12. Convolutional Neural Networks (CNN)

Part 1: Basics of CNN

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

The origin of the CNN architecture

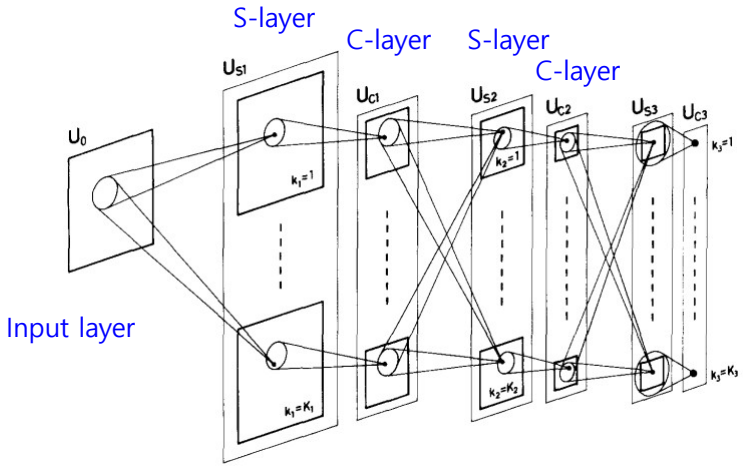
- Fukushima had proposed the Cognitron in 1975, and he proposed an improved version, the NeoCognitron network, in 1980.
- The Neocognitron consists of S-layers and C-layers. S-layer extracts features like a convolutional layer in CNN. And C-layer downsamples feature map like a pooling layer in CNN.
- The term "convolution" was used in neural networks in the 1980s. In 1989 and 1998, the term "Convolutional Neural Networks", which we use today, was used by Yann LeCun et al. in their architecture of LeNet for the task of recognizing handwritten digits.

Biol. Cybernetics 36, 193 202 (1980)

Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position

Kunihiko Fukushima
NHK Broadcasting Science Research Laboratories, Kinuta, Setagaya, Tokyo, Japan

Abstract. A neural network model for a mechanism of visual pattern recognition is proposed in this paper. The network is self-organized by "learning without a teacher", and acquires an ability to recognize stimulus patterns based on the geometrical similarity (Gestalt) of their shapes without affected by their positions. This network is given a nickname "**neocognitron**". After completion of self-organization, the network has a structure similar to the hierarchy model of the visual nervous system proposed by Hubel and Wiesel. The network consists of an input layer (photoreceptor array) followed by a cascade connection of a number of modular structures, each of which is composed of two layers of cells connected in a cascade. ...

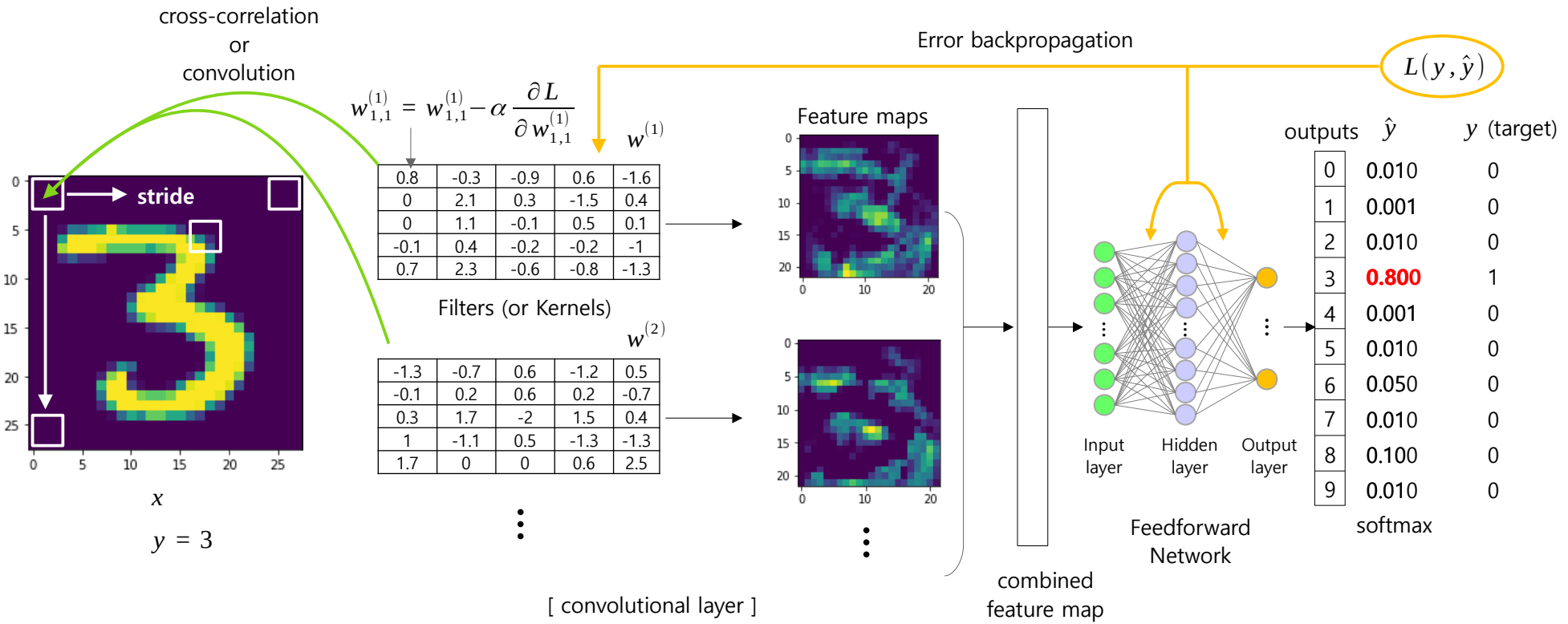


As shown in Fig. 1, the neocognitron consists of a cascade connection of a number of modular structures preceded by an input layer U_0. Each of the modular structure is composed of two layers of cells connected in a cascade. The first layer of the module consists of "S-cells", which correspond to simple cells or lower order hypercomplex cells according to the classification of Hubel and Wiesel. We call it **S-layer** and denote the S-layer in the l-th module as U_{Sl}. The second layer of the module consists of "C-cells", which correspond to complex cells or higher order hypercomplex cells. We call it **C-layer** and denote the C-layer in the l-th module as U_{Cl}. In the neocognitron, only the input synapses to S-cells are supposed to have plasticity and to be modifiable.

MX-AI

Basic concepts of Convolutional Neural Networks

- Let's see how CNN recognizes images with a simple example. This example is for a simple CNN model that classifies MNIST images.

