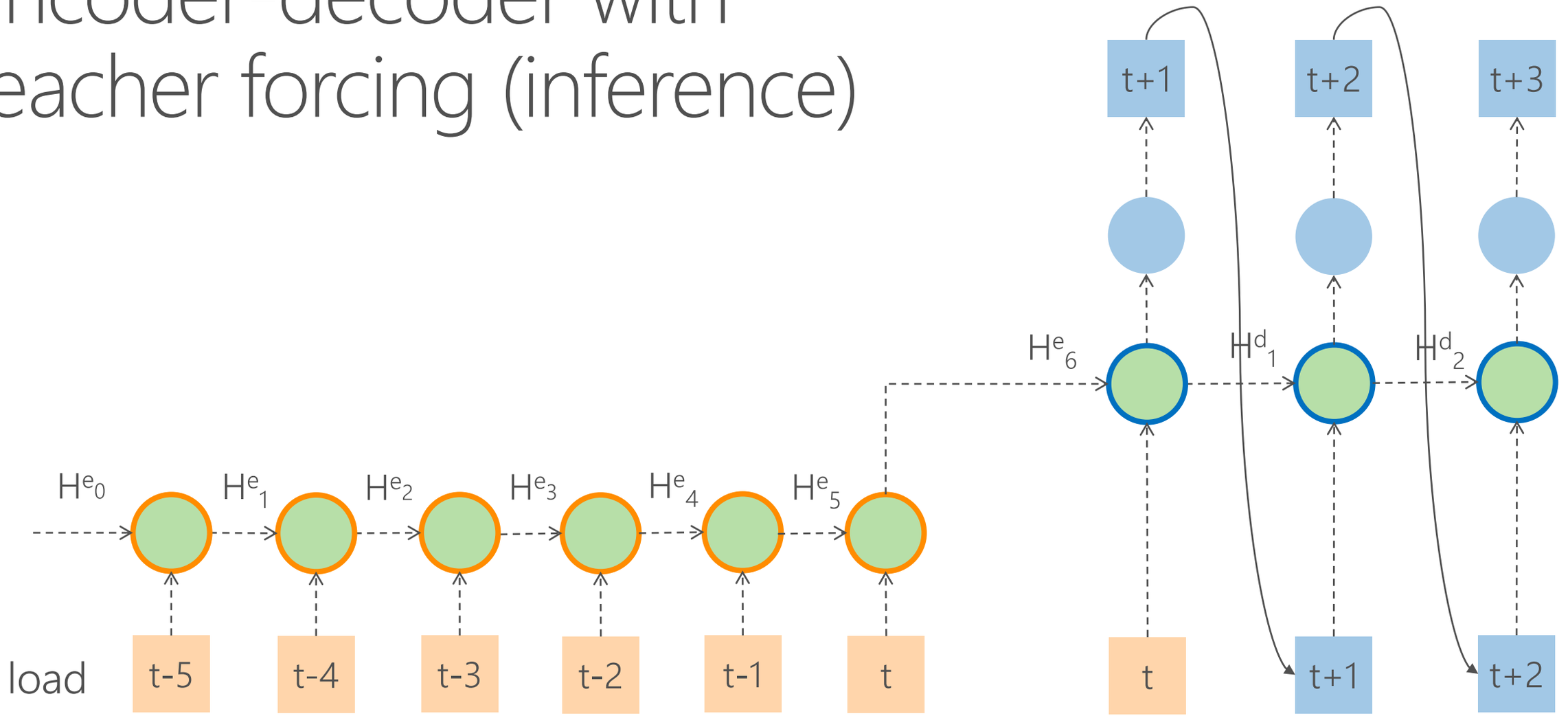



- Fairly simple to implement
- Tries to capture dependencies between forecasted time steps through decoder hidden state
- Slower to train with stacked RNN layers
- Fixed length output

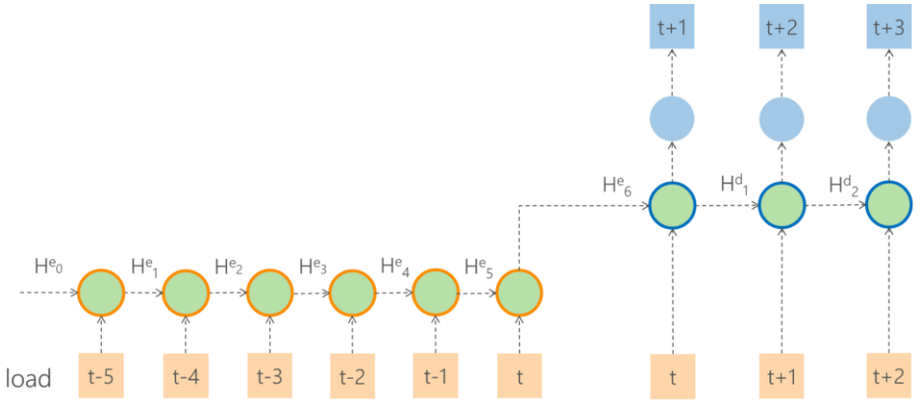
# Encoder-decoder with teacher forcing (training)



