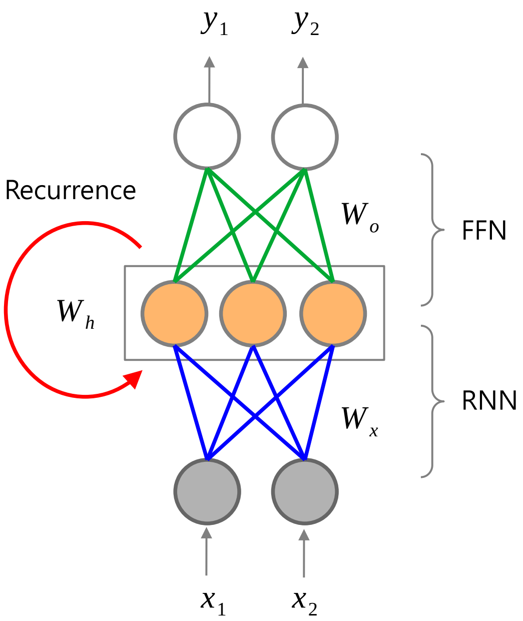


10. Recurrent Neural Networks

Part 1: RNN basics and data structures

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

[MXDL-10-08] Deep Learning / Recurrent Neural Network (RNN) - Contents

1. Simple RNN

[MXDL-10-01]
- 1-1. Time series and RNN
- 1-2. Constructing a dataset for RNN
- 1-3. Feeding data into RNN

[MXDL-10-02]
- 1-4. RNN structure
  - Many-to-One
  - Many-to-Many
- 1-5. Backpropagation Through Time (BPTT)

[MXDL-10-03]
- 1-6. Implementation of a Simple RNN model
  - Many-to-One (Keras' custom layer)
  - Many-to-One (Keras SimpleRNN class)
  - Many-to-Many (Keras SimpleRNN class)

2. Long Short Term Memory (LSTM)

[MXDL-10-04]
- 2-1. History and papers
- 2-2. LSTM cell structure
- 2-3. Gates: forget, input, output
- 2-4. Feeding data into LSTM
- 2-5. Backpropagation Through Time (BPTT) of LSTM

[MXDL-10-05]
- 2-6. LSTM implementation
  - Many-to-One (Custom LSTM layer)
  - Many-to-One and Many-to-Many (Keras LSTM class)

3. Variations of LSTM

[MXDL-10-06]
- 3-1. Peephole LSTM
  - History and structure
  - Many-to-One (Custom layer)
  - Many-to-Many (Custom layer)

[MXDL-10-07]
- 3-2. Gated Recurrent Unit (GRU)
  - GRU structure
  - Many-to-Many (Custom GRU layer)
  - Many-to-Many (Keras GRU class)

4. Multi-layer and Bi-directional

[MXDL-10-08]
- 4-1. Many-to-One & Multi-layer
- 4-2. Many-to-Many & Multi-layer
- 4-3. Many-to-One & Multi-layer & Bi-directional
- 4-4. Many-to-Many & Multi-layer & Bi-directional

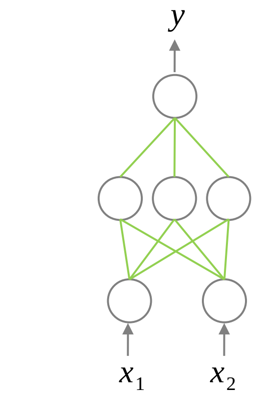
[MXDL-10-01] Deep Learning / Recurrent Neural Network (RNN) – Simple RNN

Time series and RNN

- Data where the past and present are dependent on each other, such as time series, cannot be analyzed with a feedforward network.
- A new model is required. This is the recurrent neural networks (RNN) that are widely used in time series forecasting, natural language processing, etc.

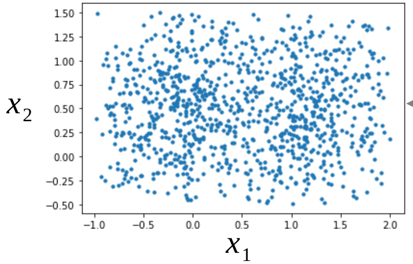
$x^{(t)}$ and $x^{(t+1)}$ are independent

- Use feedforward network.
- We can do row subsampling or randomly shuffle the data points.

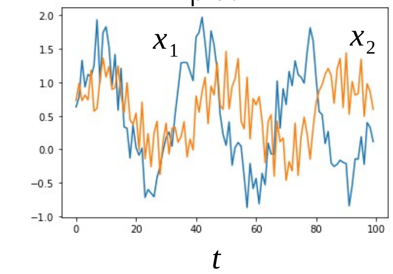


[Feed Forward Network: FFN]

No patterns scatter



Time series plot



[Training data]

feature

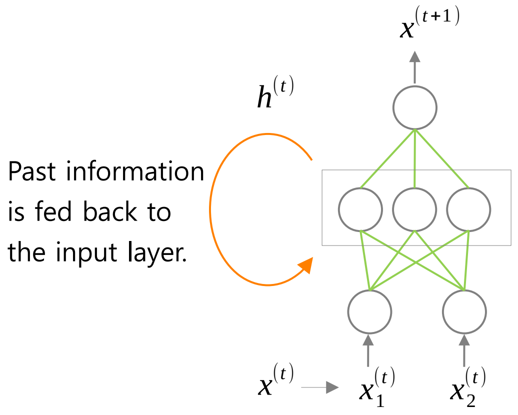
$x^{(1)}$

$x^{(2)}$

```
  t    x1     x2
 ---  ----   ----
  1   0.63,  0.73
  2   0.78,  0.98
  3   1.33,  0.73
  4   0.93,  0.81
  5   1.12,  0.74
  6   1.09,  1.18
  7   1.25,  0.57
  8   1.93,  0.61
  9   1.  ,  1.05
 10   1.74,  1.37
 11   1.83,  1.08
 12   1.53,  1.24
 13   0.94,  0.9
 14   1.42,  0.92
  ⋮     ⋮      ⋮
```

$x^{(t)}$ and $x^{(t+1)}$ are dependent

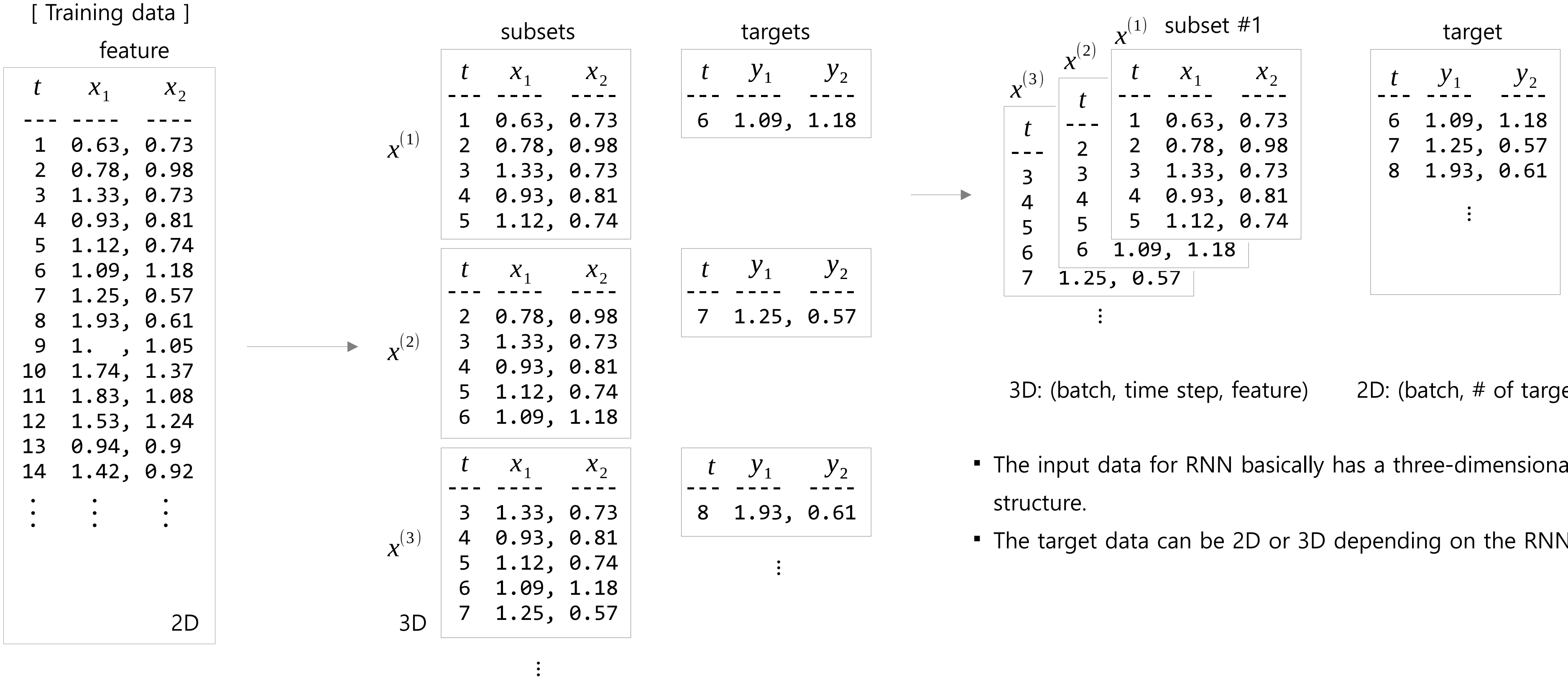
- Use recurrent neural networks.
- Time series prediction, classification and regression are all possible.
- We cannot do row subsampling and randomly shuffle the data points.



[Recurrent Neural Network: RNN]

Constructing a dataset for RNN

- We cannot just feed the training data below straight into a recurrent neural network. We need to transform this data into something that fits into a recurrent neural network.



[Training data]

feature

t	x_1	x_2
1	0.63,	0.73
2	0.78,	0.98
3	1.33,	0.73
4	0.93,	0.81
5	1.12,	0.74
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
9	1. ,	1.05
10	1.74,	1.37
11	1.83,	1.08
12	1.53,	1.24
13	0.94,	0.9
14	1.42,	0.92
⋮	⋮	⋮

2D

subsets

$x^{(1)}$

t	x_1	x_2
1	0.63,	0.73
2	0.78,	0.98
3	1.33,	0.73
4	0.93,	0.81
5	1.12,	0.74

$x^{(2)}$

t	x_1	x_2
2	0.78,	0.98
3	1.33,	0.73
4	0.93,	0.81
5	1.12,	0.74
6	1.09,	1.18

$x^{(3)}$

t	x_1	x_2
3	1.33,	0.73
4	0.93,	0.81
5	1.12,	0.74
6	1.09,	1.18
7	1.25,	0.57

⋮

3D

targets

t	y_1	y_2
6	1.09,	1.18

t	y_1	y_2
7	1.25,	0.57

t	y_1	y_2
8	1.93,	0.61

⋮

$x^{(1)}$ subset #1

t	x_1	x_2
1	0.63,	0.73
2	0.78,	0.98
3	1.33,	0.73
4	0.93,	0.81
5	1.12,	0.74

$x^{(2)}$

t		
2		
3		
4		
5		
6	1.09,	1.18

$x^{(3)}$

t		
3		
4		
5		
6		
7	1.25,	0.57

⋮

target

t	y_1	y_2
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
	⋮	

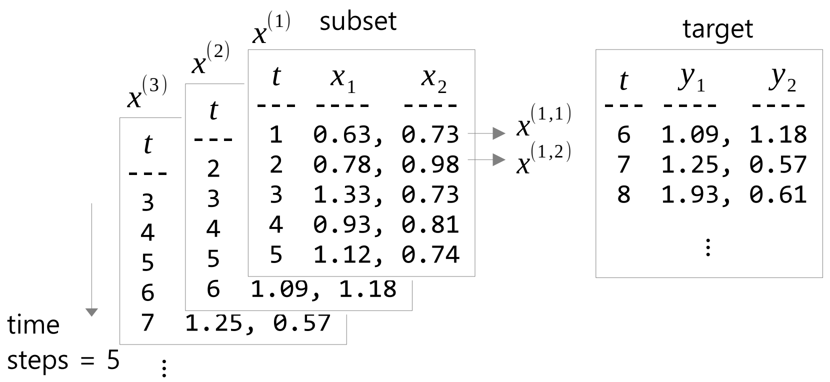
3D: (batch, time step, feature)

2D: (batch, # of targets)

- The input data for RNN basically has a three-dimensional structure.
- The target data can be 2D or 3D depending on the RNN type.

[MXDL-10-01] Deep Learning / Recurrent Neural Network (RNN) – Simple RNN

- Feeding data into an RNN.

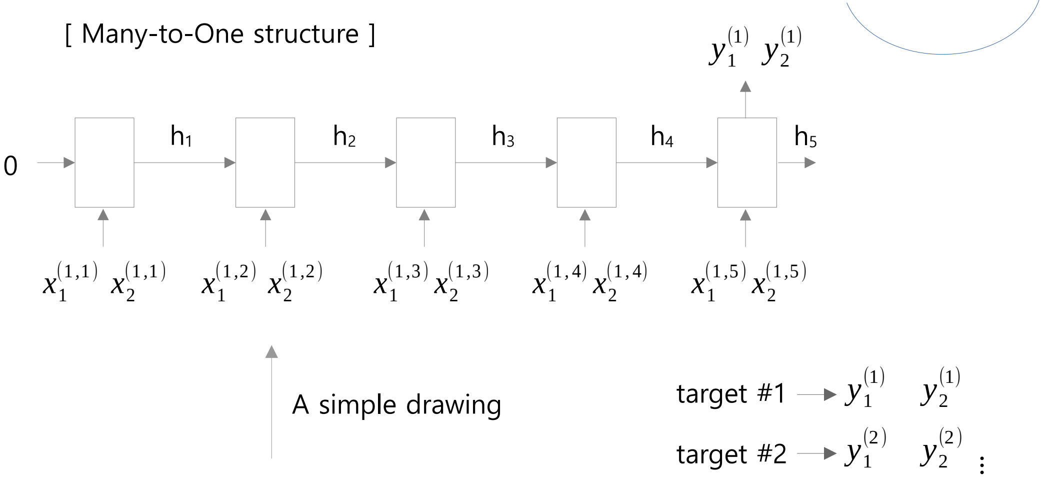


subset

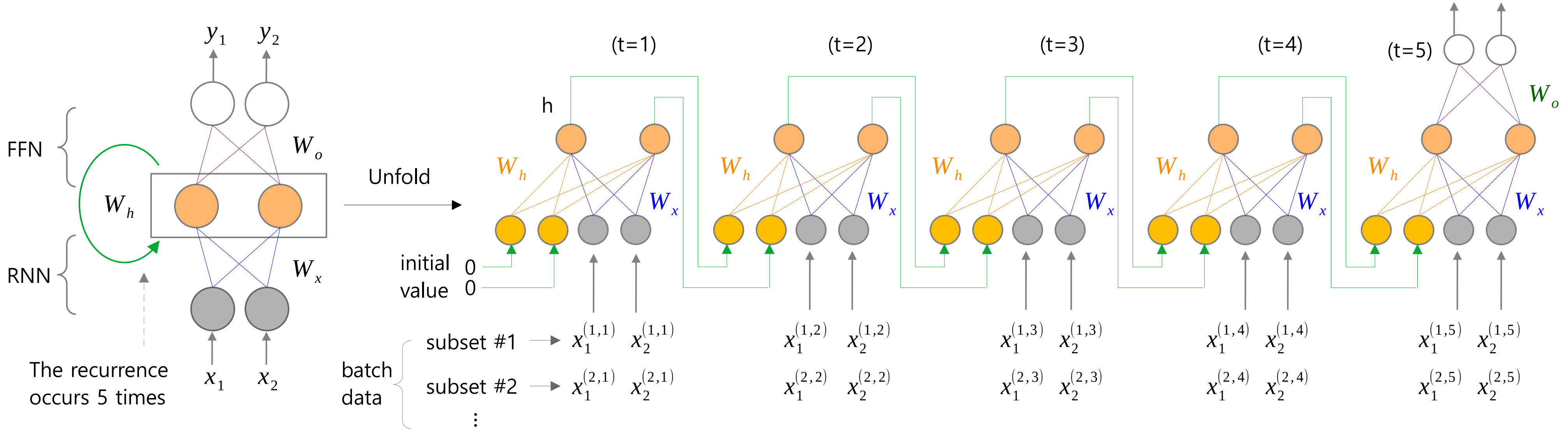
t	x_1	x_2
1	0.63,	0.73
2	0.78,	0.98
3	1.33,	0.73
4	0.93,	0.81
5	1.12,	0.74
6	1.09,	1.18
7	1.25,	0.57

target

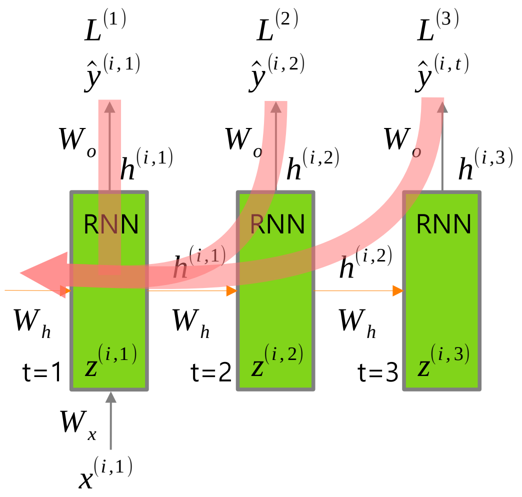
t	y_1	y_2
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
⋮		



[Many-to-One structure]
0
h₁
h₂
h₃
h₄
h₅
$y_1^{(1)}$ $y_2^{(1)}$
$x_1^{(1,1)}$ $x_2^{(1,1)}$
$x_1^{(1,2)}$ $x_2^{(1,2)}$
$x_1^{(1,3)}$ $x_2^{(1,3)}$
$x_1^{(1,4)}$ $x_2^{(1,4)}$
$x_1^{(1,5)}$ $x_2^{(1,5)}$
A simple drawing
target #1 → $y_1^{(1)}$ $y_2^{(1)}$
target #2 → $y_1^{(2)}$ $y_2^{(2)}$
⋮



y_1 y_2
FFN
RNN
W_o
W_h
W_x
x_1 x_2
The recurrence occurs 5 times
Unfold
(t=1)
(t=2)
(t=3)
(t=4)
(t=5)
h
W_h
W_x
W_o
initial value
0
0
batch data
subset #1 → $x_1^{(1,1)}$ $x_2^{(1,1)}$
subset #2 → $x_1^{(2,1)}$ $x_2^{(2,1)}$
⋮
$x_1^{(1,2)}$ $x_2^{(1,2)}$
$x_1^{(2,2)}$ $x_2^{(2,2)}$
$x_1^{(1,3)}$ $x_2^{(1,3)}$
$x_1^{(2,3)}$ $x_2^{(2,3)}$
$x_1^{(1,4)}$ $x_2^{(1,4)}$
$x_1^{(2,4)}$ $x_2^{(2,4)}$
$x_1^{(1,5)}$ $x_2^{(1,5)}$
$x_1^{(2,5)}$ $x_2^{(2,5)}$



$$\frac{\partial h^{(t)}}{\partial W_h} \rightarrow \frac{\partial h^{(t)}}{\partial W_h} + \frac{\partial h^{(t)}}{\partial h^{(t-1)}} \cdot \frac{\partial h^{(t-1)}}{\partial W_h}$$

$$= \frac{\partial h^{(t)}}{\partial W_h} + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \frac{\partial h^{(j)}}{\partial h^{(j-1)}} \right) \frac{\partial h^{(i)}}{\partial W_h}$$

10. Recurrent Neural Networks

Part 2: Backpropagation through time (BPTT)

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

[MXDL-10-02] Deep Learning / Recurrent Neural Network (RNN) – Simple RNN

- Many-to-One structure

subsets (x)

$x^{(1)}$

t	x_1	x_2
1	0.63,	0.73
2	0.78,	0.98
3	1.33,	0.73
4	0.93,	0.81
5	1.12,	0.74

$x^{(1,1)}$

$x^{(1,2)}$

$x^{(2)}$

t		
2		
3		
4		
5		
6	1.09,	1.18

$x^{(3)}$

t		
3		
4		
5		
6		
7	1.25,	0.57

⋮

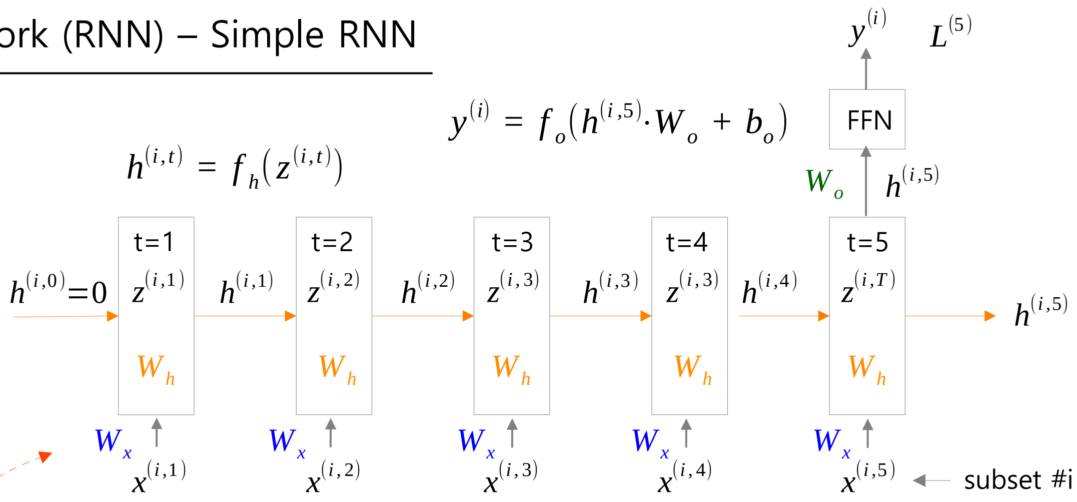
time steps = 5

target (y)

t	y_1	y_2
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
	⋮	

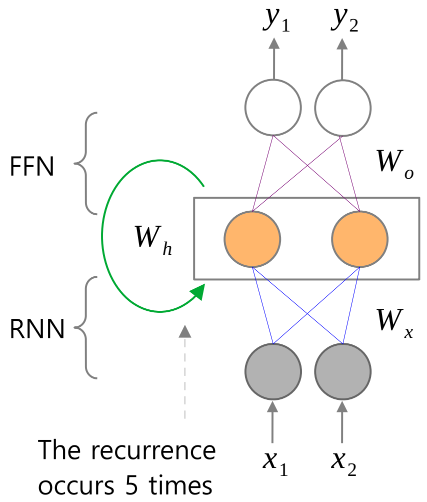
$$y^{(i)} = f_o\left(h^{(i,5)}\cdot W_o + b_o\right)$$

$$h^{(i,t)} = f_h\left(z^{(i,t)}\right)$$

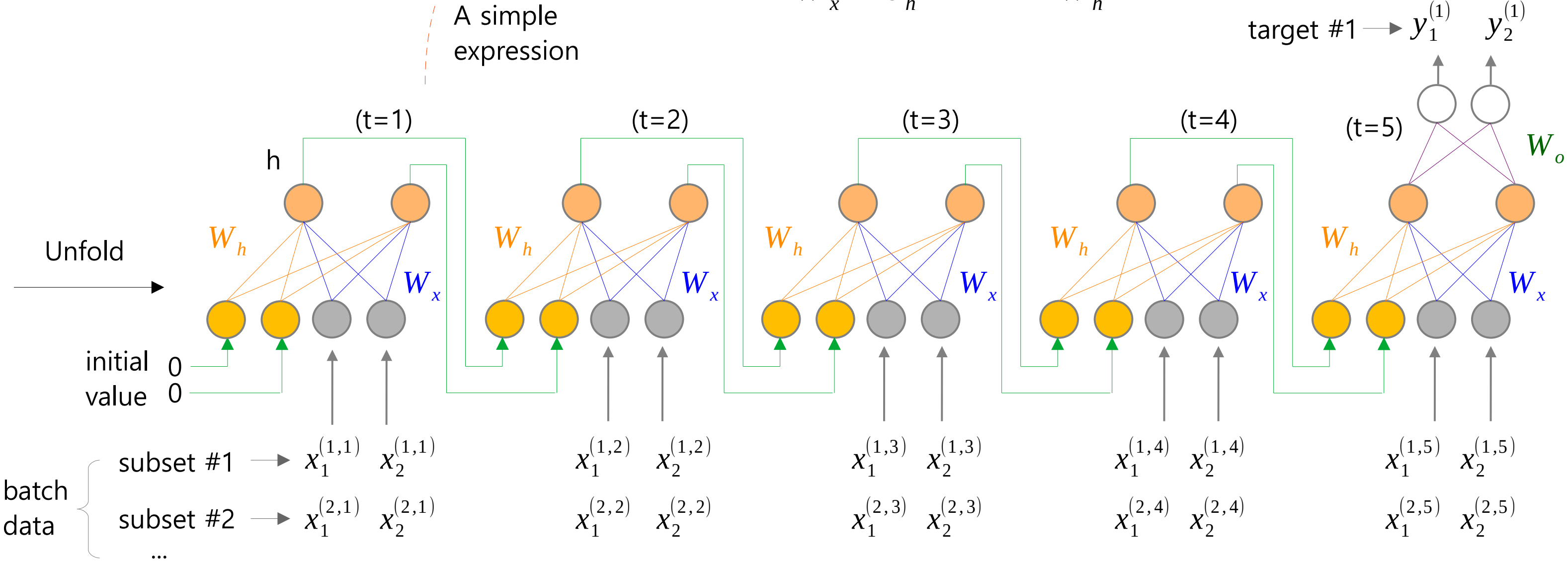


$$z^{(i,t)} = x^{(i,t)}\cdot W_x + b_h + h^{(i,t-1)}\cdot W_h$$

A simple expression



The recurrence occurs 5 times



[MXDL-10-02] Deep Learning / Recurrent Neural Network (RNN) – Simple RNN

- Many-to-Many structure

$x^{(1)}$ subsets (x)

t	x_1	x_2
1	0.63,	0.73
2	0.78,	0.98
3	1.33,	0.73
4	0.93,	0.81
5	1.12,	0.74

$\rightarrow x^{(1,1)}$, $\rightarrow x^{(1,2)}$

$x^{(2)}$

t		
2		
3		
4		
5		
6	1.09,	1.18

T = 5

⋮

$y^{(1)}$ target (y)

t	y_1	y_2
2	0.78,	0.98
3	1.33,	0.73
4	0.93,	0.81
5	1.12,	0.74
6	1.09,	1.18

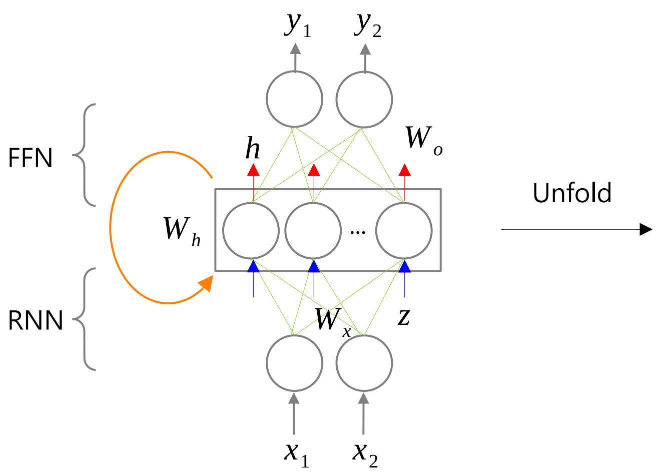
$\rightarrow y^{(1,1)}$, $\rightarrow y^{(1,2)}$

