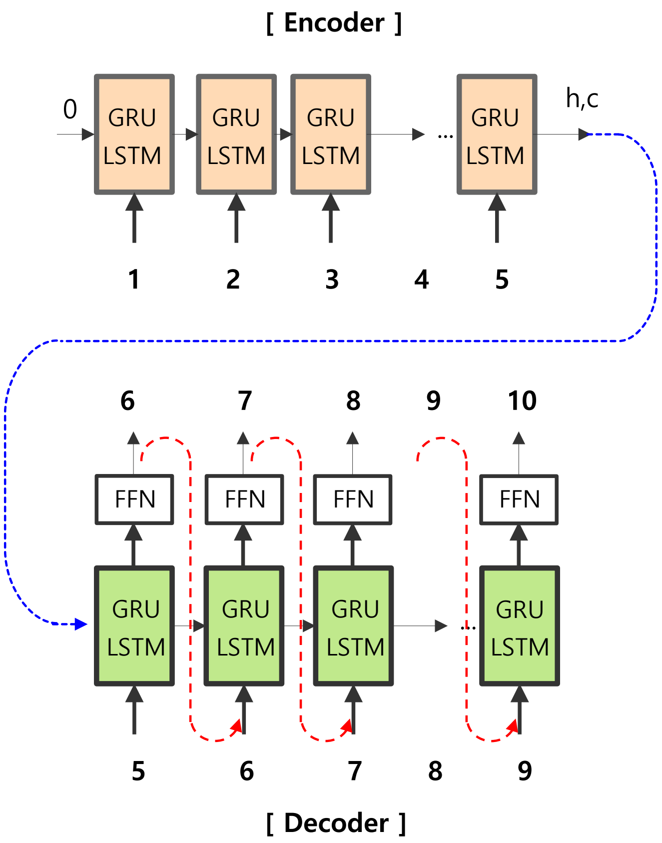
MX-AI



11. Attention Networks

Part 1: Sequence-to-Sequence model

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

www.youtube.com/@meanxai

MX-AI

MX-AI

■ RNN Encoder-Decoder model

- In 2014, Kyunghyun Cho et al. proposed RNN Encoder-Decoder model in their paper "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation."

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

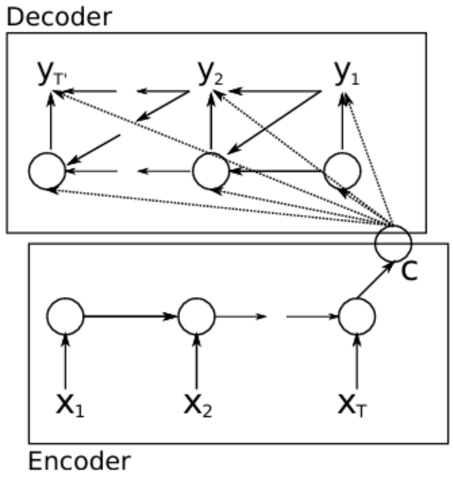
Dzmitry Bahdanau
Jacobs University, Germany
d.bahdanau@jacobs-university.de

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.



RNN Encoder-Decoder model (Sequence-to-Sequence model)

- The RNN encoder-decoder model was proposed for machine translation in the field of natural language processing, but it can be applied to any dataset composed of sequences, such as chatbots or time series prediction.
- Both the encoder and decoder are composed of RNNs such as GRU and LSTM. RNNs can be constructed as single- or multi-layer, and uni- or bi-directional. The final hidden states h (and c) of the encoder are fed as the initial hidden states of the decoder.
- Both feature x and target y in the dataset are sequence data. The feature x is input to the encoder, and the target y is output to the decoder.

- **dataset**

$$x = [x_1, x_2, x_3, \dots, x_T]$$
$$[\quad \vdots \quad]$$

$$y = [y_1, y_2, y_3, \dots, y_{T'}]$$
$$[\quad \vdots \quad]$$

- **Time series prediction**

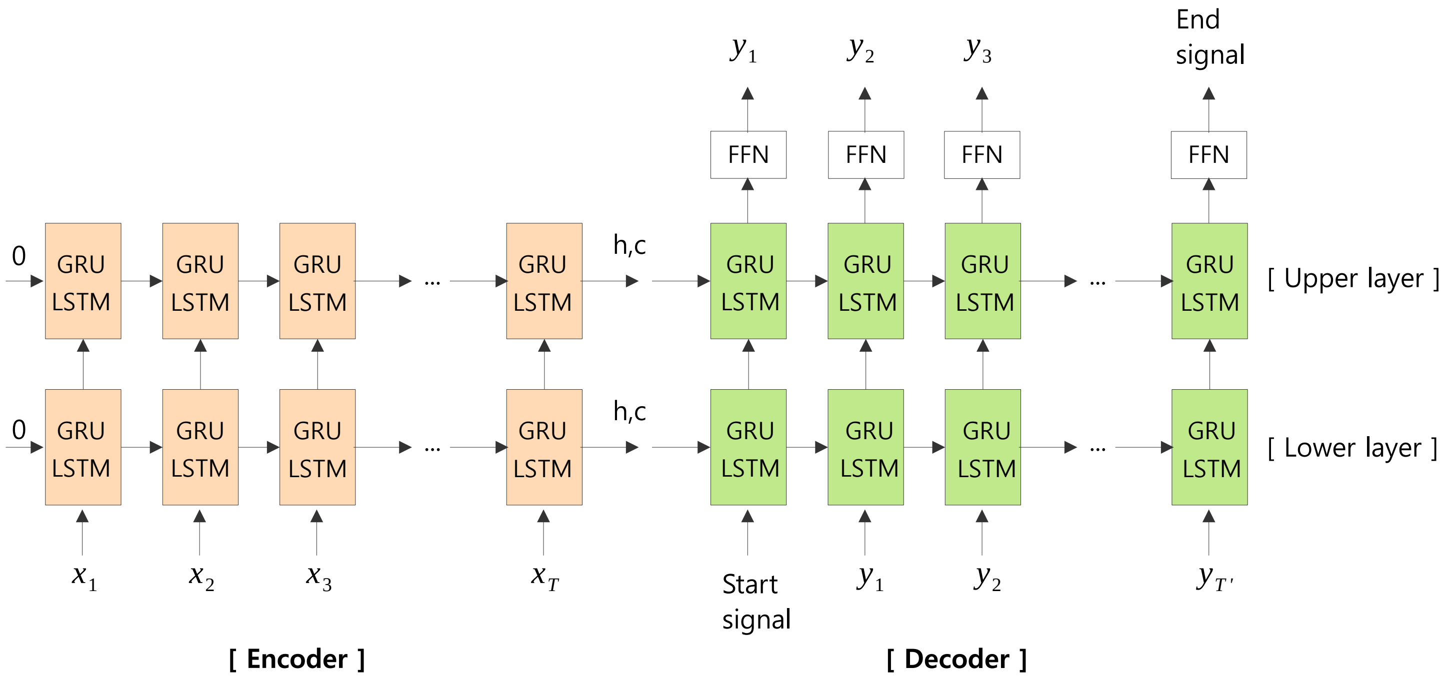
  x: Past time series
  y: Future Time series

- **Conversation (ChatBot)**

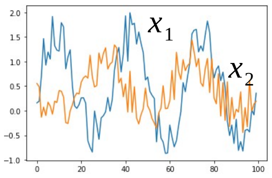
  x: Question
  y: Answer

- **Machine translation**

  x: English
  y: Korean



Constructing a dataset for time series prediction



[Time series dataset]

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	1.00,	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

Past time series → Encoder inputs

Only for the teacher forcing technique → Decoder inputs

Future time series → Decoder outputs

$x^{(1)}$ Encoder inputs

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

Decoder inputs

t	x_1	x_2
5	1.12,	0.74
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
9	1.00,	1.05

Decoder outputs $y^{(1)}$

t	x_1	x_2
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
9	1. ,	1.05
10	1.74,	1.37

$x^{(2)}$ Encoder inputs

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

Decoder inputs

t	x_1	x_2
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
9	1. ,	1.05
10	1.74,	1.37

Decoder outputs $y^{(2)}$

t	x_1	x_2
7	1.25,	0.57
8	1.93,	0.61
9	1. ,	1.05
10	1.74,	1.37
11	1.83,	1.08

$x^{(3)}$ Encoder inputs

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

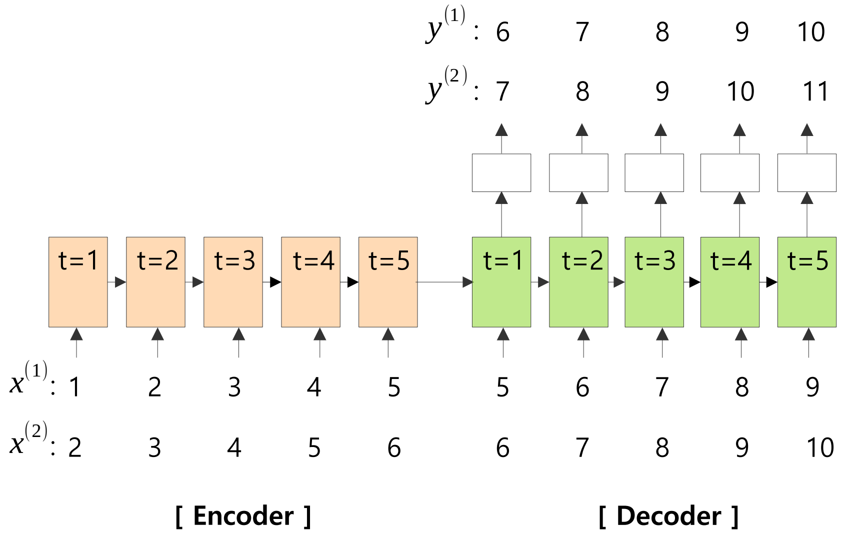
Decoder inputs

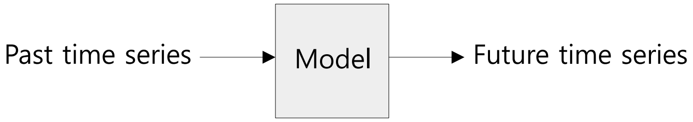
t	x_1	x_2
7	1.25,	0.57
8	1.93,	0.61
9	1. ,	1.05
10	1.74,	1.37
11	1.83,	1.08

Decoder outputs $y^{(3)}$

t	x_1	x_2
8	1.93,	0.61
9	1. ,	1.05
10	1.74,	1.37
11	1.83,	1.08
12	1.53,	1.24

3D ⋮ ⋮ ⋮





MX-AI

Training stage: Teacher forcing

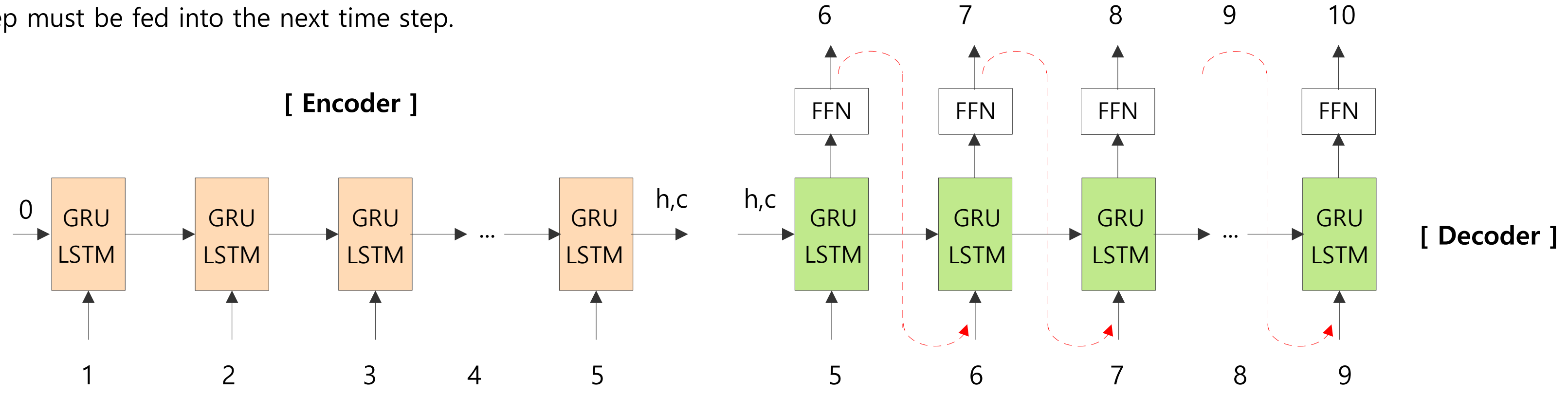
- The encoder and decoder are combined into a single model and trained using a method called teacher forcing.
- In the decoder, the output of the previous time step must be fed into the next time step.
- However, since the outputs of the previous time steps are the predicted values, during training, observations that are more accurate than the predicted values are input to the next time step. This is the teacher forcing.
- During prediction, since the future time series are unknown, the output of the previous time step must be fed into the next time step.

$y^{(1)}$

t	x_1	x_2
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
9	1.00,	1.05
10	1.74,	1.37

Decoder outputs

(Future time series)



$x^{(1)}$

Encoder inputs

(Past time series)

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

This is only for the teacher forcing during training. This cannot be used during forecasting because the future time series are unknown.

t	x_1	x_2
5	1.12,	0.74
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
9	1. ,	1.05

Decoder inputs

(Future time series)

Prediction stage

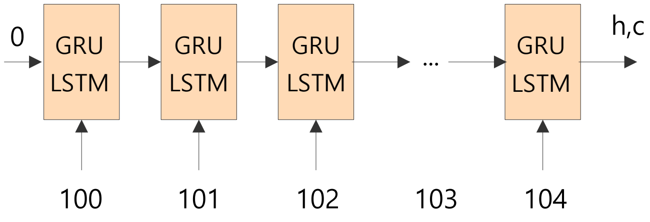
- Once training is complete, we use the following procedure to predict future time series.
- Both the encoder and decoder are trained.
- During prediction, the "decoder inputs" used in the training stage cannot be used, because the future time series are unknown.

[Time series dataset]

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
⋮	⋮	⋮
100	-0.10,	-0.20
101	-0.60,	0.70
102	-0.36,	0.33
103	0.22,	0.34
104	0.17,	0.89

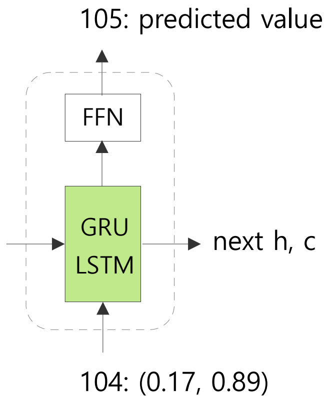
1. Build an encoder model for prediction, feed the last n data points of the time series into the model, and find the final hidden state.

[Encoder model]



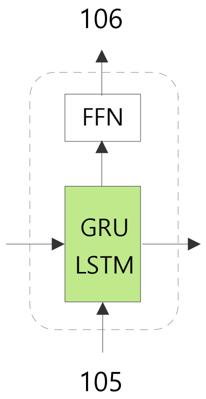
Feed the last n data points into the encoder model

2. Build a single-step decoder model for prediction, feed the hidden state of the encoder and the last data point of the time series into the model, and find the next hidden state and the next predicted value.

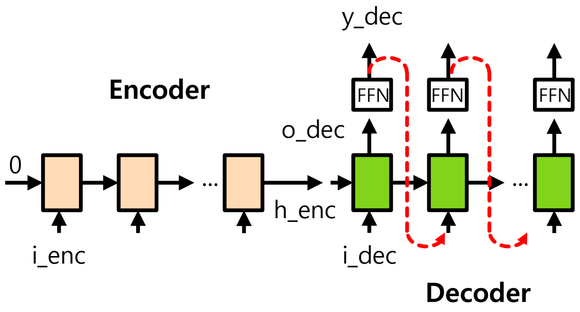


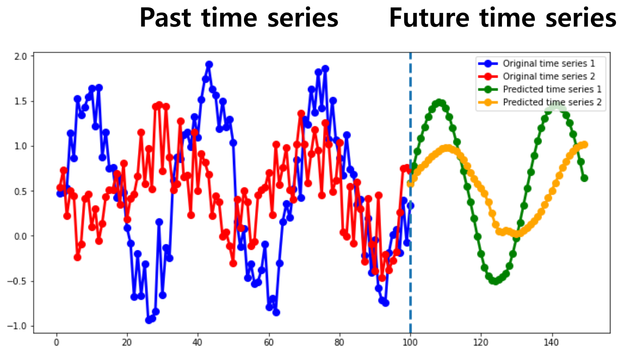
[single-step decoder model]

3. feed the next hidden state and predicted value of the first decoder back into the single decoder model. And find the next hidden state and the next predicted value. Repeat this process the given number of times.