# Encoder-decoder with teacher forcing (inference)

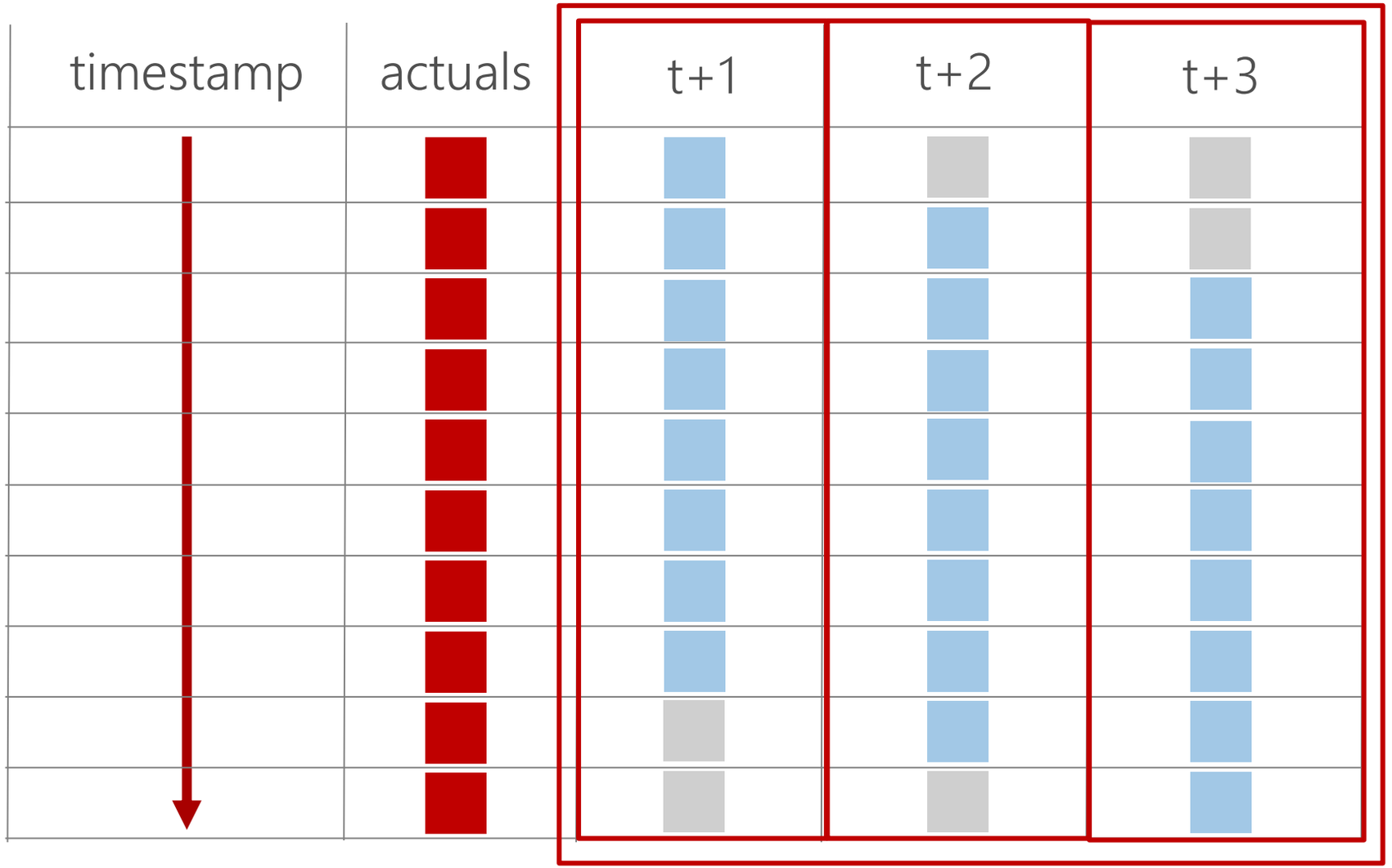


# Encoder-decoder with teacher forcing




- 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 with stacked RNN layers
- Error propagation due to recursive decoder