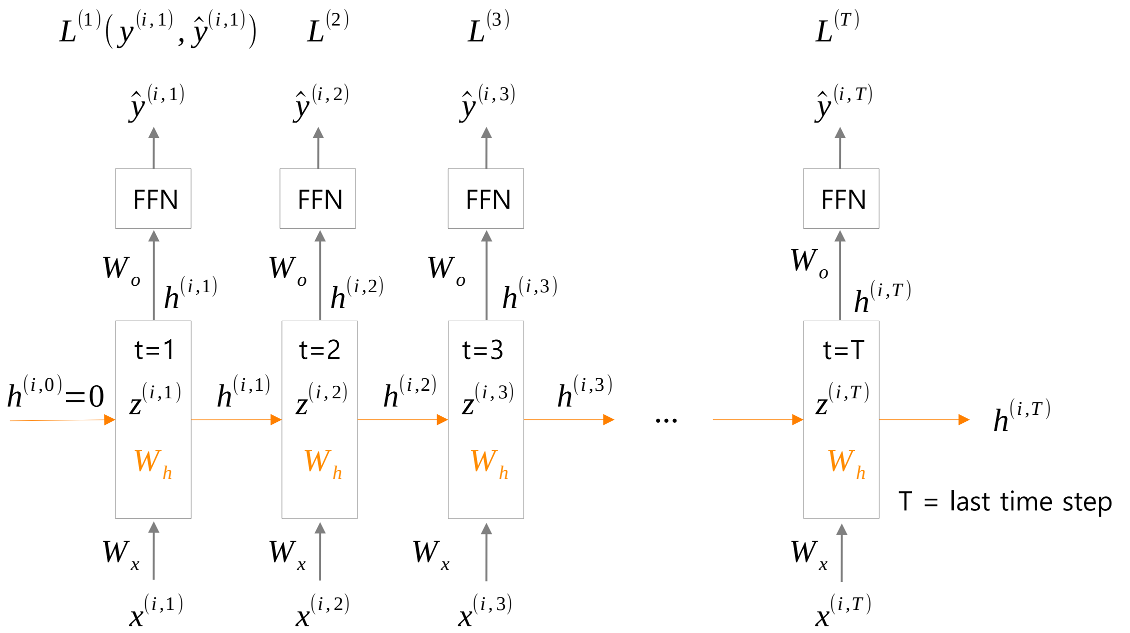
$y^{(2)}$

t		
3		
4		
5		
6		
7	1.25,	0.57

$$h^{(i,t)} = f_h\left(z^{(i,t)}\right)$$

$$z^{(i,t)} = x^{(i,t)}\cdot W_x + b_h + h^{(i,t-1)}\cdot W_h \qquad y^{(i,t)} = f_o\left(h^{(i,t)}\cdot W_o + b_o\right)$$





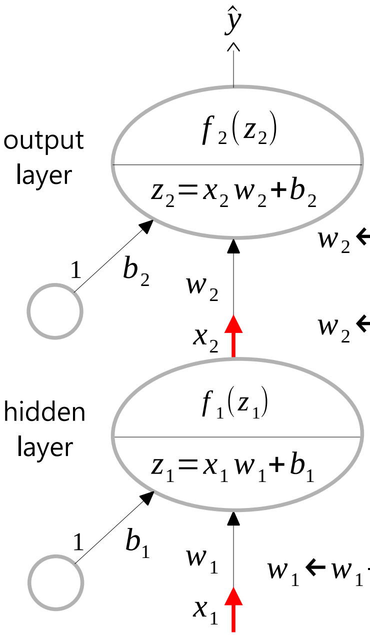
T = last time step

[MXDL-10-02] Deep Learning / Recurrent Neural Network (RNN) – Simple RNN

Backpropagation through time (BPTT)

* Source: https://d2l.ai/chapter_recurrent-neural-networks/bptt.html

- Backpropagation of feedforward network



$L(\hat{y})$

$\hat{y}=f_2(z_2)$

$x_2=f_1(z_1)$

$$w_2 \leftarrow w_2-\alpha\frac{\partial L}{\partial w_2}$$

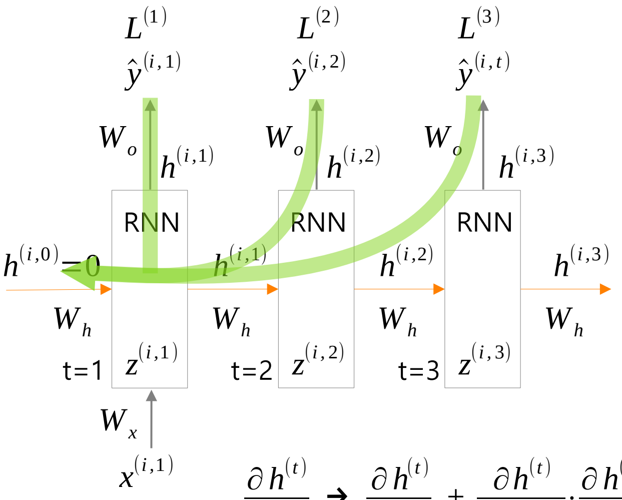
$$w_2 \leftarrow w_2-\alpha\frac{\partial L}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial z_2}\cdot\frac{\partial z_2}{\partial w_2}$$

$$w_1 \leftarrow w_1-\alpha\frac{\partial L}{\partial w_1}$$

$$w_1 \leftarrow w_1-\alpha\frac{\partial L}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial z_2}\cdot\frac{\partial z_2}{\partial x_2}\cdot\frac{\partial x_2}{\partial z_1}\cdot\frac{\partial z_1}{\partial w_1}$$

Let's assume that w₁ = w₂ = w, like an RNN (common weights)

$$w \leftarrow w-\alpha\frac{\partial L}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial z_2}\left(\frac{\partial z_2}{\partial w}+\frac{\partial z_2}{\partial z_1}\cdot\frac{\partial z_1}{\partial w}\right)$$



$$L=\frac{1}{T}\sum_{t=1}^{T}L^{(t)}\left(\hat{y}^{(t)},y^{(t)}\right)$$

$$h^{(t)}=f_h\left(x^{(t)}\cdot W_x+b_h+h^{(t-1)}\cdot W_h\right)$$

$$\frac{\partial L}{\partial W_h}=\frac{1}{T}\sum_{t=1}^{T}\frac{\partial L^{(t)}}{\partial W_h}=\frac{1}{T}\sum_{t=1}^{T}\overbrace{\frac{\partial L^{(t)}}{\partial \hat{y}^{(t)}}\cdot\frac{\partial \hat{y}^{(t)}}{\partial h^{(t)}}}^{\text{easy to compute}}\cdot\frac{\partial h^{(t)}}{\partial W_h}$$

It is a little tricky to compute.

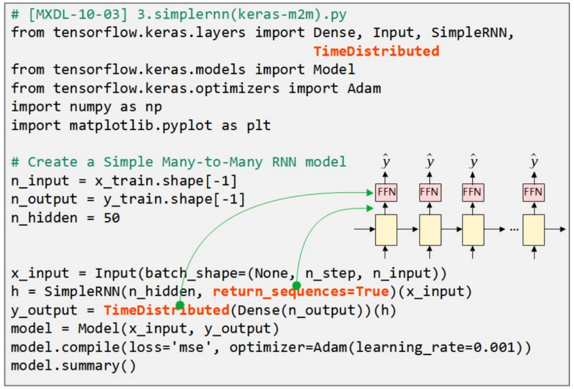
$$\frac{\partial h^{(t)}}{\partial W_h}\rightarrow\frac{\partial h^{(t)}}{\partial W_h}+\frac{\partial h^{(t)}}{\partial h^{(t-1)}}\cdot\frac{\partial h^{(t-1)}}{\partial W_h}=\frac{\partial h^{(t)}}{\partial W_h}+\sum_{i=1}^{t-1}\left(\underbrace{\prod_{j=i+1}^{t}\frac{\partial h^{(j)}}{\partial h^{(j-1)}}}\right)\frac{\partial h^{(i)}}{\partial W_h}$$

$$\prod_{j=i+1}^{t}W_h\cdot f'_h(.)=W_h^{t-i-1}\prod_{j=i+1}^{t}f'_h(.)$$

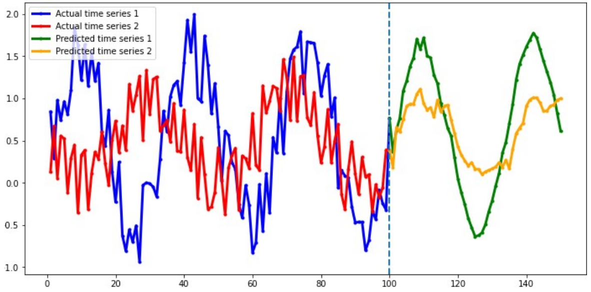
$$a_t=b_t+c_ta_{t-1},\quad t=1,2,3...$$
$$=b_t+c_t(b_{t-1}+c_{t-1}a_{t-2})$$
$$=b_t+\sum_{i=1}^{t-1}\left(\prod_{j=i+1}^{t}c_j\right)b_i$$

* Problems:
- W_h > 1 → Exploding gradient
- W_h < 1 or f'_h < 1 → Vanishing gradient

- To solve this problem, truncated BPTT was also designed. (Jaeger, 2002, Tutorial on training recurrent neural networks, covering BPTT, RTRL, EKF and the "echo state network" approach.)



```
# [MXDL-10-03] 3.simplernn(keras-m2m).py
from tensorflow.keras.layers import Dense, Input, SimpleRNN,
                                                TimeDistributed
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt

# Create a Simple Many-to-Many RNN model
n_input = x_train.shape[-1]
n_output = y_train.shape[-1]
n_hidden = 50


x_input = Input(batch_shape=(None, n_step, n_input))
h = SimpleRNN(n_hidden, return_sequences=True)(x_input)
y_output = TimeDistributed(Dense(n_output))(h)
model = Model(x_input, y_output)
model.compile(loss='mse', optimizer=Adam(learning_rate=0.001))
model.summary()
```



10. Recurrent Neural Networks

Part 3: Implementation of RNN models

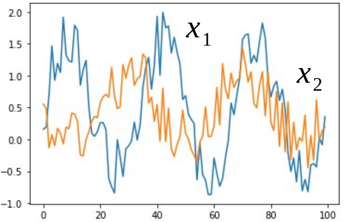
This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

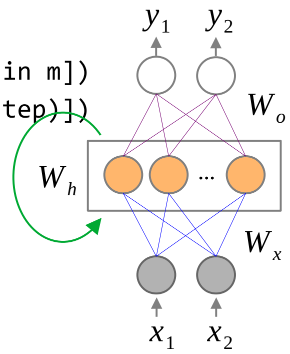
[MXDL-10-03] Deep Learning / Recurrent Neural Network (RNN) – Simple RNN

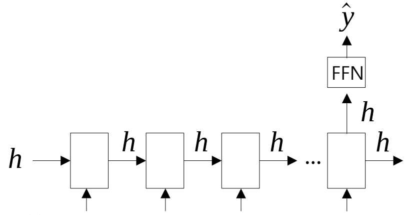
- Implementation of a simple RNN model using Keras custom layer: Many-to-One structure

```
# [MXDL-10-03] 1.simplernn(m2o).py
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Layer
from tensorflow.keras.models import Model
from tensorflow.keras import initializers
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
```



```
# Generate training data: 2 noisy sine curves
n = 1000        # the number of data points
n_step = 20     # the number of time steps
s1 = np.sin(np.pi * 0.06 * np.arange(n)) + np.random.random(n)
s2 = 0.5*np.sin(np.pi * 0.05 * np.arange(n)) + np.random.random(n)
data = np.vstack([s1, s2]).T  # shape = (1000, 2)

m = np.arange(0, n - n_step)
x_train = np.array([data[i:(i+n_step), :] for i in m])
y_train = np.array([data[i, :] for i in (m + n_step)])

# Create a simple RNN model
n_feat = x_train.shape[-1]    # 2
n_output = y_train.shape[-1]  # 2
n_hidden = 50
```





```
class MySimpleRNN(Layer):
   # nf: the number of features,
   # nh: the number of hidden units
   def __init__(self, nf, nh):
      super().__init__()
      self.nh = nh
      w_init = initializers.GlorotUniform()
      b_init = tf.zeros_initializer()
      self.wx = tf.Variable(w_init([nf, nh]), trainable = True)
      self.wh = tf.Variable(w_init([nh, nh]), trainable = True)
      self.b = tf.Variable(b_init([1, nh]), trainable = True)

   def call(self, x):
      h = tf.zeros(shape=(tf.shape(x)[0], self.nh)) # init. values
      for t in range(tf.shape(x)[1]): # Recurrence
         # shape: (None, nf)*(nf, nh) + (1, nh)*(nh, nh) + (1, nh)
         z = tf.matmul(x[:, t, :], self.wx) + \
             tf.matmul(h, self.wh) + self.b
         h = tf.math.tanh(z)
      return h
```

$$z^{(i,t)} = x^{(i,t)} \cdot W_x + b_h + h^{(i,t-1)} \cdot W_h$$

$$h^{(i,t)} = f_h\left(z^{(i,t)}\right)$$

```
# Create a Many-to-One RNN model
x_input = Input(batch_shape=(None, n_step, n_feat))
h = MySimpleRNN(n_feat, n_hidden)(x_input)
y_output = Dense(n_output)(h)
model = Model(x_input, y_output)
model.compile(loss='mse', optimizer=Adam(learning_rate=0.001))
model.summary()  # trainable parameters = 2,752
```

[MXDL-10-03] Deep Learning / Recurrent Neural Network (RNN) – Simple RNN

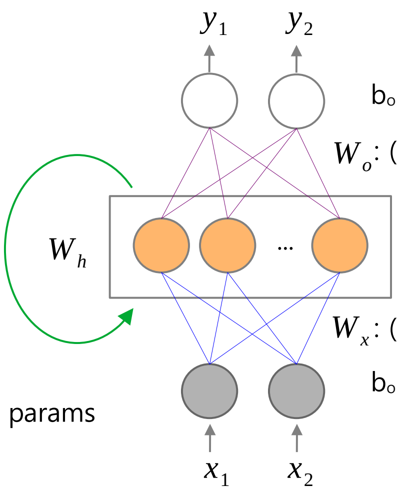
- Implementation of a simple RNN model using Keras custom layer: Many-to-One structure

```
# Training
hist = model.fit(x_train, y_train, epochs=100, batch_size=50)

# Loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'], color='red')
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()

# Predict future values for the next 50 periods.
# After predicting the next value, re-enter the predicted value
# to predict the next value. Repeat this process 50 times.
n_future = 50
n_last = 100
last_data = data[-n_last:]  # The last n_last data points
for i in range(n_future):
    # Predict the next value with the last n_step data points.
    px = last_data[-n_step:, :].reshape(1, n_step, 2)

    # Predict the next value
    y_hat = model.predict(px, verbose=0)

    # Append the predicted value to the last_data array.
    # In the next iteration, the predicted value is input
    # along with the existing data points.
    last_data = np.vstack([last_data, y_hat])
```

```
p = last_data[:-n_future, :]          # past time series
f = last_data[-(n_future + 1):, :]   # future time series

# Plot past and future time series.
plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(p) + 1)
ax2 = np.arange(len(p), len(p) + len(f))
plt.plot(ax1, p[:, 0], '-o', c='blue', markersize=3,
         label='Actual time series 1', linewidth=1)
plt.plot(ax1, p[:, 1], '-o', c='red', markersize=3,
         label='Actual time series 2', linewidth=1)
plt.plot(ax2, f[:, 0], '-o', c='green', markersize=3,
         label='Estimated time series 1')
plt.plot(ax2, f[:, 1], '-o', c='orange', markersize=3,
         label='Estimated time series 2')
plt.axvline(x=ax1[-1],  linestyle='dashed', linewidth=1)
plt.legend()
plt.show()
```

[MXDL-10-03] Deep Learning / Recurrent Neural Network (RNN) – Simple RNN

- Implementation of a simple RNN model using Keras custom layer: Many-to-One structure



b_o: (1, 2) = 2 params

W_o: (50, 2) = 100 params

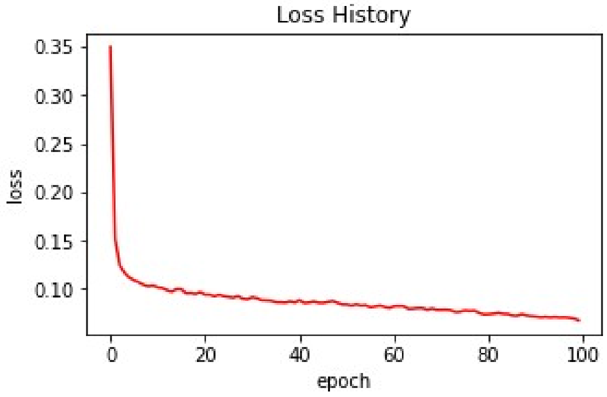
50 neurons

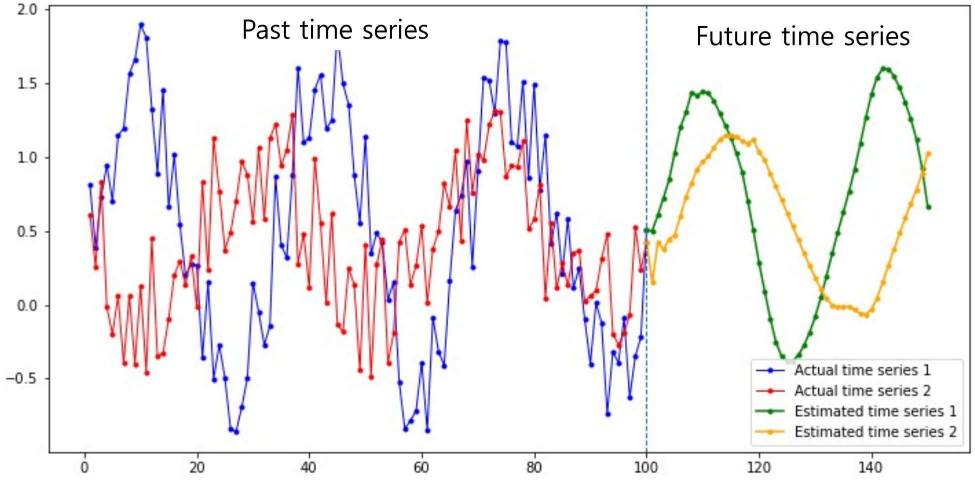
W_x: (2, 50) = 100 params

b_o: (1, 50) = 50 params

W_h: (50, 50)
= 2500 params

```
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 20, 2)]           0

 my_simple_rnn (MySimpleRNN)  (None, 50)               2650

 dense (Dense)                (None, 2)                102

=================================================================
Total params: 2,752
Trainable params: 2,752
Non-trainable params: 0
_________________________________________________________________
```





[MXDL-10-03] Deep Learning / Recurrent Neural Network (RNN) – Simple RNN

- Implementation of a simple RNN model using Keras' SimpleRNN class: Many-to-One structure

```
# [MXDL-10-03] 2.simplernn(keras-m2o).py
from tensorflow.keras.layers import Dense, Input, SimpleRNN
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt

# Generate training data: 2 noisy sine curves
n = 1000        # the number of data points
n_step = 20     # the number of time steps
s1 = np.sin(np.pi * 0.06 * np.arange(n)) + np.random.random(n)
s2 = 0.5*np.sin(np.pi * 0.05 * np.arange(n)) + np.random.random(n)
data = np.vstack([s1, s2]).T

m = np.arange(0, n - n_step)
x_train = np.array([data[i:(i+n_step), :] for i in m])
y_train = np.array([data[i, :] for i in (m + n_step)])

n_feat = x_train.shape[-1]
n_output = y_train.shape[-1]
n_hidden = 50

# Create a Many-to-One RNN model
x_input = Input(batch_shape=(None, n_step, n_feat))
h = SimpleRNN(n_hidden)(x_input)
y_output = Dense(n_output)(h)

model = Model(x_input, y_output)
model.compile(loss='mse', optimizer=Adam(learning_rate=0.001))
model.summary()
```

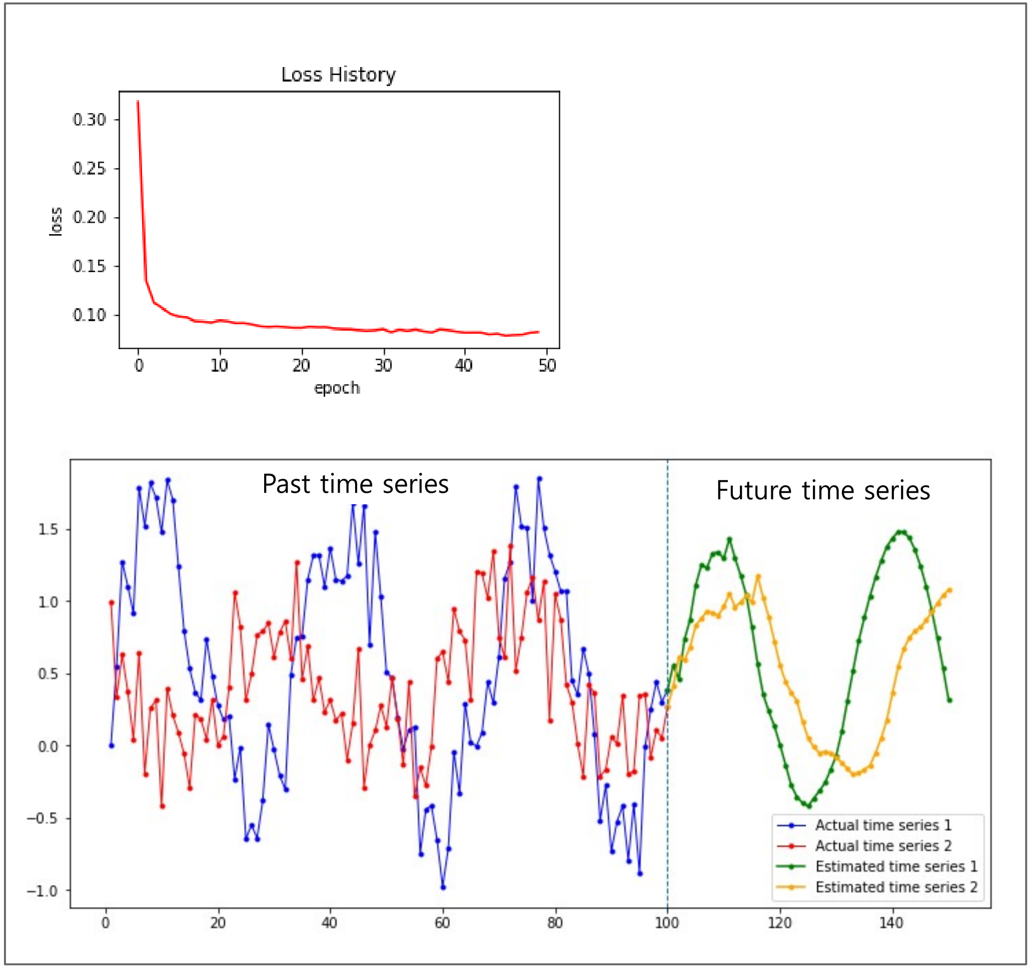
```
# Training
hist = model.fit(x_train, y_train, epochs=50, batch_size=50)

# Loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'], color='red')
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()

# Predict future values for the next 50 periods.
# After predicting the next value, re-enter the predicted value
# to predict the next value. Repeat this process 50 times.
n_future = 50
n_last = 100
last_data = data[-n_last:]  # The last n_last data points
for i in range(n_future):
    # Predict the next value with the last n_step data points.
    px = last_data[-n_step:, :].reshape(1, n_step, 2)

    # Predict the next value
    y_hat = model.predict(px, verbose=0)

    # Append the predicted value to the last_data array.
    # In the next iteration, the predicted value is input
    # along with the existing data points.
    last_data = np.vstack([last_data, y_hat])
```

[MXDL-10-03] Deep Learning / Recurrent Neural Network (RNN) – Simple RNN

- Implementation of a simple RNN model using Keras' SimpleRNN class: Many-to-One structure

```
p = last_data[:-n_future, :]        # past time series
f = last_data[-(n_future + 1):, :]  # future time series

# Plot past and future time series.
plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(p) + 1)
ax2 = np.arange(len(p), len(p) + len(f))
plt.plot(ax1, p[:, 0], '-o', c='blue', markersize=3,
         label='Actual time series 1', linewidth=1)
plt.plot(ax1, p[:, 1], '-o', c='red', markersize=3,
         label='Actual time series 2', linewidth=1)
plt.plot(ax2, f[:, 0], '-o', c='green', markersize=3,
         label='Predicted time series 1')
plt.plot(ax2, f[:, 1], '-o', c='orange', markersize=3,
         label='Predicted time series 2')
plt.axvline(x=ax1[-1],  linestyle='dashed', linewidth=1)
plt.legend()
plt.show()
```

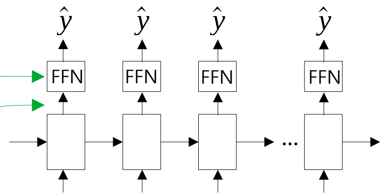
```
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 20, 2)]           0
 simple_rnn (SimpleRNN)      (None, 50)                2650
 dense (Dense)               (None, 2)                 102
=================================================================
Total params: 2,752
Trainable params: 2,752
Non-trainable params: 0
```