[single-step decoder model]





11. Attention Networks

Part 2: Implementation of a Seq2Seq model

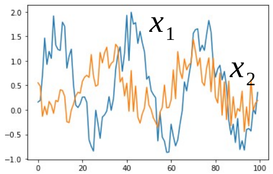
This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

[MXDL-11-02] Deep Learning / Attention Networks - Sequence-to-Sequence model

MX-AI

- Writing a code to generate the dataset for time series prediction



[Time series dataset]
(data)

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	1.00,	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

Past time series → Encoder inputs (xi_enc)

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	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

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 ⋮

Only for the teacher forcing technique → Decoder inputs (xi_dec)

t	x_1	x_2
5	1.12,	0.74
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
9	1.00,	1.05

t	x_1	x_2
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
9	1.00,	1.05
10	1.74,	1.37

t	x_1	x_2
7	1.25,	0.57
8	1.93,	0.61
9	1. ,	1.05
10	1.74,	1.37
11	1.83,	1.08

⋮

Future time series → Decoder outputs (xo_dec)

t	x_1	x_2
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
9	1. ,	1.05
10	1.74,	1.37

t	x_1	x_2
7	1.25,	0.57
8	1.93,	0.61
9	1. ,	1.05
10	1.74,	1.37
11	1.83,	1.08

t	x_1	x_2
8	1.93,	0.61
9	1. ,	1.05
10	1.74,	1.37
11	1.83,	1.08
12	1.53,	1.24

⋮

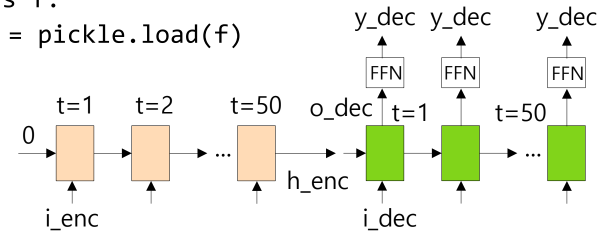
```python
# [MXDL-11-02] 1.dataset.py
import numpy as np
import pickle

# Generate a dataset consisting of two noisy sine curves
n = 5000   # the number of data points
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

# Generate the training data for a Seq2Seq model.
t = 50     # the number of sequences
m = np.arange(0, n-2*t+1)
xi_enc = np.array([data[i:(i+t), :] for i in m])
xi_dec = np.array([data[(i+t-1):(i+2*t-1), :] for i in m])
xo_dec = np.array([data[(i+t):(i+2*t), :] for i in m])

# Save the training data for later use
with open('dataset.pkl', 'wb') as f:
        pickle.dump([data, xi_enc, xi_dec, xo_dec], f)

print("\nThe shape of the dataset:", data.shape)
print("The shape of the encoder input:", xi_enc.shape)
print("The shape of the decoder input:", xi_dec.shape)
print("The shape of the decoder output:", xo_dec.shape)
```

```
The shape of the dataset: (5000, 2)
The shape of the encoder input: (4901, 50, 2)
The shape of the decoder input: (4901, 50, 2)
The shape of the decoder output: (4901, 50, 2)
```

MX-AI

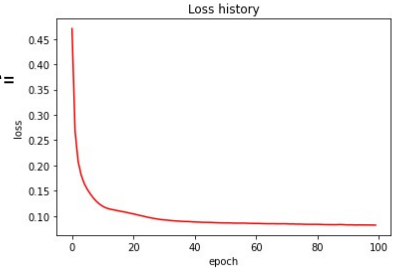
■ Seq2Seq model: Teacher forcing

```python
# [MXDL-11-02] 2.seq2seq(train).py
from tensorflow.keras.layers import Input, GRU, Dense, TimeDistributed
from tensorflow.keras.models import Model
import matplotlib.pyplot as plt
import numpy as np
import pickle

# Read the training dataset
with open('dataset.pkl', 'rb') as f:
      _, xi_enc, xi_dec, xp_dec = pickle.load(f)

n_hidden = 100
n_step = xi_enc.shape[1]   # 50
n_feat = xi_enc.shape[2]   # 2
```




```python
# Encoder
i_enc = Input(batch_shape=(None, n_step, n_feat))
h_enc = GRU(n_hidden)(i_enc)

# Decoder
i_dec = Input(batch_shape=(None, n_step, n_feat))
o_dec = GRU(n_hidden,
            return_sequences=True)(i_dec, initial_state = h_enc)
y_dec = TimeDistributed(Dense(n_feat))(o_dec)

model = Model([i_enc, i_dec], y_dec)
model.compile(loss='mse', optimizer='adam')
```

```python
# Training: teacher forcing
hist = model.fit([xi_enc, xi_dec], xp_dec,
                 batch_size=200, epochs=100)
# Save the trained model
model.save_weights("models/seq2seq.h5")

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

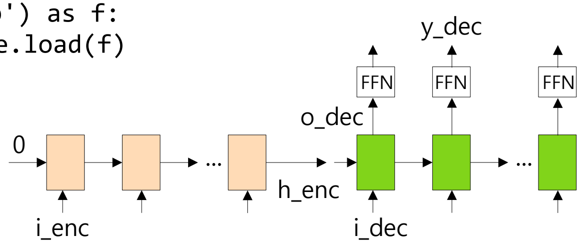



```
model.summary()

 Layer (type)            Output Shape       Param #    Connected to
=================================================================
 input_1 (InputLayer)   [(None, 50, 2)]     0          []
 input_2 (InputLayer)   [(None, 50, 2)]     0          []
 gru (GRU)              [(None, 100),       31200    ['input_1[0][0]']
                         (None, 100)]
 gru_1 (GRU)             (None, 50, 100)    31200    ['input_2[0][0]',
                                                      'gru[0][1]']
 time_distributed        (None, 50, 2)       202     ['gru_1[0][0]']
(TimeDistributed)
=================================================================
Total params: 62,602
Trainable params: 62,602
Non-trainable params: 0
```

MX-AI

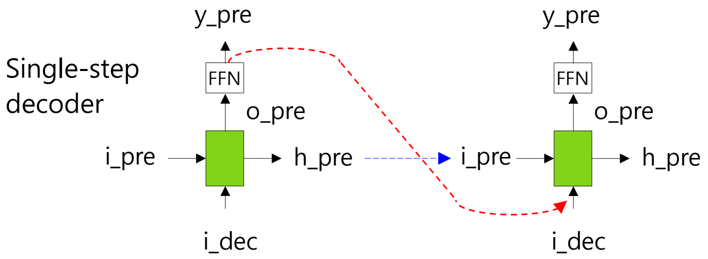
Seq2Seq model: Prediction

```python
# [MXDL-11-02] 3.seq2seq(predict).py
from tensorflow.keras.layers import Input, GRU, Dense
from tensorflow.keras.layers import TimeDistributed
from tensorflow.keras.models import Model
import matplotlib.pyplot as plt
import numpy as np
import pickle

# Read dataset
with open('dataset.pkl', 'rb') as f:
        data, _, _, _ = pickle.load(f)

n_hidden = 100
n_step = 50
n_feat = 2
```




```python
# Encoder
i_enc = Input(batch_shape=(None, n_step, n_feat))
h_enc = GRU(n_hidden)(i_enc)

# Decoder
# Instantiate a single-step GRU and a many-to-many output layer classes
# so that they can be shared later in the prediction decoder model.
SingleStepGRU = GRU(n_hidden, return_sequences=True, return_state=True)
ManyOUT = TimeDistributed(Dense(n_feat))

i_dec = Input(batch_shape=(None, 1, n_feat))
o_dec, _ = SingleStepGRU(i_dec, initial_state = h_enc)
y_dec = ManyOUT(o_dec)
model = Model([i_enc, i_dec], y_dec)
model.load_weights("models/seq2seq.h5")
```

```python
# Encoder model for time series forecasting.
Encoder = Model(i_enc, h_enc)

# Decoder for time series forecasting: single-step model
i_pre = Input(batch_shape = (None, n_hidden))
o_pre, h_pre = SingleStepGRU(i_dec, initial_state = i_pre)
y_pre = ManyOUT(o_pre)
Decoder = Model([i_dec, i_pre], [y_pre, h_pre])
```

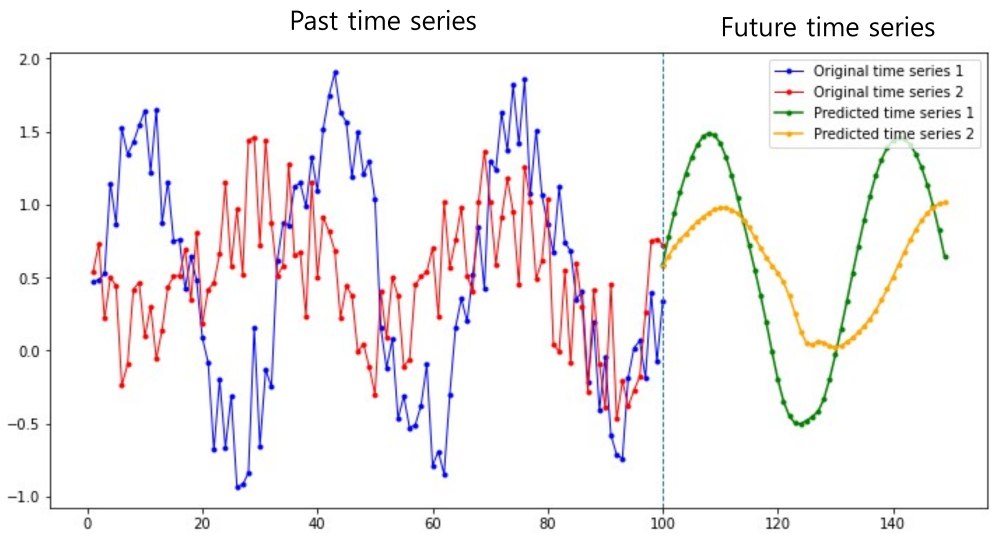


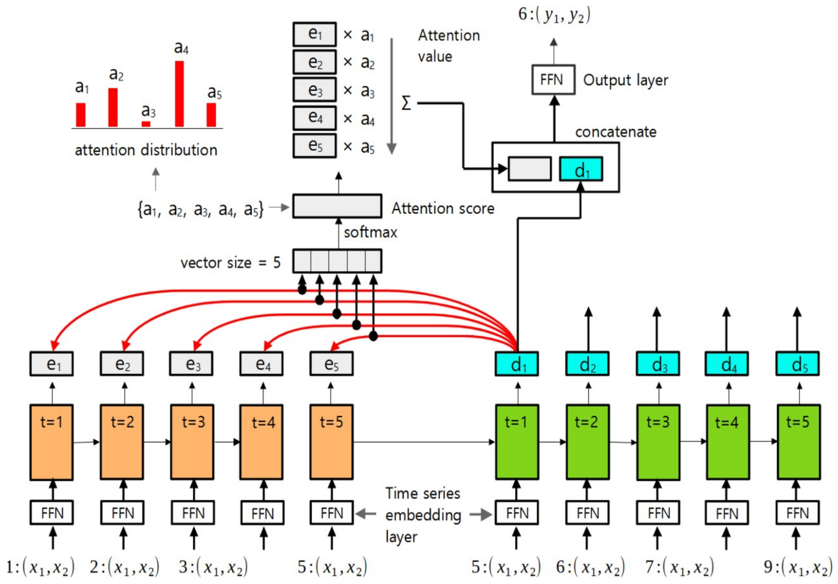

```python
# prediction
e_seed = data[-50:].reshape(-1, 50, 2)
d_seed = data[-1].reshape(-1, 1, 2)
he = Encoder.predict(e_seed, verbose=0)

n_future = 50
y_pred = []
for i in range(n_future):
    yd, hd = Decoder.predict([d_seed, he], verbose=0)
    y_pred.append(yd.reshape(2,))

    he = hd
    d_seed = yd
y_pred = np.array(y_pred)
```

MX-AI

■ Seq2Seq model: Prediction

```
# Plot the past time series and the predicted future time series.
y_past = data[-100:]
plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(y_past) + 1)
ax2 = np.arange(len(y_past), len(y_past) + len(y_pred))
plt.plot(ax1, y_past[:, 0], '-o', c='blue', markersize=3,
         label='Original time series 1', linewidth=1)
plt.plot(ax1, y_past[:, 1], '-o', c='red', markersize=3,
         label='Original time series 2', linewidth=1)
plt.plot(ax2, y_pred[:, 0], '-o', c='green', markersize=3,
         label='Predicted time series 1')
plt.plot(ax2, y_pred[:, 1], '-o', c='orange', markersize=3,
         label='Predicted time series 2')
plt.axvline(x=ax1[-1],  linestyle='dashed', linewidth=1)
plt.legend()
plt.show()
```





11. Attention Networks

Part 3: Seq2Seq-Attention model

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

www.youtube.com/@meanxai

MX-AI

■ Seq2Seq-Attention model

Effective Approaches to Attention-based Neural Machine Translation

Minh-Thang Luong **Hieu Pham** **Christopher D. Manning**
Computer Science Department, Stanford University, Stanford, CA 94305
{lmthang,hyhieu,manning}@stanford.edu

Abstract

An attentional mechanism has lately been used to improve neural machine translation (NMT) by selectively focusing on parts of the source sentence during translation. However, there has been little work exploring useful architectures for attention-based NMT. This paper examines two simple and effective classes of attentional mechanism: a global approach which always attends to all source words and a local one that only looks at a subset of source words at a time. We demonstrate the effectiveness of both approaches on the WMT translation tasks between English and German in both directions. With local attention, we achieve a significant gain of 5.0 BLEU points over non-attentional systems that already incorporate known techniques such as dropout. Our ensemble model using different attention architectures yields a new state-of-the-art result in the WMT'15 English to German translation task with 25.9 BLEU points, an improvement of 1.0 BLEU points over the existing best system backed by NMT and an n-gram reranker.

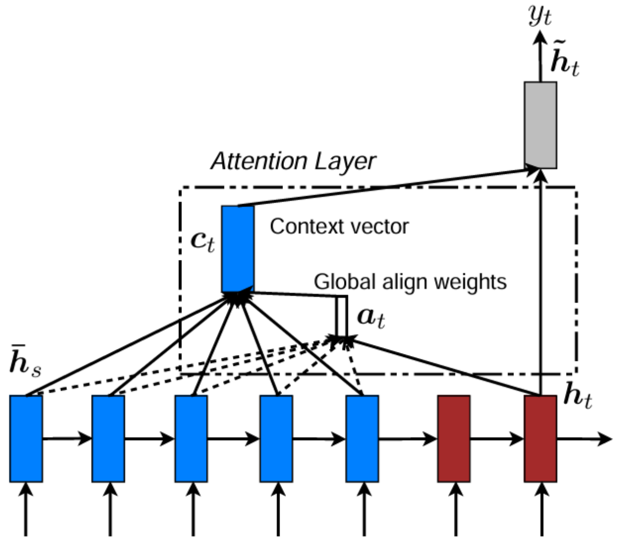


Figure 2: **Global attentional model** – at each time step t, the model infers a *variable-length* alignment weight vector a_t based on the current target state h_t and all source states $\bar{h}_s$. A global context vector c_t is then computed as the weighted average, according to a_t, over all the source states.

■ Seq2Seq-Attention model

Encoder inputs

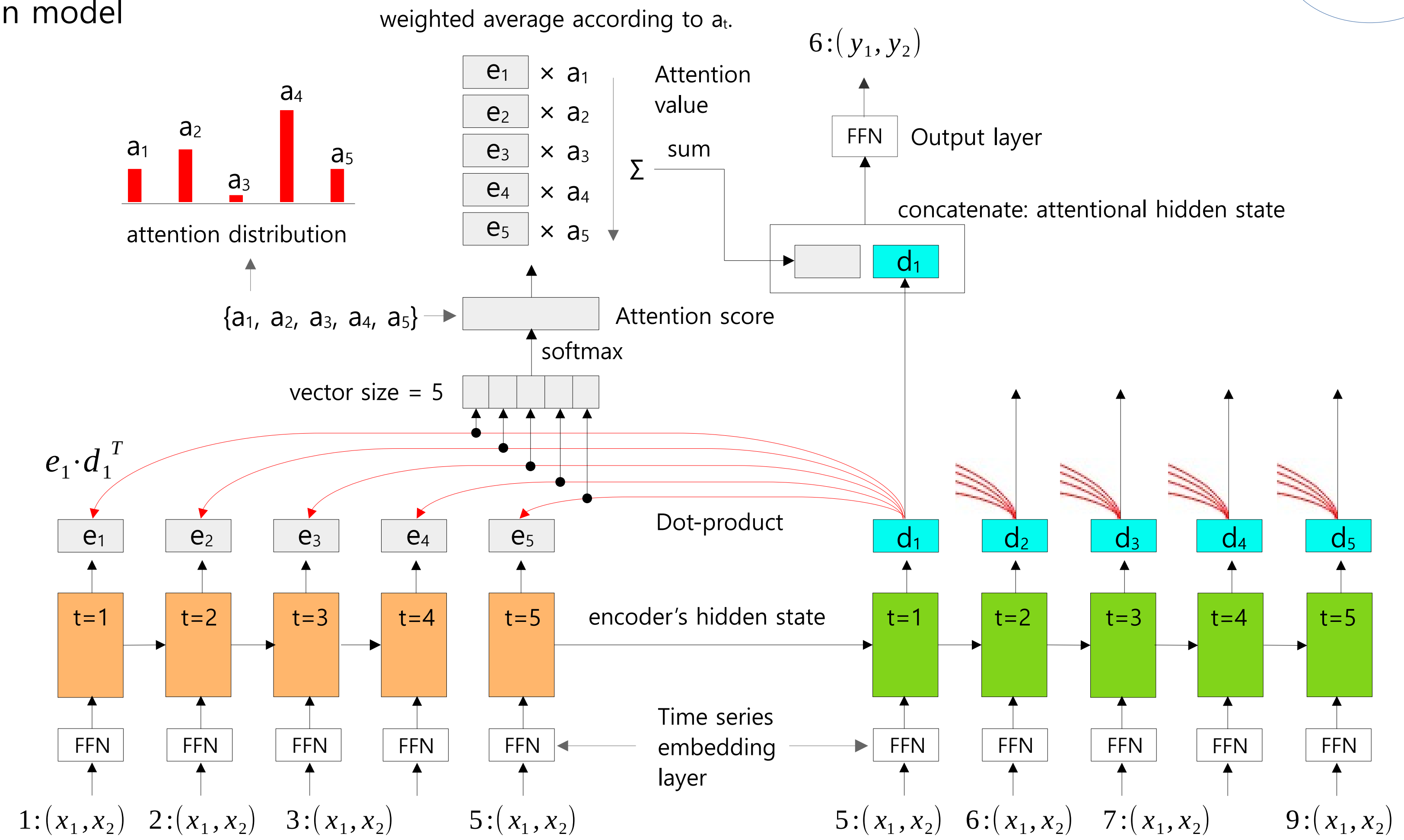
t	x_1	x_2
1	0.63,	0.73
2	0.78,	0.98
3	1.33,	0.73
4	0.93,	0.81
5	1.12,	0.74

Decoder inputs

t	x_1	x_2
5	1.12,	0.74
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
9	1.00,	1.05

Targets

t	y_1	y_2
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
9	1.	, 1.05
10	1.74,	1.37



MX-AI

■ Finding attention score and attention value

Encoder output (E) (1, 4, 3)

e1	0.707	0.616	0.852
e2	0.190	0.113	0.123
e3	0.757	0.022	0.236
e4	0.540	0.923	0.412

Decoder output (D) (1, 4, 3)

d1	0.786	0.634	0.873
d2	0.796	0.949	0.872
d3	0.704	0.314	0.912
d4	0.293	0.075	0.730

(1)

Dot-product: **P=Dot(axes=(2, 2))([D, E])**

	e1	e2	e3	e4
d1	1.690	0.328	0.815	1.369
d2	1.890	0.366	0.829	1.665
d3	1.468	0.281	0.755	1.046
d4	0.875	0.154	0.396	0.528

(1, 4, 4)

(2)

Attention score: **S=Activation('softmax')(P)**

d1	0.417	0.107	0.174	0.303
d2	0.423	0.092	0.147	0.338
d3	0.408	0.125	0.200	0.267
d4	0.356	0.173	0.220	0.251

(1, 4, 4)

(3)

Attention value: **A=Dot(axes=(2, 1))([S, E])**

0.610	0.552	0.534
0.610	0.586	0.546
0.608	0.517	0.520
0.587	0.475	0.480

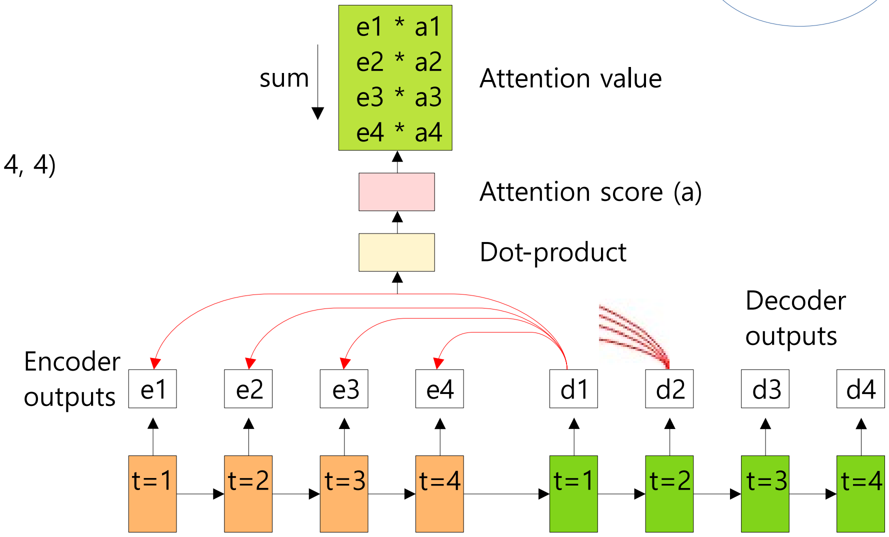
(1, 4, 3)

(4)

Attentional hidden state: **C=Concatenate()([A, D])**

c1	0.610	0.552	0.534	0.786	0.634	0.873
c2	0.610	0.586	0.546	0.796	0.949	0.872
c3	0.608	0.517	0.520	0.704	0.314	0.912
c4	0.587	0.475	0.480	0.293	0.075	0.730

(1, 4, 6)



* verification of the first row of attention value

Encoder output (E)

e1	0.707	0.616	0.852	× 0.417 = [0.295, 0.257, 0.355]
e2	0.190	0.113	0.123	× 0.107 = [0.02 , 0.012, 0.013]
e3	0.757	0.022	0.236	× 0.174 = [0.132, 0.004, 0.041]
e4	0.540	0.923	0.412	× 0.303 = [0.163, 0.279, 0.125]

sum = [0.610, 0.552, 0.534]

MX-AI

■ Finding attention score and attention value