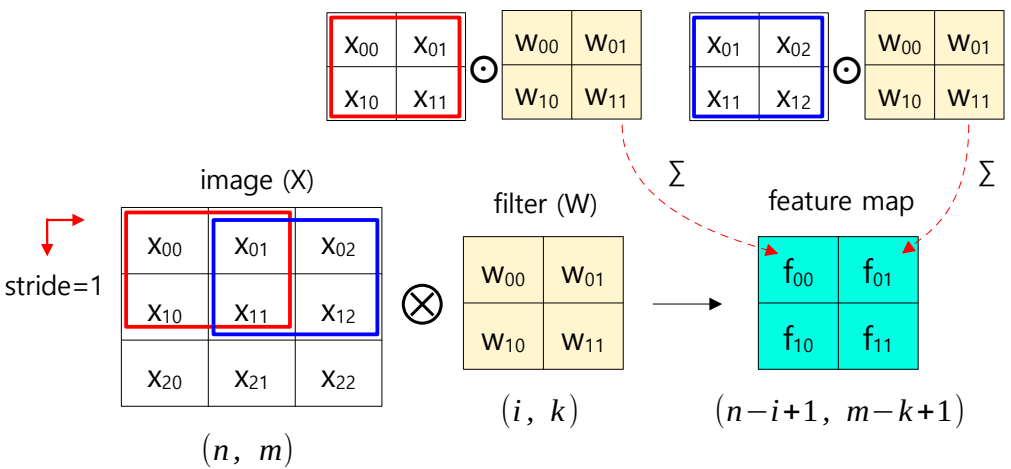


MX-AI

Convolutional layer: Cross-correlation vs. Convolution

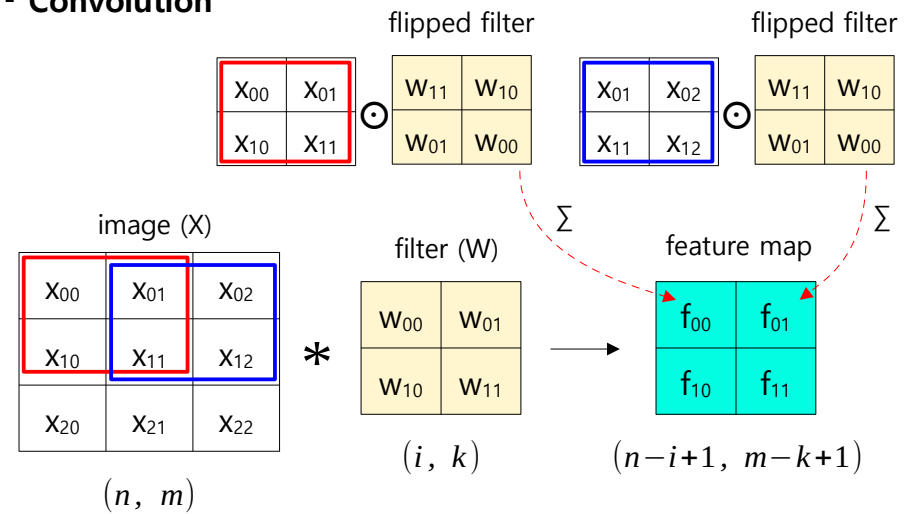
- There are two ways to apply filters on images in a convolutional layer: cross-correlation and convolution. Both are techniques used in signal processing.
- Cross-correlation allows us to measure the similarity between two signals, while convolution combines two signals to produce another signal.
- Convolution performs linear time-invariant (LTI) filtering by combining a signal with the impulse response of the system.
- CNN is called Convolutional Neural Networks, but most practical applications, such as Tensorflow and PyTorch, use cross-correlation instead of convolution.

Cross-correlation



$$f_{00} = x_{00}\,w_{00} + x_{01}\,w_{01} + x_{10}\,w_{10} + x_{11}\,w_{11} + \text{bias}$$
$$f_{01} = x_{01}\,w_{00} + x_{02}\,w_{01} + x_{11}\,w_{10} + x_{12}\,w_{11} + \text{bias}$$
$$f_{10} = x_{10}\,w_{00} + x_{11}\,w_{01} + x_{20}\,w_{10} + x_{21}\,w_{11} + \text{bias}$$
$$f_{11} = x_{11}\,w_{00} + x_{12}\,w_{01} + x_{21}\,w_{10} + x_{22}\,w_{11} + \text{bias}$$

Convolution



$$f_{00} = x_{00}\,w_{11} + x_{01}\,w_{10} + x_{10}\,w_{01} + x_{11}\,w_{00} + \text{bias}$$
$$f_{01} = x_{01}\,w_{11} + x_{02}\,w_{10} + x_{11}\,w_{01} + x_{12}\,w_{00} + \text{bias}$$
$$f_{10} = x_{10}\,w_{11} + x_{11}\,w_{10} + x_{20}\,w_{01} + x_{21}\,w_{00} + \text{bias}$$
$$f_{11} = x_{11}\,w_{11} + x_{12}\,w_{10} + x_{21}\,w_{01} + x_{22}\,w_{00} + \text{bias}$$

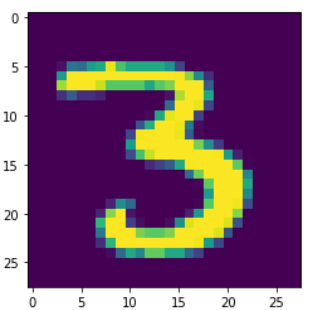
MX-AI

- Simple examples of cross-correlation and convolution calculations

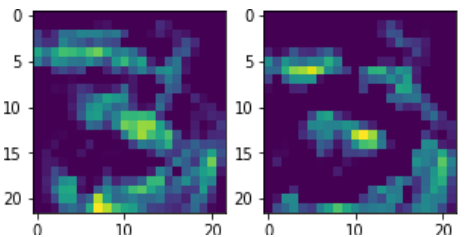
```
# [MXDL-12-01] 1.cross_correlation.py
import numpy as np
import matplotlib.pyplot as plt
import pickle

with open('data/mnist.pkl', 'rb') as f:
      x, y = pickle.load(f)
w1 = np.random.normal(size=(7,7))   # the first filter
w2 = np.random.normal(size=(7,7))   # the second filter

def cross_correlation(x, w):
    return np.sum(x * w)

def convolution(x, w):
    fw = np.flipud(np.fliplr(w))    # flipped w
    return cross_correlation(x, fw)

def feat_map(x, w, s, method):
    rw, cw = w.shape
    rx, cx = x.shape
    rf = int((rx - rw) / s + 1)
    cf = int((cx - cw) / s + 1)
    feat = np.zeros(shape=(rf, cf))
    for i in range(rf):
        for j in range(cf):
            px = x[(i*s):(i*s+rw), (j*s):(j*s+cw)]
            if method == 'CROSS':
                feat[i, j] = cross_correlation(px, w)
            else:
                feat[i, j] = convolution(px, w)
    return np.maximum(0, feat)  # ReLU

print("\nOriginal image:")
xi = x[12].reshape(28,28)
plt.imshow(xi)
plt.show()
```

```
feat1 = feat_map(xi, w1, 1, 'CROSS')
feat2 = feat_map(xi, w2, 1, 'CROSS')
print("\nFeature map by cross-correlation:")
fig, ax = plt.subplots(1, 2, figsize=(5,5))
ax[0].imshow(feat1)
ax[1].imshow(feat2)
plt.show()

feat1 = feat_map(xi, w1, 1, 'CONV')
feat2 = feat_map(xi, w2, 1, 'CONV')
print("\nFeature map by convolution:")
fig, ax = plt.subplots(1, 2, figsize=(5,5))
ax[0].imshow(feat1)
ax[1].imshow(feat2)
plt.show()
```

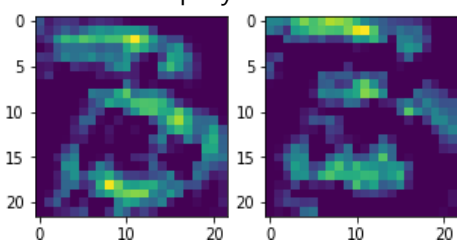
Original image:



Feature map by cross-correlation:



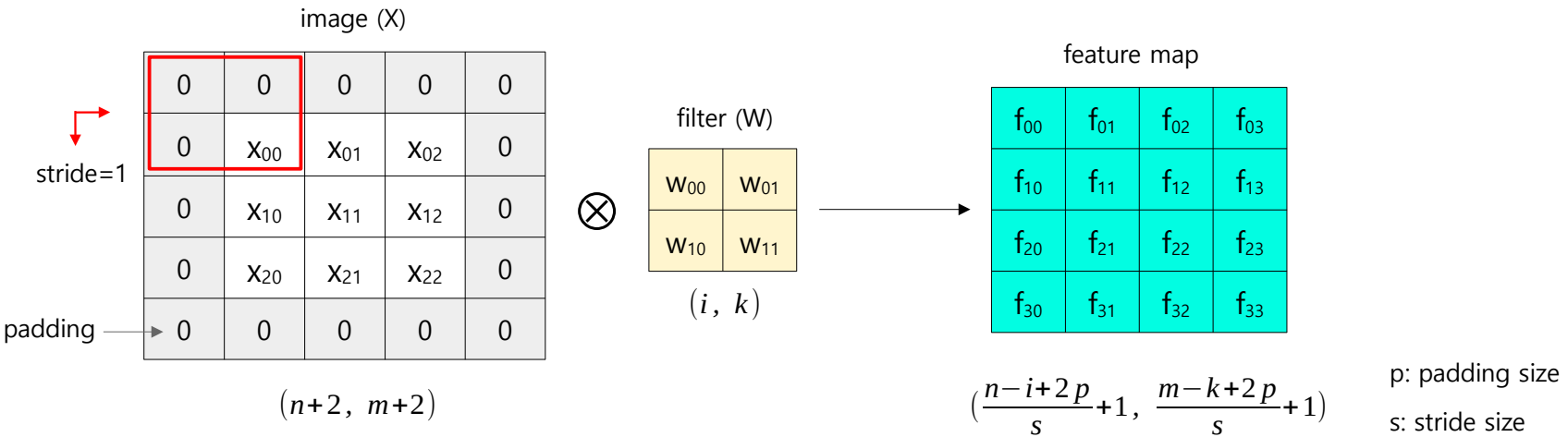
Feature map by convolution:



MX-AI

Padding

- Padding refers to adding extra pixels around the borders of an input image or feature map. The most common padding value is zero.
- Applying a filter to an input image or the output feature map of a previous convolutional layer results in the generated feature map being smaller than the input image. The more convolutional layers we have, the smaller the feature maps will be.
- This can result in the information loss at the borders of the input image. To prevent this, we use padding on the input image or feature map.