# Evaluation

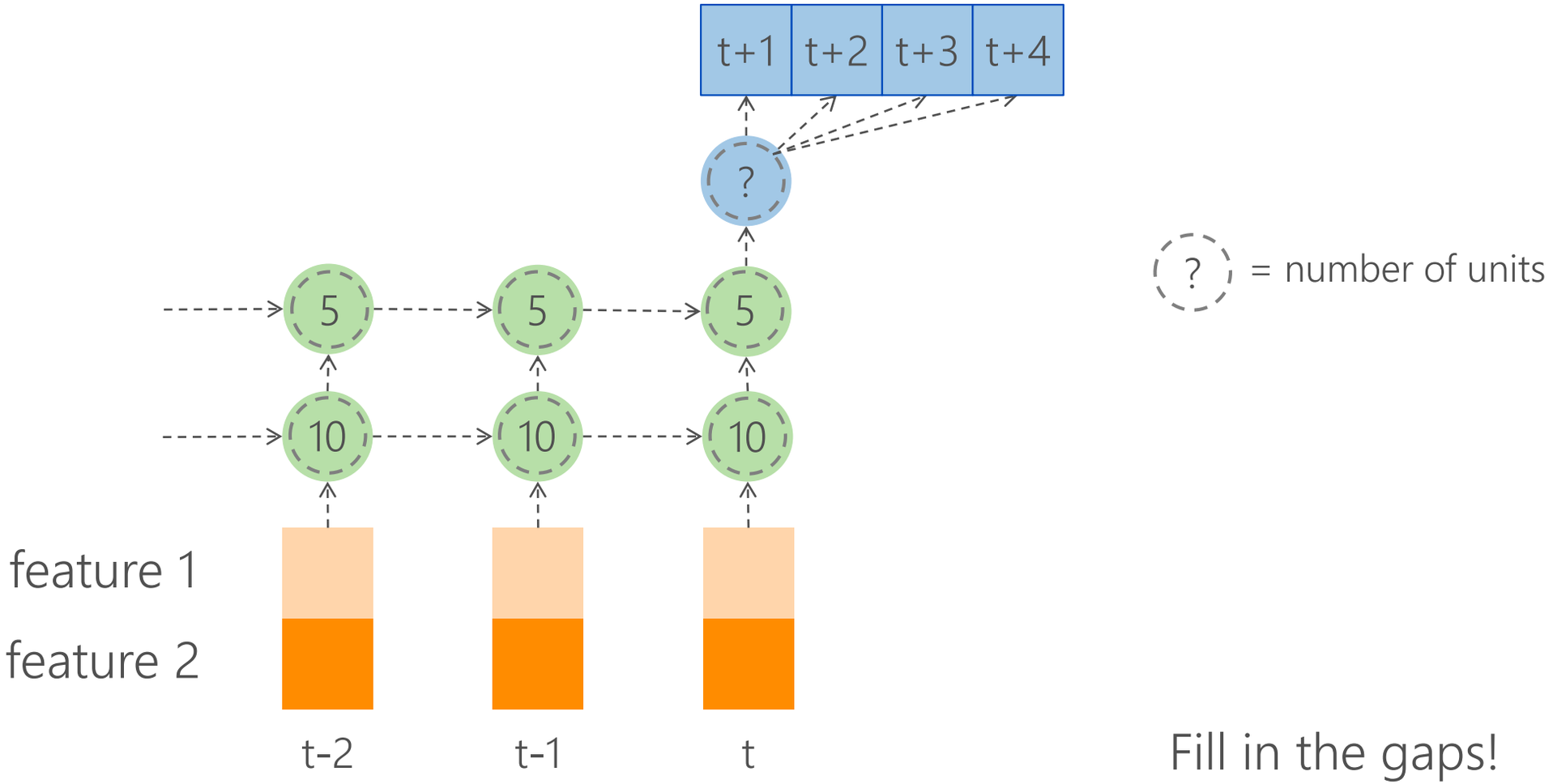



$$\frac{100}{N}\sum_{i=1}^{N}\left|\frac{y_i - \hat{y}_i}{y_i}\right|$$

# Evaluation

| h | MAPE |
|---|---|
| t+1 | 0.75% |
| t+2 | 1.30% |
| t+3 | 2.10% |
| Overall | **1.40%** |