```
# [MXDL-11-03] 4.compute_attention.py
from tensorflow.keras.layers import Dot, Activation, Concatenate
import numpy as np

E = np.array([[[0.707, 0.616, 0.852],
               [0.19 , 0.113, 0.123],
               [0.757, 0.022, 0.236],
               [0.54 , 0.923, 0.412]]])

D = np.array([[[0.786, 0.634, 0.873],
               [0.796, 0.949, 0.872],
               [0.704, 0.314, 0.912],
               [0.293, 0.075, 0.73 ]]])

def AttentionLayer(d, e):
    dot_product = Dot(axes=(2, 2))([d, e])
    score = Activation('softmax')(dot_product)
    value = Dot(axes=(2, 1))([score, e])
    output = Concatenate()([value, d])
    return dot_product, score, value, output

d, s, v, o = AttentionLayer(D, E)

print("\nDot-product:")
print(np.round(d, 3))

print("\nScore:")
print(np.round(s, 3))
```

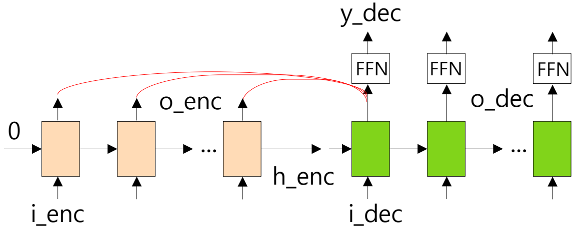
```
print("\nAttention values:")
print(np.round(v, 3))

print("\nAttentional hidden states:")
print(np.round(o, 3))
```

```
Dot-product:
[[[1.69  0.328 0.815 1.369]
  [1.89  0.366 0.829 1.665]
  [1.468 0.281 0.755 1.046]
  [0.875 0.154 0.396 0.528]]]

Score:
[[[0.417 0.107 0.174 0.303]
  [0.423 0.092 0.147 0.338]
  [0.408 0.125 0.2   0.267]
  [0.356 0.173 0.22  0.251]]]

Attention values:
[[[0.61  0.552 0.534]
  [0.61  0.586 0.546]
  [0.608 0.517 0.52 ]
  [0.587 0.475 0.48 ]]]

Attentional hidden states:
[[[0.61  0.552 0.534 0.786 0.634 0.873]
  [0.61  0.586 0.546 0.796 0.949 0.872]
  [0.608 0.517 0.52  0.704 0.314 0.912]
  [0.587 0.475 0.48  0.293 0.075 0.73 ]]]
```

MX-AI

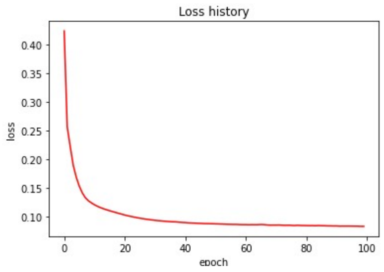
Implementing a Seq2Seq-Attention model for time series prediction: Training state (teacher forcing)

```python
# [MXDL-11-03] 5.Attention1(train).py (Simple version)
from tensorflow.keras.layers import Input, GRU, Dense, Dot
from tensorflow.keras.layers import Activation, Concatenate
from tensorflow.keras.layers import TimeDistributed
from tensorflow.keras.models import Model
from tensorflow.keras import optimizers
import matplotlib.pyplot as plt
import numpy as np
import pickle

# Read save dataset
with open('dataset.pkl', 'rb') as f:
      _, xi_enc, xi_dec, xp_dec = pickle.load(f)

n_hidden = 100
n_step = 50
n_feat = 2
n_emb = 30
```



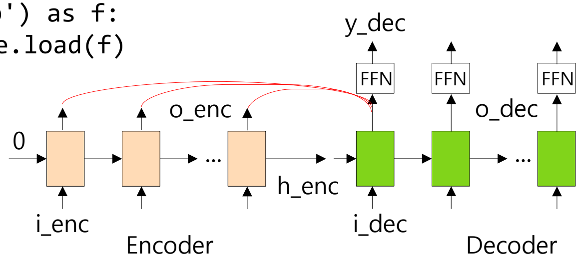

```python
def AttentionLayer(d, e):
    dot_product = Dot(axes=(2, 2))([d, e])
    score = Activation('softmax')(dot_product)
    value = Dot(axes=(2, 1))([score, e])
    return Concatenate()([value, d])

# Time series embedding layer.
EmbedInput = Dense(n_emb, activation='tanh')
```

```python
# Encoder
i_enc = Input(batch_shape=(None, n_step, n_feat))
e_enc = EmbedInput(i_enc)
o_enc, h_enc = GRU(n_hidden, return_sequences=True,
                   return_state = True)(i_enc)
# Decoder
i_dec = Input(batch_shape=(None, n_step, n_feat))
e_dec = EmbedInput(i_dec)
o_dec = GRU(n_hidden, return_sequences=True)(i_dec, initial_state=h_enc)
a_dec = AttentionLayer(o_dec, o_enc)
y_dec = TimeDistributed(Dense(n_feat))(a_dec)

model = Model([i_enc, i_dec], y_dec)
model.compile(loss='mse',
              optimizer=optimizers.Adam(learning_rate=0.001))

# Training: teacher forcing
hist = model.fit([xi_enc, xi_dec], xp_dec, batch_size=500, epochs=200)

# Save the trained model
model.save_weights("models/attention1.h5")

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

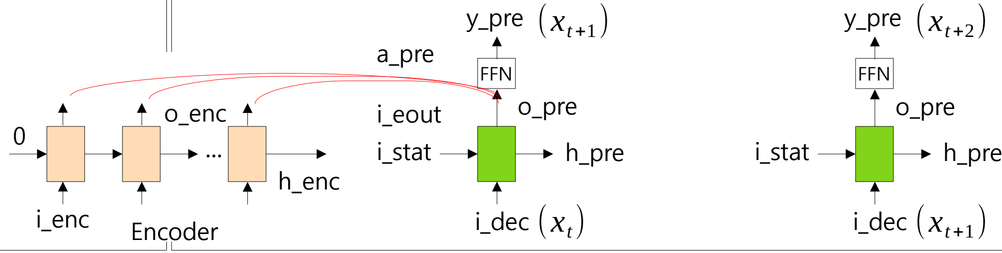


- Implementing a Seq2Seq-Attention model for time series prediction: Prediction stage

```
# [MXDL-11-03] 6.Attention1(predict).py (Simple version)
from tensorflow.keras.layers import Input, GRU, Dense, Dot, Activation
from tensorflow.keras.layers import Concatenate, TimeDistributed
from tensorflow.keras.models import Model
import matplotlib.pyplot as plt
import numpy as np
import pickle

# Read saved dataset
with open('dataset.pkl', 'rb') as f:
    data, _, _, _ = pickle.load(f)

n_hidden = 100
n_step = 50
n_feat = 2
n_emb = 30

def AttentionLayer(d, e):
    dot_product = Dot(axes=(2, 2))([d, e])
    score = Activation('softmax')(dot_product)
    value = Dot(axes=(2, 1))([score, e])
    return Concatenate()([value, d])

# Time series embedding layer.
EmbedInput = Dense(n_emb, activation='tanh')

# Trained Encoder
i_enc = Input(batch_shape=(None, n_step, n_feat))
e_enc = EmbedInput(i_enc)
o_enc, h_enc = GRU(n_hidden, return_sequences=True,
                   return_state = True)(i_enc)
```



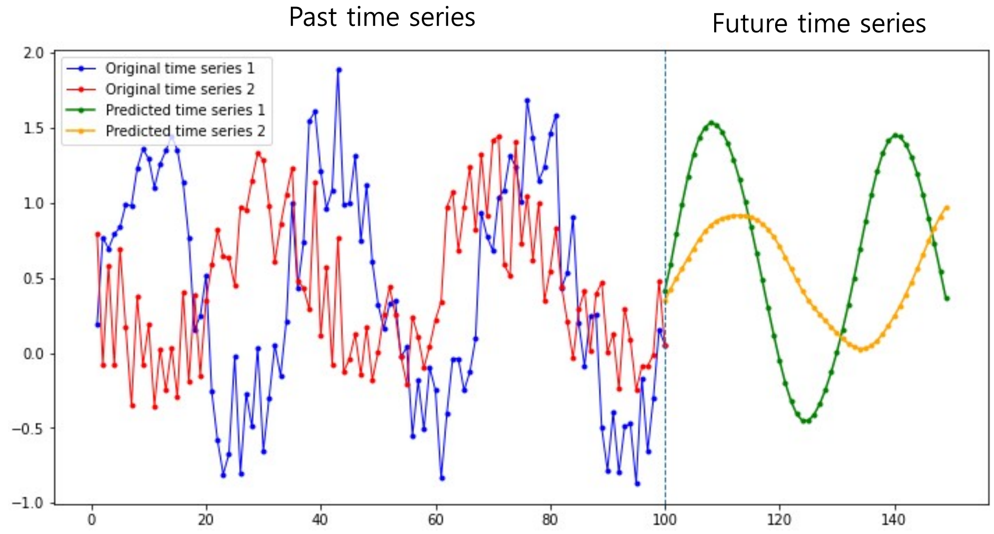

```
# Trained Decoder
TrainedGRU = GRU(n_hidden, return_sequences=True,
                 return_state=True)
TrainedFFN = TimeDistributed(Dense(n_feat))

i_dec = Input(batch_shape=(None, 1, n_feat))
e_dec = EmbedInput(i_dec)
o_dec, _ = TrainedGRU(i_dec, initial_state = h_enc)
a_dec = AttentionLayer(o_dec, o_enc)
y_dec = TrainedFFN(a_dec)

model = Model([i_enc, i_dec], y_dec)
model.load_weights("models/attention1.h5")

# Encoder model for prediction
Encoder = Model(i_enc, [o_enc, h_enc])

# Decoder model for prediction
i_stat = Input(batch_shape = (None, n_hidden))
i_henc = Input(batch_shape = (None, n_step, n_hidden))
o_pre, h_pre = TrainedGRU(i_dec, initial_state = i_stat)
a_pre = AttentionLayer(o_pre, i_henc)
y_pre = TrainedFFN(a_pre)
Decoder = Model([i_dec, i_stat, i_henc], [y_pre, h_pre])
```

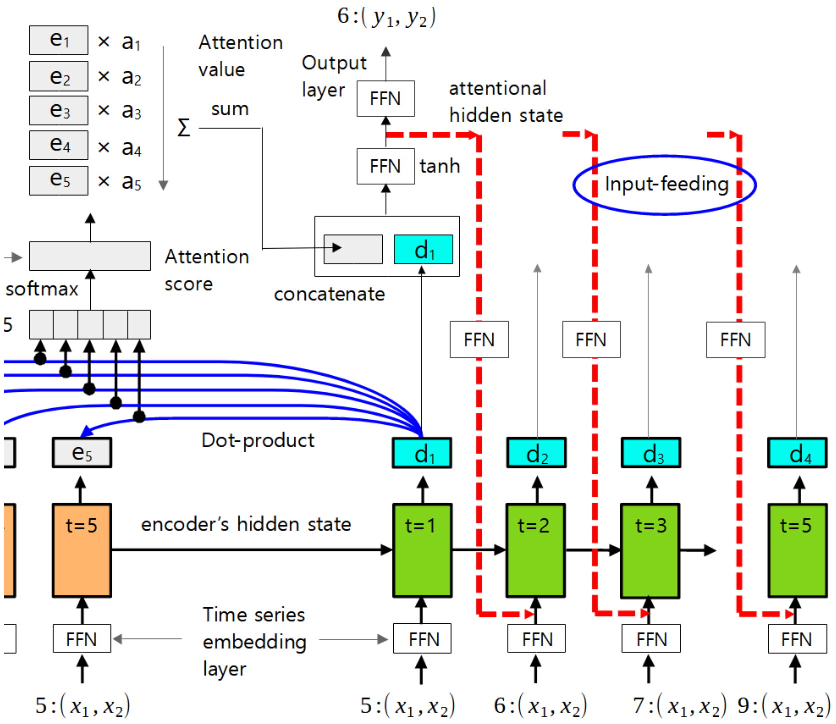


MX-AI

- Implementing a Seq2Seq-Attention model for time series prediction: Prediction stage

```python
# prediction
e_seed = data[-50:].reshape(-1, 50, 2)
d_seed = data[-1].reshape(-1, 1, 2)
oe, he = Encoder.predict(e_seed, verbose=0)

n_future = 50
y_pred = []
for i in range(n_future):
    yd, hd = Decoder.predict([d_seed, he, oe], verbose=0)
    y_pred.append(yd.reshape(2,))

    he = hd
    d_seed = yd
y_pred = np.array(y_pred)

# Plot the past time series and the predicted future time series.
y_past = data[-100:]
plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(y_past) + 1)
ax2 = np.arange(len(y_past), len(y_past) + len(y_pred))
plt.plot(ax1, y_past[:, 0], '-o', c='blue', markersize=3,
         label='Original time series 1', linewidth=1)
plt.plot(ax1, y_past[:, 1], '-o', c='red', markersize=3,
         label='Original time series 2', linewidth=1)
plt.plot(ax2, y_pred[:, 0], '-o', c='green', markersize=3,
         label='Predicted time series 1')
plt.plot(ax2, y_pred[:, 1], '-o', c='orange', markersize=3,
         label='Predicted time series 2')
plt.axvline(x=ax1[-1],  linestyle='dashed', linewidth=1)
plt.legend()
plt.show()
```





11. Attention Networks

Part 4: Seq2Seq-Attention model applying input-feeding method

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

www.youtube.com/@meanxai

MX-AI

■ Seq2Seq-Attention model

Effective Approaches to Attention-based Neural Machine Translation

Minh-Thang Luong **Hieu Pham** **Christopher D. Manning**
Computer Science Department, Stanford University, Stanford, CA 94305
{lmthang,hyhieu,manning}@stanford.edu

3.3 Input-feeding Approach

In our proposed global and local approaches, the attentional decisions are made independently, which is suboptimal. Whereas, in standard MT, a coverage set is often maintained during the translation process to keep track of which source words have been translated. Likewise, in attentional NMTs, alignment decisions should be made jointly taking into account past alignment information. To address that, we propose an input feeding approach in which attentional vectors ht are concatenated with inputs at the next time steps as illustrated in Figure 4. The effects of having such connections are two-fold: (a) we hope to make the model fully aware of previous alignment choices and (b) we create a very deep network spanning both horizontally and vertically.
Comparison to other work Bahdanau et al. (2015) use context vectors, similar to our ct, in building subsequent hidden states, which can also achieve the "coverage" effect. …

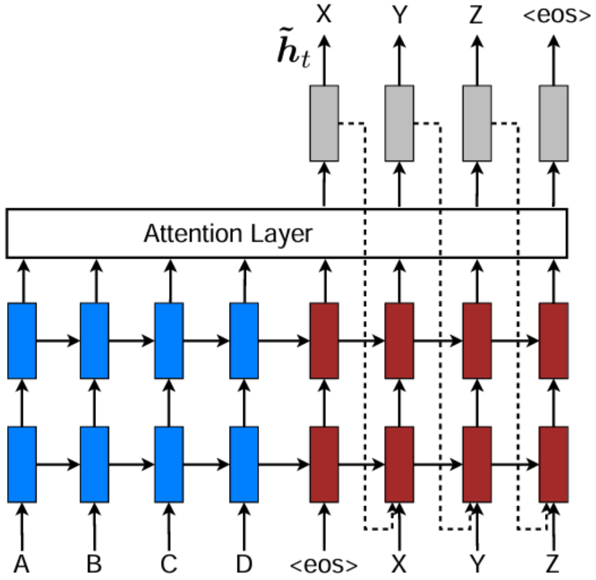


Figure 4: **Input-feeding approach** – Attentional vectors $\tilde{h}_t$ are fed as inputs to the next time steps to inform the model about past alignment decisions.