[MXDL-10-03] Deep Learning / Recurrent Neural Network (RNN) – Simple RNN

■ Implementation of a simple RNN model using Keras’ SimpleRNN class: Many-to-Many structure

```
# [MXDL-10-03] 3.simplernn(keras-m2m).py
from tensorflow.keras.layers import Dense, Input, SimpleRNN,
                                    TimeDistributed
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt

# Generate training data: 2 noisy sine curves
n = 1000        # the number of data points
n_step = 20     # the number of time steps
s1 = np.sin(np.pi * 0.06 * np.arange(n)) + np.random.random(n)
s2 = 0.5*np.sin(np.pi * 0.05 * np.arange(n)) + np.random.random(n)
data = np.vstack([s1, s2]).T
m = np.arange(0, n - n_step)
x_train = np.array([data[i:(i+n_step), :] for i in m])
y_train = np.array([data[(i+1):(i+1+n_step), :] for i in m])

# Create a Simple Many-to-Many RNN model
n_feat = x_train.shape[-1]
n_output = y_train.shape[-1]
n_hidden = 50
```



```
x_input = Input(batch_shape=(None, n_step, n_feat))
h = SimpleRNN(n_hidden, return_sequences=True)(x_input)
y_output = TimeDistributed(Dense(n_output))(h)
model = Model(x_input, y_output)
model.compile(loss='mse', optimizer=Adam(learning_rate=0.001))
model.summary()
```

```
# Training
hist = model.fit(x_train, y_train, epochs=50, batch_size=50)

# Loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'], color='red')
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()

# Predict future values for the next 50 periods.
# After predicting the next value, re-enter the predicted value
# to predict the next value. Repeat this process 50 times.
n_future = 50
n_last = 100
last_data = data[-n_last:]  # The last n_last data points
for i in range(n_future):
    # Predict the next value with the last n_step data points.
    px = last_data[-n_step:, :].reshape(1, n_step, 2)

    # Predict the next value. In the prediction stage, only the
    # output of the final time step is used.
    y_hat = model.predict(px, verbose=0)[:, -1, :]

    # Append the predicted value to the last_data array.
    # In the next iteration, the predicted value is input
    # along with the existing data points.
    last_data = np.vstack([last_data, y_hat])
```

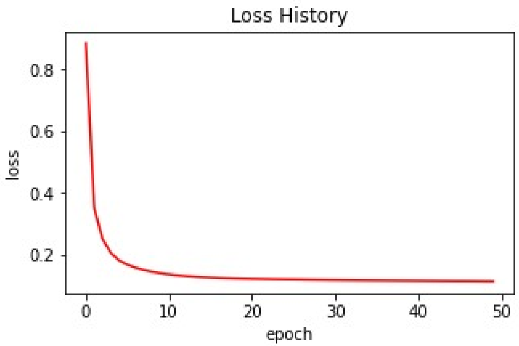
[MXDL-10-03] Deep Learning / Recurrent Neural Network (RNN) – Simple RNN

MX-AI

- Implementation of a simple RNN model using Keras' SimpleRNN class: Many-to-Many structure

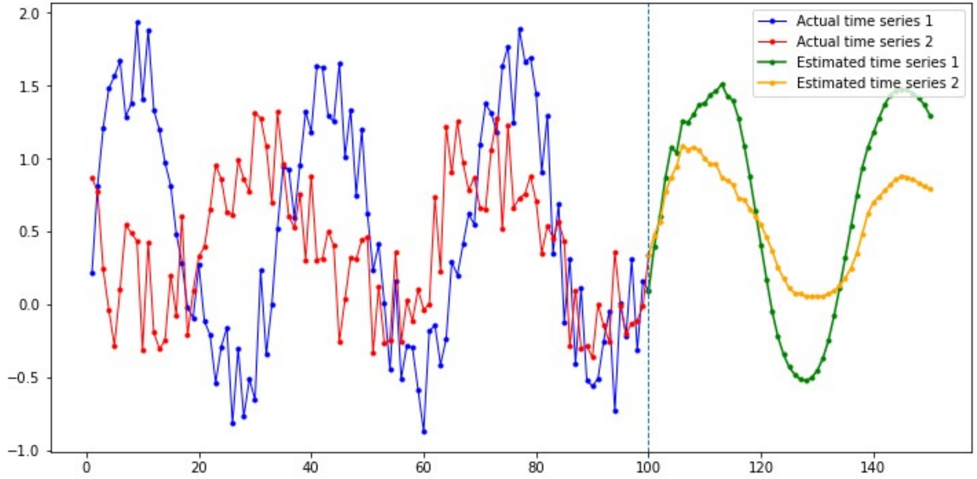
```
p = last_data[:-n_future, :]        # past time series
f = last_data[-(n_future + 1):, :]  # future time series

# Plot past and future time series.
plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(p) + 1)
ax2 = np.arange(len(p), len(p) + len(f))
plt.plot(ax1, p[:, 0], '-o', c='blue', markersize=3,
         label='Actual time series 1', linewidth=1)
plt.plot(ax1, p[:, 1], '-o', c='red', markersize=3,
         label='Actual time series 2', linewidth=1)
plt.plot(ax2, f[:, 0], '-o', c='green', markersize=3,
         label='Predicted time series 1')
plt.plot(ax2, f[:, 1], '-o', c='orange', markersize=3,
         label='Predicted time series 2')
plt.axvline(x=ax1[-1],  linestyle='dashed', linewidth=1)
plt.legend()
plt.show()
```

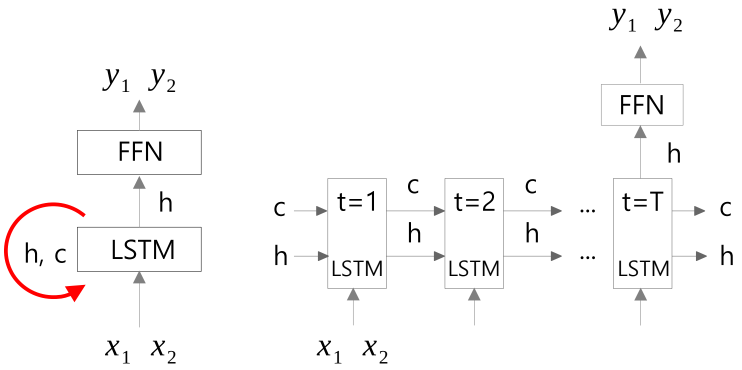
```
 Layer (type)                      Output Shape        Param #
=================================================================
 input_1 (InputLayer)             [(None, 20, 2)]     0
 simple_rnn (SimpleRNN)            (None, 20, 50)     2650
 time_distributed (TimeDistributed)  (None, 20, 2)    102
=================================================================
Total params: 2,752
Trainable params: 2,752
Non-trainable params: 0
```

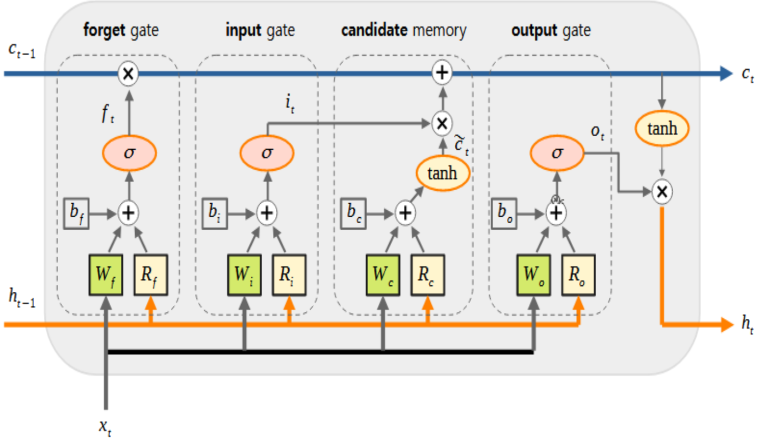


Past time series

Future time series







10. Recurrent Neural Networks

Part 4: Long Short-Term Memory (LSTM)

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

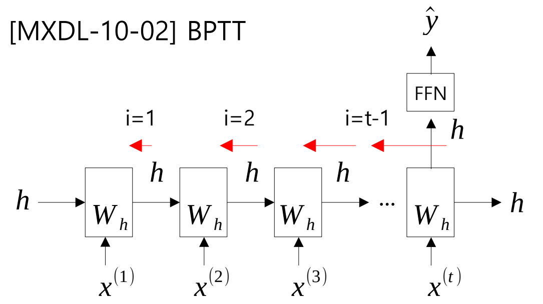
www.youtube.com/@meanxai

[MXDL-10-04] Deep Learning / Recurrent Neural Network (RNN) – LSTM

Brief history of LSTM

- The simple RNN struggles to learn long-term dependencies because of vanishing or exploding gradients. To solve this problem, many studies have been actively conducted since 1990. Sepp Hochreiter and Jürgen Schmidhuber briefly reviewed Hochreiter's analysis of this problem in 1991, then addressed it by introducing a novel, efficient, gradient based method called long short-term memory (LSTM). At that time, the LSTM had a structure consisting of cell states and input and output gates.
- In 1999, Felix Gers et al. added a forget gate to the existing LSTM through the paper, "Learning to forget: Continual prediction with LSTM"

[MXDL-10-02] BPTT



$$\frac{\partial h^{(t)}}{\partial W_h} \rightarrow \frac{\partial h^{(t)}}{\partial W_h} + \sum_{i=1}^{t-1} [(W_h^{t-i-1} \prod_{j=i+1}^{t} f'_h(.)) \frac{\partial h^{(i)}}{\partial W_h}]$$

Long Short-Term Memory

Neural Computation (1997) 9 (8): 1735–1780.

Sepp Hochreiter
Fakultat fur Informatik
Technische Universitat Munchen
80290 Munchen, Germany
hochreit@informatik.tu-muenchen.de
www7.informatik.tu-muenchen.de/~hochreit

Jurgen Schmidhuber
IDSIA
Corso Elvezia 36
6900 Lugano, Switzerland
juergen@idsia.ch
http://www.idsia.ch/~juergen

Abstract

Learning to store information over extended time intervals by recurrent backpropagation takes a very long time, mostly because of insufficient, decaying error back flow. We briefly review Hochreiter's (1991) analysis of this problem, then address it by introducing a novel, efficient, gradient based method called long short-term memory (LSTM). Truncating the gradient where this does not do harm, LSTM can learn to bridge minimal time lags in excess of 1000 discrete time steps by enforcing constant error …

Learning to Forget: Continual Prediction with LSTM

Technical Report IDSIA-01-99
January, 1999

Felix A. Gers
felix@idsia.ch

Jurgen Schmidhuber
juergen@idsia.ch

Fred Cummins
fred@idsia.ch

IDSIA, Corso Elvezia 36 6900 Lugano, Switzerland, www.idsia.ch

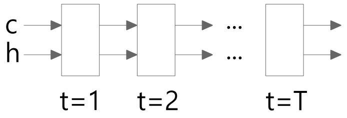
Abstract

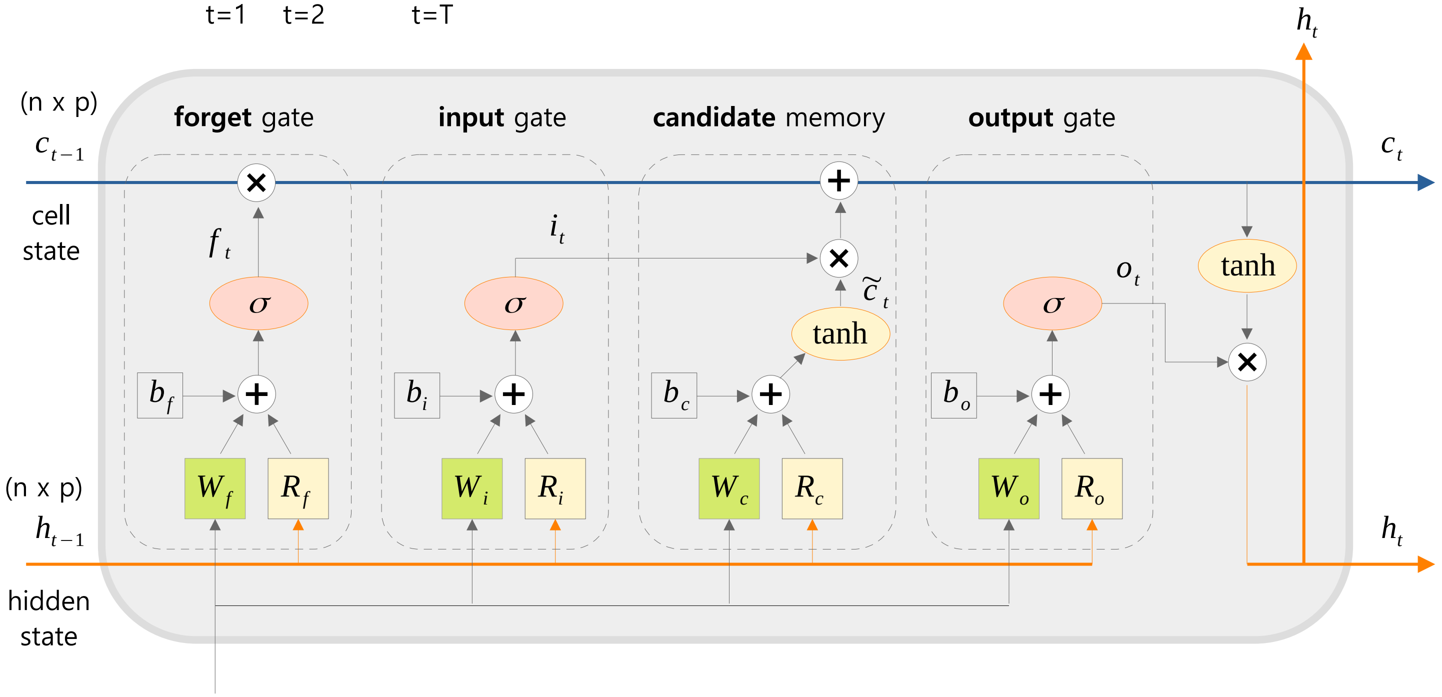
Long Short-Term Memory (LSTM, Hochreiter & Schmidhuber, 1997) can solve numerous tasks not solvable by previous learning algorithms for recurrent neural networks (RNNs). We identify a weakness of LSTM networks processing continual input streams that are not a priori segmented into subsequences with explicitly marked ends at which the network's internal state could be reset. Without resets, the state may grow indefinitely and eventually cause the network to break down. Our remedy is a novel, adaptive "**forget gate**" that enables an LSTM cell to learn to reset itself at appropriate times, thus releasing internal resources. We review illustrative benchmark problems on which standard LSTM outperforms other RNN algorithms. …

[MXDL-10-04] Deep Learning / Recurrent Neural Network (RNN) – LSTM

MX-AI

- Cell structure of LSTM





$$f_t = \sigma(x_t \cdot W_f + h_{t-1} \cdot R_f + b_f)$$

$$i_t = \sigma(x_t \cdot W_i + h_{t-1} \cdot R_i + b_i)$$

$$\tilde{c}_t = \tanh(x_t \cdot W_c + h_{t-1} \cdot R_c + b_c)$$

$$c_t = c_{t-1} \odot f_t + \tilde{c}_t \odot i_t$$

$$o_t = \sigma(x_t \cdot W_o + h_{t-1} \cdot R_o + b_o)$$

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

x_t : input data (n x m)

n : batch size
m : the number of features
p : hidden units

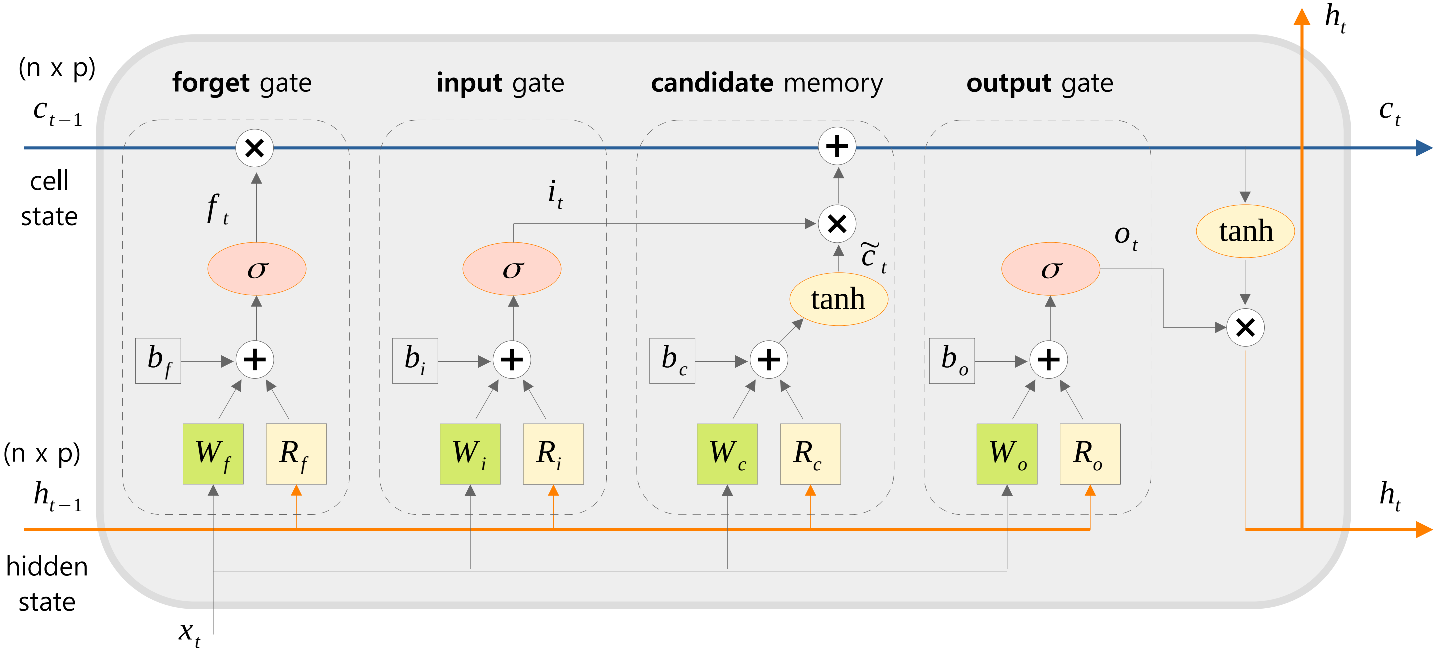
W : (m x p)
R : (p x p)

b : (1 x p)
$f_t, i_t, \tilde{c}_t, c_t, o_t$: (n x p)

MX-AI

The role of gates in LSTM

- **Forget gate**: It controls how much of the previous cell state information is passed on to the next time step. Since the activation function of the forget gate is sigmoid, ft is between 0 and 1. If ft is 0, the forget gate is closed. That is, the cell state is reset. If ft is 1.0, 100% of c(t-1) is passed to remember all past memories.
- **Candidate memory**: It uses the current input data x and the previous hidden state h(t-1) to determine new information, c tilde.
- **Input gate**: It determines how much of the new information c tilde will be reflected. If the new information is important, the value of i increases to store more information, and if it is not important, the value of i decreases to store less information.
- **Output gate**: It determines the size of the next hidden state value.



$$f_t = \sigma(x_t \cdot W_f + h_{t-1} \cdot R_f + b_f)$$

$$i_t = \sigma(x_t \cdot W_i + h_{t-1} \cdot R_i + b_i)$$

$$\tilde{c}_t = \tanh(x_t \cdot W_c + h_{t-1} \cdot R_c + b_c)$$

$$c_t = c_{t-1} \odot f_t + \tilde{c}_t \odot i_t$$

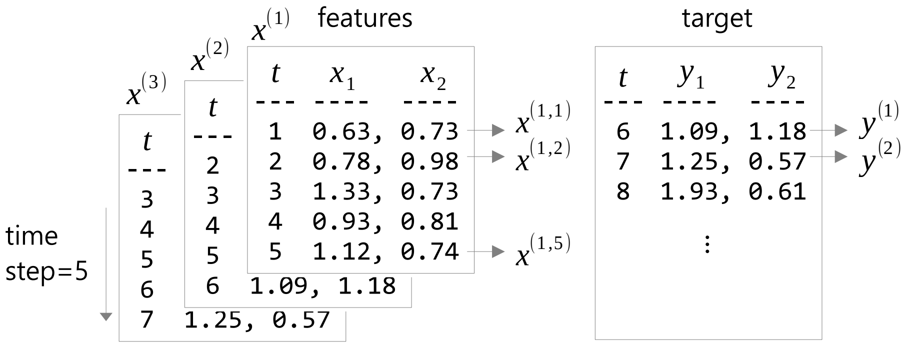
$$o_t = \sigma(x_t \cdot W_o + h_{t-1} \cdot R_o + b_o)$$

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

[MXDL-10-04] Deep Learning / Recurrent Neural Network (RNN) – LSTM

MX-AI

■ Feeding data into LSTM cell



$x^{(1)}$ features

t	x_1	x_2
1	0.63,	0.73
2	0.78,	0.98
3	1.33,	0.73
4	0.93,	0.81
5	1.12,	0.74

$x^{(2)}$

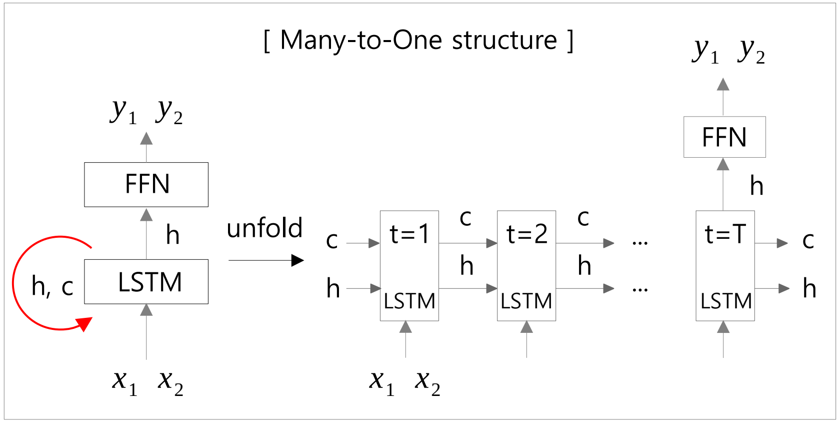
t		
2		
3		
4		
5		
6	1.09,	1.18

$x^{(3)}$

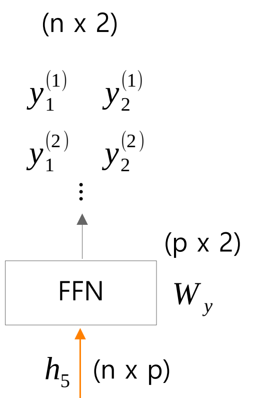
t		
3		
4		
5		
6		
7	1.25,	0.57

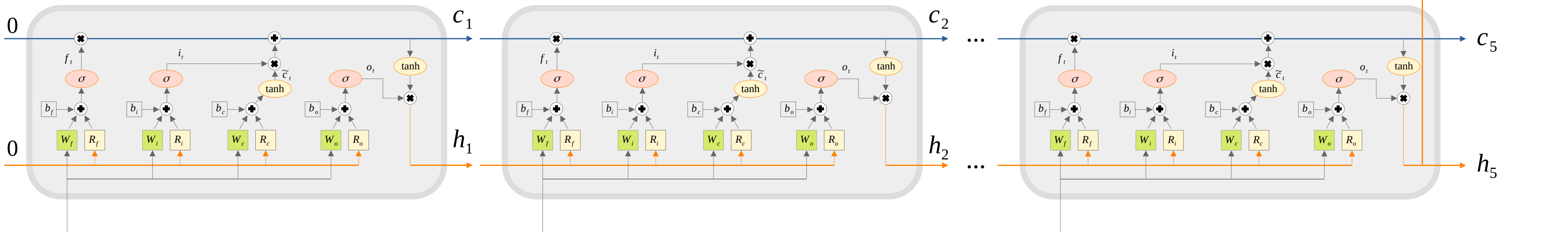
target

t	y_1	y_2
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
	⋮	



[Many-to-One structure]





$x_1^{(1,1)}$ $x_2^{(1,1)}$

$x_1^{(2,1)}$ $x_2^{(2,1)}$

⋮

(n x 2)
(n: batch size,
2: the number of features)

$x_1^{(1,2)}$ $x_2^{(1,2)}$

$x_1^{(2,2)}$ $x_2^{(2,2)}$

⋮

$W_{f,i,c,o}$
$R_{f,i,c,o}$ } (2 x p)