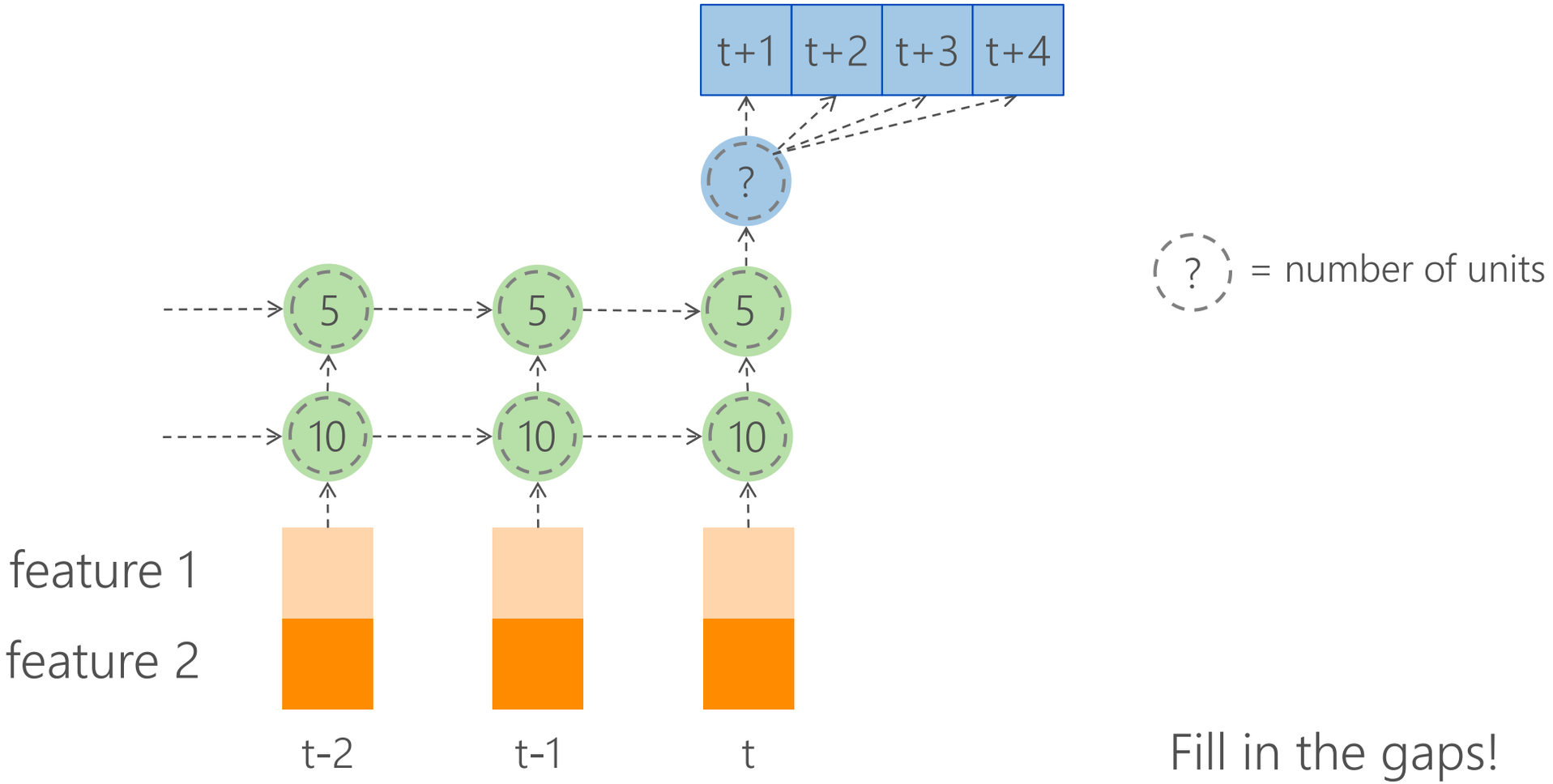
# Quiz




Fill in the gaps!

```
model = Sequential()
model.add(GRU(  , input_shape=(  ,  ),
model.add(GRU(
model.add(          )
```

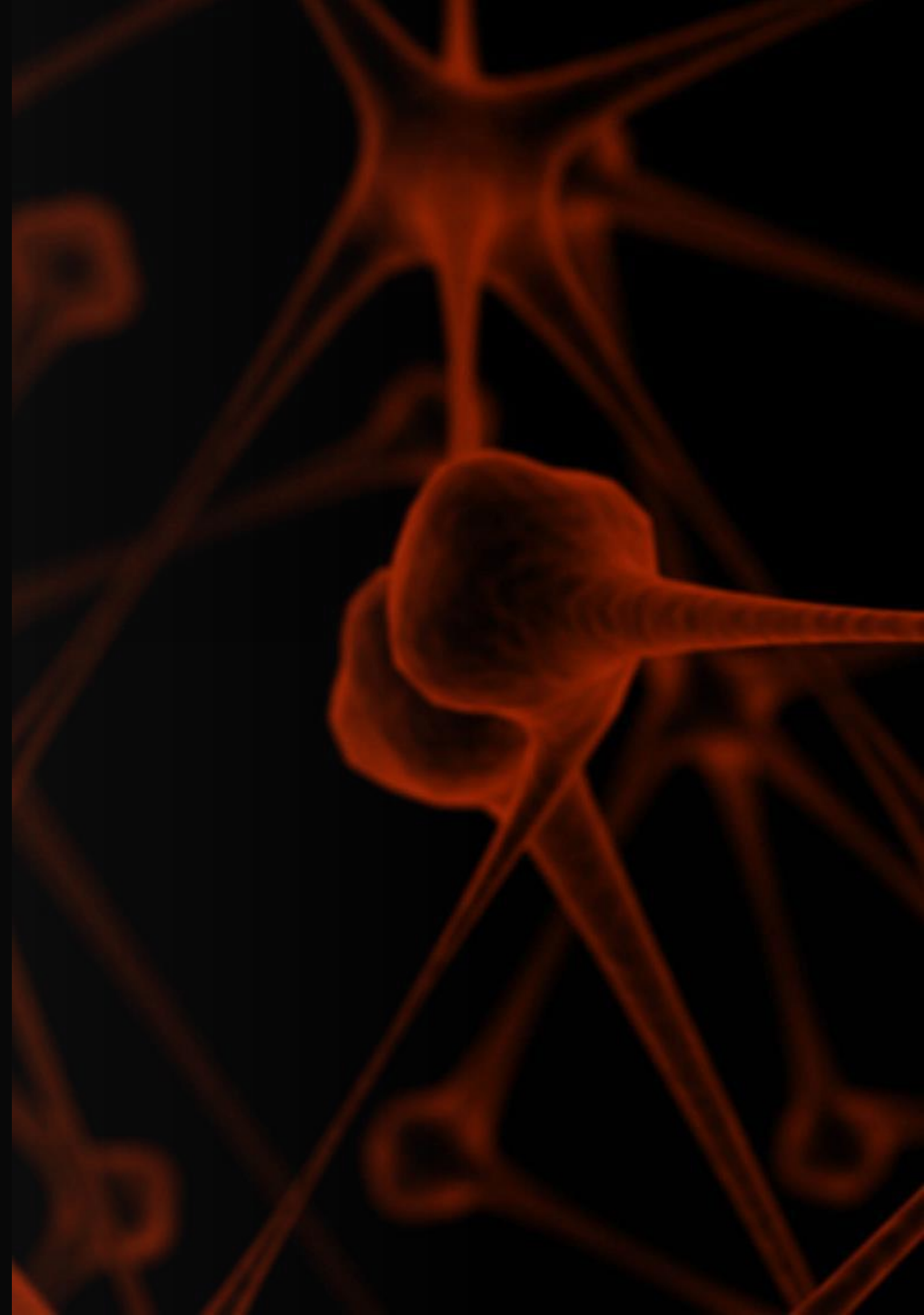
# Answers




Fill in the gaps!

```
model = Sequential()
model.add(GRU(  , input_shape=(  ,  ),
model.add(GRU(
model.add(          )
```