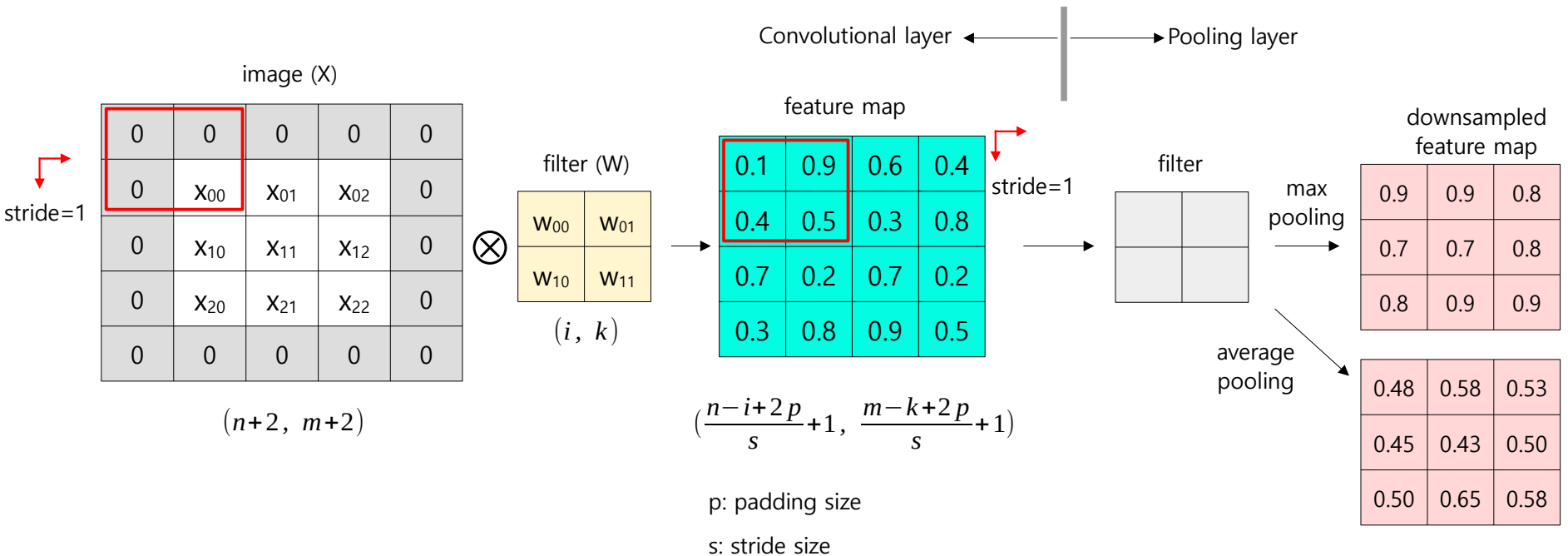
Without padding:

$f_{00} = x_{00} w_{00} + x_{01} w_{01} + x_{10} w_{10} + x_{11} w_{11} + bias$
$f_{01} = x_{01} w_{00} + x_{02} w_{01} + x_{11} w_{10} + x_{12} w_{11} + bias$
$f_{10} = x_{10} w_{00} + x_{11} w_{01} + x_{20} w_{10} + x_{21} w_{11} + bias$
$f_{11} = x_{11} w_{00} + x_{12} w_{01} + x_{21} w_{10} + x_{22} w_{11} + bias$

$(\frac{3-2+2\times1}{1}+1, \frac{3-2+2\times1}{1}+1) = (4, 4)$

Pooling Layer

- A pooling layer downsamples the output feature maps of the convolutional layer to produce new feature maps of smaller size.
- It scans the feature map using a filter to extract important pixels. The filter in the pooling layer has no trainable parameters, no element values.
- The most popular pooling methods are max-pooling and average-pooling. Max pooling extracts the largest value in the area covered by the filter, while average pooling computes the average of the pixel values in that area.

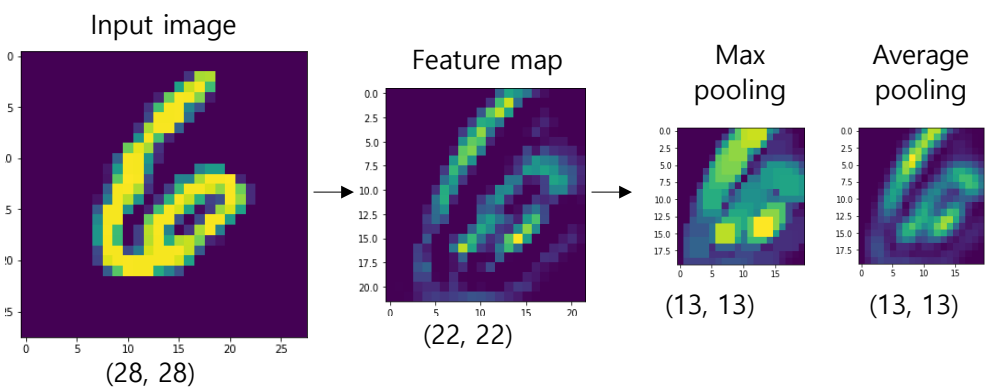


MX-AI

■ Simple example of pooling layer

```
# [MXDL-12-01] 2.pooling.py
import numpy as np
import matplotlib.pyplot as plt
import pickle

with open('data/mnist.pkl', 'rb') as f:
      x, y = pickle.load(f)

w1 = np.random.normal(size=(7,7)) # filter for convolutional layer
w2 = np.empty(shape=(3,3))        # filter for pooling layer

def cross_correlation(x, w): return np.sum(x * w)

def feat_map(x, w, s):
    rw, cw = w.shape
    rx, cx = x.shape
    rf = int((rx - rw) / s + 1)
    cf = int((cx - cw) / s + 1)
    feat = np.zeros(shape=(rf, cf))
    for i in range(rf):
        for j in range(cf):
            px = x[(i*s):(i*s+rw), (j*s):(j*s+cw)]
            feat[i, j] = cross_correlation(px, w)
    return np.maximum(0, feat)  # ReLU

def pooling(x, w, s, method):
    rw, cw = w.shape
    rx, cx = x.shape
    rf = int((rx - rw) / s + 1)
    cf = int((cx - cw) / s + 1)
    feat = np.zeros(shape=(rf, cf))
    for i in range(rf):
        for j in range(cf):
            px = x[(i*s):(i*s+rw), (j*s):(j*s+cw)]
            if method == 'max': feat[i, j] = np.max(px)
            else: feat[i, j] = np.mean(px)
    return feat
```

```
def show_image(x, size, title):
    plt.figure(figsize=size)
    plt.imshow(x)
    plt.title(title)
    plt.show()

xi = x[13].reshape(28,28)
show_image(xi, (5,5), "Input image")

feat = feat_map(xi, w1, 1)
show_image(feat, (4,4), "Feature map")

max_pool = pooling(feat, w2, 1, 'max')
show_image(max_pool, (3,3), "Max-pooling")

avg_pool = pooling(feat, w2, 1, 'avg')
show_image(avg_pool, (3,3), "Average-pooling")
```

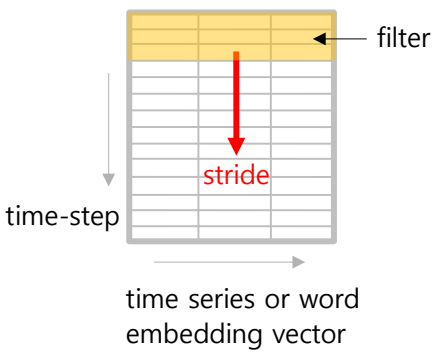


MX-AI

Types of convolutional layers

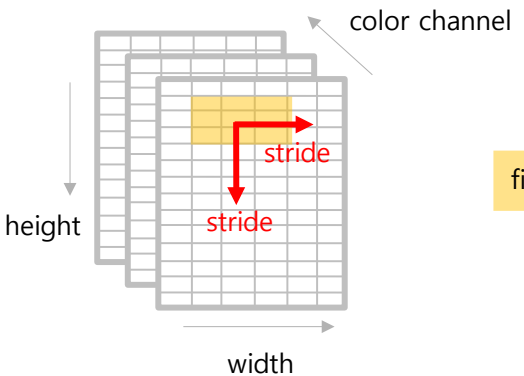
- There are three types of convolutional layers: 1D convolution, 2D convolution, and 3D convolution.
- 1D convolution can be used for sequence data such as time series and natural language data, and the filters move in only one direction.
- 2D convolution is typically used for RGB images, where the filters move in two directions.
- 3D convolution can be used for 3D image slices, such as medical imaging, or for videos, such as action recognition data. The filters slide in three dimensions.
- The structures of the input data are 3-dimensional, 4-dimensional, and 5-dimensional, respectively, including the batch dimension.