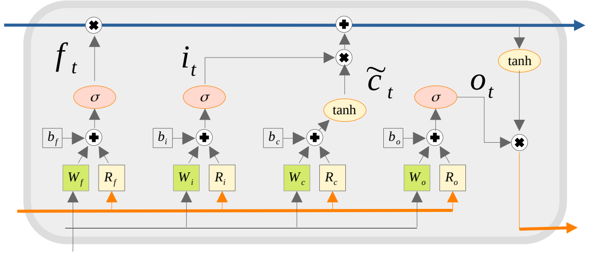
$x_1^{(1,5)}$ $x_2^{(1,5)}$ ← subset #1

$x_1^{(2,5)}$ $x_2^{(2,5)}$ ← subset #2

⋮ …

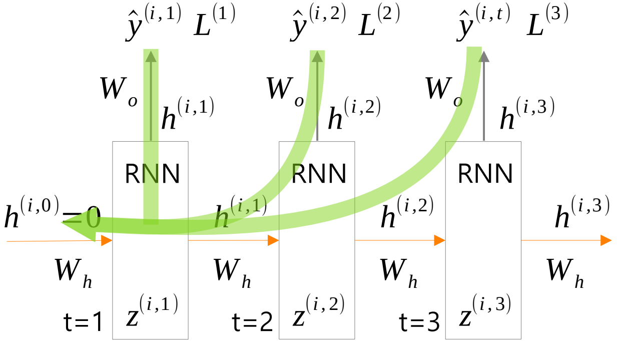
} batch data

[MXDL-10-04] Deep Learning / Recurrent Neural Network (RNN) – LSTM



Backpropagation through time (BPTT) of LSTM

- BPTT of vanilla RNN



- [MXDL-10-02] BPTT of vanilla RNN

$$\frac{\partial h^{(t)}}{\partial W_h} \rightarrow \frac{\partial h^{(t)}}{\partial W_h} + \frac{\partial h^{(t)}}{\partial h^{(t-1)}} \cdot \frac{\partial h^{(t-1)}}{\partial W_h} = \frac{\partial h^{(t)}}{\partial W_h} + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \frac{\partial h^{(j)}}{\partial h^{(j-1)}} \right) \frac{\partial h^{(i)}}{\partial W_h}$$

f: forget, i: input

⊙: element-wise product

$$\prod_{j=i+1}^{t} W_h \cdot f'_h(.) = W_h^{t-i-1} \prod_{j=i+1}^{t} f'_h(.)$$

- $W_h > 1$ → Exploding gradient
- $f'_h < 1$ or $W_h < 1$ → Vanishing gradient

- BPTT of LSTM

Source: weberna.github.io/blog/2017/11/15/LSTM-Vanishing-Gradients.html

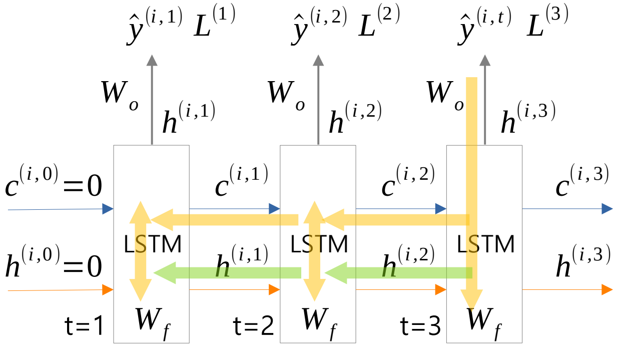
$$\frac{\partial c^{(t)}}{\partial W_f} \rightarrow \frac{\partial c^{(t)}}{\partial W_f} + \frac{\partial c^{(t)}}{\partial c^{(t-1)}} \cdot \frac{\partial c^{(t-1)}}{\partial W_f} = \frac{\partial c^{(t)}}{\partial W_f} + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \frac{\partial c^{(j)}}{\partial c^{(j-1)}} \right) \frac{\partial c^{(i)}}{\partial W_f}$$

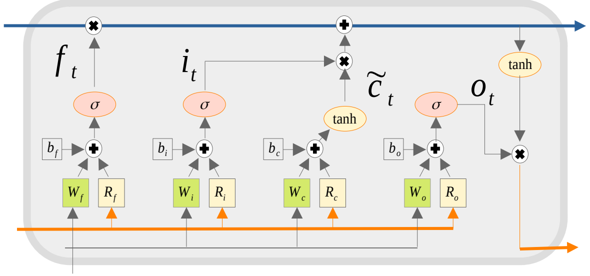
$$\frac{\partial c^{(j)}}{\partial c^{(j-1)}} = \frac{\partial c^{(j)}}{\partial f^{(j)}} \frac{\partial f^{(j)}}{\partial h^{(j-1)}} \frac{\partial h^{(j-1)}}{\partial c^{(j-1)}} + \frac{\partial c^{(j)}}{\partial i^{(j)}} \frac{\partial i^{(j)}}{\partial h^{(j-1)}} \frac{\partial h^{(j-1)}}{\partial c^{(j-1)}} + \frac{\partial c^{(j)}}{\partial \widetilde{C}^{(j)}} \frac{\partial \widetilde{C}^{(j)}}{\partial h^{(j-1)}} \frac{\partial h^{(j-1)}}{\partial c^{(j-1)}} + \frac{\partial C^{(j)}}{\partial C^{(j-1)}}$$

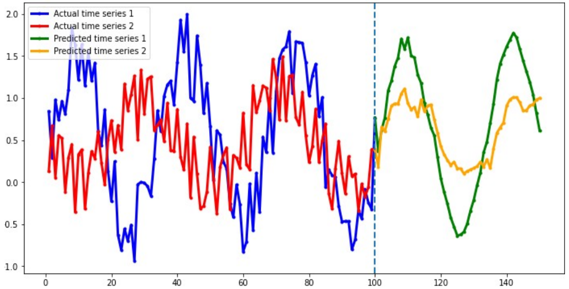
$$= c^{(j-1)} \sigma'(.) W_f o_{j-1} \tanh'(c^{(j-1)}) + \widetilde{C}^{(j)} \sigma'(.) W_i o_{t-1} \tanh'(c^{(j-1)})$$

$$+ i^{(j)} \tanh'(.) W_c o_{j-1} \tanh'(c^{(j-1)}) + f^{(j)}$$

- BPTT of LSTM







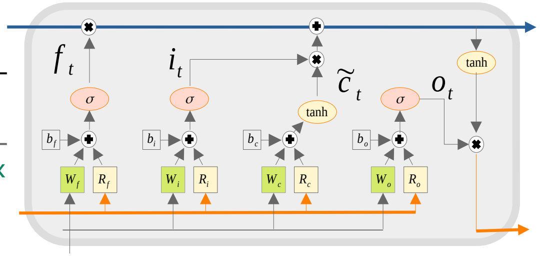
10. Recurrent Neural Networks

Part 5: Implementation of LSTM models

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

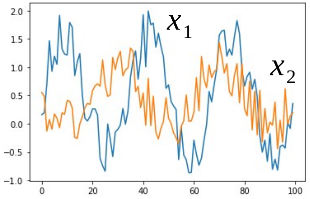
www.youtube.com/@meanxai

[MXDL-10-05] Deep Learning / Recurrent Neural Network (RNN) – LSTM



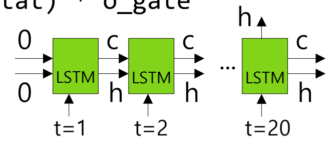
- Implementation of a LSTM model using Keras custom layer: Many-to-One

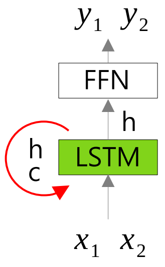
```
# [MXDL-10-05] 4.LSTM(m2o).py (Many-to-One)
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Layer
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt

# Generate training data: 2 noisy sine curves
n = 1000        # the number of data points
n_step = 20     # the number of time steps
s1 = np.sin(np.pi * 0.06 * np.arange(n)) + np.random.random(n)
s2 = 0.5*np.sin(np.pi * 0.05 * np.arange(n)) + np.random.random(n)
data = np.vstack([s1, s2]).T  # shape = (1000, 2)

m = np.arange(0, n - n_step)
x_train = np.array([data[i:(i+n_step), :] for i in m])
y_train = np.array([data[i, :] for i in (m + n_step)])

n_hidden = 50   # the number of hidden
                # units

# LSTM custom layer
class MyLSTM(Layer):
   def __init__(self, n_hidden):
      super().__init__()
      self.nh = n_hidden
```



$$f_t = \sigma\left(x_t \cdot W_f + h_{t-1} \cdot R_f + b_f\right)$$
$$i_t = \sigma\left(x_t \cdot W_i + h_{t-1} \cdot R_i + b_i\right)$$
$$\tilde{c}_t = \tanh\left(x_t \cdot W_c + h_{t-1} \cdot R_c + b_c\right)$$
$$c_t = c_{t-1} \odot f_t + \tilde{c}_t \odot i_t$$
$$o_t = \sigma\left(x_t \cdot W_o + h_{t-1} \cdot R_o + b_o\right)$$
$$h_t = \tanh\left(c_t\right) \odot o_t$$

```
      # weights & bias for data x
      self.Wf = Dense(n_hidden)
      self.Wi = Dense(n_hidden)
      self.Wc = Dense(n_hidden)
      self.Wo = Dense(n_hidden)

      # weights for h. The biases are included in w above.
      self.Rf = Dense(n_hidden, use_bias=False) # forget gate
      self.Ri = Dense(n_hidden, use_bias=False) # input gate
      self.Rc = Dense(n_hidden, use_bias=False) # candidate
      self.Ro = Dense(n_hidden, use_bias=False) # output gate

   def lstm_cell(self, x, h, c):
      f_gate = tf.math.sigmoid(self.Wf(x) + self.Rf(h))
      i_gate = tf.math.sigmoid(self.Wi(x) + self.Ri(h))
      c_tild = tf.math.tanh(self.Wc(x) + self.Rc(h))
      o_gate = tf.math.sigmoid(self.Wo(x) + self.Ro(h))
      c_stat = c * f_gate + c_tild * i_gate
      h_stat = tf.math.tanh(c_stat) * o_gate
      return h_stat, c_stat

   def call(self, x):
      # initialize h and c
      h = tf.zeros(shape=(tf.shape(x)[0], self.nh))
      c = tf.zeros(shape=(tf.shape(x)[0], self.nh))

      # Repeat lstm_cell for the number of time steps
      for t in range(x.shape[1]):
         h, c = self.lstm_cell(x[:, t, :], h, c)
      return h
```



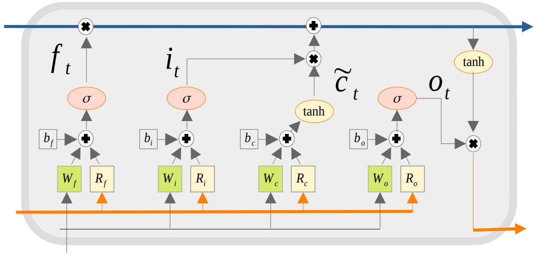


[MXDL-10-05] Deep Learning / Recurrent Neural Network (RNN) – LSTM

■ Implementation of a LSTM model using Keras custom layer: Many-to-One

```
# Build an LSTM model
n_feat = x_train.shape[-1]
n_output = y_train.shape[-1]

x_input = Input(batch_shape=(None, n_step, n_feat))
h = MyLSTM(n_hidden)(x_input)
y_output = Dense(n_output)(h)
model = Model(x_input, y_output)
model.compile(loss='mse', optimizer=Adam(learning_rate=0.001))
model.summary()   # Trainable params: 10,702

# Training
hist = model.fit(x_train, y_train, epochs=50, batch_size=50)
```

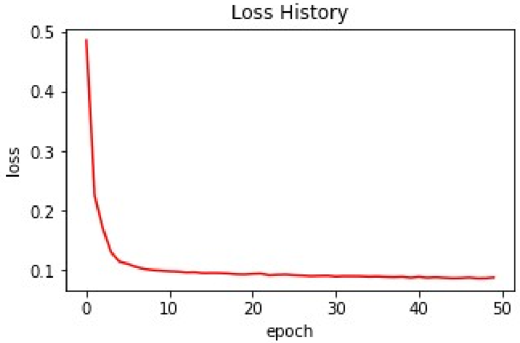
```
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)       [(None, 20, 2)]            0
 my_lstm (MyLSTM)            (None, 50)                10600
 dense_8 (Dense)             (None, 2)                 102
=================================
Total params: 10,702
Trainable params: 10,702
Non-trainable params: 0

W: (2, 50)  -> 100
R: (50, 50) -> 2500
b: (1, 50)  -> 50
```

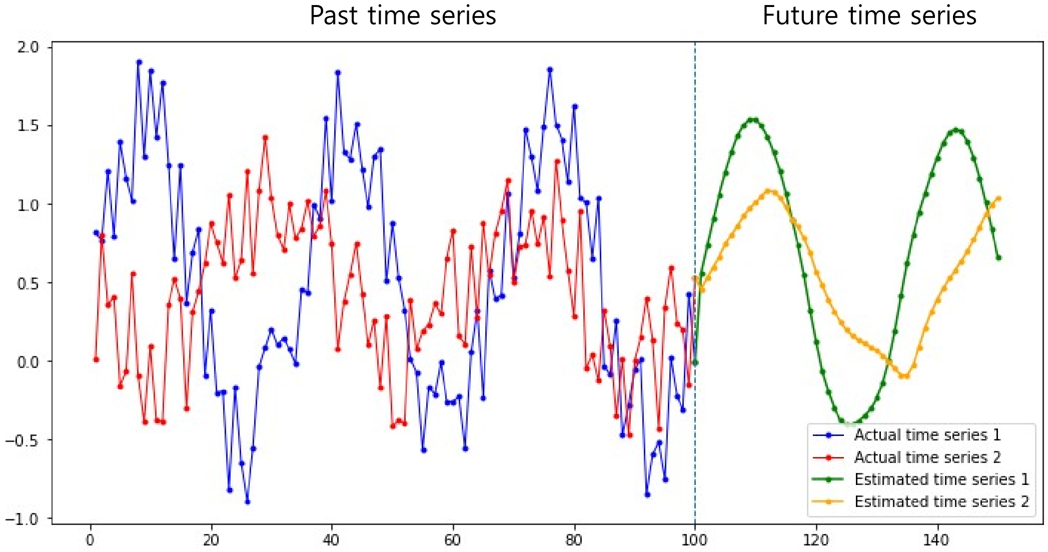


```
# Visually see the loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'], color='red')
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()

# Predict future values for the next 50 periods.
# After predicting the next value, re-enter the predicted value
# to predict the next value. Repeat this process 50 times.
n_future = 50
n_last = 100
last_data = data[-n_last:]  # The last n_last data points
for i in range(n_future):
    # Predict the next value with the last n_step data points.
    px = last_data[-n_step:, :].reshape(1, n_step, 2)

    # Predict the next value
    y_hat = model.predict(px, verbose=0)

    # Append the predicted value to the last_data array.
    # In the next iteration, the predicted value is input
    # along with the existing data points.
    last_data = np.vstack([last_data, y_hat])
```

- Implementation of a LSTM model using Keras custom layer: Many-to-One

```
p = last_data[:-n_future, :]        # past time series
f = last_data[-(n_future + 1):, :]  # future time series

# Plot past and future time series.
plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(p) + 1)
ax2 = np.arange(len(p), len(p) + len(f))
plt.plot(ax1, p[:, 0], '-o', c='blue', markersize=3,
         label='Actual time series 1', linewidth=1)
plt.plot(ax1, p[:, 1], '-o', c='red', markersize=3,
         label='Actual time series 2', linewidth=1)
plt.plot(ax2, f[:, 0], '-o', c='green', markersize=3,
         label='Estimated time series 1')
plt.plot(ax2, f[:, 1], '-o', c='orange', markersize=3,
         label='Estimated time series 2')
plt.axvline(x=ax1[-1],  linestyle='dashed', linewidth=1)
plt.legend()
plt.show()
```





[MXDL-10-05] Deep Learning / Recurrent Neural Network (RNN) – LSTM

Implementation of a LSTM model using Keras' LSTM layer: Many-to-One

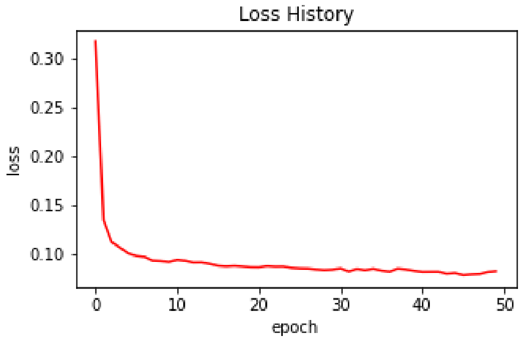
```python
# [MXDL-10-05] 5.LSTM(keras-m2o).py
from tensorflow.keras.layers import Dense, Input, LSTM
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt

# Generate training data: 2 noisy sine curves
n = 1000        # the number of data points
n_step = 20     # the number of time steps
s1 = np.sin(np.pi * 0.06 * np.arange(n)) + np.random.random(n)
s2 = 0.5*np.sin(np.pi * 0.05 * np.arange(n)) + np.random.random(n)
data = np.vstack([s1, s2]).T

m = np.arange(0, n - n_step)
x_train = np.array([data[i:(i+n_step), :] for i in m])
y_train = np.array([data[i, :] for i in (m + n_step)])

n_feat = x_train.shape[-1]
n_output = y_train.shape[-1]
n_hidden = 50

# Create a Many-to-One RNN model
x_input = Input(batch_shape=(None, n_step, n_feat))
h = LSTM(n_hidden)(x_input)
y_output = Dense(n_output)(h)

model = Model(x_input, y_output)
model.compile(loss='mse', optimizer=Adam(learning_rate=0.001))
model.summary()
```

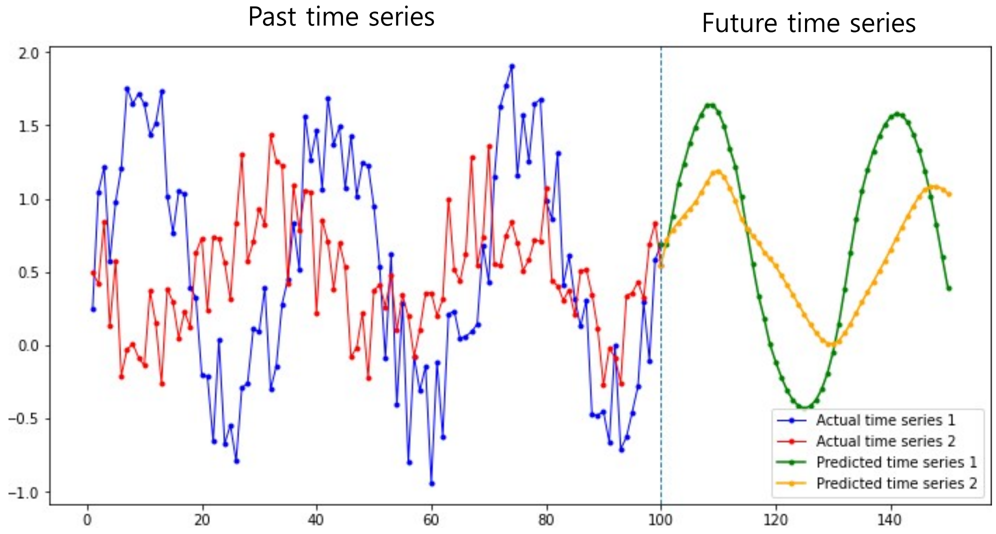
```python
# Training
hist = model.fit(x_train, y_train, epochs=50, batch_size=50)

# Loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'], color='red')
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()

# Predict future values for the next 50 periods.
# After predicting the next value, re-enter the predicted value
# to predict the next value. Repeat this process 50 times.
n_future = 50
n_last = 100
last_data = data[-n_last:]  # The last n_last data points
for i in range(n_future):
    # Predict the next value with the last n_step data points.
    px = last_data[-n_step:, :].reshape(1, n_step, 2)

    # Predict the next value
    y_hat = model.predict(px, verbose=0)

    # Append the predicted value to the last_data array.
    # In the next iteration, the predicted value is input
    # along with the existing data points.
    last_data = np.vstack([last_data, y_hat])
```

■ Implementation of a LSTM model using Keras' LSTM layer: Many-to-One

```
p = last_data[:-n_future, :]        # past time series
f = last_data[-(n_future + 1):, :]  # future time series

# Plot past and future time series.
plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(p) + 1)
ax2 = np.arange(len(p), len(p) + len(f))
plt.plot(ax1, p[:, 0], '-o', c='blue', markersize=3,
         label='Actual time series 1', linewidth=1)
plt.plot(ax1, p[:, 1], '-o', c='red', markersize=3,
         label='Actual time series 2', linewidth=1)
plt.plot(ax2, f[:, 0], '-o', c='green', markersize=3,
         label='Predicted time series 1')
plt.plot(ax2, f[:, 1], '-o', c='orange', markersize=3,
         label='Predicted time series 2')
plt.axvline(x=ax1[-1],  linestyle='dashed', linewidth=1)
plt.legend()
plt.show()
```

```
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)       [(None, 20, 2)]           0
 lstm (LSTM)                 (None, 50)                10600
 dense (Dense)               (None, 2)                 102
=================================================================
Total params: 10,702
Trainable params: 10,702
Non-trainable params: 0
```



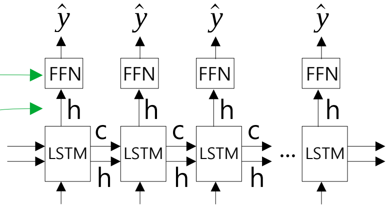


[MXDL-10-05] Deep Learning / Recurrent Neural Network (RNN) – LSTM

MX-AI

■ Implementation of a LSTM model using Keras' LSTM layer: Many-to-Many structure

```
# [MXDL-10-05] 6.LSTM(keras-m2m).py
from tensorflow.keras.layers import Dense, Input, LSTM
from tensorflow.keras.layers import TimeDistributed
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt

# Generate training data: 2 noisy sine curves
n = 1000        # the number of data points
n_step = 20     # the number of time steps
s1 = np.sin(np.pi * 0.06 * np.arange(n)) + np.random.random(n)
s2 = 0.5*np.sin(np.pi * 0.05 * np.arange(n)) + np.random.random(n)
data = np.vstack([s1, s2]).T
m = np.arange(0, n - n_step)
x_train = np.array([data[i:(i+n_step), :] for i in m])
y_train = np.array([data[(i+1):(i+1+n_step), :] for i in m])

# Create a Simple Many-to-Many RNN model
n_feat = x_train.shape[-1]
n_output = y_train.shape[-1]
n_hidden = 50


x_input = Input(batch_shape=(None, n_step, n_feat))
h = LSTM(n_hidden, return_sequences=True)(x_input)
y_output = TimeDistributed(Dense(n_output))(h)
model = Model(x_input, y_output)
model.compile(loss='mse', optimizer=Adam(learning_rate=0.001))
model.summary()
```



```
# Training
hist = model.fit(x_train, y_train, epochs=50, batch_size=50)

# Loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'], color='red')
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()

# Predict future values for the next 50 periods.
# After predicting the next value, re-enter the predicted value
# to predict the next value. Repeat this process 50 times.
n_future = 50
n_last = 100
last_data = data[-n_last:]  # The last n_last data points
for i in range(n_future):
    # Predict the next value with the last n_step data points.
    px = last_data[-n_step:, :].reshape(1, n_step, 2)

    # Predict the next value. In the prediction stage, only the
    # output of the final time step is used.
    y_hat = model.predict(px, verbose=0)[:, -1, :]

    # Append the predicted value to the last_data array.
    # In the next iteration, the predicted value is input
    # along with the existing data points.
    last_data = np.vstack([last_data, y_hat])
```

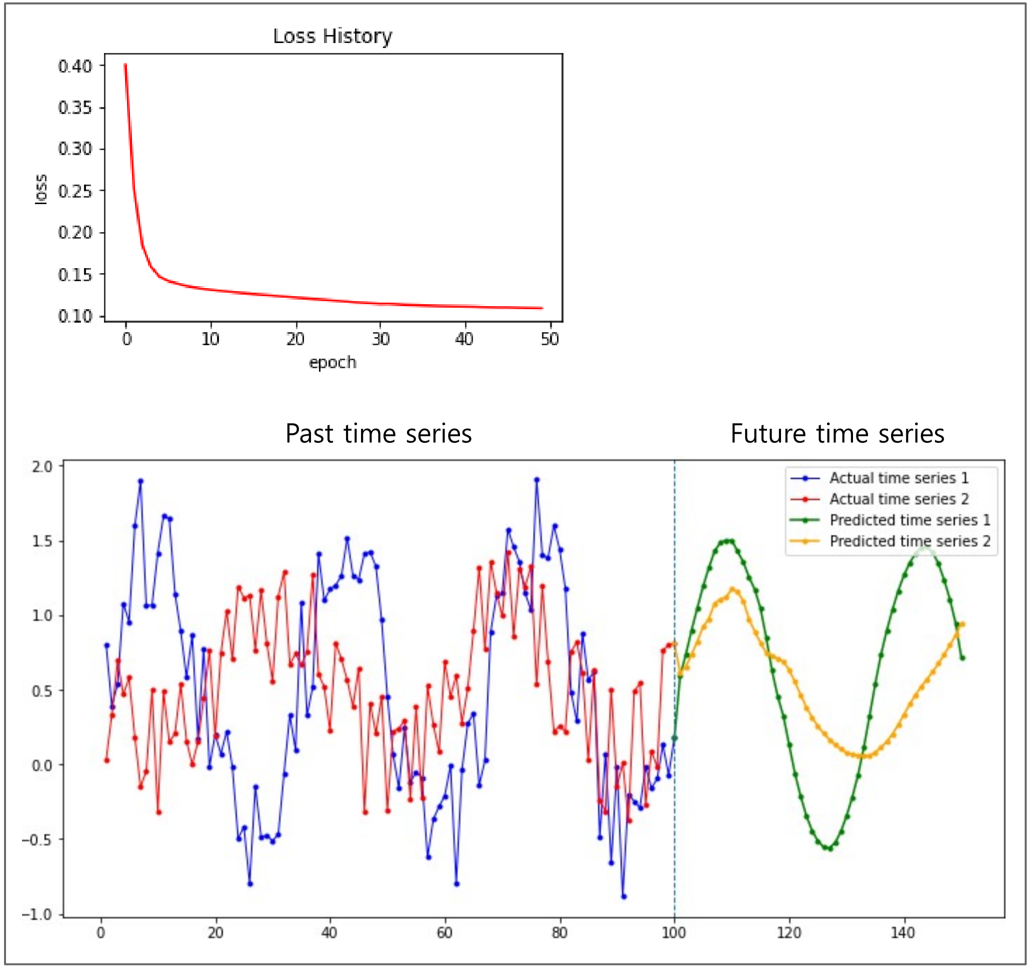
[MXDL-10-05] Deep Learning / Recurrent Neural Network (RNN) – LSTM