# Tricks for training successful RNN models



# Minibatch Stochastic Gradient Descent

Initialization: $w = w_0$

While stopping criterion not met:

Shuffle examples randomly

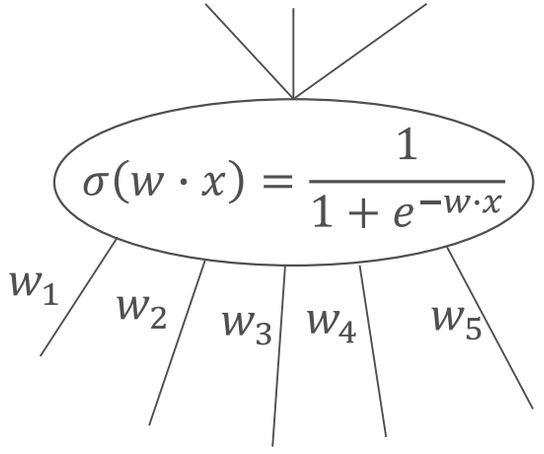
Partition examples into batches of size m

For $\mathbf{minibatch}=1\ldots\mathrm{N}/\mathrm{m}$ examples

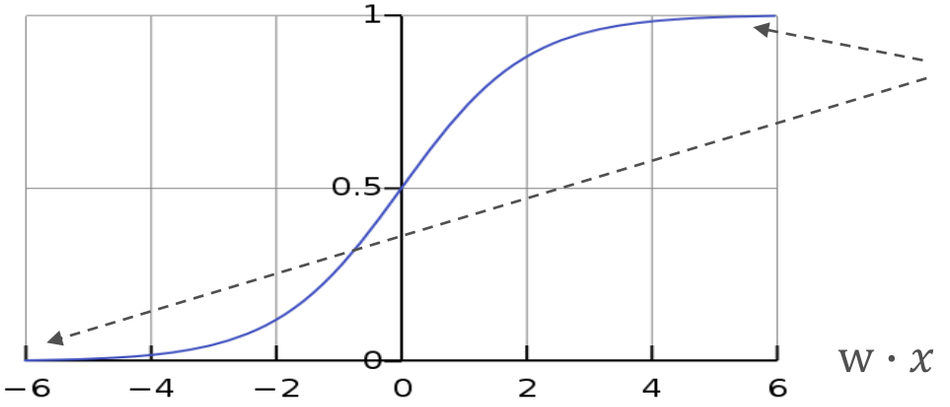
$$w = w - \frac{\alpha}{m}\sum_{i=1}^{m}\nabla L_i(w)$$

What is a good value of $w_0$?

# Initialization of weights




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

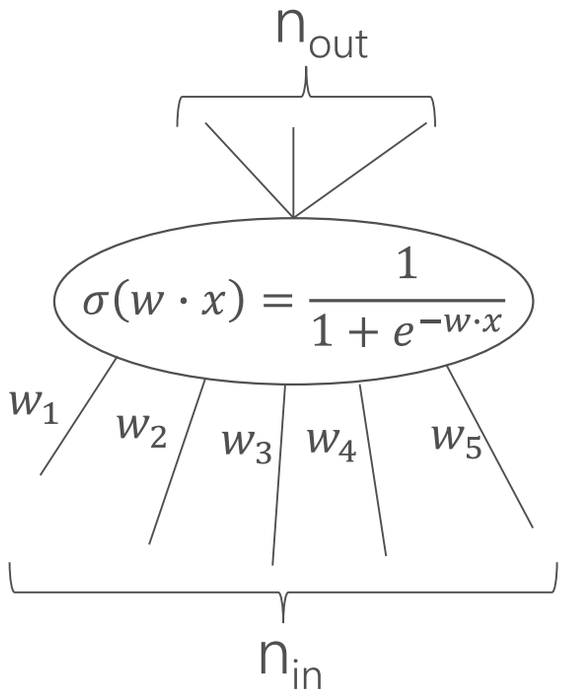



large weights
gradient=0, no
backpropagation, “dead neuron”

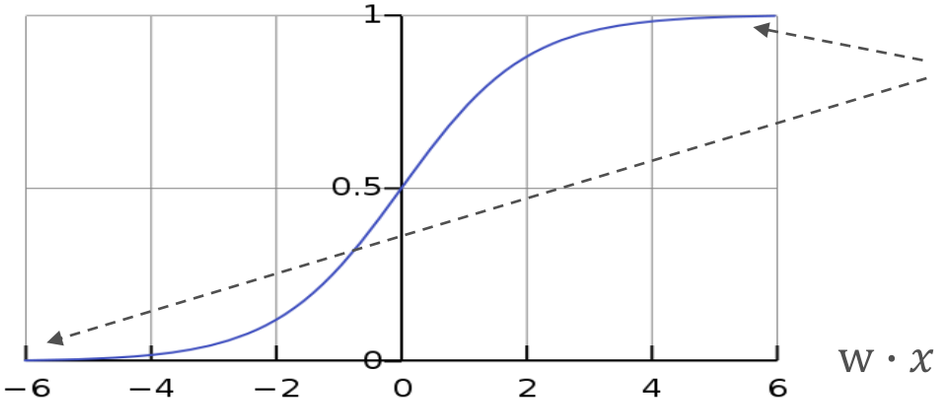
- Initialize all $w_i$ with zeros

Quiz: Is this a good initialization?