1D Convolution



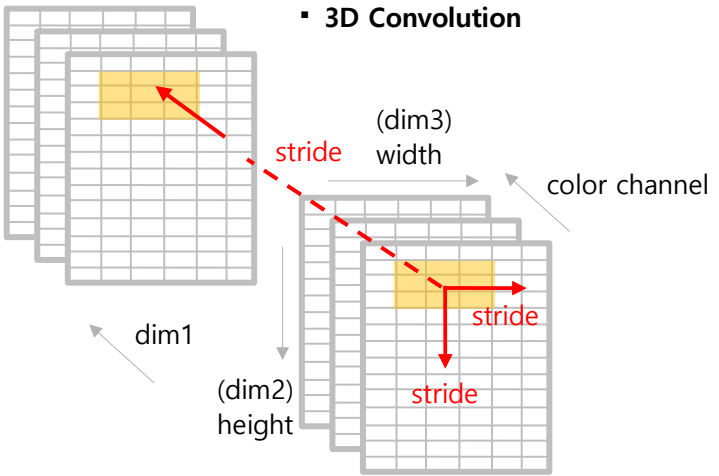
- 3-dimensional input data:
  (batch, time-step, embedding size)

2D Convolution



- 4-dimensional input data:
  (batch, height, width, channel) or
  (batch, channel, height, width)

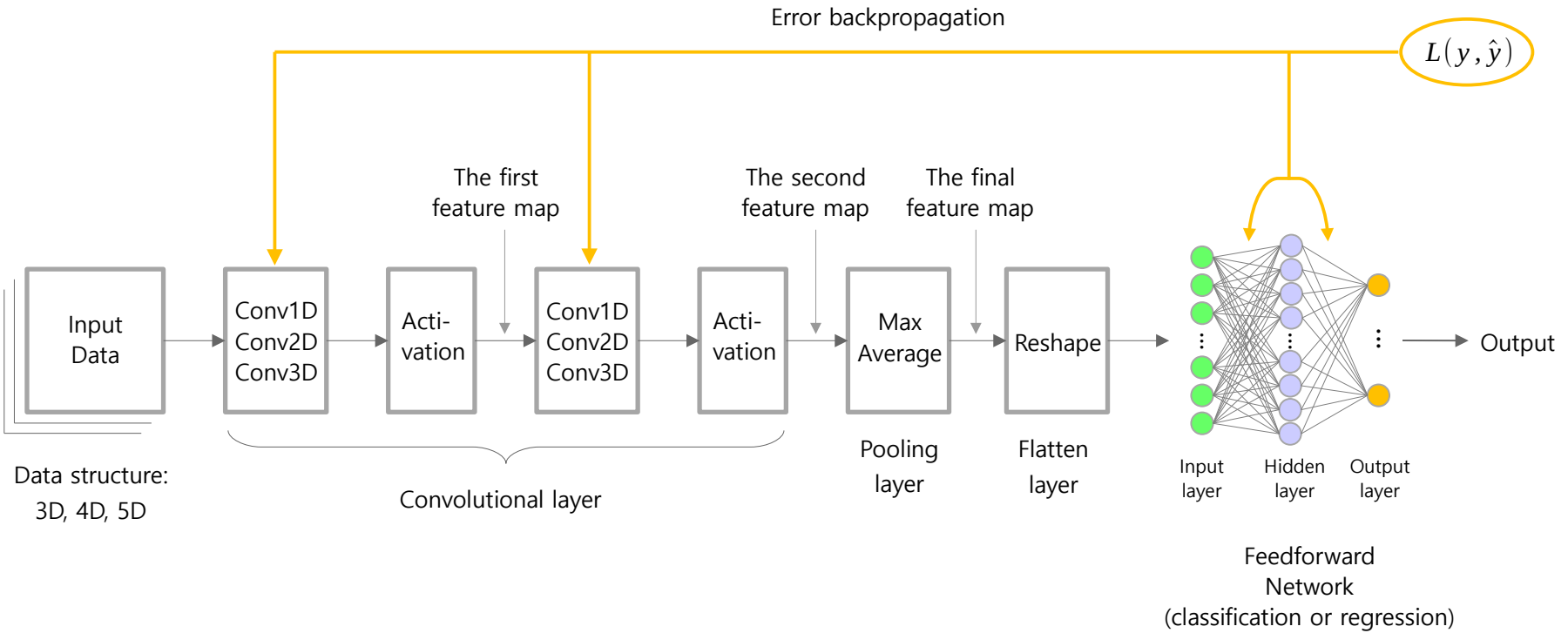
3D Convolution

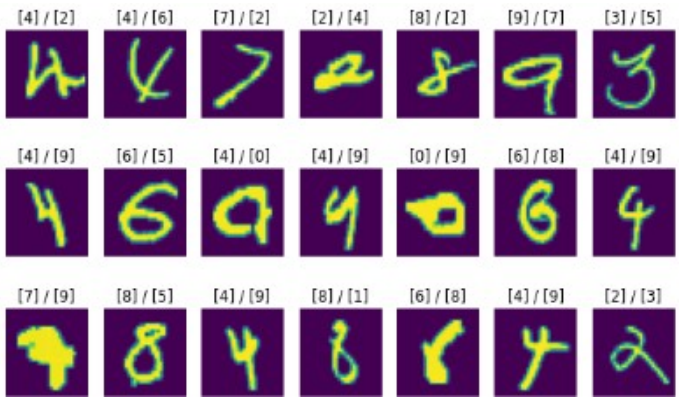


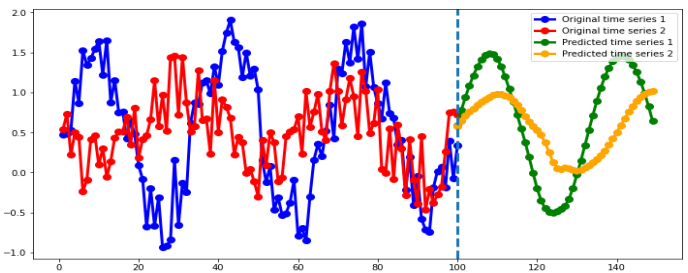
- 5-dimensional input data:
  (batch, dim1, dim2, dim3, channel)