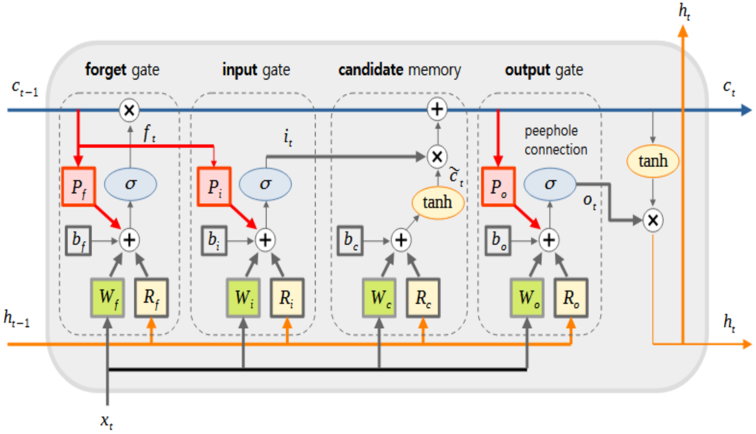
MX-AI

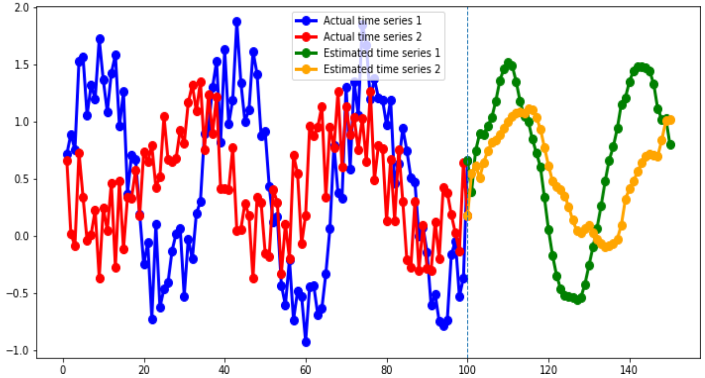
■ Implementation of a LSTM model using Keras' LSTM layer: Many-to-Many structure

```
p = last_data[:-n_future, :]         # past time series
f = last_data[-(n_future + 1):, :]  # future time series

# Plot past and future time series.
plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(p) + 1)
ax2 = np.arange(len(p), len(p) + len(f))
plt.plot(ax1, p[:, 0], '-o', c='blue', markersize=3,
         label='Actual time series 1', linewidth=1)
plt.plot(ax1, p[:, 1], '-o', c='red', markersize=3,
         label='Actual time series 2', linewidth=1)
plt.plot(ax2, f[:, 0], '-o', c='green', markersize=3,
         label='Predicted time series 1')
plt.plot(ax2, f[:, 1], '-o', c='orange', markersize=3,
         label='Predicted time series 2')
plt.axvline(x=ax1[-1],  linestyle='dashed', linewidth=1)
plt.legend()
plt.show()
```

```
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 20, 2)]           0
 lstm (LSTM)                  (None, 20, 50)           10600
 time_distributed (TimeDistr  (None, 20, 2)            102
 ibuted)
=================================================================
Total params: 10,702
Trainable params: 10,702
Non-trainable params: 0
```







10. Recurrent Neural Networks

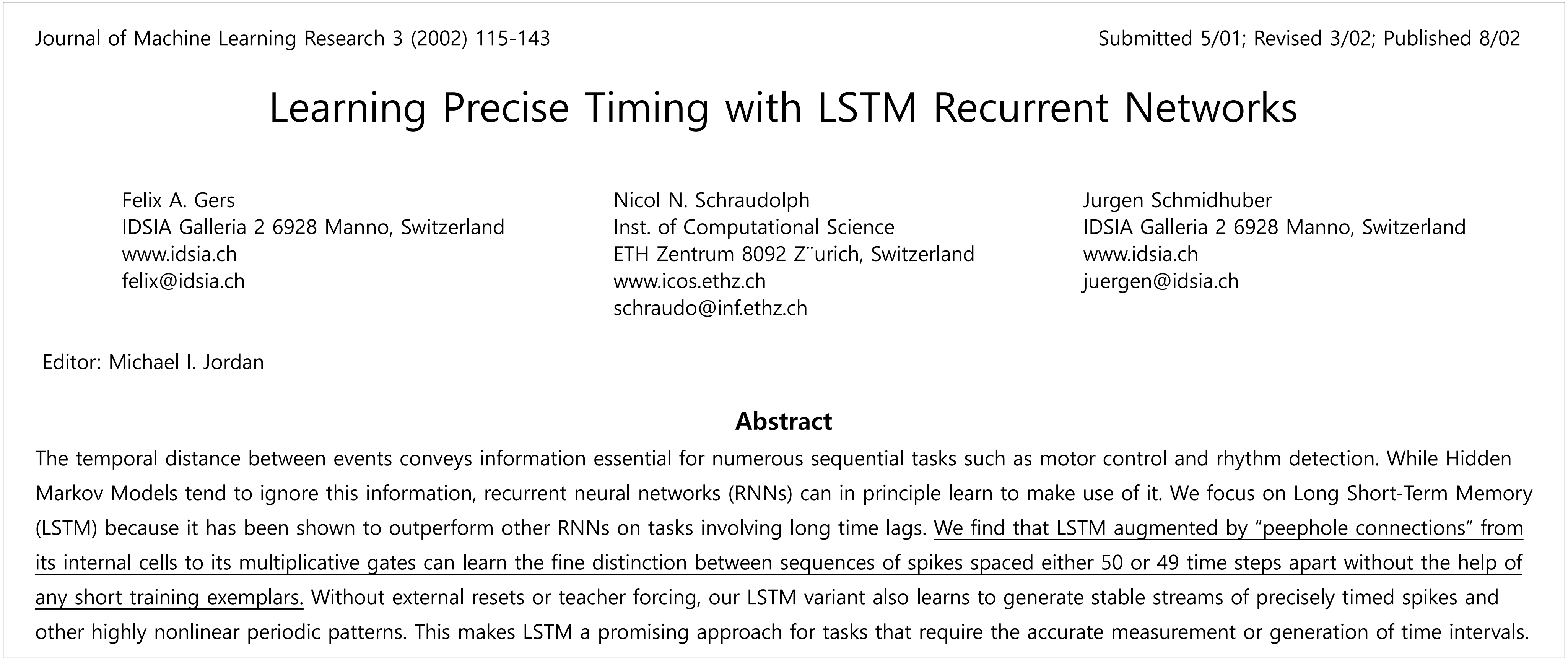
Part 6: Peephole LSTM

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

Peephole LSTM

- Felix A. Gers et al. proposed Peephole LSTM, which adds peerhole connections to traditional LSTM, in their 2000 paper "Recurrent Nets that Time and Count" and their 2002 paper "Learning Precise Timing with LSTM Recurrent Networks".



Journal of Machine Learning Research 3 (2002) 115-143 Submitted 5/01; Revised 3/02; Published 8/02

Learning Precise Timing with LSTM Recurrent Networks

Felix A. Gers
IDSIA Galleria 2 6928 Manno, Switzerland
www.idsia.ch
felix@idsia.ch

Nicol N. Schraudolph
Inst. of Computational Science
ETH Zentrum 8092 Z¨urich, Switzerland
www.icos.ethz.ch
schraudo@inf.ethz.ch

Jurgen Schmidhuber
IDSIA Galleria 2 6928 Manno, Switzerland
www.idsia.ch
juergen@idsia.ch

Editor: Michael I. Jordan

Abstract

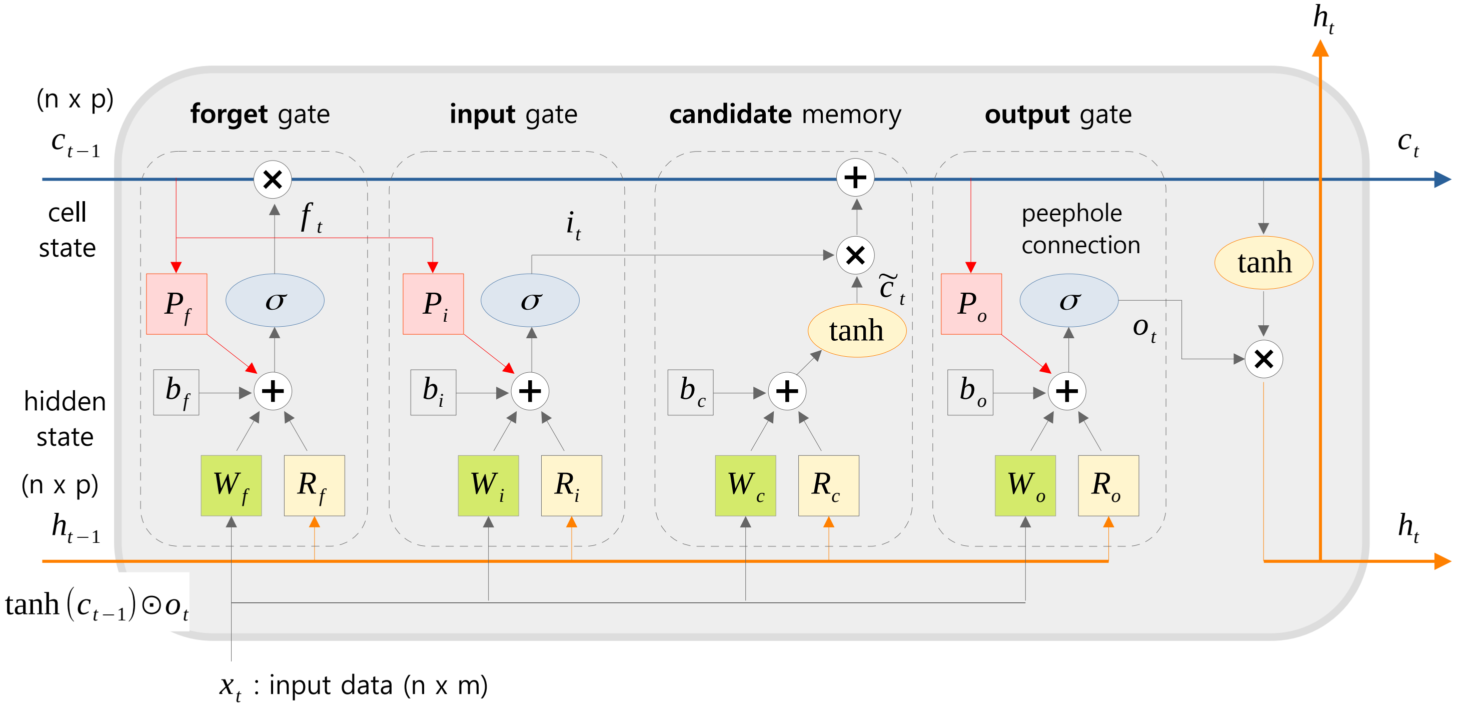
The temporal distance between events conveys information essential for numerous sequential tasks such as motor control and rhythm detection. While Hidden Markov Models tend to ignore this information, recurrent neural networks (RNNs) can in principle learn to make use of it. We focus on Long Short-Term Memory (LSTM) because it has been shown to outperform other RNNs on tasks involving long time lags. We find that LSTM augmented by "peephole connections" from its internal cells to its multiplicative gates can learn the fine distinction between sequences of spikes spaced either 50 or 49 time steps apart without the help of any short training exemplars. Without external resets or teacher forcing, our LSTM variant also learns to generate stable streams of precisely timed spikes and other highly nonlinear periodic patterns. This makes LSTM a promising approach for tasks that require the accurate measurement or generation of time intervals.

[MXDL-10-06] Deep Learning / Recurrent Neural Network (RNN) – peephole LSTM

MX-AI

peephole LSTM: Cell structure

- In traditional LSTMs, the current input data x_t and the previous hidden state h_{t-1} are input to each gate. And h_{t-1} is tanh(c_{t-1})*ot. Therefore, we can see that the information of c_{t-1} is also input to each gate. However, if the output gate is closed, that is, if o_t is 0, then the information of c_{t-1} cannot be input to each gate. Then each gate cannot receive any information from the previous state.
- Peephole LSTM solves this problem by adding peephole connections. The information of c_{t-1} is directly input to each gate through these connections. Even if the output gate is closed, information from the previous state can still be passed to each gate.



peephole connection

$$f_t = \sigma(x_t \cdot W_f + h_{t-1} \cdot R_f + c_{t-1} \cdot P_f + b_f)$$

$$i_t = \sigma(x_t \cdot W_i + h_{t-1} \cdot R_i + c_{t-1} \cdot P_i + b_i)$$

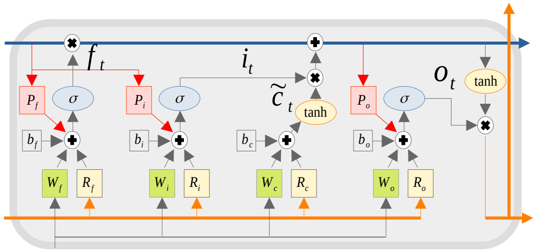
$$\tilde{c}_t = \tanh(x_t \cdot W_c + h_{t-1} \cdot R_c + b_c)$$

$$c_t = c_{t-1} \odot f_t + \tilde{c}_t \odot i_t$$

$$o_t = \sigma(x_t \cdot W_o + h_{t-1} \cdot R_o + c_t \cdot P_o + b_o)$$

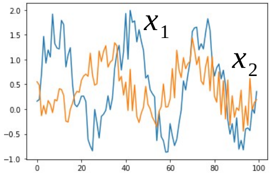
$$h_t = \tanh(c_t) \odot o_t$$

[MXDL-10-06] Deep Learning / Recurrent Neural Network (RNN) – peephole LSTM



- Implementation of a peephole LSTM model using custom layer: Many-to-One

```
# [MXDL-10-06] 7.peepholeLSTM(m2o).py
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Layer
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt

# Generate training data: 2 noisy sine curves
n = 1000        # the number of data points
n_step = 20     # the number of time steps
s1 = np.sin(np.pi * 0.06 * np.arange(n)) + np.random.random(n)
s2 = 0.5*np.sin(np.pi * 0.05 * np.arange(n)) + np.random.random(n)
data = np.vstack([s1, s2]).T  # shape = (1000, 2)

m = np.arange(0, n - n_step)
x_train = np.array([data[i:(i+n_step), :] for i in m])
y_train = np.array([data[i, :] for i in (m + n_step)])

n_hidden = 50   # the number of
                # hidden units

# peephole LSTM custom layer
class PeepholeLSTM(Layer):
   def __init__(self, n_hidden):
      super().__init__()
      self.nh = n_hidden
```



peephole connection

$$f_t = \sigma\left(x_t \cdot W_f + h_{t-1} \cdot R_f + c_{t-1} \cdot P_f + b_f\right)$$

$$i_t = \sigma\left(x_t \cdot W_i + h_{t-1} \cdot R_i + c_{t-1} \cdot P_i + b_i\right)$$

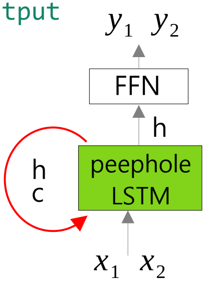
$$\tilde{c}_t = \tanh\left(x_t \cdot W_c + h_{t-1} \cdot R_c + b_c\right)$$

$$c_t = c_{t-1} \odot f_t + \tilde{c}_t \odot i_t$$

$$o_t = \sigma\left(x_t \cdot W_o + h_{t-1} \cdot R_o + c_t \cdot P_o + b_o\right)$$

$$h_t = \tanh\left(c_t\right) \odot o_t$$

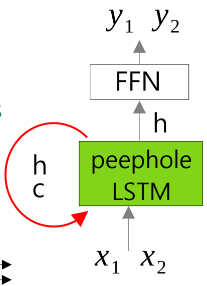
```
      # weights and bias for x
      self.Wf = Dense(n_hidden)
      self.Wi = Dense(n_hidden)
      self.Wc = Dense(n_hidden)
      self.Wo = Dense(n_hidden)

      # weights for h. The biases are included in w above.
      self.Rf = Dense(n_hidden, use_bias=False)  # forget
      self.Ri = Dense(n_hidden, use_bias=False)  # input
      self.Rc = Dense(n_hidden, use_bias=False)  # candidate state
      self.Ro = Dense(n_hidden, use_bias=False)  # output

      # peephole connections
      self.Pf = Dense(n_hidden, use_bias=False)
      self.Pi = Dense(n_hidden, use_bias=False)
      self.Po = Dense(n_hidden, use_bias=False)

   def lstm_cell(self, x, h, c):
      f_gate=tf.math.sigmoid(self.Wf(x) + self.Rf(h) + self.Pf(c))
      i_gate=tf.math.sigmoid(self.Wi(x) + self.Ri(h) + self.Pi(c))
      c_tild= tf.math.tanh(self.Wc(x) + self.Rc(h))
      c_stat = c * f_gate + c_tild * i_gate
      o_gate=tf.math.sigmoid(self.Wo(x)+self.Ro(h)+self.Po(c_stat))
      h_stat = tf.math.tanh(c_stat) * o_gate
      return h_stat, c_stat
```

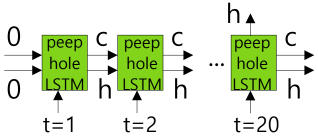


[MXDL-10-06] Deep Learning / Recurrent Neural Network (RNN) – peephole LSTM

■ Implementation of a peephole LSTM model using custom layer: Many-to-One

```
    def call(self, x):
        # initialize h, c state
        h = tf.zeros(shape=(tf.shape(x)[0], self.nh))
        c = tf.zeros(shape=(tf.shape(x)[0], self.nh))

        # Repeat lstm_cell for the number of time steps
        for t in range(x.shape[1]):
            h, c = self.lstm_cell(x[:, t, :], h, c)
        return h

# Build a peephole LSTM model
n_feat = x_train.shape[-1]
n_output = y_train.shape[-1]
x_input = Input(batch_shape=(None, n_step, n_feat))
h = PeepholeLSTM(n_hidden)(x_input)
y_output = Dense(n_output)(h)
model = Model(x_input, y_output)
model.compile(loss='mse', optimizer=Adam(learning_rate=0.001))
model.summary()   # Trainable params: 18,202
```





```
 Layer (type)                 Output Shape            Param #
=================================================================
 input_1 (InputLayer)         [(None, 20, 2)]          0
 peephole_lstm (PeepholeLSTM)  (None, 50)              18100
 dense_11 (Dense)              (None, 2)               102
=================================================================
Total params: 18,202                W: (2, 50)   -> 100    x 4
Trainable params: 18,202            R: (50, 50)  -> 2500   x 4
Non-trainable params: 0             P: (50, 50)  -> 2500   x 3
                                    b: (1, 50)   -> 50     x 4
```

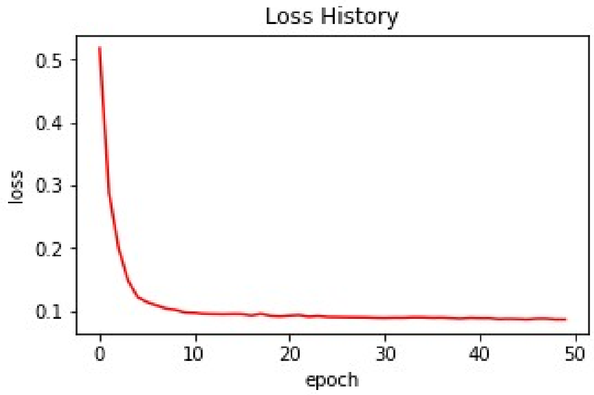
```
# Training
hist = model.fit(x_train, y_train, epochs=50, batch_size=50)

# Visually see the loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'], color='red')
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()

# Predict future values for the next 50 periods.
# After predicting the next value, re-enter the predicted value
# to predict the next value. Repeat this process 50 times.
n_future = 50
n_last = 100
last_data = data[-n_last:]  # The last n_last data points
for i in range(n_future):
    # Predict the next value with the last n_step data points.
    px = last_data[-n_step:, :].reshape(1, n_step, 2)

    # Predict the next value
    y_hat = model.predict(px, verbose=0)

    # Append the predicted value to the last_data array.
    # In the next iteration, the predicted value is input
    # along with the existing data points.
    last_data = np.vstack([last_data, y_hat])
```

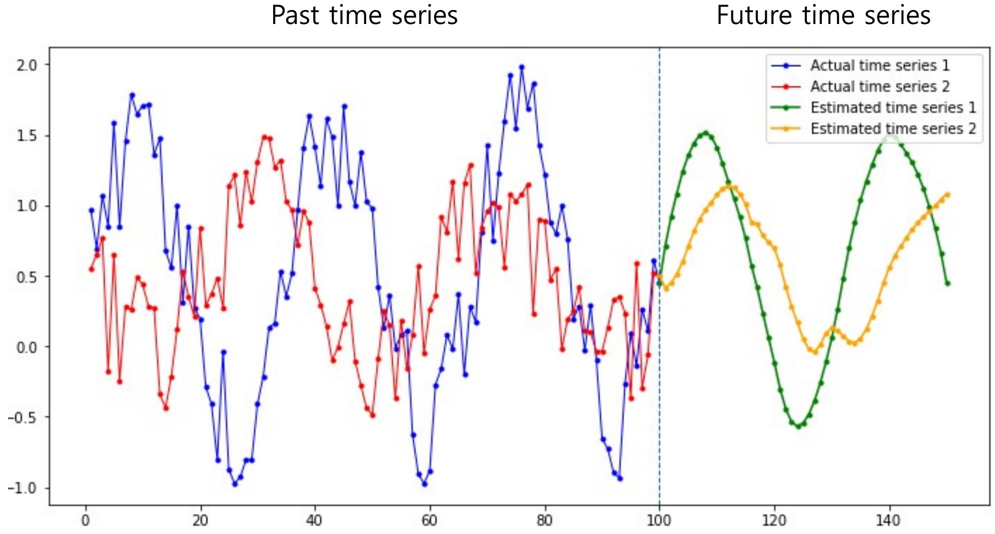
[MXDL-10-06] Deep Learning / Recurrent Neural Network (RNN) – peephole LSTM

MX-AI

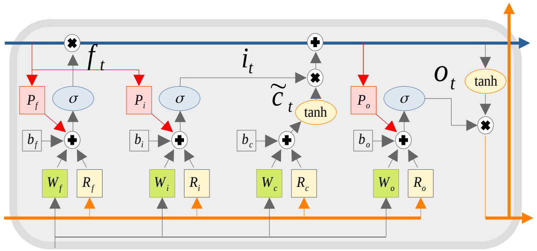
- Implementation of a peephole LSTM model using custom layer: Many-to-One

```
p = last_data[:-n_future, :]        # past time series
f = last_data[-(n_future + 1):, :]  # future time series

# Plot past and future time series.
plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(p) + 1)
ax2 = np.arange(len(p), len(p) + len(f))
plt.plot(ax1, p[:, 0], '-o', c='blue', markersize=3,
         label='Actual time series 1', linewidth=1)
plt.plot(ax1, p[:, 1], '-o', c='red', markersize=3,
         label='Actual time series 2', linewidth=1)
plt.plot(ax2, f[:, 0], '-o', c='green', markersize=3,
         label='Estimated time series 1')
plt.plot(ax2, f[:, 1], '-o', c='orange', markersize=3,
         label='Estimated time series 2')
plt.axvline(x=ax1[-1],  linestyle='dashed', linewidth=1)
plt.legend()
plt.show()
```



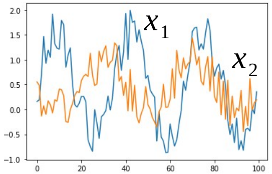


[MXDL-10-06] Deep Learning / Recurrent Neural Network (RNN) – peephole LSTM

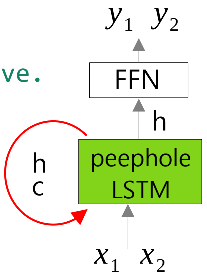


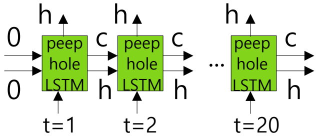
- Implementation of a peephole LSTM model using custom layer: Many-to-Many

```
# [MXDL-10-06] 7.peepholeLSTM(m2o).py
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Layer
from tensorflow.keras.layers TimeDistributed
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt

# Generate training data: 2 noisy sine curves
n = 1000        # the number of data points
n_step = 20     # the number of time steps
s1 = np.sin(np.pi * 0.06 * np.arange(n)) + np.random.random(n)
s2 = 0.5*np.sin(np.pi * 0.05 * np.arange(n)) + np.random.random(n)
data = np.vstack([s1, s2]).T  # shape = (1000, 2)

m = np.arange(0, n - n_step)
x_train = np.array([data[i:(i+n_step), :] for i in m])
y_train = np.array([data[(i+1):(i+1+n_step), :] for i in m])

n_hidden = 50   # the number of hidden units

# peephole LSTM custom layer
class PeepholeLSTM(Layer):
  def __init__(self, n_hidden, return_sequences=False):
    super().__init__()
    self.nh = n_hidden
    self.return_sequences = return_sequences
```



```
    # weights and bias for x
    self.Wf = Dense(n_hidden)
    self.Wi = Dense(n_hidden)
    self.Wc = Dense(n_hidden)
    self.Wo = Dense(n_hidden)

    # weights for h. The biases are included in w above.
    self.Rf = Dense(n_hidden, use_bias=False)
    self.Ri = Dense(n_hidden, use_bias=False)
    self.Rc = Dense(n_hidden, use_bias=False)
    self.Ro = Dense(n_hidden, use_bias=False)

    # peephole connections
    self.Pf = Dense(n_hidden, use_bias=False)
    self.Pi = Dense(n_hidden, use_bias=False)
    self.Po = Dense(n_hidden, use_bias=False)

  def lstm_cell(self, x, h, c):
    f_gate=tf.math.sigmoid(self.Wf(x) + self.Rf(h) + self.Pf(c))
    i_gate=tf.math.sigmoid(self.Wi(x) + self.Ri(h) + self.Pi(c))
    c_tild= tf.math.tanh(self.Wc(x) + self.Rc(h))
    c_stat = c * f_gate + c_tild * i_gate
    o_gate=tf.math.sigmoid(self.Wo(x)+self.Ro(h)+self.Po(c_stat))
    h_stat = tf.math.tanh(c_stat) * o_gate
    return h_stat, c_stat
```

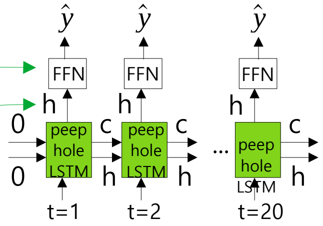




[MXDL-10-06] Deep Learning / Recurrent Neural Network (RNN) – peephole LSTM

■ Implementation of a peephole LSTM model using custom layer: Many-to-Many

```
    def call(self, x):
        # initialize h, c state
        h = [tf.zeros(shape=(tf.shape(x)[0], self.nh))]
        c = [tf.zeros(shape=(tf.shape(x)[0], self.nh))]

        # Repeat lstm_cell for the number of time steps
        for t in range(x.shape[1] - 1):
            ht, ct = self.lstm_cell(x[:, t, :], h[-1], c[-1])
            h.append(ht)
            c.append(ct)

        if self.return_sequences:
            h = tf.convert_to_tensor(h)         # (20, None, 50)
            h = tf.transpose(h, perm=[1, 0, 2]) # (None, 20, 50)
            return h      # return whole h
        else:
            return h[-1]  # return final h

# Build a peephole LSTM model
n_feat = x_train.shape[-1]
n_output = y_train.shape[-1]
x_input = Input(batch_shape=(None, n_step, n_feat))
h = PeepholeLSTM(n_hidden, return_sequences=True)(x_input)
y_output = TimeDistributed(Dense(n_output))(h)
model = Model(x_input, y_output)
model.compile(loss='mse', optimizer=Adam(learning_rate=0.001))
model.summary()   # Trainable params: 18,202
```