# Initialization of weights




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




large weights
gradient=0, no
backpropagation, “dead neuron”

- Initialize all $w_i$ with zeros - the output of all neurons will look the same
- $w_i \sim N(0, A)$ – many weights close to 0, might cause vanishing gradients problem
- Xavier Glorot uniform (default in Keras)

$$w_i \sim \text{Uniform}\left[-\frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}\right]$$

# Regularization

Objective function minimized by mini-batch SGD:

$$L(weights, biases) + \lambda \sum_i w_i^2$$

L2 regularization,
$\lambda > 0$ is a hyperparameter

Benefits of L2 regularization:

- mitigates overfitting by spreading weights across multiple connections
- reduces number of dead neurons by penalizing large weights

Other types of regularization (not covered here): dropout, zoneout

# Network tuning


# Hyperparameters

- Architecture

  number of layers, number of cells in each layer, connections between cells

- Loss function

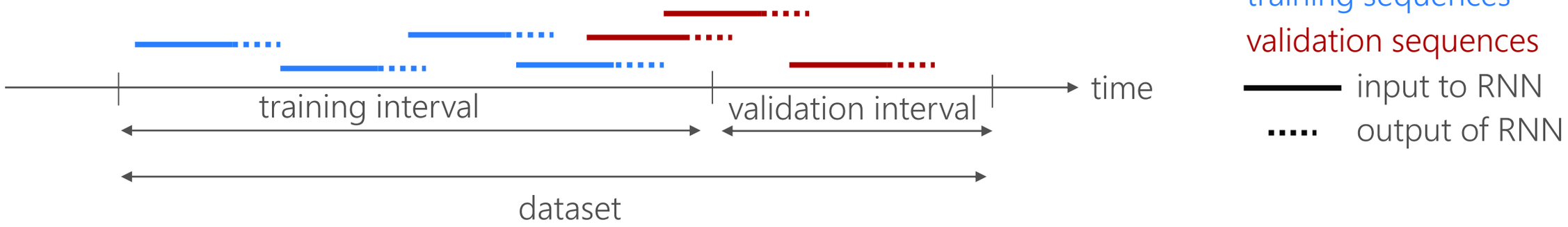
  weight of regularization term

- Optimization

  learning rate, mini-batch size, stopping criterion, initialization

# Tuning using training/validation split

Split the dataset into training and validation time intervals




1. For each combination of values of tunable hyperparameters
   - Train over training sequences
   - Compute target metric over validation sequences
2. Choose the values of hyperparameters that give the best value of target metric
3. (Optional) Use the chosen value of hyperparameters to train the model over training + validation sequences.

# Advanced techniques for tuning hyperparameters

- Bayesian optimization Bayesian optimization with robust Bayesian neural networks, talk of Chenhui Hu, 6/13, Sonora, 2:30pm
- Multi-armed bandit techniques Hyperband: A novel bandit-based approach for hyperparameter optimization
- Neural architecture search Neural architecture search with reinforcement learning
- Genetic programming Population based training of neural networks

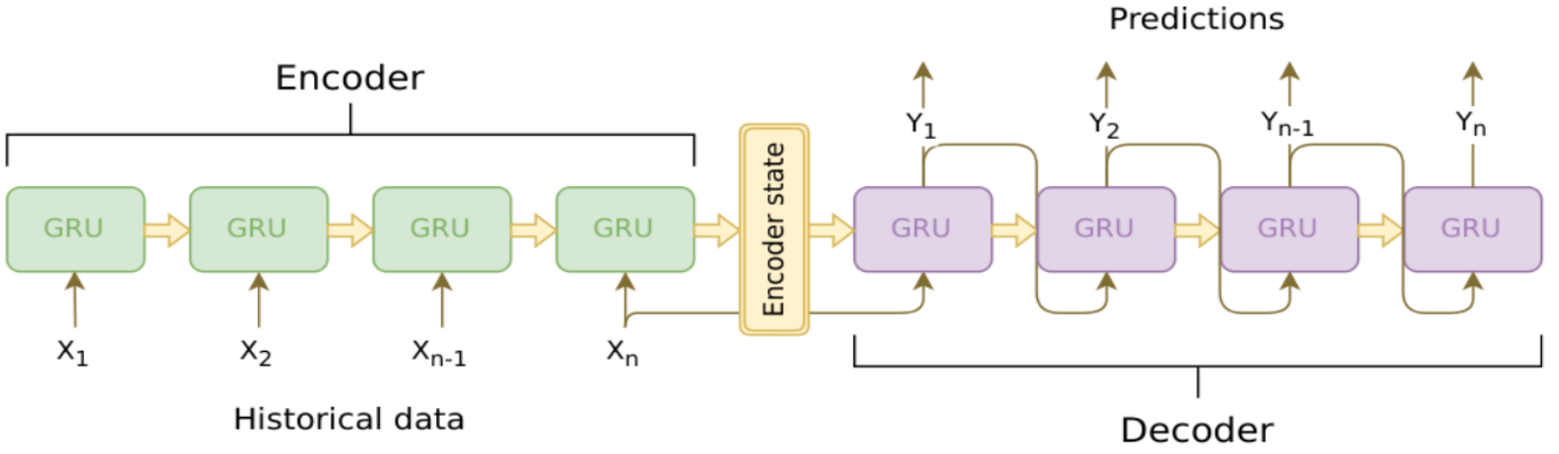
Hyperdrive – Microsoft service for tuning hyperparameters