MX-AI

- The overall structure of CNN







12. Convolutional Neural Networks (CNN)

Part 2: 1D Convolution

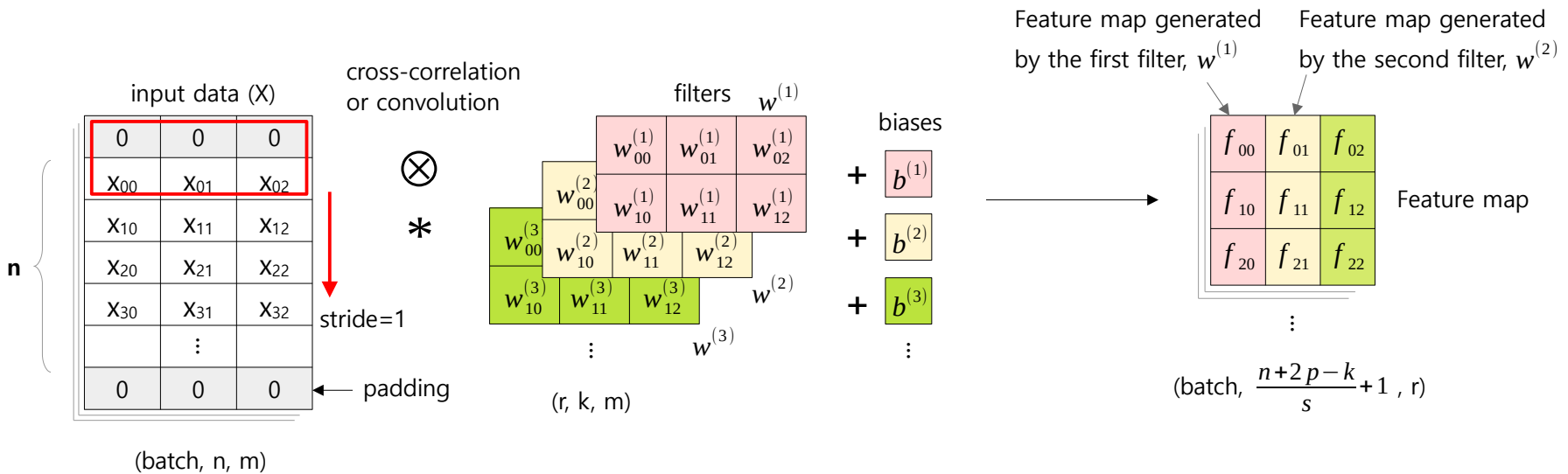
This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

■ 1D Convolutional layer

- 1D convolution is typically used for sequence data such as time series and natural language data, and the filters slide in one dimension.



$$f_{00} = 0\cdot w_{00}^{(1)} + 0\cdot w_{01}^{(1)} + 0\cdot w_{02}^{(1)} + x_{00}\cdot w_{10}^{(1)} + x_{01}\cdot w_{11}^{(1)} + x_{02}\cdot w_{12}^{(1)} + b^{(1)}$$

$$f_{10} = x_{00}\cdot w_{00}^{(1)} + x_{01}\cdot w_{01}^{(1)} + x_{02}\cdot w_{02}^{(1)} + x_{10}\cdot w_{10}^{(1)} + x_{11}\cdot w_{11}^{(1)} + x_{12}\cdot w_{12}^{(1)} + b^{(1)}$$

$$f_{20} = x_{10}\cdot w_{00}^{(1)} + x_{11}\cdot w_{01}^{(1)} + x_{12}\cdot w_{02}^{(1)} + x_{20}\cdot w_{10}^{(1)} + x_{21}\cdot w_{11}^{(1)} + x_{22}\cdot w_{12}^{(1)} + b^{(1)}$$

$$f_{01} = 0\cdot w_{00}^{(2)} + 0\cdot w_{01}^{(2)} + 0\cdot w_{02}^{(2)} + x_{00}\cdot w_{10}^{(2)} + x_{01}\cdot w_{11}^{(2)} + x_{02}\cdot w_{12}^{(2)} + b^{(2)}$$

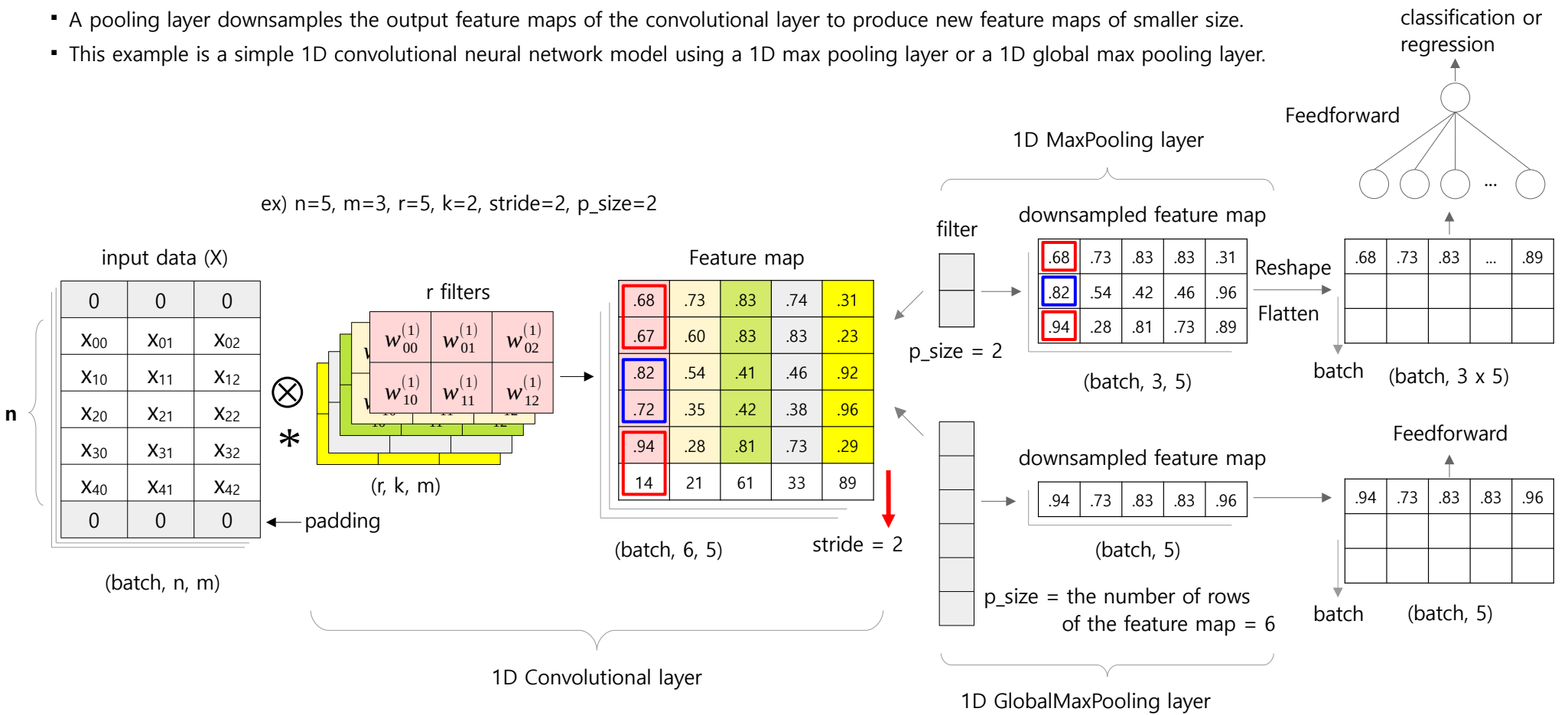
$$f_{11} = x_{00}\cdot w_{00}^{(2)} + x_{01}\cdot w_{01}^{(2)} + x_{02}\cdot w_{02}^{(2)} + x_{10}\cdot w_{10}^{(2)} + x_{11}\cdot w_{11}^{(2)} + x_{12}\cdot w_{12}^{(2)} + b^{(2)}$$

$$f_{21} = x_{10}\cdot w_{00}^{(2)} + x_{11}\cdot w_{01}^{(2)} + x_{12}\cdot w_{02}^{(2)} + x_{20}\cdot w_{10}^{(2)} + x_{21}\cdot w_{11}^{(2)} + x_{22}\cdot w_{12}^{(2)} + b^{(2)}$$

MX-AI

1D Pooling layer

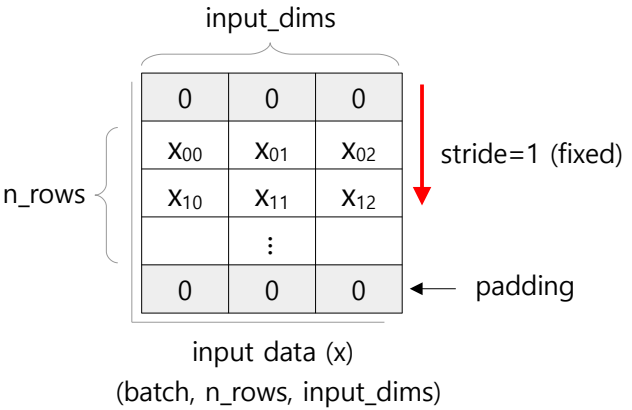
- A pooling layer downsamples the output feature maps of the convolutional layer to produce new feature maps of smaller size.
- This example is a simple 1D convolutional neural network model using a 1D max pooling layer or a 1D global max pooling layer.