MX-AI

Seq2Seq-Attention model applying input-feeding method

Encoder inputs

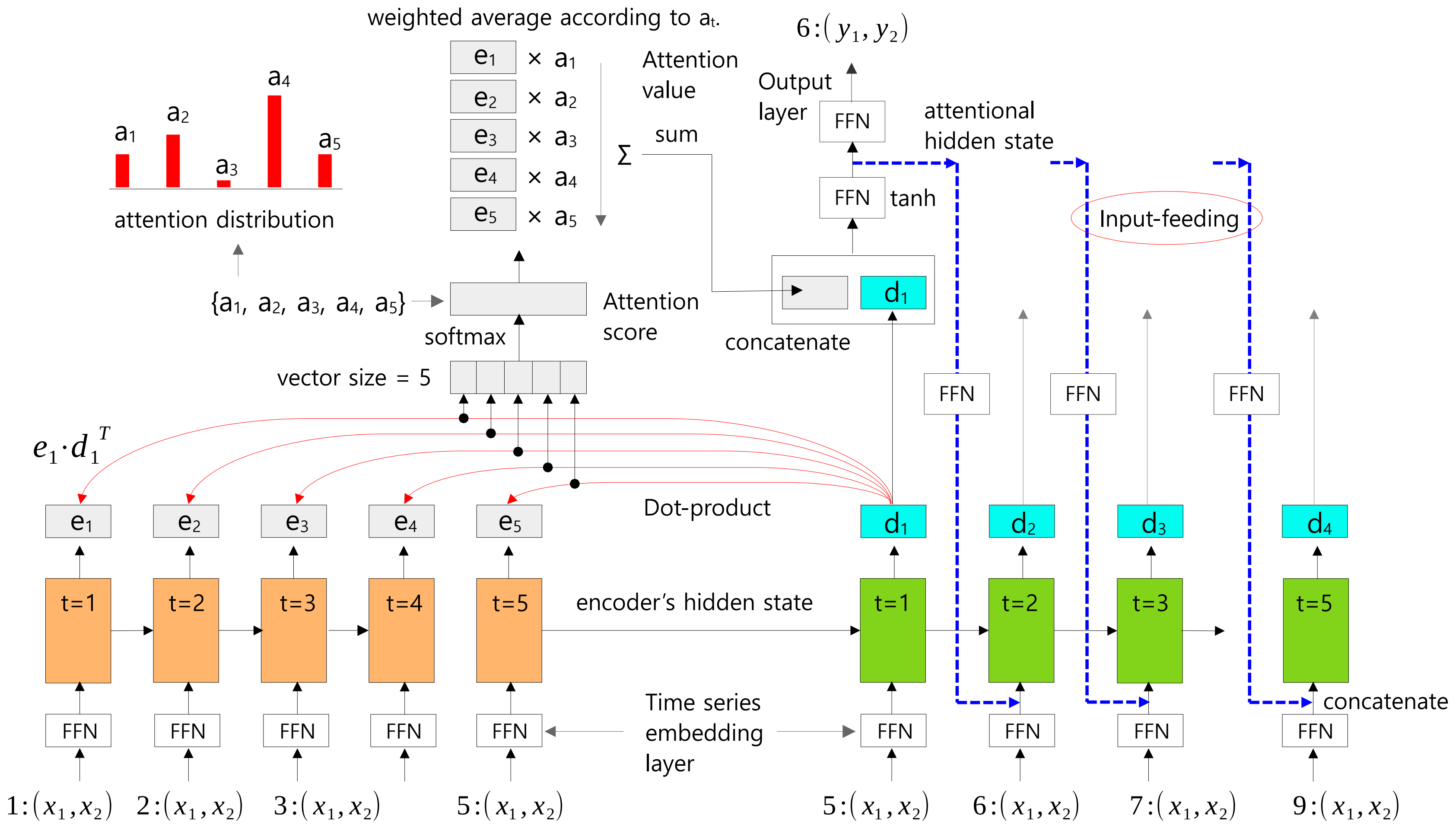
t	x_1	x_2
1	0.63,	0.73
2	0.78,	0.98
3	1.33,	0.73
4	0.93,	0.81
5	1.12,	0.74

Decoder inputs

t	x_1	x_2
5	1.12,	0.74
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
9	1.00,	1.05

Targets

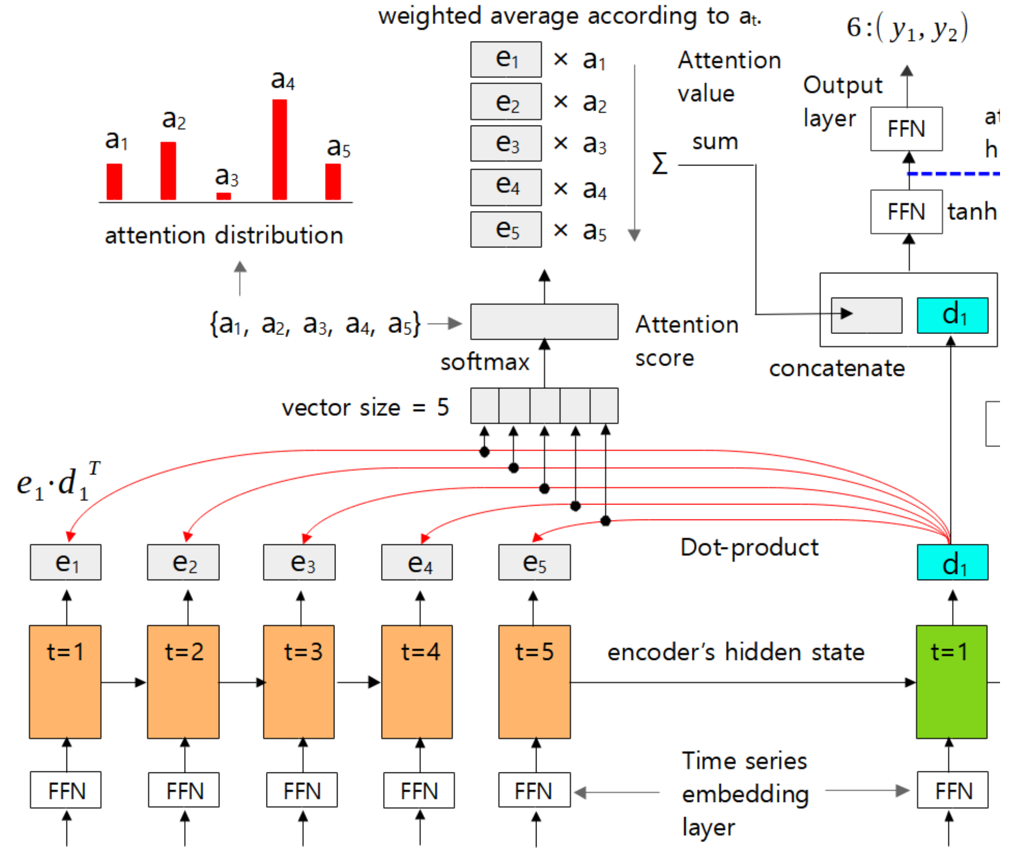
t	y_1	y_2
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
9	1. ,	1.05
10	1.74,	1.37



MX-AI

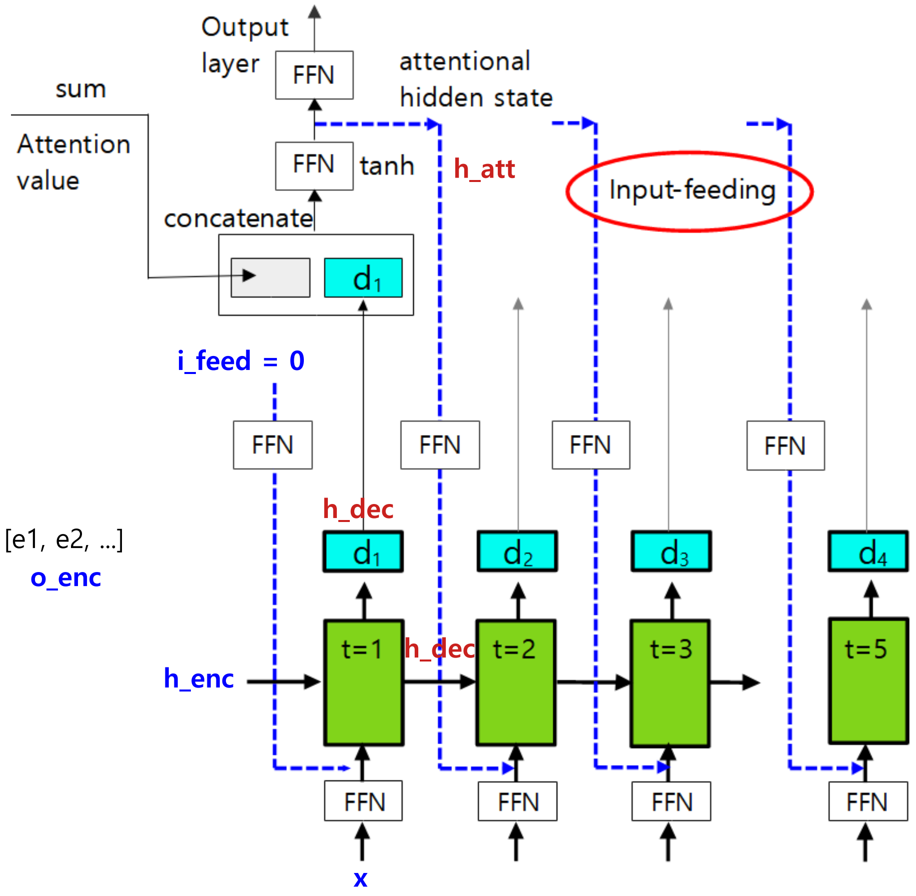
■ Implementing a Seq2Seq-Attention model applying input-feeding method: Build an attention class

```python
# [MXDL-11-04] Attention.py
# Attention Networks for time series prediction
# Minh-Thang Luong et, al., 2015, Effective Approaches
# to Attention-based Neural Machine Translation
# 3.3 Input-feeding approach: Attentional vectors are concatenated
# with inputs at the next steps.
import tensorflow as tf
from tensorflow.keras.layers import Dense, GRU, Dot, Activation
from tensorflow.keras.layers import Concatenate, Reshape

class AttentionLayer:
    def __init__(self, n_hidden):
        self.attentionFFN = Dense(n_hidden, activation='tanh')

    def __call__(self, d, e):
        dot_product = Dot(axes=(2, 2))([d, e])
        score = Activation('softmax')(dot_product)
        value = Dot(axes=(2, 1))([score, e])
        concat = Concatenate()([value, d])
        h_attention = self.attentionFFN(concat)
        return h_attention    # attentional hidden state

class Encoder:
    def __init__(self, n_hidden):
        self.n_hidden = n_hidden
        self.encoderGRU = GRU(n_hidden,
                              return_sequences=True,
                              return_state = True)

    def __call__(self, x):
        return self.encoderGRU(x)
```



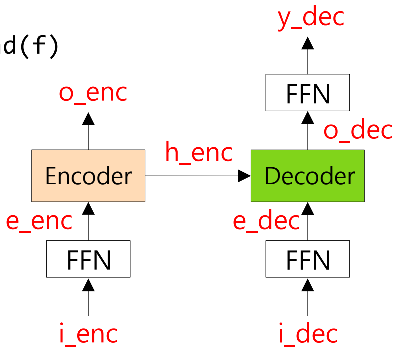
■ Implementing a Seq2Seq-Attention model applying input-feeding method: Build an attention class

```
class Decoder:
    def __init__(self, n_hidden, n_feed):
        self.n_hidden = n_hidden
        self.n_feed = n_feed
        self.decoderGRU = GRU(n_hidden)
        self.inputFeedingFFN = Dense(n_feed, activation='tanh')
        self.attention = AttentionLayer(n_hidden)

    def __call__(self, x, o_enc, h_enc):
        outputs = []   # outputs of decoder (many-to-many)
        i_feed = tf.zeros(shape=(tf.shape(x)[0], self.n_hidden))
        for t in range(x.shape[1]):
            i_cat = self.inputFeedingFFN(i_feed)
            i_cat = Concatenate()([i_cat, x[:, t, :]])
            i_cat = Reshape([1, -1])(i_cat)
            h_dec = self.decoderGRU(i_cat, initial_state = h_enc)

            # Find attentional hidden state
            h_att = self.attention(Reshape((1, -1))(h_dec), o_enc)

            # Update encoder's hidden state and the input-feeding
            # vector for the next step
            h_enc = h_dec
            i_feed = Reshape((-1,))(h_att)

            # Collect outputs at all time steps.
            outputs.append(Reshape((self.n_hidden,))(h_att))

        outputs = tf.convert_to_tensor(outputs)
        outputs = tf.transpose(outputs, perm=[1, 0, 2])
        return outputs
```

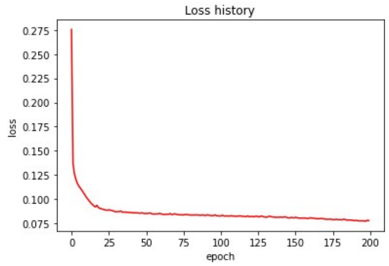


■ Implementing a Seq2Seq-Attention model applying input-feeding method: Teacher forcing

```
# [MXDL-11-04] 6.Attention(train).py (Luong's version)
from tensorflow.keras.layers import Input, Dense, TimeDistributed
from Attention import Encoder, Decoder
from tensorflow.keras.models import Model
from tensorflow.keras import optimizers
import matplotlib.pyplot as plt
import numpy as np
import pickle

# Read saved dataset
with open('dataset.pkl', 'rb') as f:
      _, xi_enc, xi_dec, xp_dec = pickle.load(f)

n_hidden = 100
n_emb = 30
n_feed = 30
n_step = xi_enc.shape[1]
n_feat = xi_enc.shape[2]

# Time series embedding layer.
EmbedInput = Dense(n_emb, activation='tanh')

# Encoder
i_enc = Input(batch_shape=(None, n_step, n_feat))
e_enc = EmbedInput(i_enc)
o_enc, h_enc = Encoder(n_hidden)(e_enc)
```



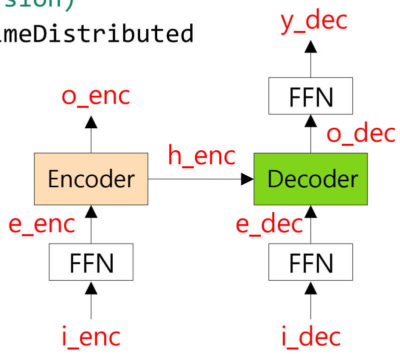

```
# Decoder
i_dec = Input(batch_shape=(None, n_step, n_feat))
e_dec = EmbedInput(i_dec)
o_dec = Decoder(n_hidden, n_feed)(e_dec, o_enc, h_enc)
y_dec = TimeDistributed(Dense(n_feat))(o_dec)

model = Model([i_enc, i_dec], y_dec)
model.compile(loss='mse',
              optimizer=optimizers.Adam(learning_rate=0.001))

# Training: teacher forcing
hist = model.fit([xi_enc, xi_dec], xp_dec,
                 batch_size=500, epochs=200)

# Save the trained model
model.save_weights("models/attention2.h5")

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

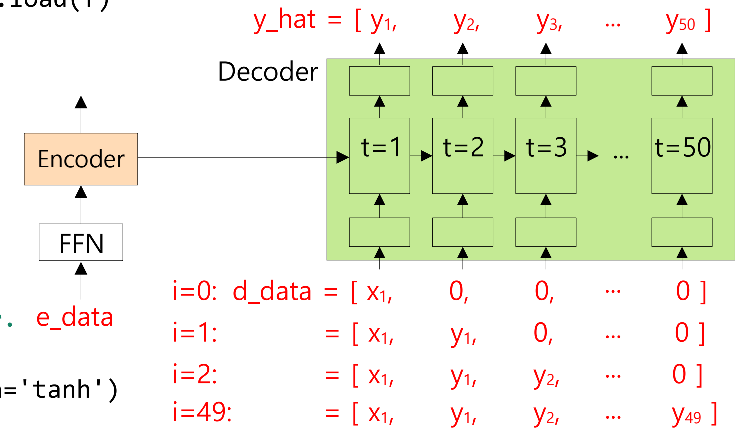


MX-AI

■ Implementing a Seq2Seq-Attention model applying input-feeding method: Prediction stage

```python
# [MXDL-11-04] 7.Attention2(predict).py (Luong's version)
from tensorflow.keras.layers import Input, Dense, TimeDistributed
from Attention import Encoder, Decoder
from tensorflow.keras.models import Model
from tensorflow.keras import optimizers
import matplotlib.pyplot as plt
import numpy as np
import pickle

# Read saved dataset
with open('dataset.pkl', 'rb') as f:
      data, _, _, _ = pickle.load(f)

n_hidden = 100
n_emb = 30
n_feed = 30
n_step = 50
n_feat = data.shape[1]

# Time series embedding layer.
EmbedInput = Dense(n_emb,
                   activation='tanh')
# Trained Encoder
i_enc = Input(batch_shape=(None, n_step, n_feat))
e_enc = EmbedInput(i_enc)
o_enc, h_enc = Encoder(n_hidden)(e_enc)
```

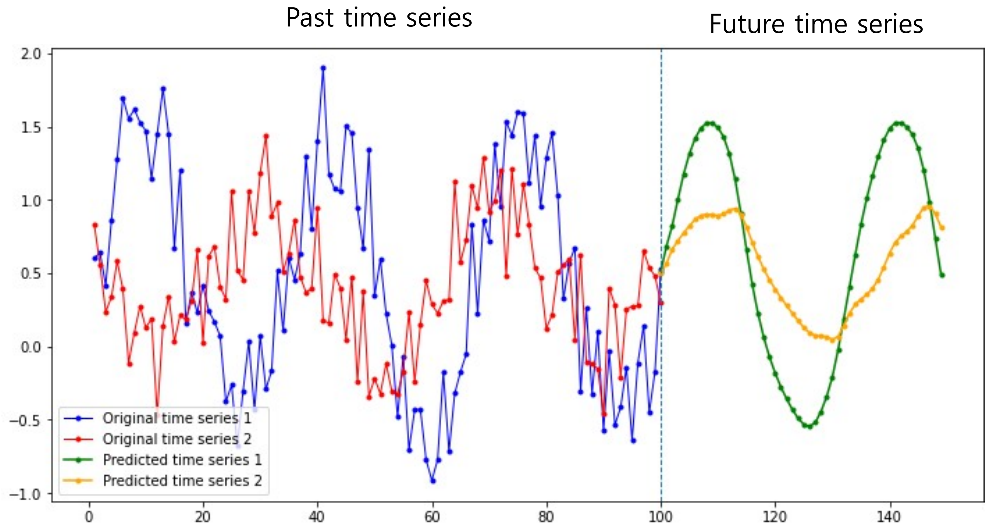


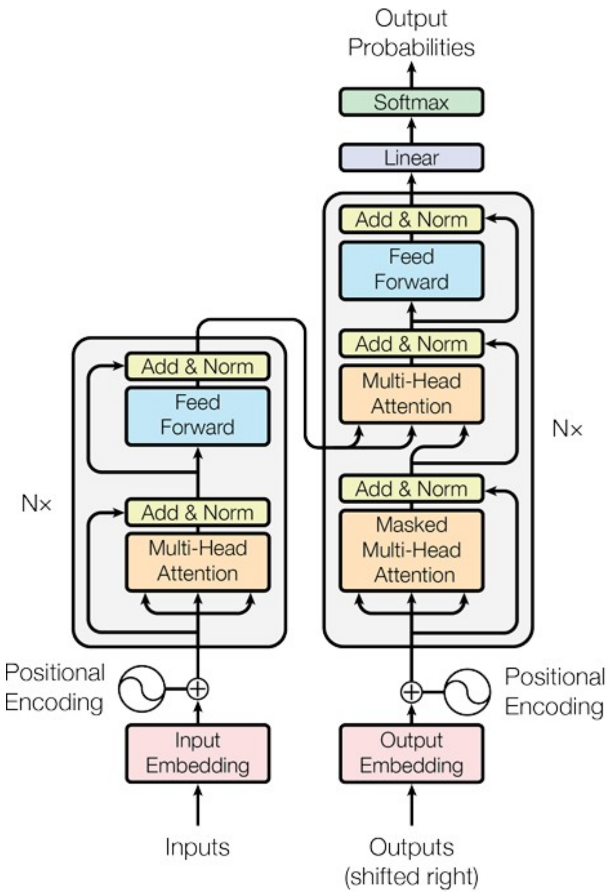