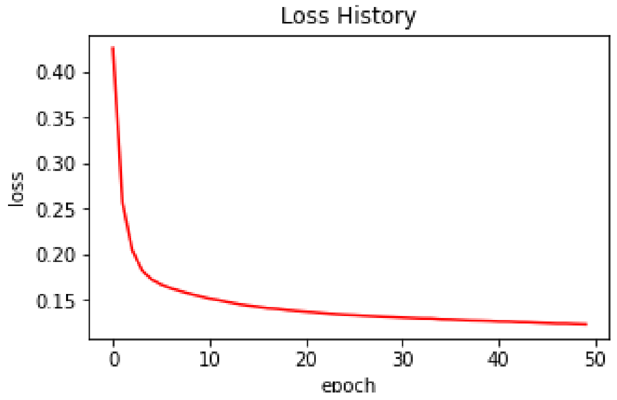
```
# Training
hist = model.fit(x_train, y_train, epochs=50, batch_size=50)

# Visually see the loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'], color='red')
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()

# Predict future values for the next 50 periods.
# After predicting the next value, re-enter the predicted value
# to predict the next value. Repeat this process 50 times.
n_future = 50
n_last = 100
last_data = data[-n_last:]  # The last n_last data points
for i in range(n_future):
    # Predict the next value with the last n_step data points.
    px = last_data[-n_step:, :].reshape(1, n_step, 2)

    # Predict the next value
    y_hat = model.predict(px, verbose=0)[:, -1, :]

    # Append the predicted value to the last_data array.
    # In the next iteration, the predicted value is input
    # along with the existing data points.
    last_data = np.vstack([last_data, y_hat])
```

[MXDL-10-06] Deep Learning / Recurrent Neural Network (RNN) – peephole LSTM

MX-AI

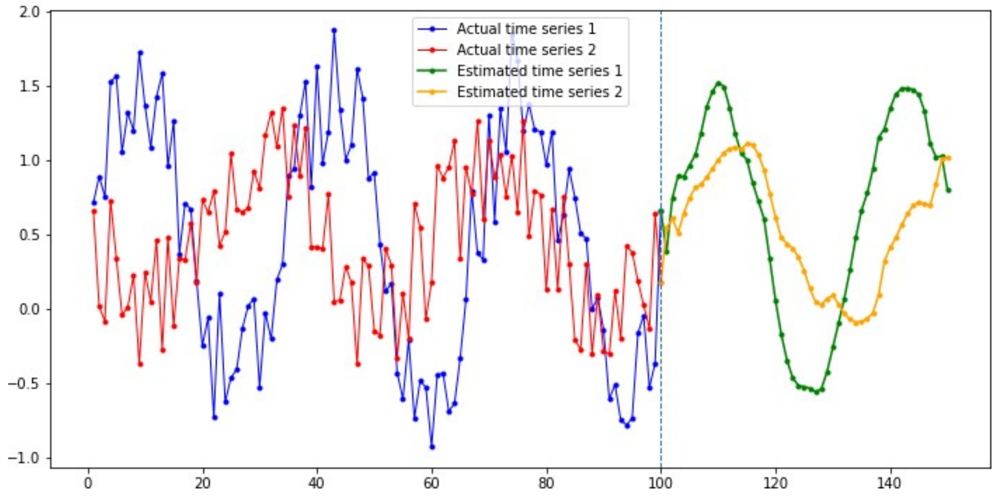
- Implementation of a peephole LSTM model using custom layer: Many-to-Many

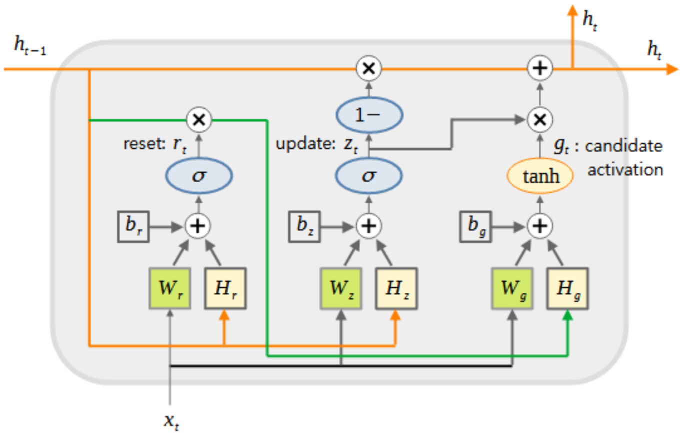
```
p = last_data[:-n_future, :]        # past time series
f = last_data[-(n_future + 1):, :]  # future time series

# Plot past and future time series.
plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(p) + 1)
ax2 = np.arange(len(p), len(p) + len(f))
plt.plot(ax1, p[:, 0], '-o', c='blue', markersize=3,
         label='Actual time series 1', linewidth=1)
plt.plot(ax1, p[:, 1], '-o', c='red', markersize=3,
         label='Actual time series 2', linewidth=1)
plt.plot(ax2, f[:, 0], '-o', c='green', markersize=3,
         label='Estimated time series 1')
plt.plot(ax2, f[:, 1], '-o', c='orange', markersize=3,
         label='Estimated time series 2')
plt.axvline(x=ax1[-1],  linestyle='dashed', linewidth=1)
plt.legend()
plt.show()
```

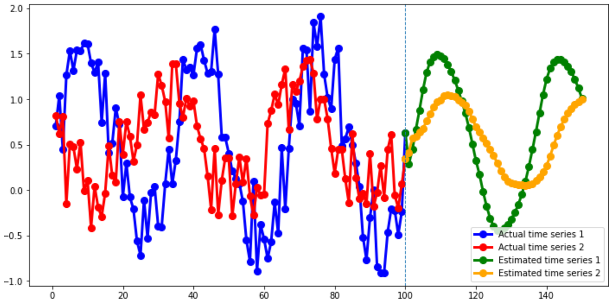


Past time series

Future time series







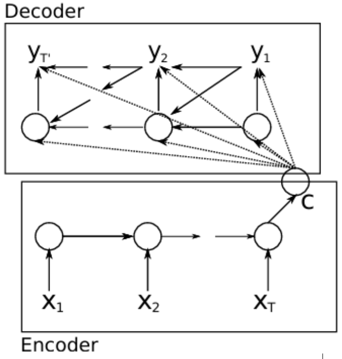
10. Recurrent Neural Networks

Part 7: Gated Recurrent Unit (GRU)

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

[MXDL-10-07] Deep Learning / Recurrent Neural Network (RNN) – GRU



Gated Recurrent Unit (GRU)

- In 2014, Kyunghyun Cho et al. proposed gated recurrent units (GRUs) in their paper "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation." Although the term GRU is not used in this paper, the architecture of GRU is presented in Section 2.3.

Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation

Kyunghyun Cho
Bart van Merrienboer Caglar Gulcehre
Universite de Montreal
firstname.lastname@umontreal.ca

Dzmitry Bahdanau
Jacobs University, Germany
d.bahdanau@jacobs-university.de

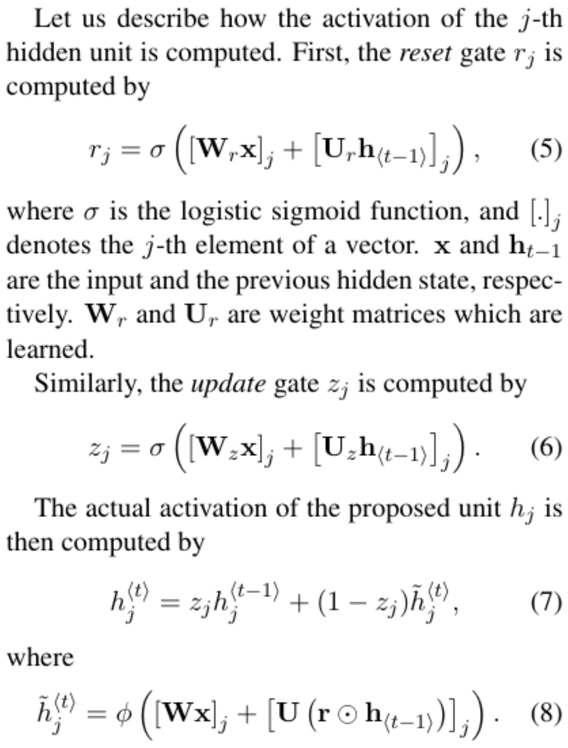
Fethi Bougares Holger Schwenk
Universite du Maine, France
firstname.lastname@lium.univ-lemans.fr

Yoshua Bengio
Universit´e de Montr´eal, CIFAR Senior Fellow
find.me@on.the.web

Abstract

In this paper, we propose a novel neural network model called RNN Encoder Decoder that consists of two recurrent neural networks (RNN). One RNN encodes a sequence of symbols into a fixed length vector representation, and the other decodes the representation into another sequence of symbols. The encoder and decoder of the proposed model are jointly trained to maximize the conditional probability of a target sequence given a source sequence. The performance of a statistical machine translation system is empirically found to improve by using the conditional probabilities of phrase pairs computed by the RNN Encoder–Decoder as an additional feature in the existing log-linear model. Qualitatively, we show that the proposed model learns a semantically and syntactically meaningful representation of linguistic phrases.

2.3 Hidden Unit that Adaptively Remembers and Forgets



Let us describe how the activation of the j-th hidden unit is computed. First, the *reset* gate r_j is computed by

$$r_j = \sigma\left(\left[\mathbf{W}_r\mathbf{x}\right]_j + \left[\mathbf{U}_r\mathbf{h}_{\langle t-1\rangle}\right]_j\right), \qquad (5)$$

where σ is the logistic sigmoid function, and $[.]_j$ denotes the j-th element of a vector. $\mathbf{x}$ and $\mathbf{h}_{t-1}$ are the input and the previous hidden state, respectively. $\mathbf{W}_r$ and $\mathbf{U}_r$ are weight matrices which are learned.

Similarly, the *update* gate z_j is computed by

$$z_j = \sigma\left(\left[\mathbf{W}_z\mathbf{x}\right]_j + \left[\mathbf{U}_z\mathbf{h}_{\langle t-1\rangle}\right]_j\right). \qquad (6)$$

The actual activation of the proposed unit h_j is then computed by

$$h_j^{\langle t\rangle} = z_j h_j^{\langle t-1\rangle} + (1 - z_j)\tilde{h}_j^{\langle t\rangle}, \qquad (7)$$

where

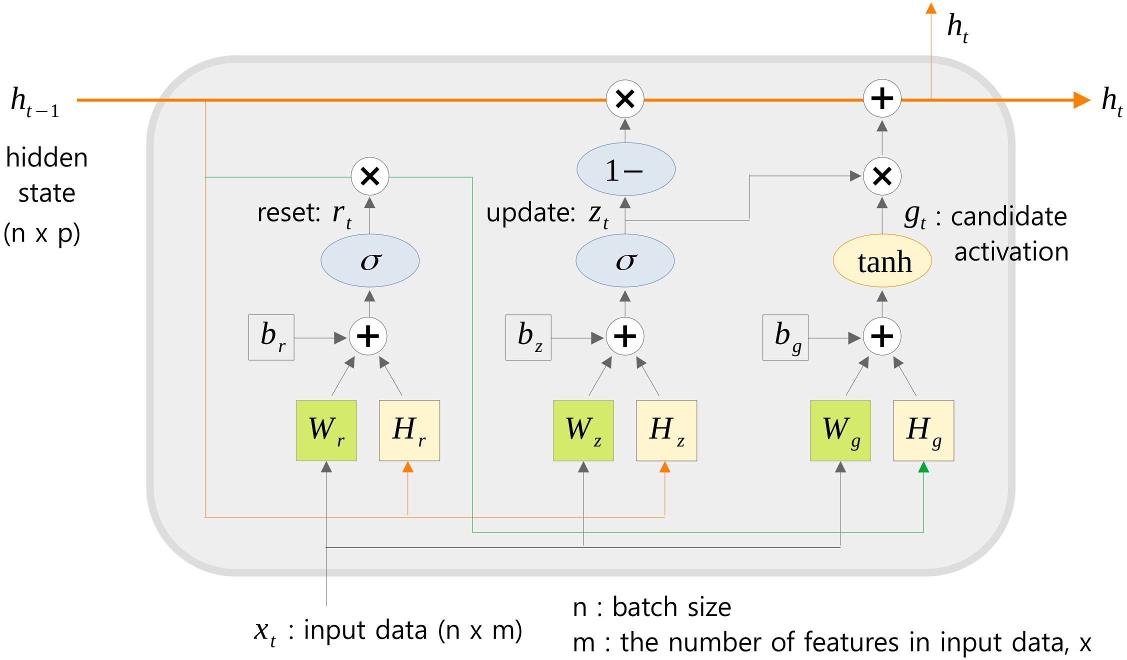
$$\tilde{h}_j^{\langle t\rangle} = \phi\left(\left[\mathbf{W}\mathbf{x}\right]_j + \left[\mathbf{U}\left(\mathbf{r}\odot\mathbf{h}_{\langle t-1\rangle}\right)\right]_j\right). \qquad (8)$$

[MXDL-10-07] Deep Learning / Recurrent Neural Network (RNN) – GRU

MX-AI

Gated Recurrent Unit (GRU)

- Gated Recurrent Unit (GRU) is a variant of LSTM that improves the learning speed of the networks by simplifying the structure of the LSTM into two gates: a reset gate and an update gate. Unlike LSTM, GRU cells do not have a separate cell state (c), but only a hidden state (h). The gradients can flow uninterrupted through the hidden state, similar to shortcut connection in a highway network.
- The reset gate r_t is similar to the forget gate in LSTM and controls how much of the previous hidden state information is used to compute candidate activations.
- The candidate activation g_t is new information computed with the current input x and the previous hidden state.
- The update gate zt determines how much of the candidate activations to apply when computing the next hidden state.



n : batch size
m : the number of features in input data, x
p : hidden units

$$r_t = \sigma(x_t \cdot W_r + h_{t-1} \cdot H_r + b_r)$$

$$z_t = \sigma(x_t \cdot W_z + h_{t-1} \cdot H_z + b_z)$$

$$g_t = \tanh(x_t \cdot W_g + r_t \odot (h_{t-1} \cdot H_g) + b_g) \quad \text{---- (v1)}$$

$$g_t = \tanh(x_t \cdot W_g + (r_t \odot h_{t-1}) \cdot H_g + b_g) \quad \text{---- (v3)}$$

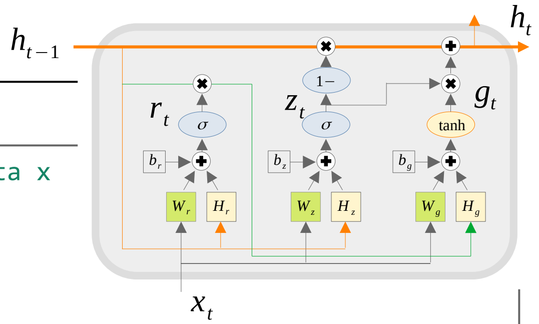
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot g_t$$

* The current state is a weighted average of h_{t-1} and g_t with z_t.

v1 – https://arxiv.org/pdf/1406.1078v1.pdf (equation 8)

v3 – https://arxiv.org/pdf/1406.1078v3.pdf (equation 8)

[MXDL-10-07] Deep Learning / Recurrent Neural Network (RNN) – GRU



■ Implementation of a GRU model using custom layer: Many-to-Many

```
# [MXDL-10-07] 9.GRU(m2m).py (Custom GRU layer)
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Layer,
TimeDistributed
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt

# Generate training data: 2 noisy sine curves
n = 1000        # the number of data points
n_step = 20     # the number of time steps
s1 = np.sin(np.pi * 0.06 * np.arange(n)) + np.random.random(n)
s2 = 0.5*np.sin(np.pi * 0.05 * np.arange(n)) + np.random.random(n)
data = np.vstack([s1, s2]).T
m = np.arange(0, n - n_step)
x_train = np.array([data[i:(i+n_step), :] for i in m])
y_train = np.array([data[(i+1):(i+1+n_step), :] for i in m])

n_hidden = 50  # the number of
               # hidden units

# GRU custom layer
class MyGRU(Layer):
    def __init__(self, n_hidden, return_sequences=False):
        super().__init__()
        self.nh = n_hidden
        self.return_sequences = return_sequences
```

$$r_t = \sigma\left(x_t \cdot W_r + h_{t-1} \cdot H_r + b_r\right)$$

$$z_t = \sigma\left(x_t \cdot W_z + h_{t-1} \cdot H_z + b_z\right)$$

$$g_t = \tanh\left(x_t \cdot W_g + \left(r_t \odot h_{t-1}\right) \cdot H_g + b_g\right)$$

$$h_t = \left(1 - z_t\right) \odot h_{t-1} + z_t \odot g_t$$

```
        # weights and bias for data x
        self.Wr = Dense(n_hidden)
        self.Wz = Dense(n_hidden)
        self.Wg = Dense(n_hidden)

        # weights for h. The biases are included in w above.
        self.Hr = Dense(n_hidden, use_bias=False)  # reset
        self.Hz = Dense(n_hidden, use_bias=False)  # update
        self.Hg = Dense(n_hidden, use_bias=False)  # candidate

    def gru_cell(self, x, h):
        rt = tf.math.sigmoid(self.Wr(x) + self.Hr(h))
        zt = tf.math.sigmoid(self.Wz(x) + self.Hz(h))
        gt = tf.math.tanh(self.Wg(x) + self.Hg(rt * h))
        h_stat = (1. - zt) * h + zt * gt
        return h_stat

    def call(self, x):
        h = [tf.zeros(shape=(tf.shape(x)[0], self.nh))] # initialize
        for t in range(x.shape[1] – 1): # Repeat gru_cell
            ht = self.gru_cell(x[:, t, :], h[-1])
            h.append(ht)

        if self.return_sequences:
            h = tf.convert_to_tensor(h)         # (20, None, 50)
            h = tf.transpose(h, perm=[1, 0, 2]) # (None, 20, 50)
            return h        # return whole h
        else:
            return h[-1]    # return final h
```

[MXDL-10-07] Deep Learning / Recurrent Neural Network (RNN) – GRU

■ Implementation of a GRU model using custom layer: Many-to-Many

```
# Build a GRU model
n_feat = x_train.shape[-1]
n_output = y_train.shape[-1]

x_input = Input(batch_shape=(None, n_step, n_feat))
h = MyGRU(n_hidden, return_sequences=True)(x_input)
y_output = TimeDistributed(Dense(n_output))(h)
model = Model(x_input, y_output)
model.compile(loss='mse', optimizer=Adam(learning_rate=0.001))
model.summary()
```

```
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 20, 2)]           0
 my_gru_1 (MyGRU)             (None, 20, 50)           7950
 time_distributed_4           (None, 20, 2)            102
(TimeDistributed)
=================================================================
Total params: 8,052              W: (2, 50)  -> 100   x 3
Trainable params: 8,052          H: (50, 50) -> 2500  x 3
Non-trainable params: 0          b: (1, 50)  -> 50    x 3
```

```
# Training
hist = model.fit(x_train, y_train, epochs=50, batch_size=50)
```

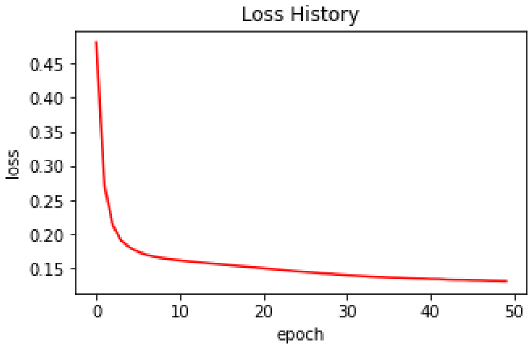
```
# Visually see the loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'], color='red')
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()

# Predict future values for the next 50 periods.
# After predicting the next value, re-enter the predicted value
# to predict the next value. Repeat this process 50 times.
n_future = 50
n_last = 100
last_data = data[-n_last:]  # The last n_last data points
for i in range(n_future):
    # Predict the next value with the last n_step data points.
    px = last_data[-n_step:, :].reshape(1, n_step, 2)

    # Predict the next value
    y_hat = model.predict(px, verbose=0)[:, -1, :]

    # Append the predicted value to the last_data array.
    # In the next iteration, the predicted value is input
    # along with the existing data points.
    last_data = np.vstack([last_data, y_hat])

p = last_data[:-n_future, :]          # past time series
f = last_data[-(n_future + 1):, :]  # future time series
```

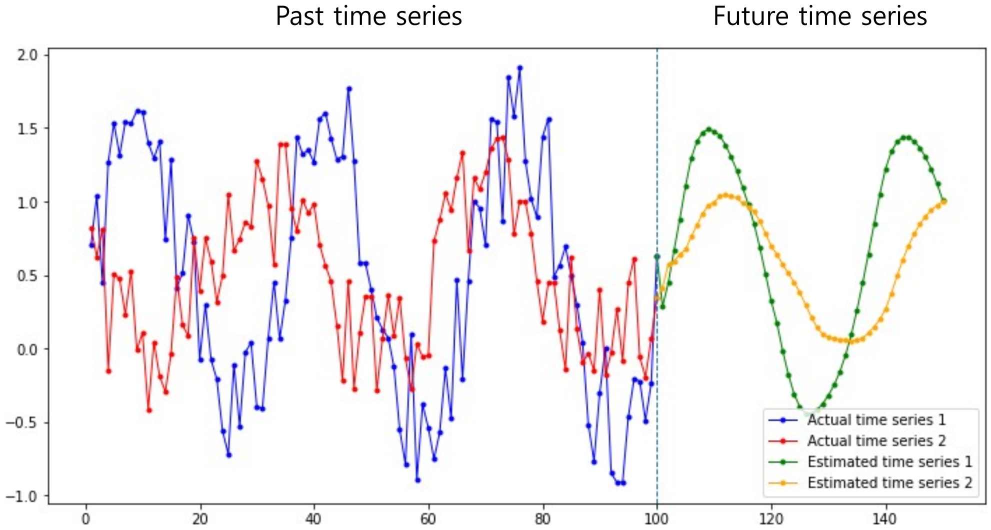
[MXDL-10-07] Deep Learning / Recurrent Neural Network (RNN) – GRU

MX-AI

■ Implementation of a GRU model using custom layer: Many-to-Many

```python
# Plot past and future time series.
plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(p) + 1)
ax2 = np.arange(len(p), len(p) + len(f))
plt.plot(ax1, p[:, 0], '-o', c='blue', markersize=8,
         label='Actual time series 1', linewidth=3)
plt.plot(ax1, p[:, 1], '-o', c='red', markersize=8,
         label='Actual time series 2', linewidth=3)
plt.plot(ax2, f[:, 0], '-o', c='green', markersize=8,
         label='Estimated time series 1', linewidth=3)
plt.plot(ax2, f[:, 1], '-o', c='orange', markersize=8,
         label='Estimated time series 2', linewidth=3)
plt.axvline(x=ax1[-1],  linestyle='dashed', linewidth=1)
plt.legend()
plt.show()
```





[MXDL-10-07] Deep Learning / Recurrent Neural Network (RNN) – GRU

■ Implementation of a GRU model using Keras' GRU layer: Many-to-Many

```python
# [MXDL-10-07] 10.GRU(keras_m2m).py (Many-to-Many)
from tensorflow.keras.layers import Dense, Input, GRU
from tensorflow.keras.layers import TimeDistributed
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt

# Generate training data: 2 noisy sine curves
n = 1000        # the number of data points
n_step = 20     # the number of time steps
s1 = np.sin(np.pi * 0.06 * np.arange(n)) + np.random.random(n)
s2 = 0.5*np.sin(np.pi * 0.05 * np.arange(n)) + np.random.random(n)
data = np.vstack([s1, s2]).T
m = np.arange(0, n - n_step)
x_train = np.array([data[i:(i+n_step), :] for i in m])
y_train = np.array([data[(i+1):(i+1+n_step), :] for i in m])

n_hidden = 50   # the number of hidden units

# Build a GRU model
n_feat = x_train.shape[-1]
n_output = y_train.shape[-1]

x_input = Input(batch_shape=(None, n_step, n_feat))
```

```python
# https://www.tensorflow.org/api_docs/python/tf/keras/layers/GRU
# Keras' GRU by default adds bias to W and H respectively.
# rt = σ(x.Wr + b1  + h_t-1. Hr + b2)
# zt = σ(x.Wz + b3  + h_t-1. Hz + b4)
# gt = tanh(x.Wg + b5 + (rt*h_t-1.Hg) + b6)
# The reason is to meet the requirements of cuDNN.
# GRU(reset_after=False): 1 bias is used.  It's slow to train.
# GRU(reset_after=True):  2 biases are used.  it's fast to train
h = GRU(n_hidden, return_sequences=True, reset_after=True)(x_input)
y_output = TimeDistributed(Dense(n_output))(h)

model = Model(x_input, y_output)
model.compile(loss='mse',
              optimizer=Adam(learning_rate=0.001))
model.summary()

# Training
hist = model.fit(x_train, y_train, epochs=50, batch_size=50)

# Visually see the loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'], color='red')
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()
```

MX-AI

■ Implementation of a GRU model using Keras' GRU layer: Many-to-Many

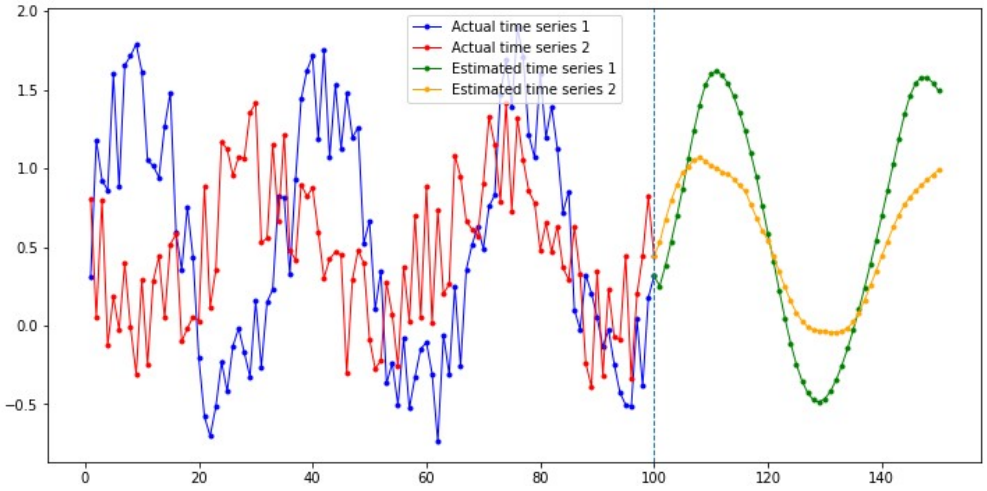
```
 Layer (type)                Output Shape              Param #
=================================================================
 input_4 (InputLayer)        [(None, 20, 2)]           0
 gru_3 (GRU)                  (None, 20, 50)           8100
 time_distributed_1           (None, 20, 2)            102
(TimeDistributed)
=================================================================
Total params: 8,202
Trainable params: 8,202
Non-trainable params: 0
```