MX-AI

■ Custom Conv1D class

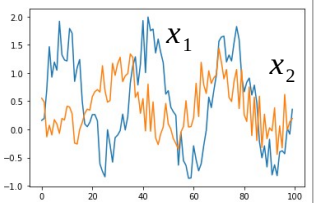
```
import tensorflow as tf
class MyConv1D(tf.keras.layers.Layer):
      def __init__(self, input_dims, filters, kernel_size, padding='VALID'):
            super().__init__()
            self.k_size = kernel_size
            self.pad = padding
            w_init = tf.random_normal_initializer()
            self.w = tf.Variable(w_init([filters, kernel_size, input_dims]), trainable=True)
            self.b = tf.Variable(tf.zeros_initializer()([filters, ]), trainable=True)

      def call(self, x):
            n_rows = x.shape[1]
            if self.pad == 'SAME':
                  n_pad_top = n_pad_bot = self.k_size // 2
                  px = tf.pad(x, [[0,0], [n_pad_top, n_pad_bot], [0,0]])
                  n_outs = n_rows    # the number of rows in the feature map
            else: # no padding
                  px = x
                  n_outs = n_rows - self.k_size + 1

            # Compute the cross-correlations as we move down the filters.
            cc = []
            for k in range(n_outs):
                  p = px[:, k:(k + self.k_size), :]
                  cc.append(self.w[tf.newaxis, :, :, :] * p[:, tf.newaxis, :, :])

            cc = tf.stack(cc)  # [n_outs, None, filters, kernel_size, input_dims]
            conv = tf.reduce_sum(cc, [3, 4])            # [n_outs, None, filters]
            conv = tf.transpose(conv, [1, 0, 2])        # [None, n_outs, filters]
            conv += self.b[tf.newaxis, tf.newaxis, :]   # [None, None, filters]
            return conv                                 # [None, n_outs, filters]
```



MX-AI

- Time series prediction using the custom Conv1D class

```
# [MXDL-12-02] 3.custom_conv1d(timeseries).py
from tensorflow.keras.layers import Input, Dense, Activation
from tensorflow.keras.layers import Flatten, AveragePooling1D
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from MyConv1D import MyConv1D
import numpy as np
import matplotlib.pyplot as plt

# Generate training data: 2 noisy sine curves
n = 3000        # the number of data points
n_step = 30     # 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 = (3000, 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_emb = 20                  # time series embedding size
k_size = 5                  # kernel size
n_kernel = 10               # number of filters
p_size = 10                 # pooling filter size
n_feat = x_train.shape[-1]  # the number of features
```




```
# Build a CNN model
x_input = Input(batch_shape=(None, n_step, n_feat))
emb = Dense(n_emb, activation='tanh')(x_input)
conv = MyConv1D(n_emb, n_kernel, k_size, padding="SAME")(emb)
conv = Activation('relu')(conv)
pool = AveragePooling1D(pool_size=p_size, strides=1)(conv)
flat = Flatten()(pool)
y_output = Dense(y_train.shape[1])(flat)

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

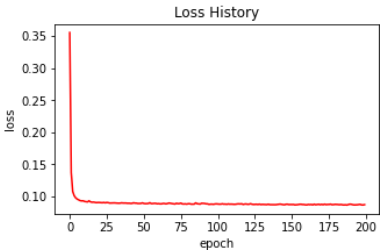
```
 Layer (type)                 Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 30, 2)]           0
 dense (Dense)                (None, 30, 20)           60
 my_conv1d (MyConv1D)         (None, 30, 10)           1010
 activation (Activation)      (None, 30, 10)           0
 average_pooling1d            (None, 21, 10)           0
 (AveragePooling1D)
 flatten (Flatten)            (None, 210)              0
 dense_1 (Dense)              (None, 2)                422
=================================================================
Total params: 1,492
Trainable params: 1,492
Non-trainable params: 0
```

```
# Training
hist = model.fit(x_train, y_train, epochs=200, batch_size=100)
```

MX-AI

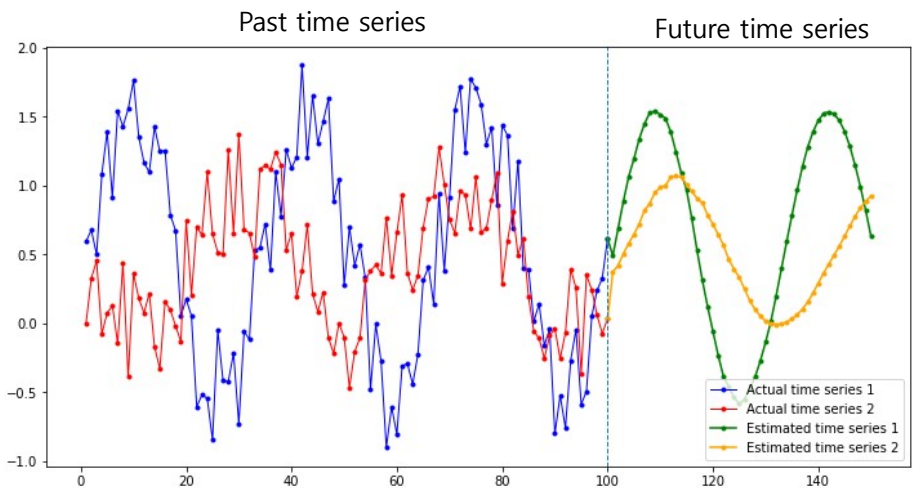
- Time series prediction using the custom Conv1D class

```
# 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])

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()
```

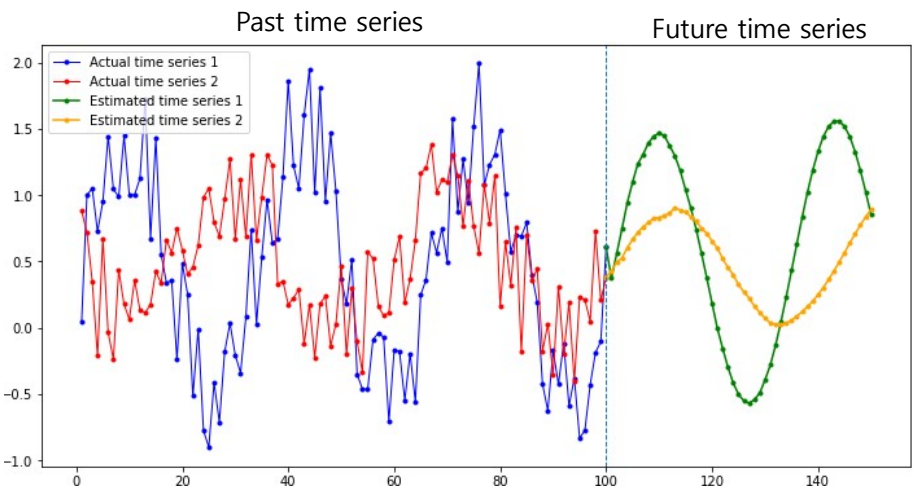


MX-AI

■ Time series prediction using Keras' Conv1D class

```
# [MXDL-12-02] 4.keras_conv1d(timeseries).py
from tensorflow.keras.layers import Input, Dense, Conv1D, Activation
from tensorflow.keras.layers import Flatten, AveragePooling1D
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from MyConv1D import MyConv1D
import numpy as np
import matplotlib.pyplot as plt

# Generate training data: 2 noisy sine curves
# [skipped]

n_emb = 20                    # time series embedding size
k_size = 5                    # kernel size
n_kernel = 10                 # number of kernels
p_size = 10                   # pooling filter size
n_feat = x_train.shape[-1]    # the number of features

# Build a CNN model
x_input = Input(batch_shape=(None, n_step, n_feat))
emb = Dense(n_emb, activation='tanh')(x_input)
conv = Conv1D(n_kernel, k_size, padding="SAME")(emb)
conv = Activation('relu')(conv)
pool = AveragePooling1D(pool_size=p_size, strides=1)(conv)
flat = Flatten()(pool)
y_output = Dense(y_train.shape[1])(flat)