```python
# Trained Decoder
i_dec = Input(batch_shape=(None, n_step, n_feat))
e_dec = EmbedInput(i_dec)
o_dec = Decoder(n_hidden, n_feed)(e_dec, o_enc, h_enc)
y_dec = TimeDistributed(Dense(n_feat))(o_dec)

model = Model([i_enc, i_dec], y_dec)
model.load_weights("models/attention2.h5")

# prediction
n_future = 50
e_data = data[-50:].reshape(-1, 50, 2)
d_data = np.zeros(shape=(1, 50, 2))
d_data[0, 0, :] = data[-1]

for i in range(n_future):
    y_hat = model.predict([e_data, d_data], verbose=0)
    y_hat = y_hat[0, :, :]  # remove the first
                            # dimension

    if i < n_future - 1:
        d_data[0, i+1, :] = y_hat[i, :]

    print(i+1, ':', y_hat[i, :])
```

- Implementing a Seq2Seq-Attention model applying input-feeding method: Prediction stage

```
# Plot the past time series and the predicted future time series.
y_past = data[-100:]
plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(y_past) + 1)
ax2 = np.arange(len(y_past), len(y_past) + len(y_hat))
plt.plot(ax1, y_past[:, 0], '-o', c='blue', markersize=3,
         label='Original time series 1', linewidth=1)
plt.plot(ax1, y_past[:, 1], '-o', c='red', markersize=3,
         label='Original time series 2', linewidth=1)
plt.plot(ax2, y_hat[:, 0], '-o', c='green', markersize=3,
         label='Predicted time series 1')
plt.plot(ax2, y_hat[:, 1], '-o', c='orange', markersize=3,
         label='Predicted time series 2')
plt.axvline(x=ax1[-1],  linestyle='dashed', linewidth=1)
plt.legend()
plt.show()
```





11. Attention Networks

Part 5: Transformer model

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

www.youtube.com/@meanxai

MX-AI

■ Transformer model

Attention Is All You Need

Ashish Vaswani∗
Google Brain
avaswani@google.com

Llion Jones∗
Google Research
llion@google.com

NoamShazeer∗
Google Brain
noam@google.com

Niki Parmar∗
Google Research
nikip@google.com

Aidan N. Gomez∗ †
University of Toronto
aidan@cs.toronto.edu

Jakob Uszkoreit∗
Google Research
usz@google.com

Łukasz Kaiser∗
Google Brain
lukaszkaiser@google.com

Illia Polosukhin∗ ‡
illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 Englishto-German translation task, improving over the existing best results, including ensembles, by over 2BLEU. On the WMT2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

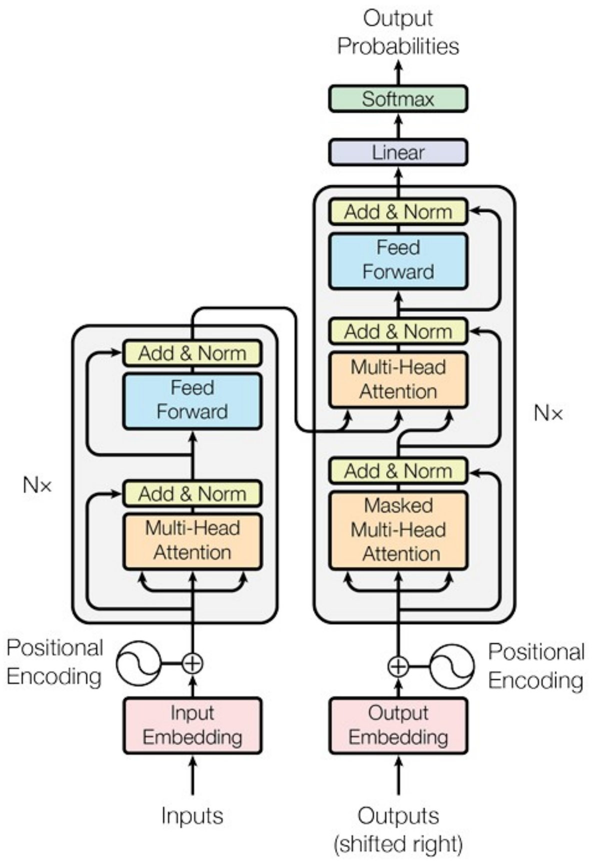


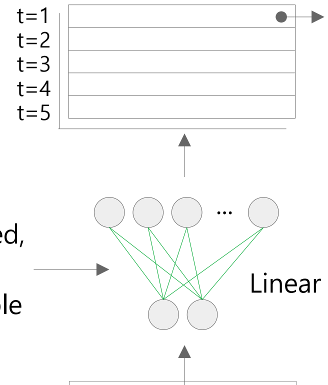
Figure 1: The Transformer - model architecture.

MX-AI

- Feeding time series datasets to a Transformer model during training for teacher forcing

  - Input Embedding for time series

Embedding vectors



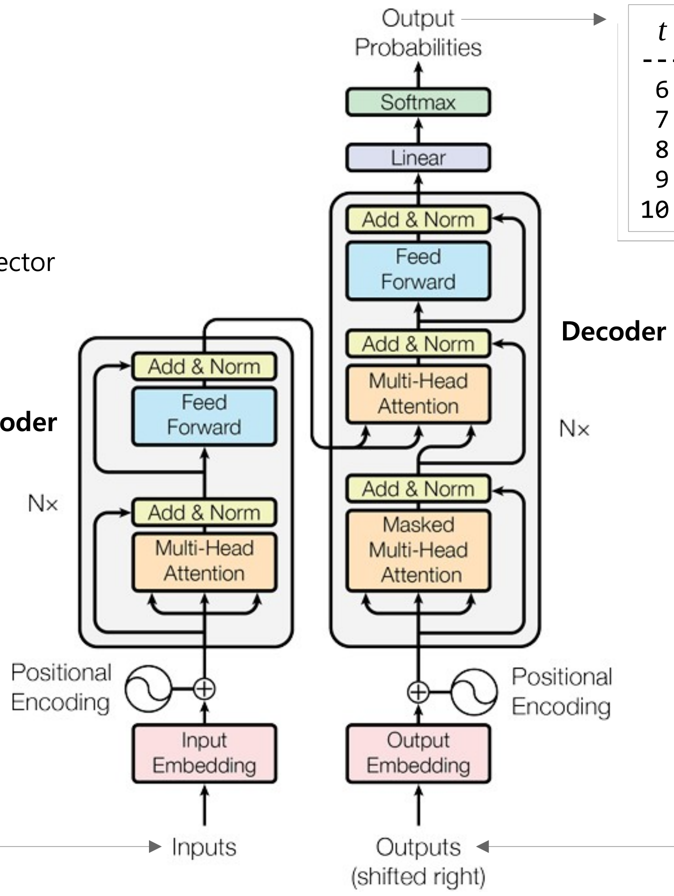
embedding vector of [0.63, 0.73]

Linear feedforward

In natural language processing, word embedding layers are used, but since we are dealing with time series data, we use a simple linear feedforward layer as the embedding layer.

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

Encoder-inputs



Targets

t	y_1	y_2
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
9	1.00,	1.05
10	1.74,	1.37

t	x_1	x_2
5	1.12,	0.74
6	1.09,	1.18
7	1.25,	0.57
8	1.93,	0.61
9	1.00,	1.05

Decoder-inputs

■ Positional Encoding

3.5 Positional Encoding

Since our model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence. To this end, we add "positional encodings" to the input embeddings at the bottoms of the encoder and decoder stacks. The positional encodings have the same dimension d_{model} as the embeddings, so that the two can be summed. There are many choices of positional encodings, learned and fixed [9].

In this work, we use sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = \sin(pos / 10000^{2i / d\_model})$$

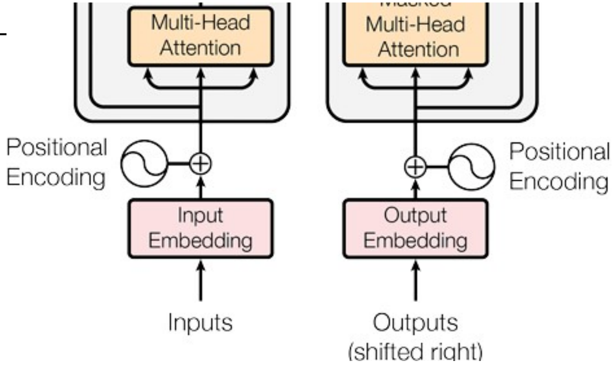
$$PE_{(pos,2i+1)} = \cos(pos / 10000^{2i / d\_model})$$

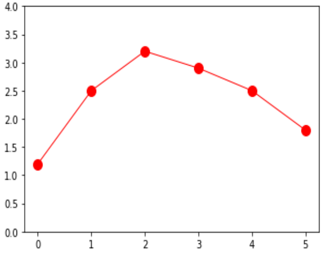
where pos is the position and i is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$. We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.

We also experimented with using learned positional embeddings [9] instead, and found that the two versions produced nearly identical results (see Table 3 row (E)). We chose the sinusoidal version because it may allow the model to extrapolate to sequence lengths longer than the ones encountered during training.

Time series

t	x1
1	1.2
2	2.5
3	3.2
4	2.9
5	2.5
6	1.8





Seq2Seq: 1.2 → 2.5 → 3.2 → 2.9 → 2.5 → 1.8

Transformer:

1.2
2.5
3.2
2.9
2.5
1.8

→ Unlike Seq2Seq, Transformers process all the data points in parallel, which significantly speeds up training.

$d_{model} = 8$

Time series

t	x1
1	1.2
2	2.5
3	3.2
4	2.9
5	2.5
6	1.8

→

Time series embedding vectors

1	2	3	4	5	6	7	8

+

Positional Encoding (d_{model} = 8)

1	2	3	4	5	6	7	8
0	0	0	0	1	1	1	1
0.84	0.1	0.01	0	0.54	1	1	1
0.91	0.2	0.02	0	-0.42	0.98	1	1
0.14	0.3	0.03	0	-0.99	0.96	1	1
-0.76	0.39	0.04	0	-0.65	0.92	1	1
-0.96	0.48	0.05	0	0.28	0.88	1	1

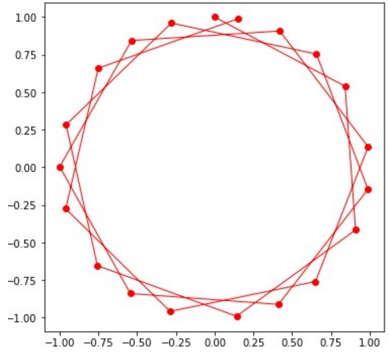
MX-AI

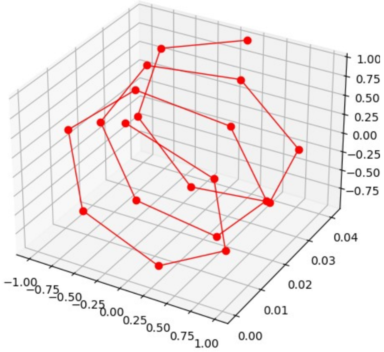
Time series embedding and Positional Encoding

```
# [MXDL-11-05] 9.positional_encoding.py
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics.pairwise import euclidean_distances

# reference: https://github.com/suyash/transformer
def positional_encoding(position, d_model):
    position_dims = np.arange(position)[:, np.newaxis]
    embed_dims = np.arange(d_model)[np.newaxis, :]
    angle_rates = 1 / np.power(10000.0, \
                            (2 * (embed_dims//2))/d_model)
    angle_rads = position_dims * angle_rates

    sines = np.sin(angle_rads[:, 0::2])
    cosines = np.cos(angle_rads[:, 1::2])
    return np.concatenate([sines, cosines], axis=-1)

PE = positional_encoding(6, 8)
print(np.round(PE, 3), '\n')

for i in range(PE.shape[0] - 1):
    d = euclidean_distances(PE[i].reshape(1,-1), PE[i+1].reshape(1,-1))
    norm = np.linalg.norm(PE[i])
    dot = np.dot(PE[i], PE[i+1])
    print("%d - %d : distance = %.4f, norm = %.4f, dot = %.4f" % (i,
i+1, d[0,0], norm, dot))

PE = positional_encoding(20, 2)
plt.figure(figsize=(6,6))
plt.plot(PE[:,0], PE[:,1], marker='o', linewidth=1.0, color='red')
plt.show()
```

```
PE = positional_encoding(20, 3)
fig = plt.figure(figsize=(6,6), dpi=100)
ax = fig.add_subplot(1,1,1, projection='3d')
ax.plot(PE[:,0], PE[:,1], PE[:, 2],
        marker='o', linewidth=1.0, color='red')
plt.show()
```

```
[[ 0.     0.     0.     0.     1.     1.     1.     1.   ]
 [ 0.841  0.1    0.01   0.001  0.54   0.995  1.     1.   ]
 [ 0.909  0.199  0.02   0.002 -0.416  0.98   1.     1.   ]
 [ 0.141  0.296  0.03   0.003 -0.99   0.955  1.     1.   ]
 [-0.757  0.389  0.04   0.004 -0.654  0.921  0.999  1.   ]
 [-0.959  0.479  0.05   0.005  0.284  0.878  0.999  1.   ]]