# Other tricks used by winning RNN models

# Web traffic forecasting competition

Forecast traffic of 145K Wikipedia pages

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




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

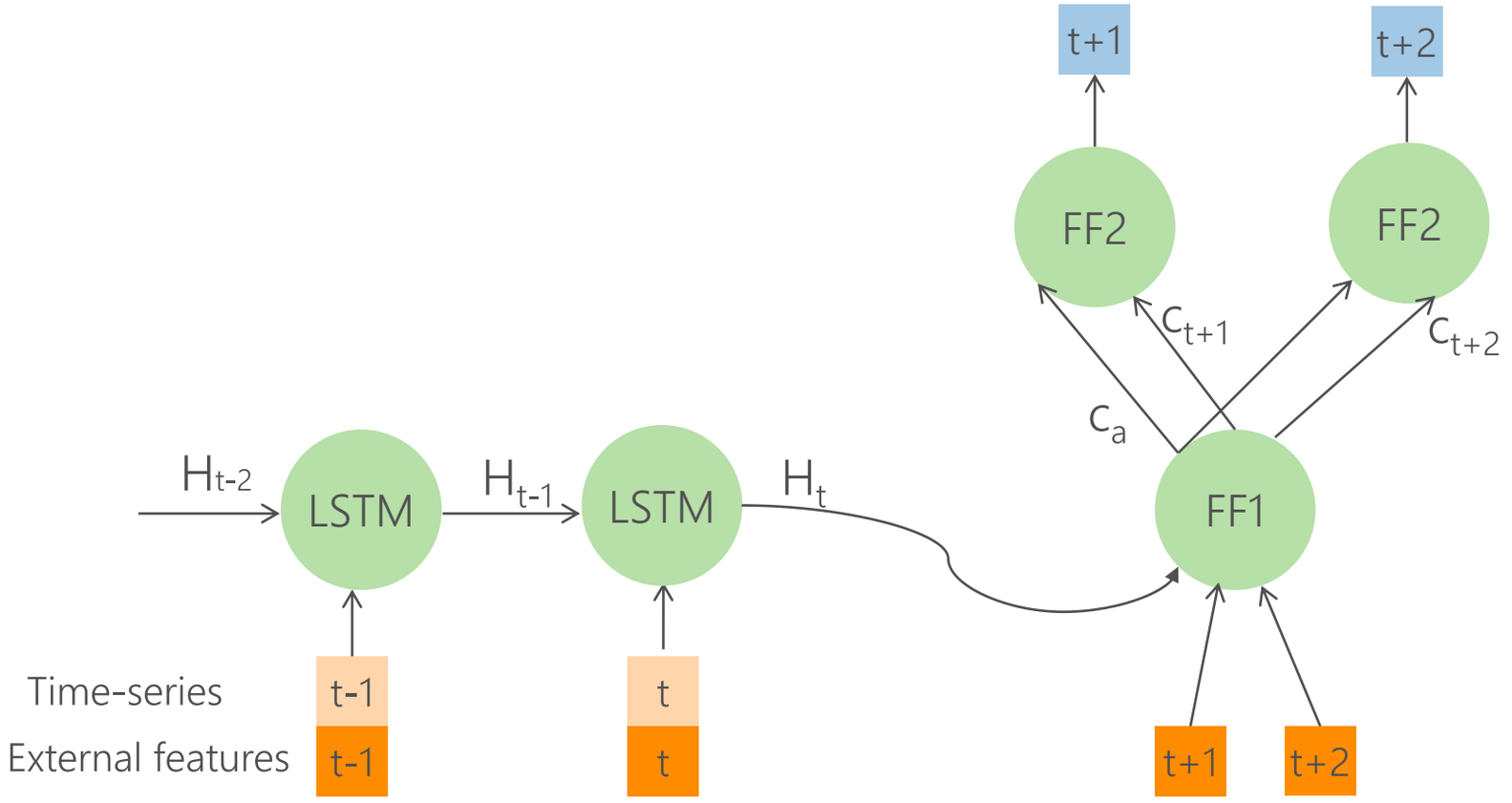
$$x_t \rightarrow (x_t, x_{t\text{-quarter}}, x_{t\text{-year}})$$

- COCOB optimizer that doesn't have learning rate hyperparameter, custom implementation
- sophisticated technique for building ensemble of RNN models
- tuning of hyperparameters using Bayesian optimization and SMAC3 package

# GEFCom 2014 competition

Energy load and price forecasting in New England

Best model (2017): LSTM encoder + feed-forward neural network arxiv.org/abs/1711.11053