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

```
 Layer (type)                 Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 30, 2)]           0
 dense (Dense)                (None, 30, 20)           60
 conv1d (Conv1D)              (None, 30, 10)           1010
 activation (Activation)      (None, 30, 10)           0
 average_pooling1d            (None, 21, 10)           0
 (AveragePooling1D)
 flatten (Flatten)            (None, 210)              0
 dense_1 (Dense)              (None, 2)                422
=================================================================
Total params: 1,492
Trainable params: 1,492
Non-trainable params: 0
```

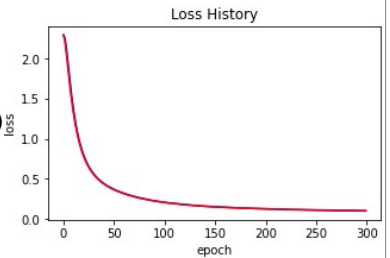


MX-AI

- MNIST classification using the custom Conv1D class

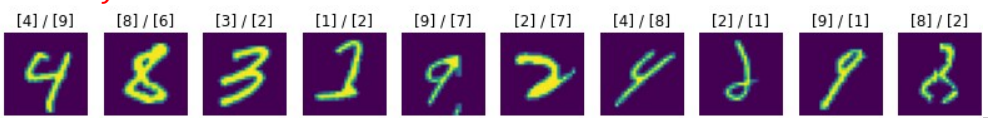
```
# [MXDL-12-02] 5.custom_conv1d(mnist).py
from tensorflow.keras.layers import Input, Dense, Activation
from tensorflow.keras.layers import Flatten, MaxPooling1D
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from MyConv1D import MyConv1D
import numpy as np
import matplotlib.pyplot as plt
import pickle

# Read an MNIST dataset
with open('data/mnist.pkl', 'rb') as f:
      x, y = pickle.load(f)

x = x.reshape(-1, 28, 28) / 255
y = y.reshape(-1,1)
x_train, x_test, y_train, y_test = train_test_split(x, y)
n_class = len(set(y_train.reshape(-1,)))

k_size = 5                  # kernel size
n_kernel = 20               # number of kernels
p_size = 10                 # pooling filter size
n_row = x_train.shape[1]    # number of rows of an image
n_col = x_train.shape[2]    # number of columns of an image

# Build a CNN model
x_input = Input(batch_shape=(None, n_row, n_col))
conv = MyConv1D(n_col, n_kernel, k_size)(x_input)
conv = Activation('relu')(conv)
pool = MaxPooling1D(pool_size=p_size, strides=1)(conv)
flat = Flatten()(pool)
y_output = Dense(n_class, activation='softmax')(flat)
model = Model(x_input, y_output)
```

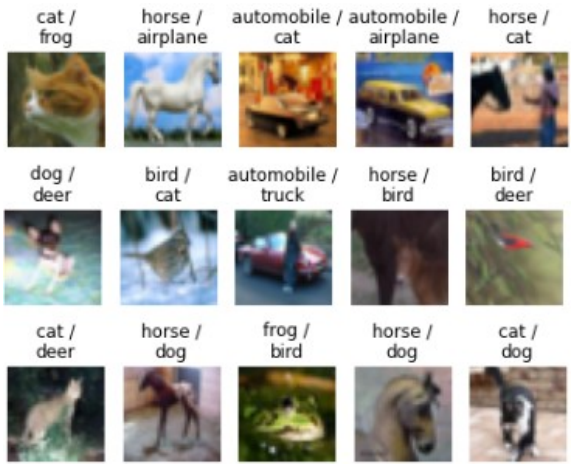
```
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam')
hist = model.fit(x_train, y_train, epochs=300, batch_size=1000,
                 validation_data = [x_test, y_test])

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

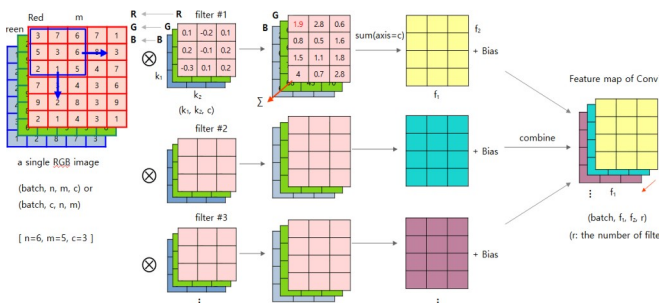



```
y_prob = model.predict(x_test)
y_pred = np.argmax(y_prob, axis=1).reshape(-1,1)
acc = (y_test == y_pred).mean()
print('* Accuracy of the test data ={:.4f}'.format(acc))

# 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]
    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 the test data =0.9720
```







12. Convolutional Neural Networks (CNN)

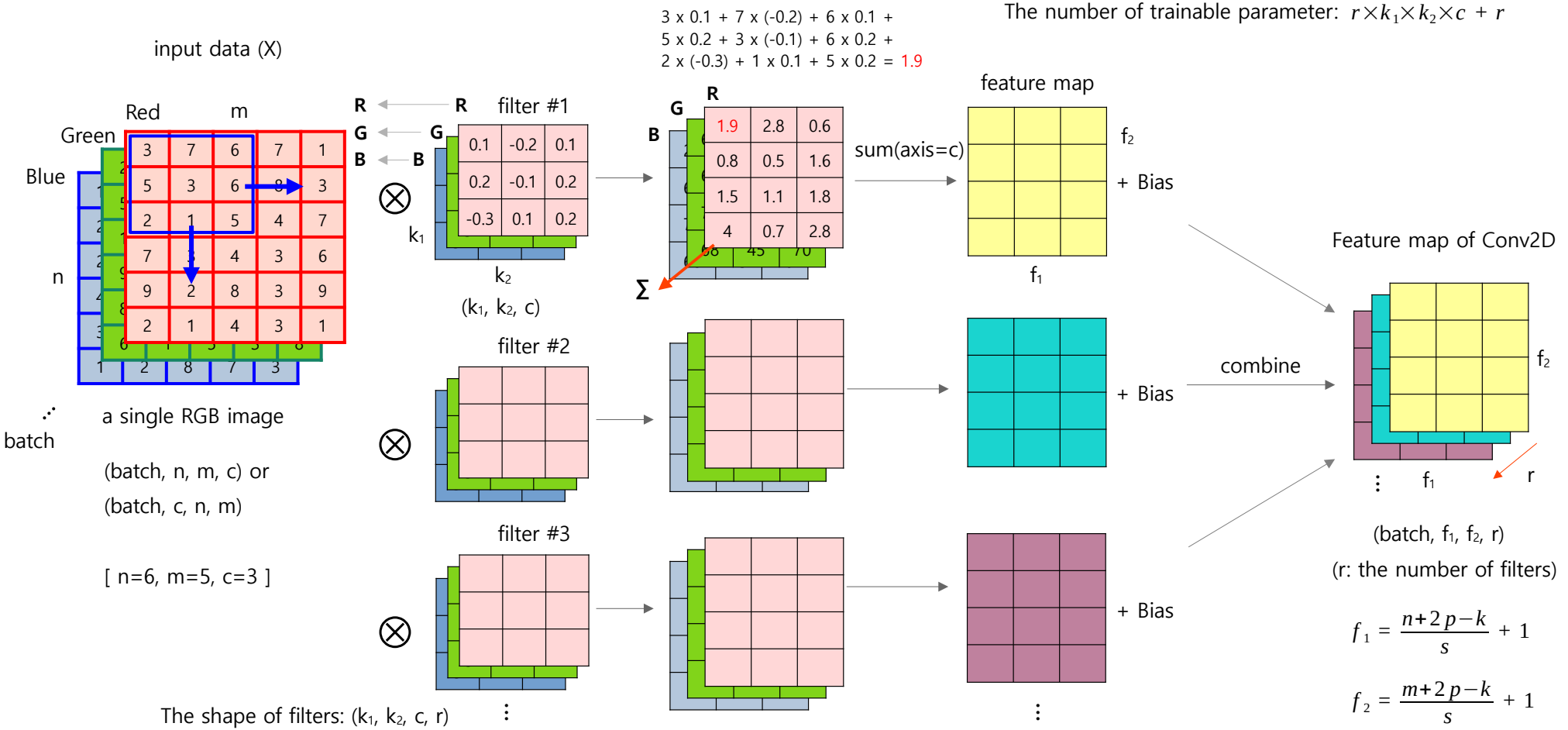
Part 3: 2D Convolution

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

www.youtube.com/@meanxai

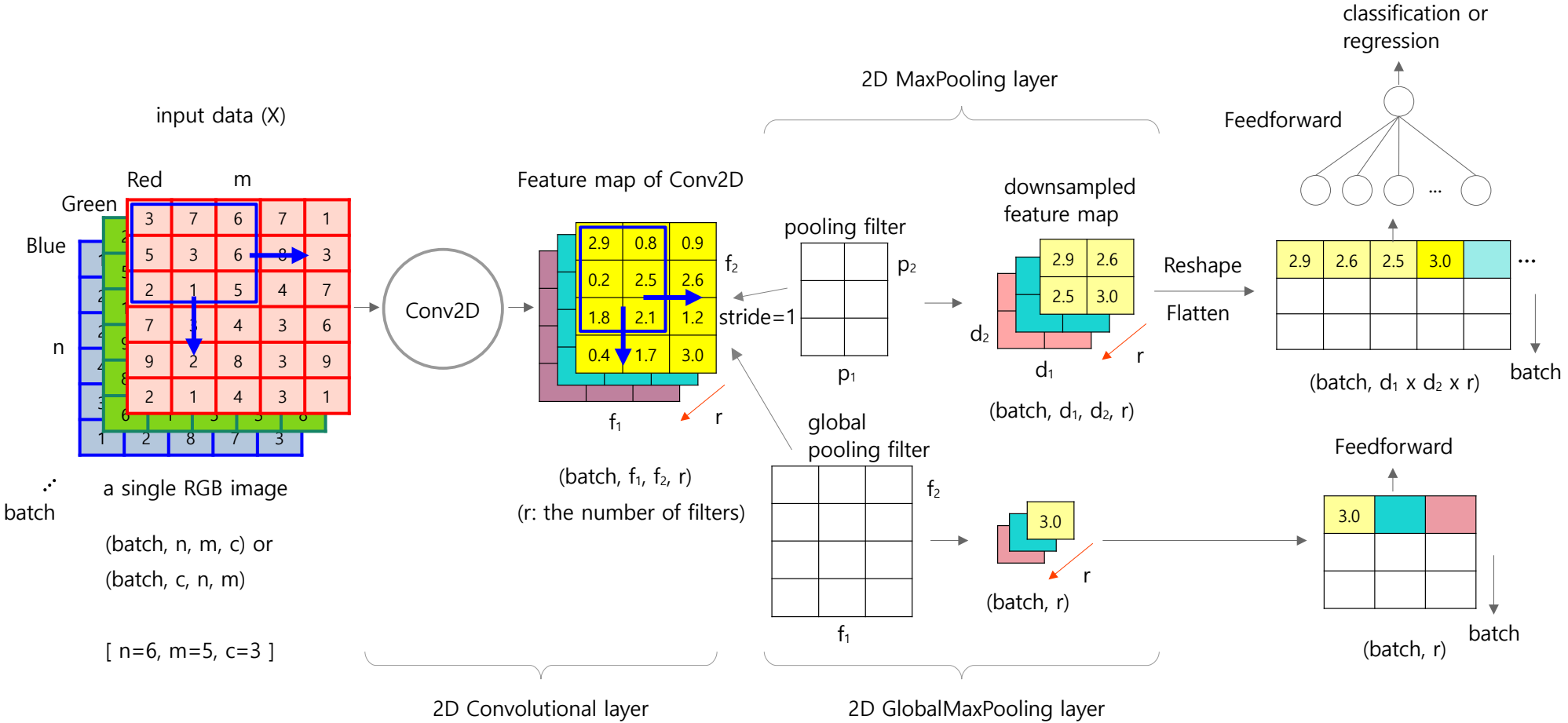
MX-AI

■ 2D Convolutional layer (Conv2D)



MX-AI

■ 2D Pooling layer: MaxPooling2D and GlobalMaxPooling2D



MX-AI

- CIFAR-10 image classification using Keras' Conv2D and AvergePooling2D