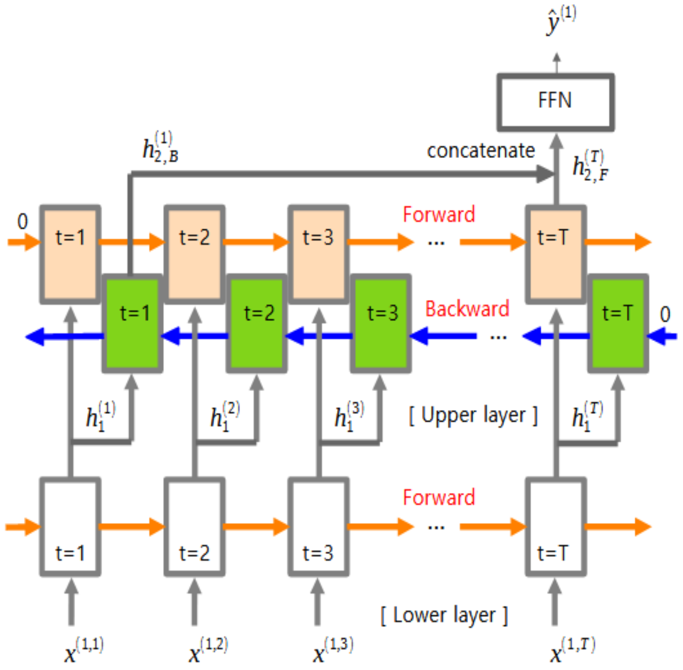
```python
# Predict future values for the next 50 periods.
# After predicting the next value, re-enter the predicted value
# to predict the next value. Repeat this process 50 times.
n_future = 50
n_last = 100
last_data = data[-n_last:]  # The last n_last data points
for i in range(n_future):
    # Predict the next value with the last n_step data points.
    px = last_data[-n_step:, :].reshape(1, n_step, 2)

    # Predict the next value
    y_hat = model.predict(px, verbose=0)[:, -1, :]

    # Append the predicted value to the last_data array.
    # In the next iteration, the predicted value is input
    # along with the existing data points.
    last_data = np.vstack([last_data, y_hat])

p = last_data[:-n_future, :]        # past time series
f = last_data[-(n_future + 1):, :]  # future time series
```

```python
# Plot past and future time series.
plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(p) + 1)
ax2 = np.arange(len(p), len(p) + len(f))
plt.plot(ax1, p[:, 0], '-o', c='blue', markersize=3,
         label='Actual time series 1', linewidth=1)
plt.plot(ax1, p[:, 1], '-o', c='red', markersize=3,
         label='Actual time series 2', linewidth=1)
plt.plot(ax2, f[:, 0], '-o', c='green', markersize=3,
         label='Estimated time series 1', linewidth=1)
plt.plot(ax2, f[:, 1], '-o', c='orange', markersize=3,
         label='Estimated time series 2', linewidth=1)
plt.axvline(x=ax1[-1],  linestyle='dashed', linewidth=1)
plt.legend()
plt.show()
```

Past time series

Future time series





10. Recurrent Neural Networks

Part 8: Multi-layer & Bi-directional

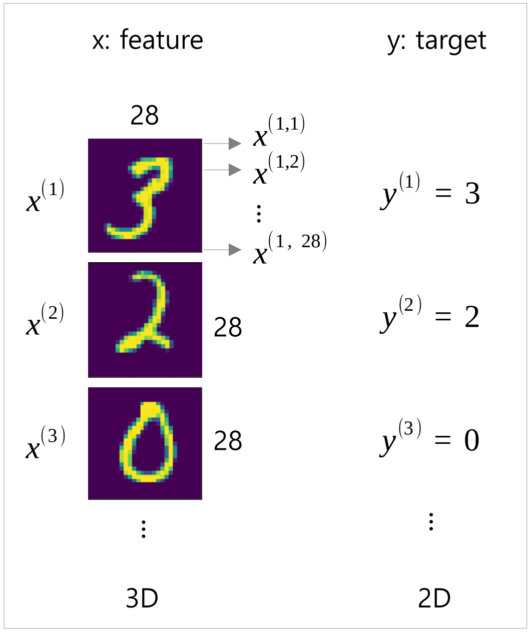
This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

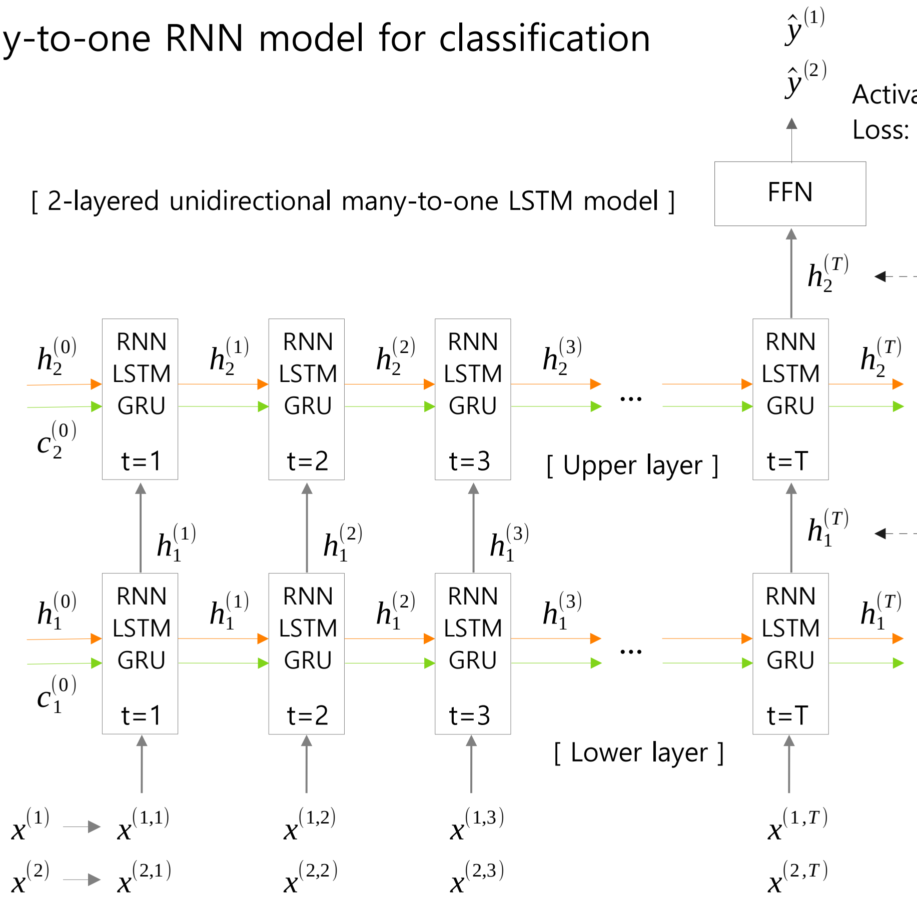
[MXDL-10-08] Deep Learning / Recurrent Neural Network (RNN) – Multi-layer RNN

- 2-layered unidirectional many-to-one RNN model for classification

[MNIST dataset]



[2-layered unidirectional many-to-one LSTM model]



Activation function: softmax
Loss: cross entropy

Since it is a many-to-one model, only the output of the last time step is used at the top layer.
h_2.shape = (None, p)

Keras:
h2 = LSTM(p, **return_sequences=False**)(h1)

The outputs of all time steps in the lower layer are passed to the upper layer.
h_1.shape = (None, T, p)
p: # hidden units

Keras:
h1 = LSTM(p, **return_sequences=True**)(x)

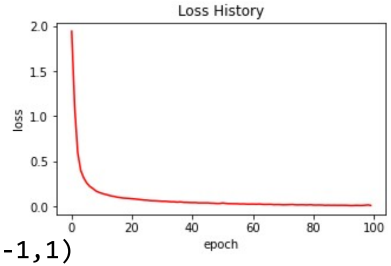
- Recurrent neural networks can be used for both regression and classification. Regression or classification is performed in FFN.
- In addition to FFN, you can use various networks such as DQN and Actor-Critic.

[MXDL-10-08] Deep Learning / Recurrent Neural Network (RNN) – Multi-layer RNN

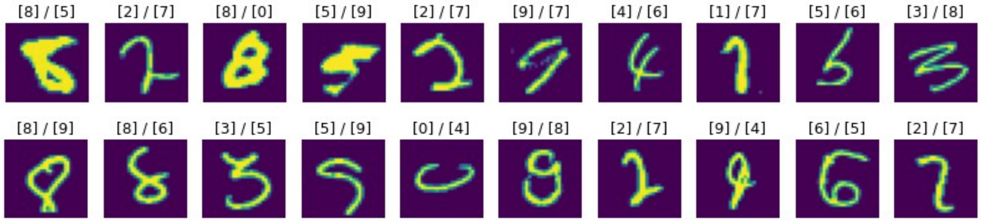
■ 2-layered unidirectional many-to-one RNN model for classification

```
# [MXDL-10-08] 11.m2o_2layer.py
from tensorflow.keras.layers import Dense, Input, LSTM
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_openml
import numpy as np
import matplotlib.pyplot as plt

# Read a MNIST dataset
mnist = fetch_openml('mnist_784', parser='auto')
x = np.array(mnist['data']).reshape(-1, 28, 28) / 255
y = np.array(mnist['target']).astype('int').reshape(-1,1)

# Split the dataset into training data and test data
x_train, x_test, y_train, y_test = train_test_split(x, y)
n_step = x_train.shape[1]

# Build a 2-layered many-to-one LSTM model
n_feat = x_train.shape[-1]
n_output = len(set(y.reshape(-1,)))
n_hidden = 50

x_input = Input(batch_shape=(None, n_step, n_feat))
h1 = LSTM(n_hidden, return_sequences=True)(x_input)
h2 = LSTM(n_hidden, return_sequences=False)(h1)
y_output = Dense(n_output, activation='softmax')(h2)
model = Model(x_input, y_output)
model.compile(loss='sparse_categorical_crossentropy',
              optimizer=Adam(learning_rate=0.001))

# Training
hist = model.fit(x_train, y_train, epochs=100, batch_size=1000)
```

```
# Visually see the loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'], color='red')
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()

y_prob = model.predict(x_test)
y_pred = np.argmax(y_prob, axis=1).reshape(-1,1)
acc = (y_test == y_pred).mean()
print('Accuracy of test data ={:.4f}'.format(acc))

# Let's check out some misclassified images.
n_sample = 10
miss_cls = np.where(y_test != y_pred)[0]
miss_sam = np.random.choice(miss_cls, n_sample)
fig, ax = plt.subplots(1, n_sample, figsize=(14,4))
for i, miss in enumerate(miss_sam):
    x = x_test[miss] * 255
    ax[i].imshow(x.reshape(28, 28))
    ax[i].axis('off')
    ax[i].set_title(str(y_test[miss]) + ' / ' + str(y_pred[miss]))

Accuracy of test data = 0.9815
```

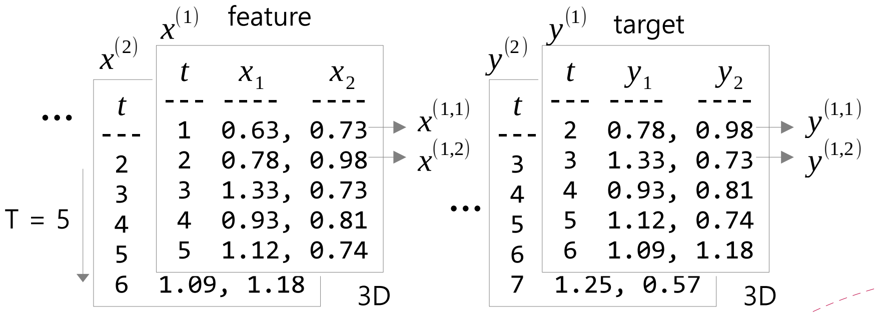


[Examples of misclassified image]



[MXDL-10-08] Deep Learning / Recurrent Neural Network (RNN) – Multi-layer RNN

- 2-layered unidirectional many-to-many RNN model for time series prediction



t	x_1	x_2
1	0.63,	0.73
2	0.78,	0.98
3	1.33,	0.73
4	0.93,	0.81
5	1.12,	0.74

t		
2		
3		
4		
5		
6	1.09,	1.18

t	y_1	y_2
2	0.78,	0.98
3	1.33,	0.73
4	0.93,	0.81
5	1.12,	0.74
6	1.09,	1.18

t		
3		
4		
5		
6		
7	1.25,	0.57

Since it is a many-to-many model, the outputs of all time steps are used at the top layer.

h₂.shape = (None, T, p)

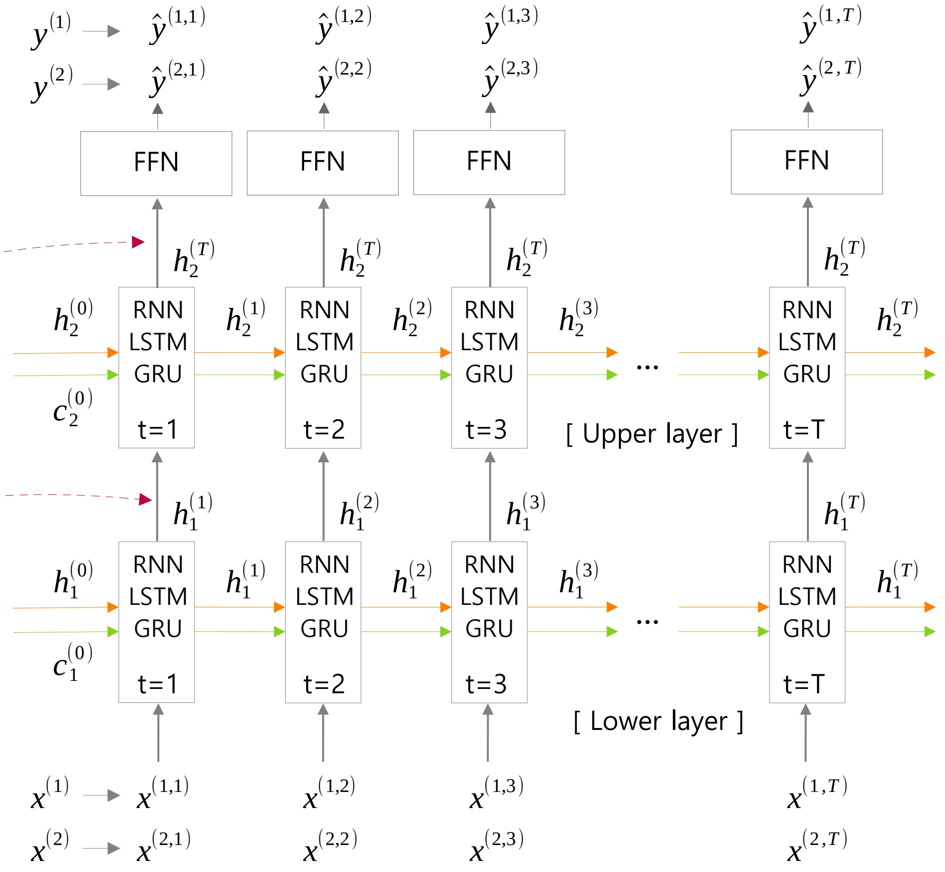
Keras:

h2 = LSTM(p, **return_sequences = True**)(h1)

y = **TimeDistributed**(Dense(2))(h2)

The outputs of all time steps in the lower layer are passed to the upper layer.
h₁.shape = (None, T, p)
p: # hidden units

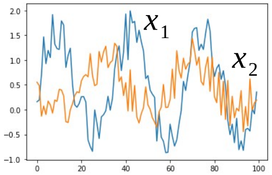
Keras:
h1 = LSTM(p, **return_sequences=True**)(x)



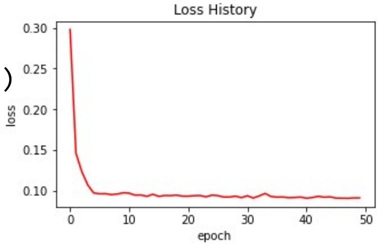
[MXDL-10-08] Deep Learning / Recurrent Neural Network (RNN) – Multi-layer RNN

- 2-layered unidirectional many-to-many RNN model for time series prediction

```
# [MXDL-10-08] 12.m2m_2layer.py
from tensorflow.keras.layers import Dense, Input, LSTM
from tensorflow.keras.layers import TimeDistributed
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt

# Generate training data: 2 noisy sine curves
n = 1000        # the number of data points
n_step = 20     # the number of time steps
s1 = np.sin(np.pi * 0.06 * np.arange(n)) + np.random.random(n)
s2 = 0.5*np.sin(np.pi * 0.05 * np.arange(n)) + np.random.random(n)
data = np.vstack([s1, s2]).T
m = np.arange(0, n - n_step)
x_train = np.array([data[i:(i+n_step), :] for i in m])
y_train = np.array([data[(i+1):(i+1+n_step), :] for i in m])

n_hidden = 50   # the number of hidden units

# Build a 2-layered many-to-many LSTM model
n_feat = x_train.shape[-1]
n_output = y_train.shape[-1]
```



```
x_input = Input(batch_shape=(None, n_step, n_feat))
h1 = LSTM(n_hidden, return_sequences=True)(x_input)
h2 = LSTM(n_hidden, return_sequences=True)(h1)
y_output = TimeDistributed(Dense(n_output))(h2)

model = Model(x_input, y_output)
model.compile(loss='mean_squared_error',
              optimizer=Adam(learning_rate=0.001))

# Training
hist = model.fit(x_train, y_train, epochs=50, batch_size=50)

# Visually see the loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'], color='red')
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()
```



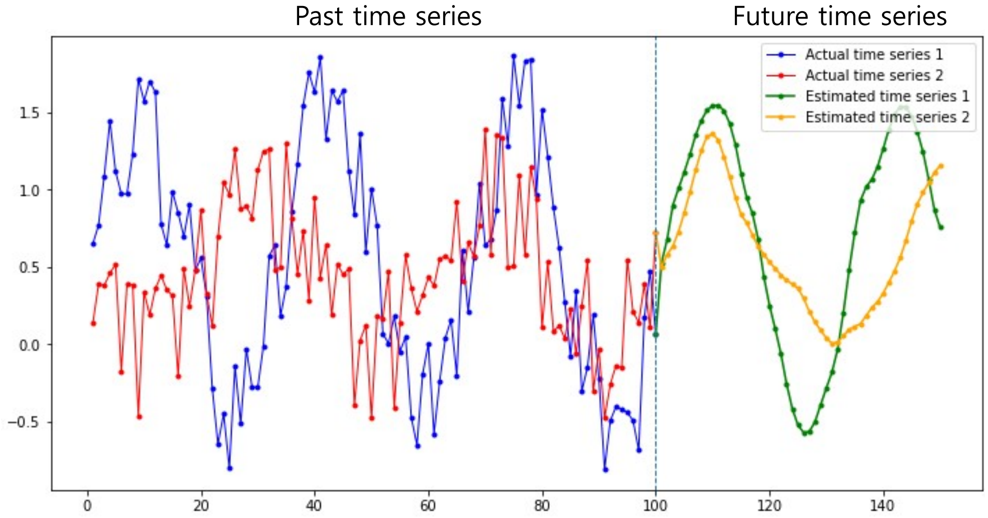
```
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 20, 2)]           0
 lstm (LSTM)                  (None, 20, 50)           10600
 lstm_1 (LSTM)                (None, 20, 50)           20200
 time_distributed (TimeDistr  (None, 20, 2)            102
 ibuted)
```

[MXDL-10-08] Deep Learning / Recurrent Neural Network (RNN) – Multi-layer RNN

- 2-layered unidirectional many-to-many RNN model for time series prediction

```
# Predict future values for the next 50 periods.
# After predicting the next value, re-enter the predicted value
# to predict the next value. Repeat this process 50 times.
n_future = 50
n_last = 100
last_data = data[-n_last:]  # The last n_last data points
for i in range(n_future):
    # Predict the next value with the last n_step data points.
    px = last_data[-n_step:, :].reshape(1, n_step, 2)

    # Predict the next value
    y_hat = model.predict(px, verbose=0)[:, -1, :]

    # Append the predicted value to the last_data array.
    # In the next iteration, the predicted value is input
    # along with the existing data points.
    last_data = np.vstack([last_data, y_hat])

p = last_data[:-n_future, :]        # past time series
f = last_data[-(n_future + 1):, :]  # future time series

# Plot past and future time series.
plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(p) + 1)
ax2 = np.arange(len(p), len(p) + len(f))
```

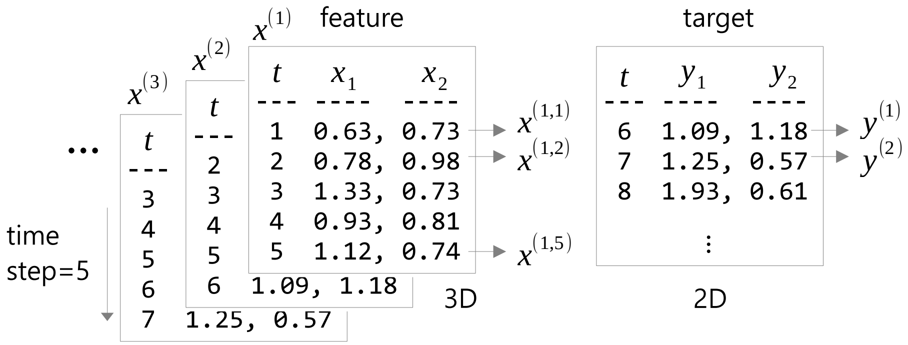
```
plt.plot(ax1, p[:, 0], '-o', c='blue', markersize=3,
         label='Actual time series 1', linewidth=1)
plt.plot(ax1, p[:, 1], '-o', c='red', markersize=3,
         label='Actual time series 2', linewidth=1)
plt.plot(ax2, f[:, 0], '-o', c='green', markersize=3,
         label='Estimated time series 1', linewidth=1)
plt.plot(ax2, f[:, 1], '-o', c='orange', markersize=3,
         label='Estimated time series 2', linewidth=1)
plt.axvline(x=ax1[-1],  linestyle='dashed', linewidth=1)
plt.legend()
plt.show()
```



[MXDL-10-08] Deep Learning / Recurrent Neural Network (RNN) – Bi-directional RNN

MX-AI

■ 2-layered bi-directional many-to-one RNN model for time series prediction



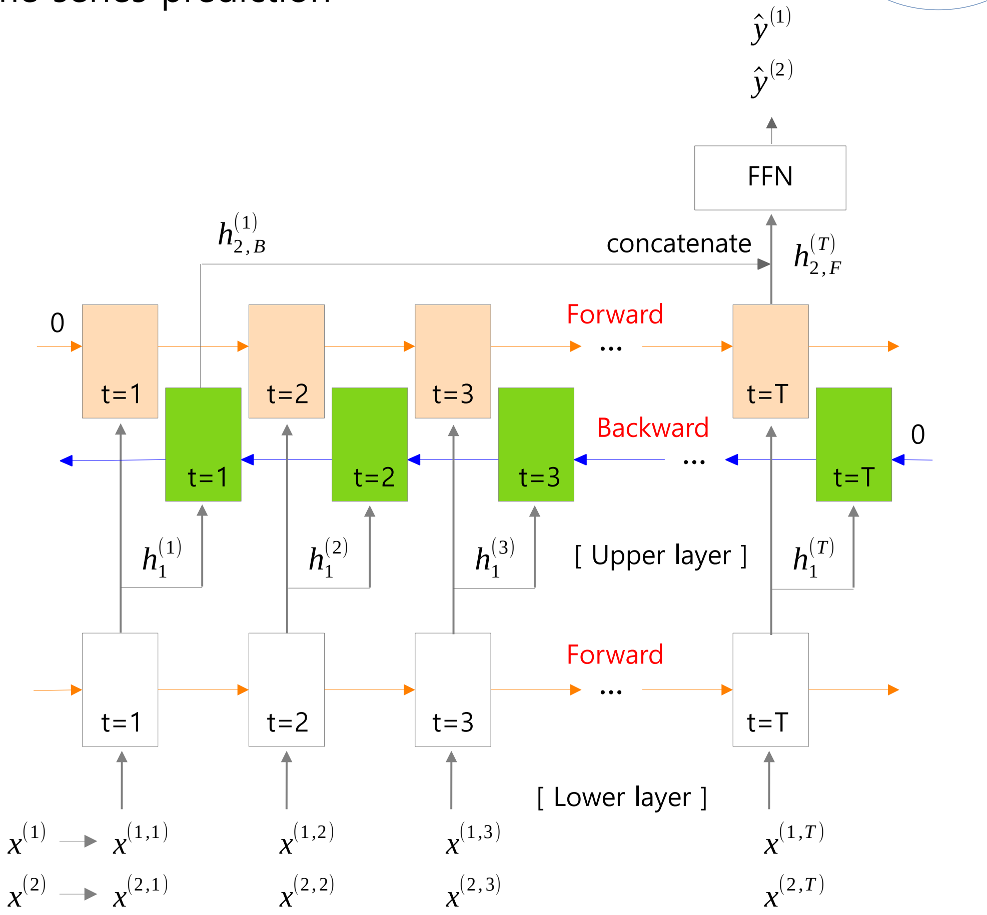
$x^{(1)}$ feature

t	x_1	x_2
1	0.63,	0.73
2	0.78,	0.98
3	1.33,	0.73
4	0.93,	0.81
5	1.12,	0.74

target

t	y_1	y_2
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
⋮		

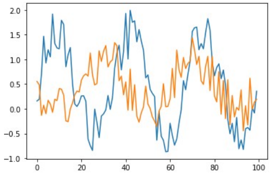
- The figure on the right shows that the lower layer is unidirectional RNN, the upper layer is bidirectional, and the overall model is a many-to-one RNN.
- The upper layer consists of a forward recurrent layer and a backward recurrent layer.
- First, the outputs of each time step of the lower layer are fed into the forward recurrent layer from the first time step to the last.
- Next, the outputs of each time step of the lower layer are also fed into the backward recurrent layer in reverse order, from the last time step to the first.
- Finally, the hidden states of the last time step of the two recurrent layers are combined.