0 - 1 : distance = 0.9641, norm = 2.0000, dot = 3.5353
1 - 2 : distance = 0.9641, norm = 2.0000, dot = 3.5353
2 - 3 : distance = 0.9641, norm = 2.0000, dot = 3.5353
3 - 4 : distance = 0.9641, norm = 2.0000, dot = 3.5353
4 - 5 : distance = 0.9641, norm = 2.0000, dot = 3.5353
```





[MXDL-11-05] Transformer

■ Multi-Head Attention

$d_{model} = 6$

$d_K = \frac{6}{2} = 3$

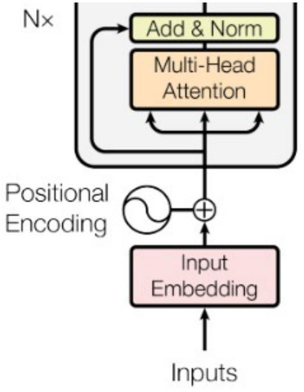
num_layers = 1

num_heads = 2

x

	1	2	3	4	5	6
t=1	0.489	0.585	0.797	1.881	1.509	1.483
t=2	1.580	0.095	0.411	0.791	1.036	1.593
t=3	1.554	1.012	0.609	-0.105	1.426	1.872
t=4	0.437	0.874	0.612	-0.509	1.443	1.176
t=5	-0.161	1.062	0.424	0.139	1.039	1.895

Input embedding + Positional encoding



[www.youtube.com/@meanxai]

head = 0

W_0^Q

0.855	0.273
0.779	0.521
0.041	0.775
0.968	0.973
0.667	0.009
0.801	0.632

$Q = x \cdot W_0^Q$

4.922	3.836
4.175	2.583
4.493	2.516
2.492	1.309
3.054	2.180

W_0^K

0.516	0.098
0.925	0.063
0.175	0.842
0.412	0.375
0.562	0.909
0.686	0.025

$K = x \cdot W_0^K$

3.574	2.871
2.976	1.785
3.888	2.033
2.548	1.763
2.914	1.452

W_0^V

0.221	0.682
0.660	0.554
0.542	0.307
0.807	0.466
0.154	0.910
0.850	0.380

$V = x \cdot W_0^V$

3.938	3.714
2.786	3.171
3.068	3.767
1.816	2.492
2.778	2.337

$Q \cdot K^T$

	t=1	t=2	t=3	t=4	t=5
t=1	28.604	21.495	26.933	19.306	19.913
t=2	22.338	17.036	21.482	15.194	15.917
t=3	23.284	17.864	22.583	15.887	16.748
t=4	12.666	9.754	12.350	8.659	9.164
t=5	17.172	12.979	16.303	11.625	12.064

$Q \cdot K^T / \sqrt{d_k}$

	t=1	t=2	t=3	t=4	t=5
t=1	16.515	12.410	15.550	11.147	11.497
t=2	12.897	9.836	12.403	8.772	9.190
t=3	13.443	10.314	13.038	9.173	9.669
t=4	7.313	5.631	7.130	4.999	5.291
t=5	9.914	7.493	9.412	6.712	6.965

	t=1	t=2	t=3	t=4	t=5
t=1	0.710	0.012	0.270	0.003	0.005
t=2	0.589	0.028	0.359	0.010	0.014
t=3	0.572	0.025	0.382	0.008	0.013
t=4	0.444	0.083	0.370	0.044	0.059
t=5	0.560	0.050	0.339	0.023	0.029

$softmax\left(Q \cdot K^T / \sqrt{d_k}\right)$

3.676	3.711
3.556	3.686
3.545	3.693
3.359	3.554
3.504	3.637

$softmax(.) \cdot V$

attention value

head = 1

W_1^Q

0.214	0.364
0.055	0.998
0.255	0.601
0.922	0.095
0.938	0.151
0.778	0.059

$Q = x \cdot W_1^Q$

4.644	1.734
3.389	1.242
3.241	2.257
2.096	1.637
2.709	1.537

W_1^K

0.946	0.289
0.065	0.398
0.862	0.498
0.584	0.353
0.073	0.902
0.306	0.239

$K = x \cdot W_1^K$

2.850	3.153
2.880	2.294
2.677	2.854
1.165	2.182
1.019	2.027

W_1^V

0.129	0.069
0.980	0.556
0.248	0.615
0.967	0.997
0.731	0.268
0.829	0.650

$V = x \cdot W_1^V$

4.985	4.094
3.241	2.516
3.837	2.539
2.602	1.536
3.590	2.489

$Q \cdot K^T$

	t=1	t=2	t=3	t=4	t=5
t=1	18.703	17.354	17.380	9.195	8.245
t=2	13.574	12.610	12.616	6.659	5.969
t=3	16.355	14.515	15.119	8.703	7.877
t=4	11.135	9.793	10.282	6.014	5.453
t=5	12.566	11.328	11.638	6.511	5.875

$Q \cdot K^T / \sqrt{d_k}$

	t=1	t=2	t=3	t=4	t=5
t=1	10.798	10.019	10.035	5.309	4.760
t=2	7.837	7.280	7.284	3.845	3.446
t=3	9.443	8.380	8.729	5.025	4.548
t=4	6.429	5.654	5.937	3.472	3.148
t=5	7.255	6.540	6.719	3.759	3.392

	t=1	t=2	t=3	t=4	t=5
t=1	0.518	0.238	0.241	0.002	0.001
t=2	0.459	0.263	0.264	0.008	0.006
t=3	0.539	0.186	0.264	0.007	0.004
t=4	0.463	0.213	0.283	0.024	0.017
t=5	0.470	0.230	0.275	0.014	0.010

$softmax\left(Q \cdot K^T / \sqrt{d_k}\right)$

attention value

4.287	3.336
4.195	3.238
4.336	3.366
4.207	3.228
4.220	3.250

$softmax(.) \cdot V$

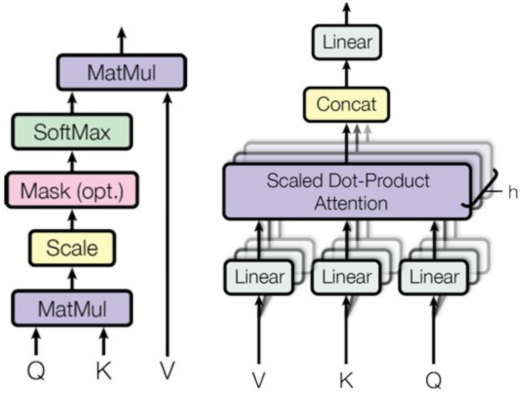


Fig 2: Scaled Dot-Product Attention, Multi-Head Attention

Multi-head attention value

	1	2	3	4	5	6
t=1	7.979	10.691	7.796	10.756	8.408	8.120
t=2	7.788	10.486	7.600	10.510	8.247	7.957
t=3	7.926	10.728	7.768	10.668	8.367	8.134
t=4	7.597	10.339	7.467	10.234	8.035	7.827
t=5	7.748	10.465	7.587	10.449	8.192	7.934

W_M

0.612	0.099	0.591	0.937	0.429	0.156
0.499	0.971	0.156	0.703	0.948	0.732
0.288	0.849	0.797	0.658	0.329	0.479
0.792	0.924	0.488	0.564	0.570	0.832

product

concatenate

3.676	3.711	4.287	3.336
3.556	3.686	4.195	3.238
3.545	3.693	4.336	3.366
3.359	3.554	4.207	3.228
3.504	3.637	4.220	3.250

concatenated attention value

MX-AI

Outputs of Encoder

x

	1	2	3	4	5	6
t=1	0.489	0.585	0.797	1.881	1.509	1.483
t=2	1.580	0.095	0.411	0.791	1.036	1.593
t=3	1.554	1.012	0.609	-0.105	1.426	1.872
t=4	0.437	0.874	0.612	-0.509	1.443	1.176
t=5	-0.161	1.062	0.424	0.139	1.039	1.895

Input embedding + Positional encoding

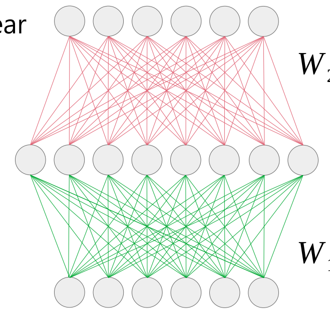
+

Multi-head attention value

	1	2	3	4	5	6
t=1	7.979	10.691	7.796	10.756	8.408	8.120
t=2	7.788	10.486	7.600	10.510	8.247	7.957
t=3	7.926	10.728	7.768	10.668	8.367	8.134
t=4	7.597	10.339	7.467	10.234	8.035	7.827
t=5	7.748	10.465	7.587	10.449	8.192	7.934

=

	1	2	3	4	5	6
t=1	8.47	11.28	8.59	12.64	9.92	9.60
t=2	9.37	10.58	8.01	11.30	9.28	9.55
t=3	9.48	11.74	8.38	10.56	9.79	10.01
t=4	8.03	11.21	8.08	9.73	9.48	9.00
t=5	7.59	11.53	8.01	10.59	9.23	9.83



Position-wise Feed Forward Networks

r

	1	2	3	4	5	6
t=1	-1.093	0.811	-1.012	1.732	-0.111	-0.328
t=2	-0.299	0.863	-1.606	1.555	-0.386	-0.126
t=3	-0.502	1.706	-1.576	0.554	-0.199	0.016
t=4	-1.13	1.803	-1.084	0.438	0.208	-0.235
t=5	-1.362	1.502	-1.056	0.819	-0.17	0.267

keras.layers.LayerNormalization

W_1

0.99	0.59	0.18	0.50	0.19	0.56	0.45	0.37
0.17	0.82	0.95	0.69	0.33	0.33	0.80	0.31
0.39	0.44	0.85	0.92	0.21	0.87	0.56	0.12
0.56	0.01	0.33	0.88	0.19	0.99	0.41	0.14
0.05	0.40	0.24	0.39	0.51	0.27	0.94	0.33
0.13	0.14	0.28	0.77	0.89	0.15	0.22	0.01

b_1

0.44	0.69	0.63	0.4	0.28	0.04	0.51	0.9

$ReLu(r \cdot W_1 + b_1)$

0.023	0.192	0.797	0.710	0.108	0.451	0.634	0.828
0.499	0.358	0.416	0.489	0.157	0.176	0.413	0.953
0.000	1.027	0.960	0.298	0.435	0.000	0.810	1.066
0.000	1.079	1.347	0.367	0.413	0.000	1.160	1.038
0.000	0.631	1.218	0.644	0.602	0.000	0.742	0.796

W_2

0.288	0.098	0.755	0.024	0.389	0.603
0.271	0.073	0.848	0.794	0.229	0.704
0.588	0.268	0.000	0.296	0.455	0.807
0.450	0.960	0.988	0.442	0.964	0.229
0.312	0.898	0.692	0.868	0.663	0.768
0.319	0.760	0.316	0.756	0.581	0.418
0.009	0.496	0.330	0.153	0.017	0.335
0.524	0.644	0.106	0.362	0.398	0.240

b_2

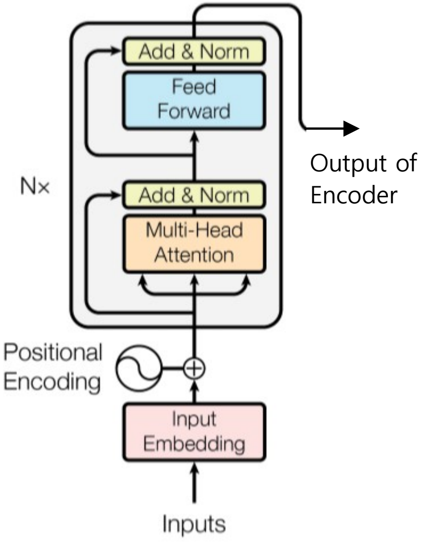
0.88	0.02	0.01	0.86	0.31	0.77

$ReLu(r \cdot W_1 + b_1) \cdot W_2 + b2$

	1	2	3	4	5	6
t=1	2.345	2.220	1.405	2.391	2.081	2.405
t=2	2.195	1.771	1.575	2.172	1.838	2.332
t=3	2.560	2.118	1.856	2.976	1.993	3.197
t=4	2.814	2.428	2.066	3.187	2.227	3.655
t=5	2.670	2.433	1.926	2.927	2.355	3.245

+

	1	2	3	4	5	6
t=1	1.25	3.03	0.39	4.12	1.97	2.08
t=2	1.90	2.63	-0.03	3.73	1.45	2.21
t=3	2.06	3.82	0.28	3.53	1.79	3.21
t=4	1.68	4.23	0.98	3.62	2.44	3.42
t=5	1.31	3.94	0.87	3.75	2.18	3.51



Outputs of Encoder (z)

	1	2	3	4	5	6
t=1	-1.093	0.811	-1.012	1.732	-0.111	-0.328
t=2	-0.299	0.863	-1.606	1.555	-0.386	-0.126
t=3	-0.502	1.706	-1.576	0.554	-0.199	0.016
t=4	-1.13	1.803	-1.084	0.438	0.208	-0.235
t=5	-1.362	1.502	-1.056	0.819	-0.17	0.267

keras.layers.LayerNormalization

MX-AI

Masked Multi-Head Attention and Outputs of Decoder

	t1	t2	t3	t4	t5	t6
Seq2Seq:	1.2 →	2.5 →	3.2 →	2.9 →	2.5 →	1.8

Transformer:

1.2
2.5
3.2
2.9
2.5
1.8

Decoder Input embedding + Positional encoding

x

	1	2	3	4	5	6
t=1	0.489	0.585	0.797	1.881	1.509	1.483
t=2	1.580	0.095	0.411	0.791	1.036	1.593
t=3	1.554	1.012	0.609	-0.105	1.426	1.872
t=4	0.437	0.874	0.612	-0.509	1.443	1.176
t=5	-0.161	1.062	0.424	0.139	1.039	1.895

head = 0

W_0^Q

0.855	0.273
0.779	0.521
0.041	0.775
0.968	0.973
0.667	0.009
0.801	0.632

$Q = x \cdot W_0^Q$

4.922	3.836
4.175	2.583
4.493	2.516
2.492	1.309
3.054	2.180

W_0^K

0.516	0.098
0.925	0.063
0.175	0.842
0.412	0.375
0.562	0.909
0.686	0.025

$K = x \cdot W_0^K$

3.574	2.871
2.976	1.785
3.888	2.033
2.548	1.763
2.914	1.452

W_0^V

0.221	0.682
0.660	0.554
0.542	0.307
0.807	0.466
0.154	0.910
0.850	0.380

$V = x \cdot W_0^V$

3.938	3.714
2.786	3.171
3.068	3.767
1.816	2.492
2.778	2.337

head = 1

$Q \cdot K^T$

	t=1	t=2	t=3	t=4	t=5
t=1	28.604	21.495	26.933	19.306	19.913
t=2	22.338	17.036	21.482	15.194	15.917
t=3	23.284	17.864	22.583	15.887	16.748
t=4	12.666	9.754	12.350	8.659	9.164
t=5	17.172	12.979	16.303	11.625	12.064

$Q \cdot K^T / \sqrt{d_k}$

	t=1	t=2	t=3	t=4	t=5
t=1	16.515	12.410	15.550	11.147	11.497
t=2	12.897	9.836	12.403	8.772	9.190
t=3	13.443	10.314	13.038	9.173	9.669
t=4	7.313	5.631	7.130	4.999	5.291
t=5	9.914	7.493	9.412	6.712	6.965

$+$ Look-ahead mask

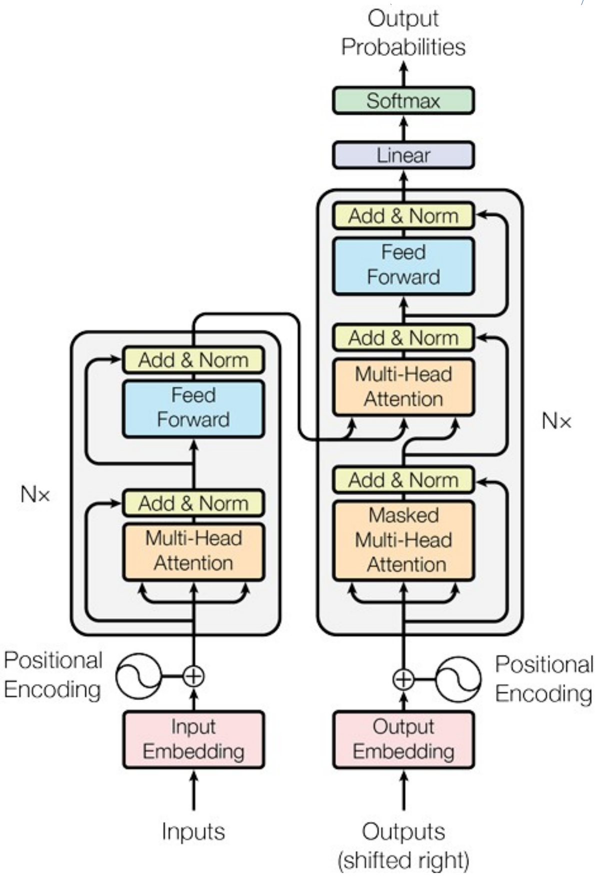
t=1	0	1	1	1	1
t=2	0	0	1	1	1
t=3	0	0	0	1	1
t=4	0	0	0	0	1
t=5	0	0	0	0	0

$\times (-\infty)$

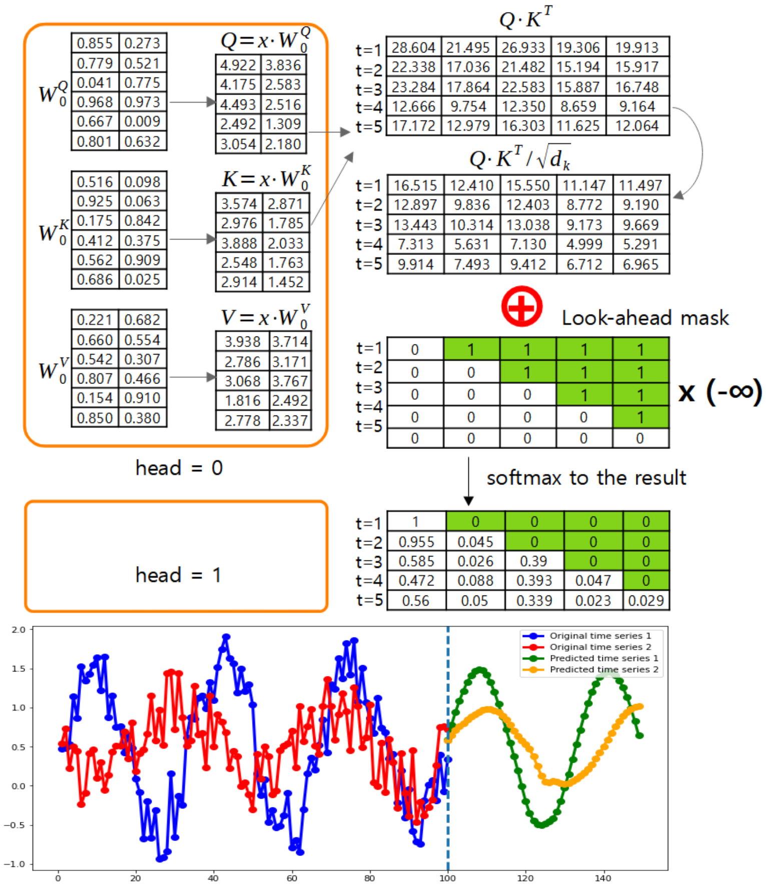
softmax to the result

	t=1	t=2	t=3	t=4	t=5
t=1	1	0	0	0	0
t=2	0.955	0.045	0	0	0
t=3	0.585	0.026	0.39	0	0
t=4	0.472	0.088	0.393	0.047	0
t=5	0.56	0.05	0.339	0.023	0.029

The process of obtaining the multi-head attention value using this score is identical to that of the encoder.



MX-AI



11. Attention Networks

Part 6: Time series forecasting using Transformer

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

www.youtube.com/@meanxai

MX-AI

■ Transformer code written in Keras

```
# Transformer code: https://github.com/suyash/transformer
# https://github.com/suyash/transformer/blob/master/transformer
      /transformer.py
# Since the Transformer code is for natural language
# processing, it needs some modifications to be used for time
# series forecasting.
class Decoder:
    def __init__(self,
                 num_layers,
                 d_model,
                 num_heads,
                 d_ff,
                 # vocab_size,
                 dropout_rate,
                 ffn_activation=tf.keras.activations.relu,
                 scope="decoder"):
        self.d_model = d_model
        self.num_layers = num_layers
        self.scope = scope

        # self.embedding = Embedding(input_dim=vocab_size,
        #                            output_dim=d_model,
        #                            name="%s/embedding" % scope)
        self.pos_encoding = PositionalEncoding(d_model,
                            name="%s/positional_encoding" % scope)

        self.dec_layers = [
            DecoderLayer(d_model=d_model,
                         num_heads=num_heads,
                         d_ff=d_ff,
                         dropout_rate=dropout_rate,
                         ffn_activation=ffn_activation,
                      scope="%s/decoder_layer_%d" % (scope, i))
            for i in range(num_layers)
        ]
```

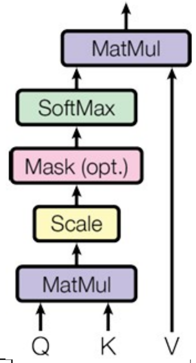
```
        self.dropout = Dropout(dropout_rate,
                               name="%s/dropout" % self.scope)

    def __call__(self, x, enc_output, lookahead_mask, padding_mask):
        # x = self.embedding(x)
        # x = MultiplyConstant(self.d_model,
        #                      name="%s/multiply" % self.scope)(x)
        x = MultiplyConstant(
                  tf.math.sqrt(tf.cast(self.d_model, tf.float32)),
                  name="%s/multiply" % self.scope)(x)
        x = Add(name="%s/add" % self.scope)([x, self.pos_encoding(x)])

        dec_attention_weights = {}
        enc_dec_attention_weights = {}

        for i in range(self.num_layers):
            x, dec_attention, enc_dec_attention = self.dec_layers[i](
                x, enc_output, lookahead_mask, padding_mask)

            dec_attention_weights["layer_%d" % i] = dec_attention
            enc_dec_attention_weights["layer_%d" % i] = enc_dec_attention

        return x, dec_attention_weights, enc_dec_attention_weights
```

MX-AI

■ Transformer code written in Keras

```
class Encoder:
    def __init__(self,
                 num_layers,
                 d_model,
                 num_heads,
                 d_ff,
                 # vocab_size,
                 dropout_rate,
                 ffn_activation=tf.keras.activations.relu,
                 scope="encoder"):
        self.d_model = d_model
        self.num_layers = num_layers
        self.scope = scope

        # self.embedding = Embedding(input_dim=vocab_size,
        #                            output_dim=d_model,
        #                            name="%s/embedding" % scope)
        self.pos_encoding = PositionalEncoding(d_model,
                                 name="%s/positional_encoding" % scope)

        self.enc_layers = [
            EncoderLayer(d_model=d_model,
                         num_heads=num_heads,
                         d_ff=d_ff,
                         dropout_rate=dropout_rate,
                         ffn_activation=ffn_activation,
                      scope="%s/encoder_layer_%d" % (scope, i))
            for i in range(num_layers)
        ]
```

```
        self.dropout = Dropout(dropout_rate,
                               name="%s/dropout" % self.scope)

    def __call__(self, x, padding_mask):
        # x = self.embedding(x)
        # x = MultiplyConstant(self.d_model,
        #                      name="%s/multiply" % self.scope)(x)
        x = MultiplyConstant(
                 tf.math.sqrt(tf.cast(self.d_model, tf.float32)),
                 name="%s/multiply" % self.scope)(x)
        x = Add(name="%s/add" % self.scope)([x, self.pos_encoding(x)])

        enc_attention_weights = {}

        for i in range(self.num_layers):
            x, enc_attention = self.enc_layers[i](x, padding_mask)
            enc_attention_weights["layer_%d" % i] = enc_attention

        return x, enc_attention_weights
```



- Transformer code written in Keras

```
class PaddingMask(Layer):
    def __call__(self, inputs):
        # seq = tf.cast(tf.math.equal(inputs, 0), tf.float32)
        # return seq[:, tf.newaxis, tf.newaxis, :]

        n_batch = tf.shape(inputs)[0]  # batch size (None)
        n_tstep = tf.shape(inputs)[1]  # time steps
        seq = tf.zeros([n_batch, n_tstep])
        return seq[:, tf.newaxis, tf.newaxis, :]  # (None, 1, 1, n)
```

Padding mask

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

```
class PaddingAndLookaheadMask(Layer):
    def __call__(self, inputs):
        # size = tf.shape(inputs)[1]
        # lhm = 1 - tf.linalg.band_part(tf.ones((size, size)), -1, 0)