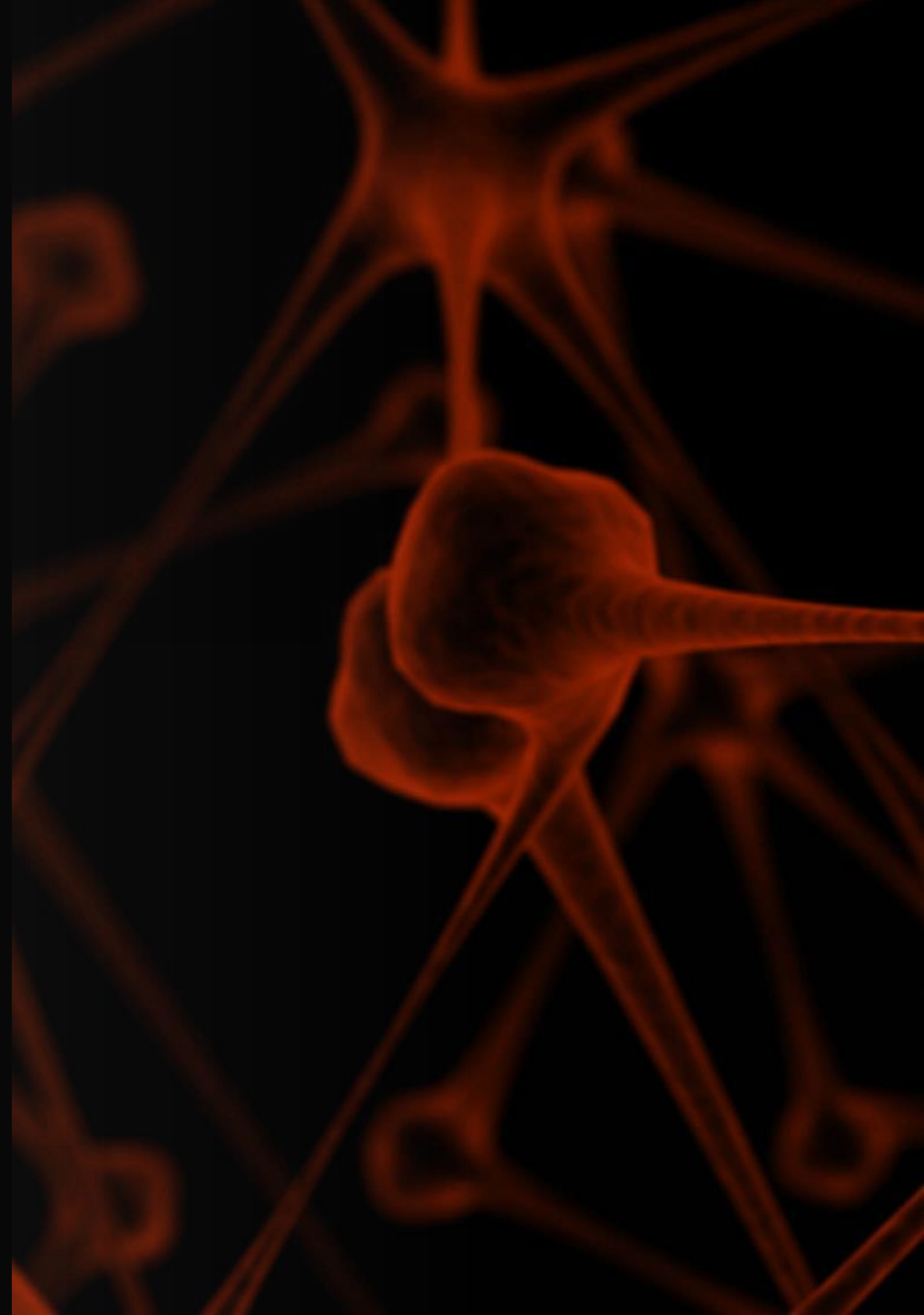

$c_a$ - horizon-agnostic context

$c_{t+i}$ - horizon-dependent context

FF2 has the same weights for all horizons

External features: weekday/holiday indicators, day of week, hour of the day, weather data

# Conclusions



# Techniques not covered

- Probabilistic prediction Multi-horizon quantile-based forecaster

- Advanced architectures
  - dropout / zoneout Zoneout: Regularizing RNNs by randomly preserving hidden activations
  - memory networks memory networks

- Advanced training techniques
  - state-of-the-art optimization algorithms
  - recurrent batch normalization recurrent batch normalization
  - stateful training stateful training

- Temporal convolutional neural networks An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling

# Learning resources

- Source code of examples of how to train RNN models with Keras
  URL: https://github.com/yijingchen/RNNForTimeSeriesForecastTutorial

- Blogs
  machinelearningmastery.org
  dacatay.com

- Competitions
  Kaggle

- State-of-the-art results
  Conferences: NIPS, ICLR, ICML
  Hot-off-the-oven papers: arxiv-sanity.org

# References


Dropout: A simple way to prevent neural networks from overfitting,
Zoneout: Regularizing RNNs by randomly preserving hidden activations
Batch normalization: Accelerating deep network training by reducing internal covariate shift,
Recurrent batch normalization
Learning rate schedules and adaptive learning rate methods for deep learning
An overview of gradient descent optimization algorithms
Learning phrase representations using rnn encoderdecoder for statistical machine translation
LSTM: A Search Space Odyssey
Bayesian optimization with robust Bayesian neural networks
Hyperband: A novel bandit-based approach for hyperparameter optimization
Neural architecture search with reinforcement learning
Multi-horizon quantile-based forecaster
Population based training of neural networks
Memory networks
Stateful training
An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling
Winning solution of web traffic forecasting competition
Probabilistic energy forecasting: Global Energy Forecasting Competition 2014 and beyond

# Key Takeaways

- RNNs are very powerful models, sometimes RNNs are the most accurate models.
- RNNs share many techniques/tricks with feed-forward neural networks
- Tuning hyperparameters and other tricks are important for creating an accurate model

# Q & A