MX-AI

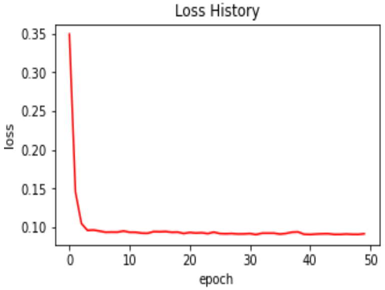
■ 2-layered bi-directional many-to-one RNN model for time series prediction

```
# [MXDL-10-08] 13.m2o_2layer_bi.py
from tensorflow.keras.layers import Dense, Input, LSTM
from tensorflow.keras.layers import Bidirectional
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt
```



```
# Generate training data: 2 noisy sine curves
n = 1000        # the number of data points
n_step = 20     # the number of time steps
s1 = np.sin(np.pi * 0.06 * np.arange(n)) + np.random.random(n)
s2 = 0.5*np.sin(np.pi * 0.05 * np.arange(n)) + np.random.random(n)
data = np.vstack([s1, s2]).T  # shape = (1000, 2)

m = np.arange(0, n - n_step)
x_train = np.array([data[i:(i+n_step), :] for i in m])
y_train = np.array([data[i, :] for i in (m + n_step)])

# Build a many-to-one, 2-layered, bi-directional LSTM model
n_feat = x_train.shape[-1]
n_output = y_train.shape[-1]
n_hidden = 50
```

```
x_input = Input(batch_shape=(None, n_step, n_feat))
h1 = LSTM(n_hidden, return_sequences=True)(x_input)
h2 = Bidirectional(LSTM(n_hidden), merge_mode='concat')(h1)
y_output = Dense(n_output)(h2)

model = Model(x_input, y_output)
model.compile(loss='mean_squared_error',
              optimizer=Adam(learning_rate=0.001))
# Training
hist = model.fit(x_train, y_train, epochs=50, batch_size=50)

# Visually see the loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'], color='red'
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()
```



```
 Layer (type)                  Output Shape              Param #
=================================================================
 input_1 (InputLayer)          [(None, 20, 2)]           0
 lstm (LSTM)                    (None, 20, 50)           10600
 bidirectional (Bidirectional)  (None, 100)              40400
 dense (Dense)                  (None, 2)                202
```

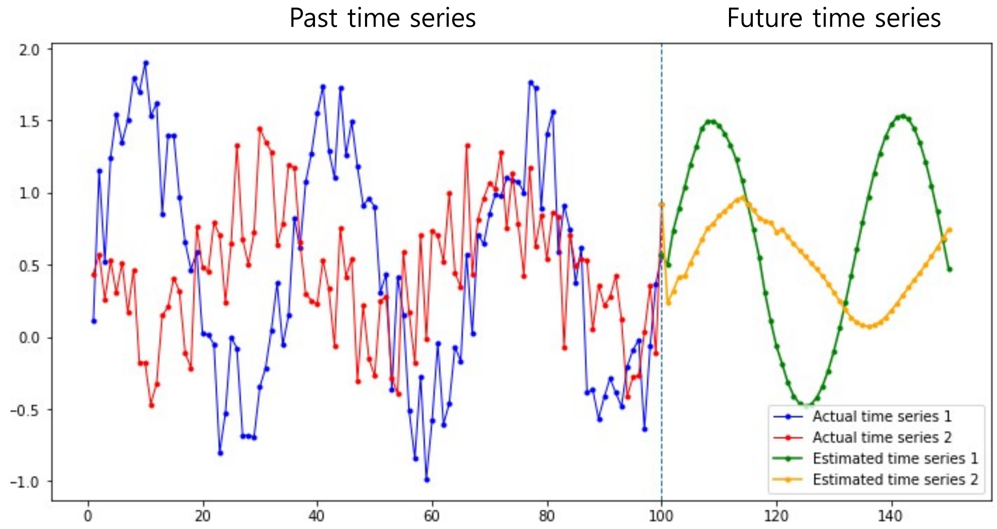
[MXDL-10-08] Deep Learning / Recurrent Neural Network (RNN) – Bi-directional RNN

MX-AI

■ 2-layered bi-directional many-to-one RNN model for time series prediction

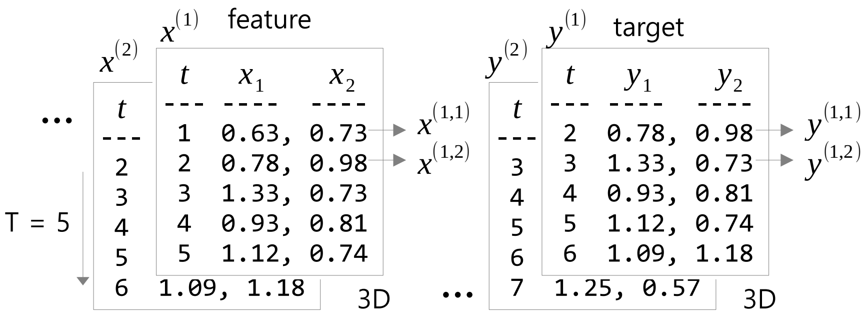
```
# Predict future values for the next 50 periods.
# After predicting the next value, re-enter the predicted value
# to predict the next value. Repeat this process 50 times.
n_future = 50
n_last = 100
last_data = data[-n_last:]  # The last n_last data points
for i in range(n_future):
    # Predict the next value with the last n_step data points.
    px = last_data[-n_step:, :].reshape(1, n_step, 2)

    # Predict the next value
    y_hat = model.predict(px, verbose=0)

    # Append the predicted value to the last_data array.
    # In the next iteration, the predicted value is input
    # along with the existing data points.
    last_data = np.vstack([last_data, y_hat])

p = last_data[:-n_future, :]        # past time series
f = last_data[-(n_future + 1):, :]  # future time series

# Plot past and future time series.
plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(p) + 1)
ax2 = np.arange(len(p), len(p) + len(f))
```

```
plt.plot(ax1, p[:, 0], '-o', c='blue', markersize=3,
         label='Actual time series 1', linewidth=1)
plt.plot(ax1, p[:, 1], '-o', c='red', markersize=3,
         label='Actual time series 2', linewidth=1)
plt.plot(ax2, f[:, 0], '-o', c='green', markersize=3,
         label='Estimated time series 1', linewidth=1)
plt.plot(ax2, f[:, 1], '-o', c='orange', markersize=3,
         label='Estimated time series 2', linewidth=1)
plt.axvline(x=ax1[-1],  linestyle='dashed', linewidth=1)
plt.legend()
plt.show()
```



[MXDL-10-08] Deep Learning / Recurrent Neural Network (RNN) – Bi-directional RNN

■ 2-layered bi-directional many-to-many RNN model for time series prediction



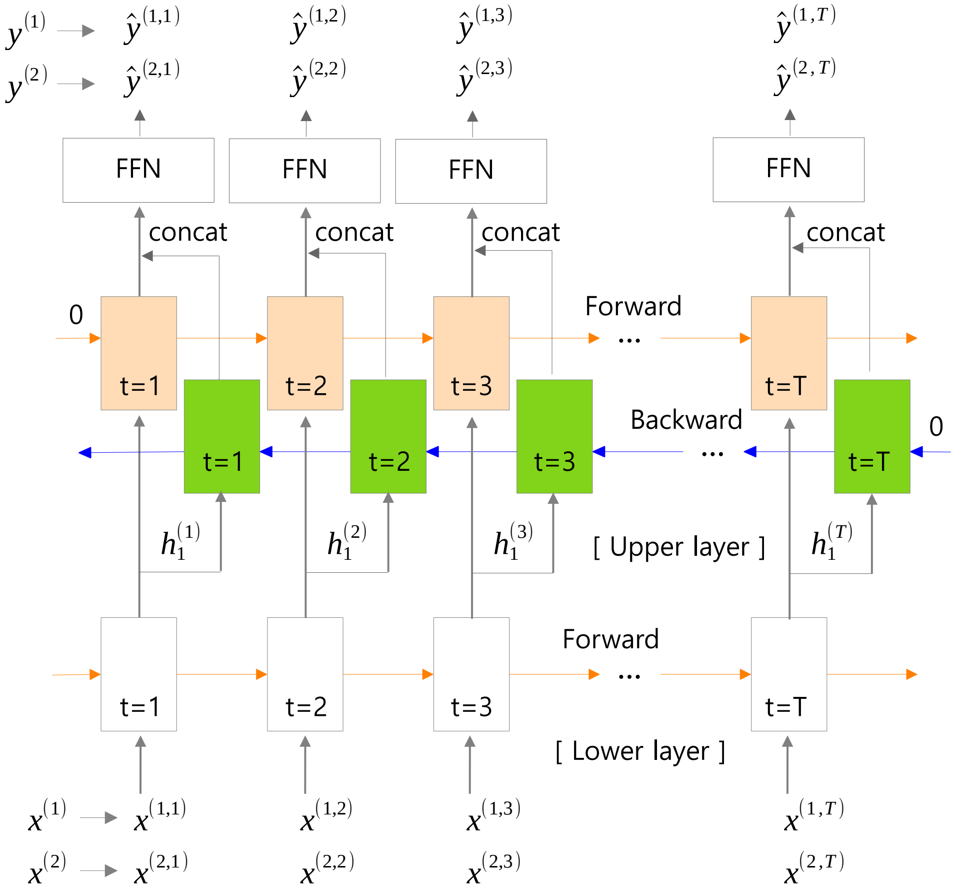
$x^{(1)}$ feature

t	x_1	x_2
1	0.63,	0.73
2	0.78,	0.98
3	1.33,	0.73
4	0.93,	0.81
5	1.12,	0.74

$y^{(1)}$ target

t	y_1	y_2
2	0.78,	0.98
3	1.33,	0.73
4	0.93,	0.81
5	1.12,	0.74
6	1.09,	1.18

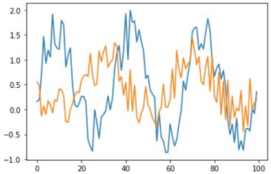
- The figure on the right shows that the lower layer is unidirectional RNN, the upper layer is bidirectional, and the overall model is a many-to-many RNN.
- The upper layer consists of a forward recurrent layer and a backward recurrent layer.
- First, the outputs of each time step of the lower layer are fed into the forward recurrent layer from the first time step to the last.
- Next, the outputs of each time step of the lower layer are also fed into the backward recurrent layer in reverse order, from the last time step to the first.
- Finally, the hidden states of the two recurrent layers are combined at each time step and passed to the output layer.



[MXDL-10-08] Deep Learning / Recurrent Neural Network (RNN) – Bi-directional RNN

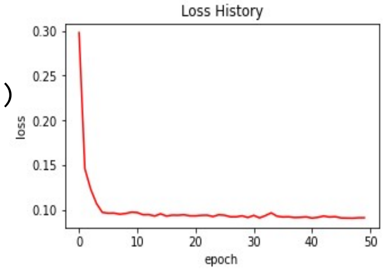
- 2-layered bi-directional many-to-many RNN model for time series prediction

```
# [MXDL-10-08] 14.m2m_2layer_bi.py
from tensorflow.keras.layers import Dense, Input, LSTM
from tensorflow.keras.layers import Bidirectional, TimeDistributed
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt
```



```
# Generate training data: 2 noisy sine curves
n = 1000        # the number of data points
n_step = 20     # the number of time steps
s1 = np.sin(np.pi * 0.06 * np.arange(n)) + np.random.random(n)
s2 = 0.5*np.sin(np.pi * 0.05 * np.arange(n)) + np.random.random(n)
data = np.vstack([s1, s2]).T
m = np.arange(0, n - n_step)
x_train = np.array([data[i:(i+n_step), :] for i in m])
y_train = np.array([data[(i+1):(i+1+n_step), :] for i in m])

# Build a many-to-many, 2-layered, bi-directional LSTM model
n_feat = x_train.shape[-1]
n_output = y_train.shape[-1]
n_hidden = 50
```

```
x_input = Input(batch_shape=(None, n_step, n_feat))
h1 = LSTM(n_hidden, return_sequences=True)(x_input)
h2 = Bidirectional(LSTM(n_hidden, return_sequences=True))(h1)
y_output = TimeDistributed(Dense(n_output))(h2)

model = Model(x_input, y_output)
model.compile(loss='mean_squared_error',
              optimizer=Adam(learning_rate=0.001))

# Training
hist = model.fit(x_train, y_train, epochs=50, batch_size=50)

# Visually see the loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'], color='red')
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()
```



```
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 20, 2)]           0
 lstm (LSTM)                  (None, 20, 50)           10600
 bidirectional (Bidirectional) (None, 20, 100)         40400
 time_distributed (TimeDistr  (None, 20, 2)            202
 ibuted)
```

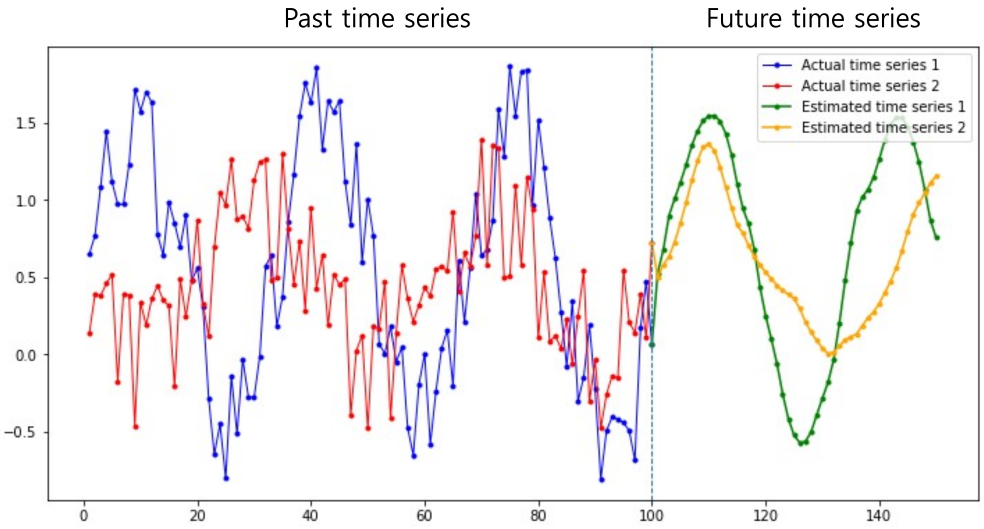
[MXDL-10-08] Deep Learning / Recurrent Neural Network (RNN) – Bi-directional RNN

MX-AI

- 2-layered bi-directional many-to-many RNN model for time series prediction

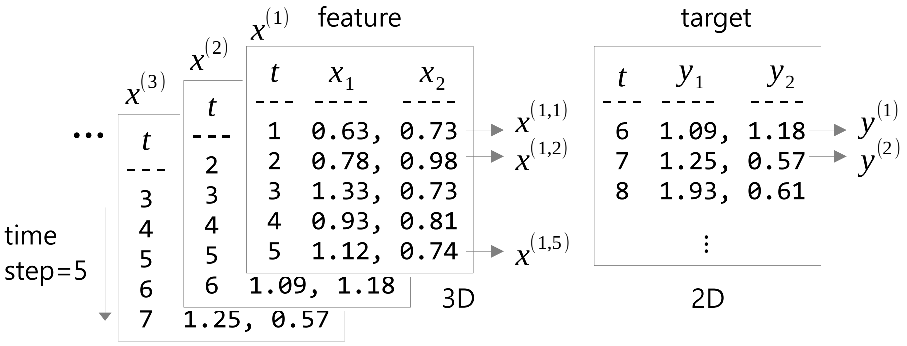
```
# Predict future values for the next 50 periods.
# After predicting the next value, re-enter the predicted value
# to predict the next value. Repeat this process 50 times.
n_future = 50
n_last = 100
last_data = data[-n_last:]  # The last n_last data points
for i in range(n_future):
    # Predict the next value with the last n_step data points.
    px = last_data[-n_step:, :].reshape(1, n_step, 2)

    # Predict the next value
    y_hat = model.predict(px, verbose=0)[:, -1, :]

    # Append the predicted value to the last_data array.
    # In the next iteration, the predicted value is input
    # along with the existing data points.
    last_data = np.vstack([last_data, y_hat])

p = last_data[:-n_future, :]        # past time series
f = last_data[-(n_future + 1):, :]  # future time series

# Plot past and future time series.
plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(p) + 1)
ax2 = np.arange(len(p), len(p) + len(f))
```

```
plt.plot(ax1, p[:, 0], '-o', c='blue', markersize=3,
         label='Actual time series 1', linewidth=1)
plt.plot(ax1, p[:, 1], '-o', c='red', markersize=3,
         label='Actual time series 2', linewidth=1)
plt.plot(ax2, f[:, 0], '-o', c='green', markersize=3,
         label='Estimated time series 1', linewidth=1)
plt.plot(ax2, f[:, 1], '-o', c='orange', markersize=3,
         label='Estimated time series 2', linewidth=1)
plt.axvline(x=ax1[-1],  linestyle='dashed', linewidth=1)
plt.legend()
plt.show()
```

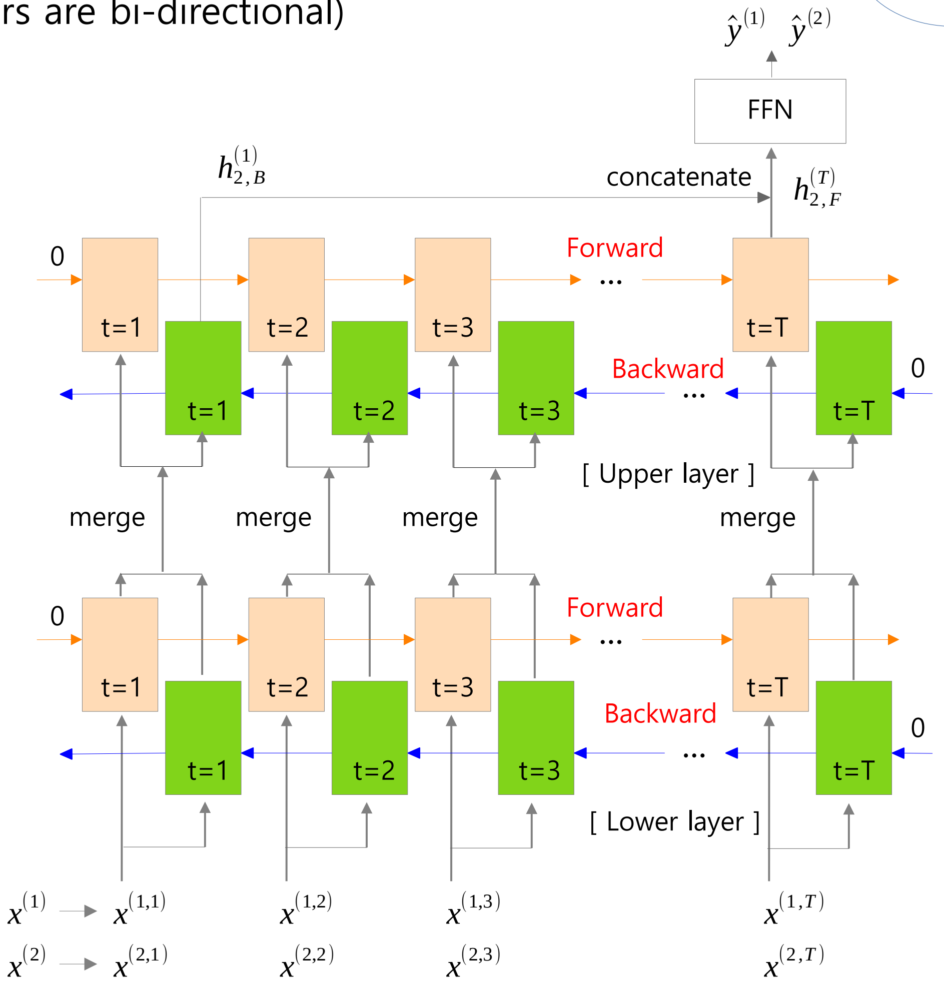


[MXDL-10-08] Deep Learning / Recurrent Neural Network (RNN) – Bi-directional RNN

- 2-layered bi-directional many-to-one model (Both layers are bi-directional)



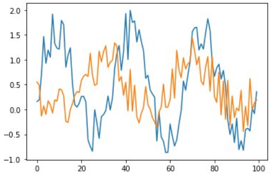
- The figure on the right shows that both the lower and upper layers are bidirectional, and the overall structure is many-to-one.
- In the lower layers, the hidden states of the two recurrent layers are combined at each time step and passed to each time step of the upper layers.
- In the upper layers, the hidden states of the last time steps of the two recurrent layers are combined and passed to the feedforward network.



$\hat{y}^{(1)}$ $\hat{y}^{(2)}$
FFN
concatenate
$h_{2,B}^{(1)}$
$h_{2,F}^{(T)}$
0
Forward
t=1
t=2
t=3
...
t=T
Backward
0
[Upper layer]
merge
merge
merge
merge
0
Forward
Backward
0
[Lower layer]
$x^{(1)}$ $x^{(1,1)}$ $x^{(1,2)}$ $x^{(1,3)}$ $x^{(1,T)}$
$x^{(2)}$ $x^{(2,1)}$ $x^{(2,2)}$ $x^{(2,3)}$ $x^{(2,T)}$

- 2-layered bi-directional many-to-one model (Both layers are bi-directional)

```
# [MXDL-10-08] 15.m2o_2layer_bibi.py
from tensorflow.keras.layers import Dense, Input, LSTM
from tensorflow.keras.layers import Bidirectional
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt
```



```
# Generate training data: 2 noisy sine curves
n = 1000        # the number of data points
n_step = 20     # the number of time steps
s1 = np.sin(np.pi * 0.06 * np.arange(n)) + np.random.random(n)
s2 = 0.5*np.sin(np.pi * 0.05 * np.arange(n)) + np.random.random(n)
data = np.vstack([s1, s2]).T  # shape = (1000, 2)

m = np.arange(0, n - n_step)
x_train = np.array([data[i:(i+n_step), :] for i in m])
y_train = np.array([data[i, :] for i in (m + n_step)])

# 2-layered bi-directional many-to-one model
# Both layers are bi-directional
n_feat = x_train.shape[-1]
n_output = y_train.shape[-1]
n_hidden = 50
```

```
x_input = Input(batch_shape=(None, n_step, n_feat))
h1 = Bidirectional(LSTM(n_hidden, return_sequences=True))(x_input)
h2 = Bidirectional(LSTM(n_hidden))(h1)
y_output = Dense(n_output)(h2)
model = Model(x_input, y_output)
model.compile(loss='mean_squared_error',
              optimizer=Adam(learning_rate=0.001))

# Training
hist = model.fit(x_train, y_train, epochs=50, batch_size=50)

# Visually see the loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'], color='red')
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()
```

```
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)       [(None, 20, 2)]            0
 bidirectional (Bidirectional  (None, 20, 100)         21200
 bidirectional_1 (Bidirectional)  (None, 100)          60400
 dense (Dense)               (None, 2)                 202
```

[MXDL-10-08] Deep Learning / Recurrent Neural Network (RNN) – Bi-directional RNN

MX-AI

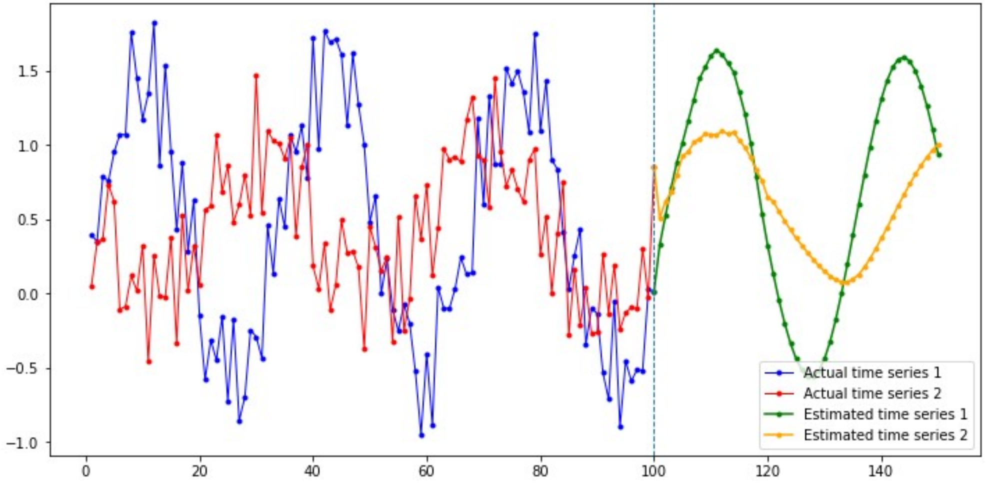
- 2-layered bi-directional many-to-one model (Both layers are bi-directional)

```python
# Predict future values for the next 50 periods.
# After predicting the next value, re-enter the predicted value
# to predict the next value. Repeat this process 50 times.
n_future = 50
n_last = 100
last_data = data[-n_last:]  # The last n_last data points
for i in range(n_future):
    # Predict the next value with the last n_step data points.
    px = last_data[-n_step:, :].reshape(1, n_step, 2)

    # Predict the next value
    y_hat = model.predict(px, verbose=0)

    # Append the predicted value to the last_data array.
    # In the next iteration, the predicted value is input
    # along with the existing data points.
    last_data = np.vstack([last_data, y_hat])

p = last_data[:-n_future, :]         # past time series
f = last_data[-(n_future + 1):, :]  # future time series

# Plot past and future time series.
plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(p) + 1)
ax2 = np.arange(len(p), len(p) + len(f))
```

```python
plt.plot(ax1, p[:, 0], '-o', c='blue', markersize=3,
         label='Actual time series 1', linewidth=1)
plt.plot(ax1, p[:, 1], '-o', c='red', markersize=3,
         label='Actual time series 2', linewidth=1)
plt.plot(ax2, f[:, 0], '-o', c='green', markersize=3,
         label='Estimated time series 1', linewidth=1)
plt.plot(ax2, f[:, 1], '-o', c='orange', markersize=3,
         label='Estimated time series 2', linewidth=1)
plt.axvline(x=ax1[-1],  linestyle='dashed', linewidth=1)
plt.legend()
plt.show()
```

Past time series

Future time series


Actual time series 1
Actual time series 2
Estimated time series 1
Estimated time series 2