        # seq = tf.cast(tf.math.equal(inputs, 0), tf.float32)
        # seq = seq[:, tf.newaxis, tf.newaxis, :]
        # return tf.maximum(lhm, seq)

        n_batch = tf.shape(inputs)[0]  # batch size (None)
        n_tstep = tf.shape(inputs)[1]  # time steps
        lhm = 1-tf.linalg.band_part(tf.ones((n_tstep, n_tstep)), -1, 0)
        lhm = lhm[tf.newaxis, tf.newaxis, :, :]       # (1, 1, n, n)
        lhm = tf.repeat(lhm, repeats=n_batch, axis=0) # (None, 1, n, n)

        return lhm
```

* For the natural language processing:

```
inputs = [ I, am, AI ] → [ 29, 15, 8, 0, 0 ]
padding_mask =           [  0,  0, 0, 1, 1 ]
```

W_0^Q

0.855	0.273
0.779	0.521
0.041	0.775
0.968	0.973
0.667	0.009
0.801	0.632

$Q = x \cdot W_0^Q$

4.922	3.836
4.175	2.583
4.493	2.516
2.492	1.309
3.054	2.180

W_0^K

0.516	0.098
0.925	0.063
0.175	0.842
0.412	0.375
0.562	0.909
0.686	0.025

$K = x \cdot W_0^K$

3.574	2.871
2.976	1.785
3.888	2.033
2.548	1.763
2.914	1.452

W_0^V

0.221	0.682
0.660	0.554
0.542	0.307
0.807	0.466
0.154	0.910
0.850	0.380

$V = x \cdot W_0^V$

3.938	3.714
2.786	3.171
3.068	3.767
1.816	2.492
2.778	2.337

head = 0

head = 1

$Q \cdot K^T$

	I	am	AI	PAD	PAD
I	28.604	21.495	26.933	19.306	19.913
am	22.338	17.036	21.482	15.194	15.917
AI	23.284	17.864	22.583	15.887	16.748
PAD	12.666	9.754	12.350	8.659	9.164
PAD	17.172	12.979	16.303	11.625	12.064

$Q \cdot K^T / \sqrt{d_k}$

	I	am	AI	PAD	PAD
I	16.515	12.410	15.550	11.147	11.497
am	12.897	9.836	12.403	8.772	9.190
AI	13.443	10.314	13.038	9.173	9.669
PAD	7.313	5.631	7.130	4.999	5.291
PAD	9.914	7.493	9.412	6.712	6.965

$+$ Padding mask

I	0	0	0	1	1
am	0	0	0	1	1
AI	0	0	0	1	1
PAD	0	0	0	1	1
PAD	0	0	0	1	1

$\times (-\infty)$

softmax to the result

	I	am	AI	PAD	PAD
I	0.716	0.012	0.273	0	0
am	0.603	0.028	0.368	0	0
AI	0.585	0.026	0.39	0	0
PAD	0.495	0.092	0.413	0	0
PAD	0.59	0.052	0.357	0	0

$\bullet$

$V = x \cdot W_0^V$

3.938	3.714
2.786	3.171
3.068	3.767
1.816	2.492
2.778	2.337

MX-AI

■ Transformer code written in Keras

```
class PaddingMask(Layer):
    def __call__(self, inputs):
        # seq = tf.cast(tf.math.equal(inputs, 0), tf.float32)

        n_batch = tf.shape(inputs)[0]  # batch size (None)
        n_tstep = tf.shape(inputs)[1]  # time steps
        seq = tf.zeros([n_batch, n_tstep])
        return seq[:, tf.newaxis, tf.newaxis, :]  # (None, 1, 1, n)

class PaddingAndLookaheadMask(Layer):
    def __call__(self, inputs):
        # size = tf.shape(inputs)[1]
        # lhm = 1 - tf.linalg.band_part(tf.ones((size, size)), -1, 0)

        # seq = tf.cast(tf.math.equal(inputs, 0), tf.float32)
        # seq = seq[:, tf.newaxis, tf.newaxis, :]
        # return tf.maximum(lhm, seq)

        n_batch = tf.shape(inputs)[0]  # batch size (None)
        n_tstep = tf.shape(inputs)[1]  # time steps
        lhm = 1-tf.linalg.band_part(tf.ones((n_tstep, n_tstep)), -1, 0)
        lhm = lhm[tf.newaxis, tf.newaxis, :, :]       # (1, 1, n, n)
        lhm = tf.repeat(lhm, repeats=n_batch, axis=0) # (None, 1, n, n)

        return lhm
```

Look-ahead mask

0	1	1	1	1
0	0	1	1	1
0	0	0	1	1
0	0	0	0	1
0	0	0	0	0

x

W_0^Q

0.855	0.273
0.779	0.521
0.041	0.775
0.968	0.973
0.667	0.009
0.801	0.632

$Q = x \cdot W_0^Q$

4.922	3.836
4.175	2.583
4.493	2.516
2.492	1.309
3.054	2.180

W_0^K

0.516	0.098
0.925	0.063
0.175	0.842
0.412	0.375
0.562	0.909
0.686	0.025

$K = x \cdot W_0^K$

3.574	2.871
2.976	1.785
3.888	2.033
2.548	1.763
2.914	1.452

W_0^V

0.221	0.682
0.660	0.554
0.542	0.307
0.807	0.466
0.154	0.910
0.850	0.380

$V = x \cdot W_0^V$

3.938	3.714
2.786	3.171
3.068	3.767
1.816	2.492
2.778	2.337

head = 0

head = 1

$Q \cdot K^T$

	t=1	t=2	t=3	t=4	t=5
t=1	28.604	21.495	26.933	19.306	19.913
t=2	22.338	17.036	21.482	15.194	15.917
t=3	23.284	17.864	22.583	15.887	16.748
t=4	12.666	9.754	12.350	8.659	9.164
t=5	17.172	12.979	16.303	11.625	12.064

$Q \cdot K^T / \sqrt{d_k}$

	t=1	t=2	t=3	t=4	t=5
t=1	16.515	12.410	15.550	11.147	11.497
t=2	12.897	9.836	12.403	8.772	9.190
t=3	13.443	10.314	13.038	9.173	9.669
t=4	7.313	5.631	7.130	4.999	5.291
t=5	9.914	7.493	9.412	6.712	6.965

$+$ Look-ahead mask

t=1	0	1	1	1	1
t=2	0	0	1	1	1
t=3	0	0	0	1	1
t=4	0	0	0	0	1
t=5	0	0	0	0	0

$\times (-\infty)$

softmax to the result

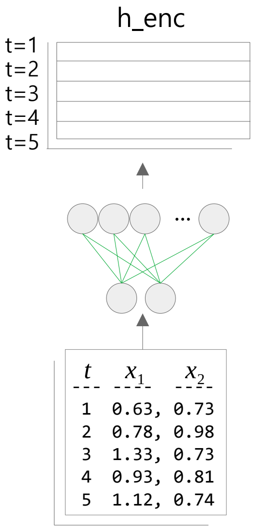
	t=1	t=2	t=3	t=4	t=5
t=1	1	0	0	0	0
t=2	0.955	0.045	0	0	0
t=3	0.585	0.026	0.39	0	0
t=4	0.472	0.088	0.393	0.047	0
t=5	0.56	0.05	0.339	0.023	0.029

[Masked Multi-head attention]

MX-AI

■ Time series forecasting using Transformer: Training stage (Teacher forcing)

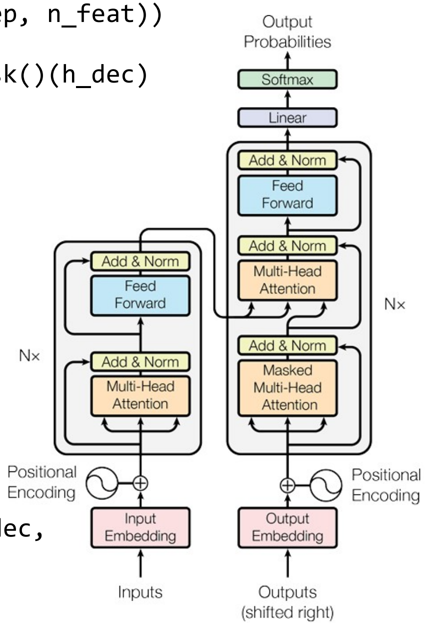
```python
# [MXDL-11-06] 10.transformer(train).py
# Transformer code: https://github.com/suyash/transformer
# Since the Transformer code is for natural language processing, it
# needs some modifications to be used for time series forecasting.
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
import matplotlib.pyplot as plt
import pickle
from transformer import Encoder, Decoder
from transformer import PaddingMask,\
                        PaddingAndLookaheadMask

# Read dataset
with open('dataset.pkl', 'rb') as f:
      _, xi_enc, xi_dec, xp_dec = pickle.load(f)

n_tstep = xi_enc.shape[1]  # 50
n_feat = xi_enc.shape[2]   # 2
d_model = 100

# Encoder
EmbDense = Dense(d_model, use_bias=False)
i_enc = Input(batch_shape=(None, n_tstep, n_feat))
h_enc = EmbDense(i_enc)
padding_mask = PaddingMask()(h_enc)
encoder = Encoder(num_layers = 1,
                  d_model = d_model,
                  num_heads = 5,
                  d_ff = 64,
                  dropout_rate=0.5)
o_enc, _ = encoder(h_enc, padding_mask)
```

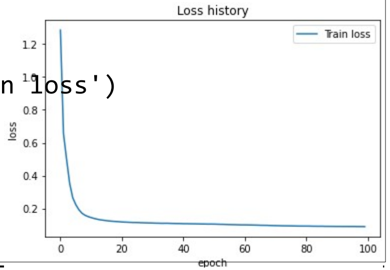


Padding mask

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

```python
# Decoder
i_dec = Input(batch_shape=(None, n_tstep, n_feat))
h_dec = EmbDense(i_dec)
lookahead_mask = PaddingAndLookaheadMask()(h_dec)
decoder = Decoder(num_layers = 1,
                  d_model = d_model,
                  num_heads = 5,
                  d_ff = 64,
                  dropout_rate=0.5)
o_dec, _, _ = decoder(h_dec, o_enc,
                      lookahead_mask,
                      padding_mask)
y_dec = Dense(n_feat)(o_dec)
model = Model([i_enc, i_dec], y_dec)
model.compile(loss='mse',
              optimizer='adam')

# Training: teacher forcing
hist = model.fit([xi_enc, xi_dec], xp_dec,
                 epochs=100,
                 batch_size = 200)

# Save the trained model
model.save("models/transformer.h5")

# Visually see the loss history
plt.plot(hist.history['loss'], label='Train loss')
plt.legend()
plt.title("Loss history")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()
```



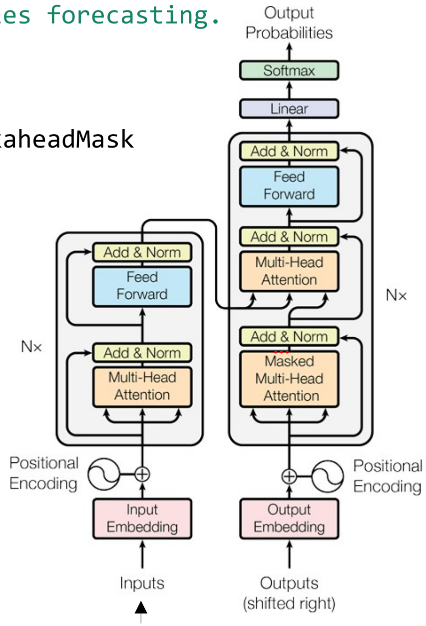


MX-AI

■ Time series forecasting using Transformer: Prediction stage

```
# [MXDL-11-06] 11.transformer(predict).py
# Transformer code: https://github.com/suyas
# Since the Transformer code is for natural
# needs some modifications to be used for time series forecasting.
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from transformer import Encoder, Decoder
from transformer import PaddingMask, PaddingAndLookaheadMask
import matplotlib.pyplot as plt
import numpy as np
import pickle

# Read dataset
with open('dataset.pkl', 'rb') as f:
      data, _, _, _ = pickle.load(f)

n_tstep = 50
n_feat = data.shape[1]   # 2
d_model = 100

# Trained Encoder
EmbDense = Dense(d_model, use_bias=False)
i_enc = Input(batch_shape=(None, n_tstep, n_feat))
h_enc = EmbDense(i_enc)
padding_mask = PaddingMask()(h_enc)
encoder = Encoder(num_layers = 1,
                  d_model = d_model,
                  num_heads = 5,
                  d_ff = 64,
                  dropout_rate=0.5)
o_enc, _ = encoder(h_enc, padding_mask)
```

y_hat = [y_1, y_2, y_3, ... y_{50}]



e_data

i=0:	d_data = [	x_1,	0,	0,	...	0]
i=1:	= [	x_1,	y_1,	0,	...	0]
i=2:	= [	x_1,	y_1,	y_2,	...	0]
i=49:	= [	x_1,	y_1,	y_2,	...	y_{49}]

```
# Trained Decoder
i_dec = Input(batch_shape=(None, None, n_feat))
h_dec = EmbDense(i_dec)
lookahead_mask = PaddingAndLookaheadMask()(h_dec)
decoder = Decoder(num_layers = 1,
                  d_model = d_model,
                  num_heads = 5,
                  d_ff = 64,
                  dropout_rate=0.5)
o_dec, _, _ = decoder(h_dec, o_enc, lookahead_mask,
                                    padding_mask)
y_dec = Dense(n_feat)(o_dec)

model = Model(inputs=[i_enc, i_dec], outputs=y_dec)
model.load_weights("models/transformer.h5")

# prediction
n_future = 50
e_data = data[-n_tstep:].reshape(-1, n_tstep, n_feat)
d_data = np.zeros(shape=(1, n_future, n_feat))
d_data[0, 0, :] = data[-1]

for i in range(n_future):
    y_hat = model.predict([e_data, d_data], verbose=0)

    if i < n_future - 1:
        d_data[0, i+1, :] = y_hat[0, i, :]

    print(i+1, ':', y_hat[0, i, :])
```

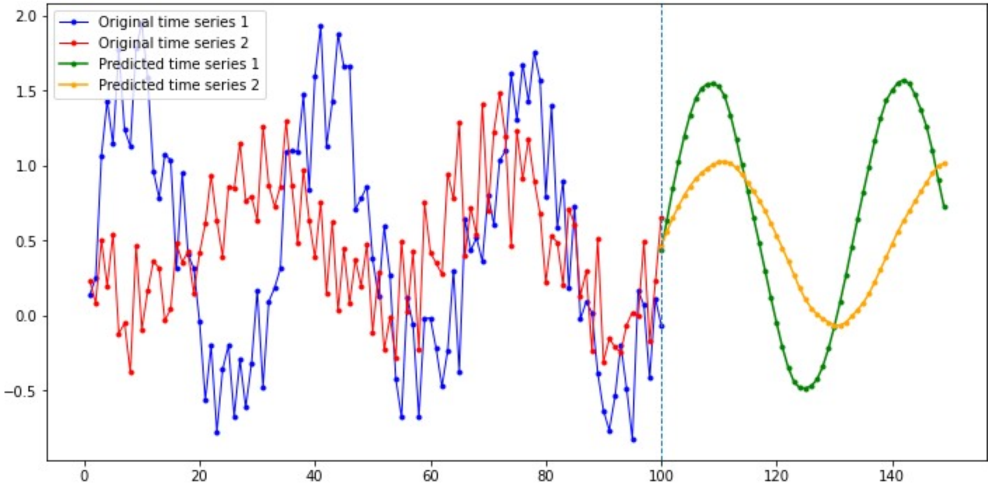
MX-AI

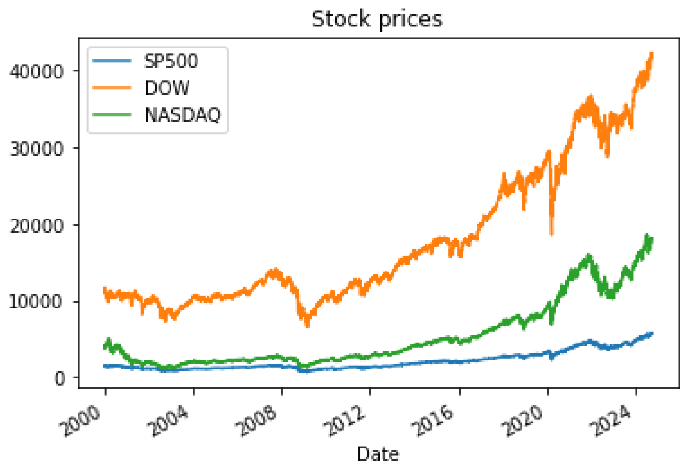
- Time series forecasting using Transformer: Prediction stage

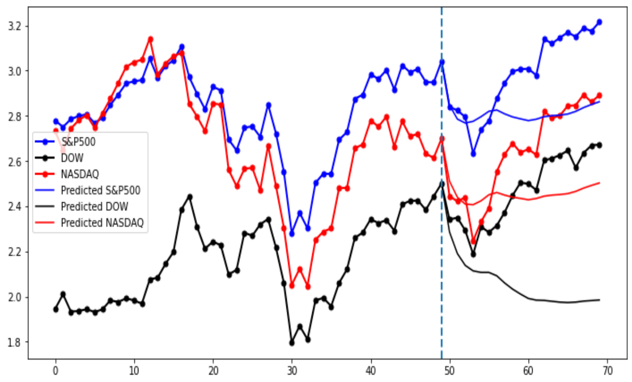
```
# Plot the past time series and the predicted future time series.
y_past = data[-100:]
y_hat = np.vstack([y_past[-1], y_hat[0,:,:]])

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

Past time series

Future time series







11. Attention Networks

Part 7: Stock price forecasting using a Transformer model

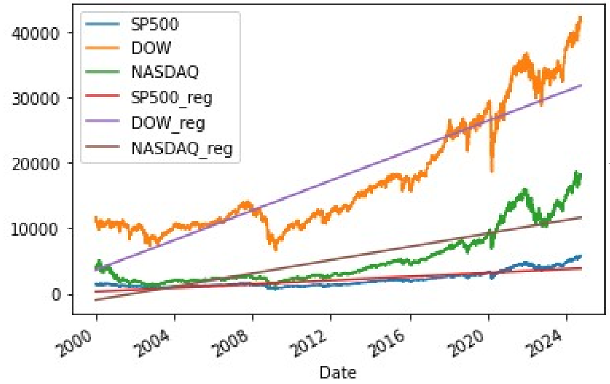
This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

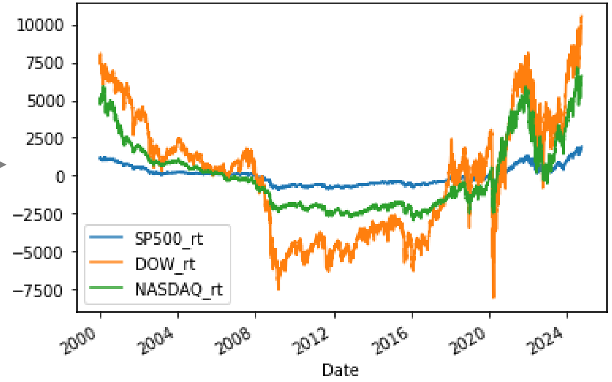
Stock price prediction

- We are going to try to predict stock prices using a Transformer model.
- Since stock prices are also time series, the Transformer model can be applied. However, stock prices are difficult to be predicted because they are characterized by non-stationary stochastic processes, random walks.
- It can be said that future stock prices are determined not only by past memories but also by future events, information shock, etc. Past memories can be technically analyzed, but future events cannot. Therefore, predicting future prices seems impossible. The transformer model can only learn from the past memories that the stock chart has.
- Nonetheless, to better understand how the Transformer works, let's apply it to stock price prediction.
- For a more stable prediction, let's predict the normalized stock price with the trend removed, rather than the raw stock price.

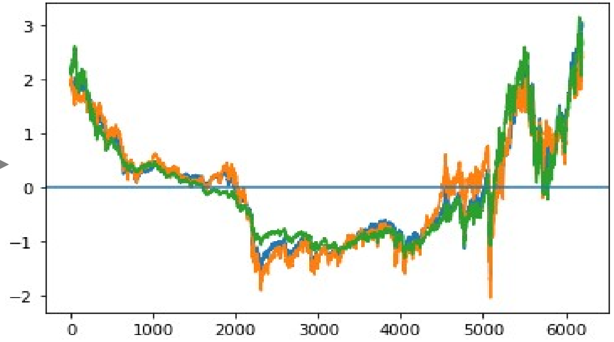
Stock price



Stock prices with trend removed



Normalized stock prices



* Please note that this experiment is not intended for stock investment but to familiarize you with the Transformer model.

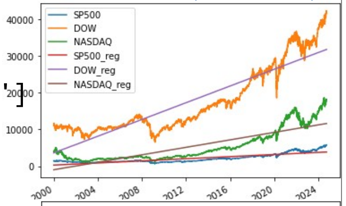
[MXDL-11-07] Deep Learning / Attention Networks: Transformer model

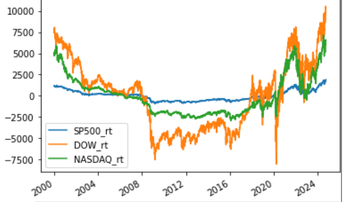
MX-AI

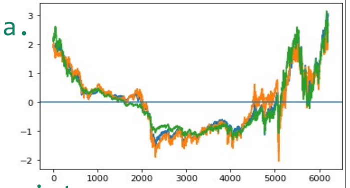
■ Data preprocessing

```python
# [MXDL-11-07] 12.stock_data.py
import numpy as np
import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
import pickle

start = "2000-01-01"
end = '2024-10-19'
sp500 = yf.download('^GSPC', start, end)  # S&P500
dow = yf.download('^DJI', start, end)     # DOW
nasdaq = yf.download('^IXIC', start, end) # NASDAQ

price = pd.DataFrame()
price['SP500'] = sp500['Adj Close']
price['DOW'] = dow['Adj Close']
price['NASDAQ'] = nasdaq['Adj Close']
price = price.dropna()

def regression(x, y):
    model = LinearRegression()
    model.fit(x, y)
    return model.predict(x)

# Finding regression line, trend of stock price
x = np.arange(price.shape[0],dtype='float').reshape(-1, 1)
price['SP500_reg'] = regression(x, price['SP500'])
price['DOW_reg'] = regression(x, price['DOW'])
price['NASDAQ_reg'] = regression(x, price['NASDAQ'])
price.plot()
plt.title("Stock prices")
plt.show()
```

```python
# Removing trend from stock price
price['SP500_rt']=price['SP500'] - price['SP500_reg']
price['DOW_rt']=price['DOW'] - price['DOW_reg']
price['NASDAQ_rt']=price['NASDAQ'] - price['NASDAQ_reg']
price = price[['SP500_rt', 'DOW_rt', 'NASDAQ_rt']]
price.plot()
plt.title("Stock prices with trend removed")
plt.show()

# Normalize stock prices
price = np.array(price)
mean = np.mean(price, axis=0)
std = np.std(price, axis=0)
price = (price - mean) / std

# Split the price data into training data and test data.
x_train = price[:-20]   # training data
x_test = price[-20:]    # test data

# Generate training dataset for a transformer model.
t = 60                  # the number of sequences
n = x_train.shape[0]    # the number of training data points
m = np.arange(0, n-2*t+1)
xi_enc=np.array([x_train[i:(i+t), :] for i in m])          # encoder input
xi_dec=np.array([x_train[(i+t-1):(i+2*t-1), :] for i in m]) # decoder input
xo_dec=np.array([x_train[(i+t):(i+2*t), :] for i in m])     # decoder output

# Save the training and test data for later use
with open('stock_data.pkl', 'wb') as f:
      pickle.dump([x_train, x_test, xi_enc, xi_dec, xo_dec], f)

plt.plot(x_train)
plt.title("Normalized stock prices")
plt.axhline(0)
plt.show()
```





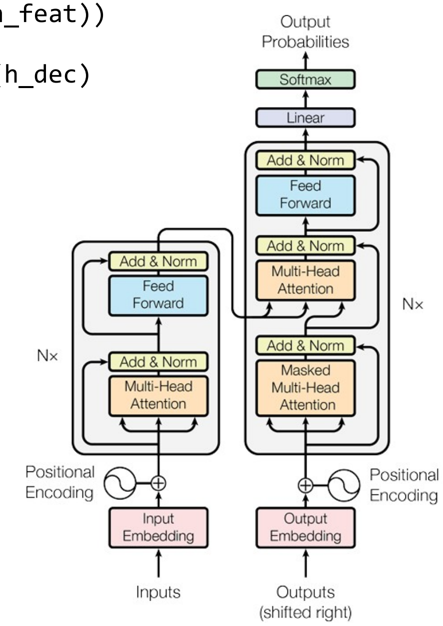


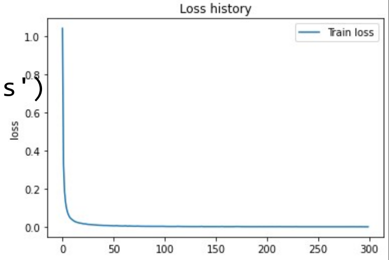
MX-AI

■ Stock price forecasting using Transformer: Training stage (Teacher forcing)

```
# [MXDL-11-07] 13.transformer(stock_train).py
# Transformer code: https://github.com/suyash/transformer
# Since the Transformer code is for natural language
# processing, it needs some modifications to be used for time
# series forecasting.
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
import matplotlib.pyplot as plt
import pickle
from transformer import Encoder, Decoder
from transformer import PaddingMask,\
                        PaddingAndLookaheadMask

# Read dataset
with open('dataset.pkl', 'rb') as f:
      _, _, xi_enc, xi_dec, xp_dec = pickle.load(f)

n_tstep = xi_enc.shape[1]  # 60
n_feat = xi_enc.shape[2]   # 3
d_model = 120

# Encoder
EmbDense = Dense(d_model, use_bias=False)
i_enc = Input(batch_shape=(None, n_tstep, n_feat))
h_enc = EmbDense(i_enc)
padding_mask = PaddingMask()(h_enc)
encoder = Encoder(num_layers = 2,
                  d_model = d_model,
                  num_heads = 4,
                  d_ff = 128,
                  dropout_rate=0.5)
o_enc, _ = encoder(h_enc, padding_mask)
```

```
# Decoder
i_dec = Input(batch_shape=(None, n_tstep, n_feat))
h_dec = EmbDense(i_dec)
lookahead_mask = PaddingAndLookaheadMask()(h_dec)
decoder = Decoder(num_layers = 2,
                  d_model = d_model,
                  num_heads = 4,
                  d_ff = 128,
                  dropout_rate=0.5)
o_dec, _, _ = decoder(h_dec, o_enc,
                      lookahead_mask,
                      padding_mask)
y_dec = Dense(n_feat)(o_dec)
model = Model([i_enc, i_dec], y_dec)
model.compile(loss='mse',
              optimizer='adam')

# Training: teacher forcing
hist = model.fit([xi_enc, xi_dec], xp_dec,
                 epochs=100,
                 batch_size = 200)

# Save the trained model
model.save("models/transformer_stock.h5")

# Visually see the loss history
plt.plot(hist.history['loss'], label='Train loss')
plt.legend()
plt.title("Loss history")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()
```



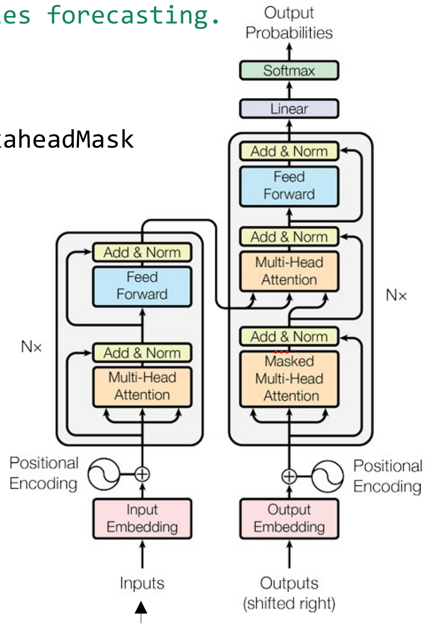


MX-AI

■ Stock price forecasting using Transformer: Prediction stage

```python
# [MXDL-11-07] 14.transformer(stock_predict).py
# Transformer code: https://github.com/suyas
# Since the Transformer code is for natural
# needs some modifications to be used for time series forecasting.
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from transformer import Encoder, Decoder
from transformer import PaddingMask, PaddingAndLookaheadMask
import matplotlib.pyplot as plt
import numpy as np
import pickle

# Read dataset
with open('stock_data.pkl', 'rb') as f:
      x_train, x_test, _, _, _ = pickle.load(f)

n_tstep = 60
n_feat = x_train.shape[1]   # 3
d_model = 120

# Trained Encoder
EmbDense = Dense(d_model, use_bias=False)
i_enc = Input(batch_shape=(None, n_tstep, n_feat))
h_enc = EmbDense(i_enc)
padding_mask = PaddingMask()(h_enc)
encoder = Encoder(num_layers = 2,
                  d_model = d_model,
                  num_heads = 4,
                  d_ff = 128,
                  dropout_rate=0.5)
o_enc, _ = encoder(h_enc, padding_mask)
```

y_hat = [y_1, y_2, y_3, ... y_{20}]



e_data

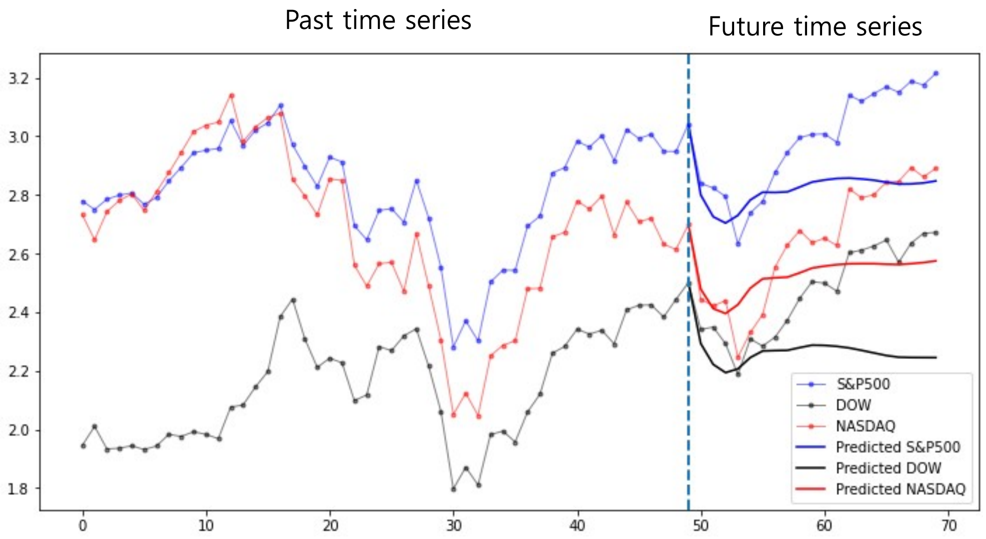
i=0: d_data =	[x_1,	0,	0,	...	0]
i=1: =	[x_1,	y_1,	0,	...	0]
i=2: =	[x_1,	y_1,	y_2,	...	0]
i=19: =	[x_1,	y_1,	y_2,	...	y_{20}]

```python
# Trained Decoder
i_dec = Input(batch_shape=(None, None, n_feat))
h_dec = EmbDense(i_dec)
lookahead_mask = PaddingAndLookaheadMask()(h_dec)
decoder = Decoder(num_layers = 2,
                  d_model = d_model,
                  num_heads = 4,
                  d_ff = 128,
                  dropout_rate=0.5)
o_dec, _, _ = decoder(h_dec, o_enc, lookahead_mask,
                      padding_mask)
y_dec = Dense(n_feat)(o_dec)

model = Model(inputs=[i_enc, i_dec], outputs=y_dec)
model.load_weights("models/transformer_stock.h5")

# prediction
n_past = 50
n_future = 20
e_data = x_train[-n_tstep:].reshape(-1,n_tstep,n_feat)
d_data = np.zeros(shape=(1, n_future, n_feat))
d_data[0, 0, :] = x_train[-1]

for i in range(n_future):
    y_hat = model.predict([e_data, d_data], verbose=0)

    if i < n_future - 1:
        d_data[0, i+1, :] = y_hat[0, i, :]

    print(i+1, ':', y_hat[0, i, :])
```

MX-AI

■ Stock price forecasting using Transformer: Prediction stage

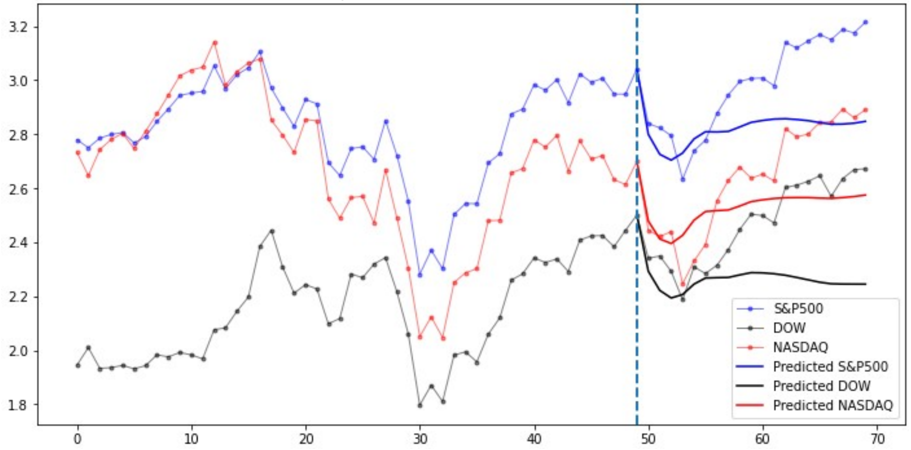
```
# Plot the past time series and the predicted future time series.
y_hat = np.vstack([x_train[-1], y_hat[0,:20,:]])
y_past = np.vstack([x_train[-n_past:], x_test])

plt.figure(figsize=(12, 6))
ax1 = np.arange(1, len(y_past) + 1)
ax2 = np.arange(n_past-1, n_past + n_future)
plt.plot(y_past[:, 0], '-o', c='blue', markersize=3,
         alpha=0.5, label='S&P500', linewidth=1)
plt.plot(y_past[:, 1], '-o', c='black', markersize=3,
         alpha=0.5, label='DOW', linewidth=1)
plt.plot(y_past[:, 2], '-o', c='red', markersize=3,
         alpha=0.5, label='NASDAQ', linewidth=1)
plt.plot(ax2, y_hat[:, 0], c='blue', label='Predicted S&P500')
plt.plot(ax2, y_hat[:, 1], c='black',label='Predicted DOW')
plt.plot(ax2, y_hat[:, 2], c='red',  label='Predicted NASDAQ')
plt.axvline(x=n_past-1,  linestyle='dashed', linewidth=2)
plt.legend()
plt.show()
```



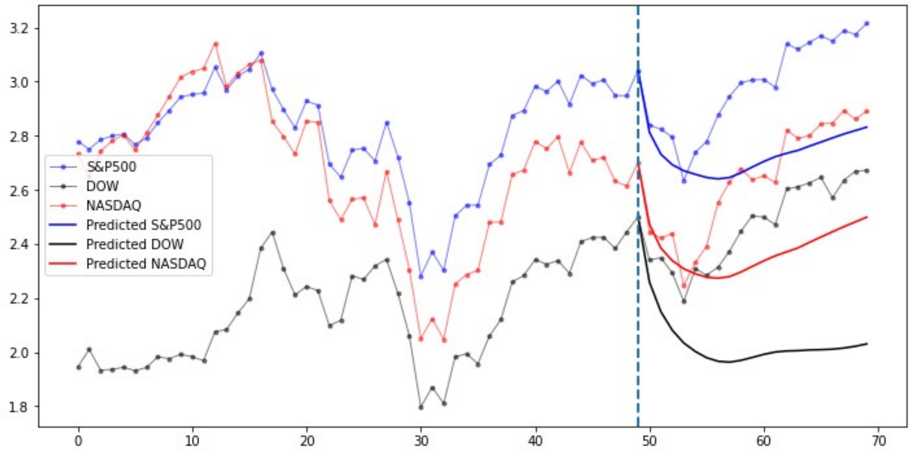
[MXDL-11-07] Deep Learning / Attention Networks: Transformer model

- Results

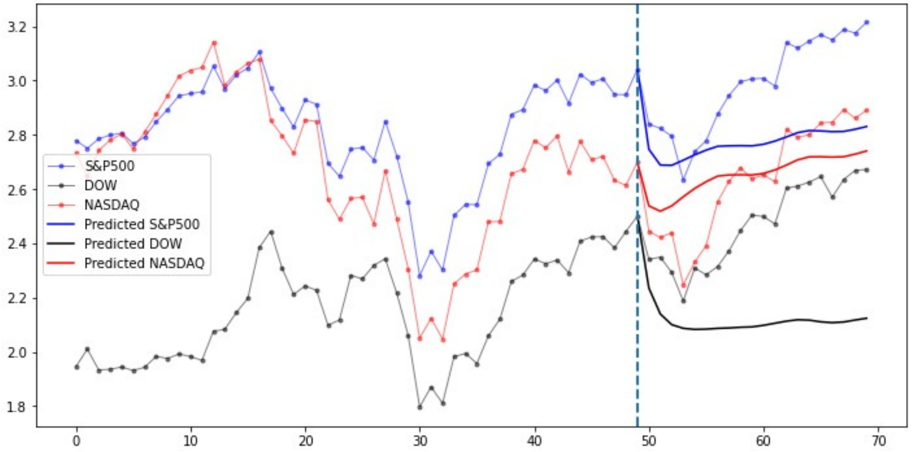
d_model = 120, num_layers = 2, num_heads = 4, d_ff = 128 : First run



The second run results



d_model = 120, num_layers = 4, num_heads = 8, d_ff = 256: First run



The second run results