```
# [MXDL-11-03] 7.conv2D(cifar10).py
from tensorflow.keras.layers import Input, Dense, Conv2D, Flatten
from tensorflow.keras.layers import Dropout, BatchNormalization
from tensorflow.keras.layers import Activation, AveragePooling2D
from tensorflow.keras.layers import RandomFlip, RandomRotation
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt
import pickle
```


```
# Load CIFAR10 dataset
# from tensorflow.keras.datasets import cifar10
# (x_train, y_train), (x_test, y_test) = cifar10.load_data()
# x_train = x_train / 255.0
# x_test = x_test / 255.0
# with open('data/cifar10.pkl', 'wb') as f:
#     pickle.dump([x_train, y_train, x_test, y_test], f)

with open('data/cifar10.pkl', 'rb') as f:
      x_train, y_train, x_test, y_test = pickle.load(f)

categories = ['airplane', 'automobile', 'bird', 'cat', 'deer',
              'dog', 'frog', 'horse', 'ship', 'truck']
```

```
# Build a 2D-CNN model
def Conv2D_Pool(x, filters, k_size, p_size):
    x = Conv2D(filters, k_size)(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = AveragePooling2D(p_size)(x)
    return x

x_input = Input(batch_shape=(None, *x_train.shape[1:]))
x = RandomFlip(mode='horizontal')(x_input)    # image augmentation
x = RandomRotation(0.1)(x)
h = Conv2D_Pool(x, 16, k_size=(3,3), p_size=(2,2))
h = Conv2D_Pool(h, 32, k_size=(3,3), p_size=(2,2))
h = Conv2D_Pool(h, 64, k_size=(3,3), p_size=(2,2))
h = Flatten()(h)
h = Dense(128, activation='relu')(h)
h = Dropout(0.5)(h)
y_output = Dense(10, activation='softmax')(h)

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

# Training
hist = model.fit(x_train, y_train,
                 epochs = 300,
                 batch_size = 1000,
                 validation_data = (x_test, y_test),
                 shuffle = True)
```

MX-AI

- CIFAR-10 image classification using Keras' Conv2D and AvergePooling2D

$$f_1 = \frac{n+2p-k}{s} + 1$$

```
# Build a 2D-CNN model
def Conv2D_Pool(x, filters, k_size, p_size):
    x = Conv2D(filters, k_size)(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = AveragePooling2D(p_size)(x)
    return x

x_input = Input(batch_shape=(None, *x_train.shape[1:]))
x = RandomFlip(mode='horizontal')(x_input)
x = RandomRotation(0.1)(x)
h = Conv2D_Pool(x, 16, k_size=(3,3), p_size=(2,2))
h = Conv2D_Pool(h, 32, k_size=(3,3), p_size=(2,2))
h = Conv2D_Pool(h, 64, k_size=(3,3), p_size=(2,2))
h = Flatten()(h)
h = Dense(128, activation='relu')(h)
h = Dropout(0.5)(h)
y_output = Dense(10, activation='softmax')(h)

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

# Training
hist = model.fit(x_train, y_train,
                 epochs = 300,
                 batch_size = 1000,
                 validation_data = (x_test, y_test),
                 shuffle = True)
```

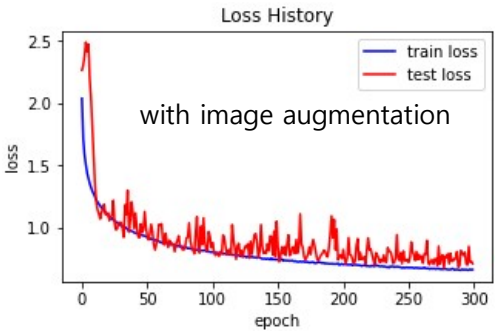
	Layer (type)	Output Shape	Param #
image augmentation	input_1 (InputLayer)	[(None, 32, 32, 3)]	0
	random_flip (RandomFlip)	(None, 32, 32, 3)	0
	random_rotation (RandomRotation)	(None, 32, 32, 3)	0
L #1	conv2d (Conv2D)	(None, 30, 30, 16)	448
	batch_normalization (BatchNormalization)	(None, 30, 30, 16)	64
	activation (Activation)	(None, 30, 30, 16)	0
	average_pooling2d (AveragePooling2D)	(None, 15, 15, 16)	0
L #2	conv2d_1 (Conv2D)	(None, 13, 13, 32)	4640
	batch_normalization_1 (BatchNormalization)	(None, 13, 13, 32)	128
	activation_1 (Activation)	(None, 13, 13, 32)	0
	average_pooling2d_1 (AveragePooling2D)	(None, 6, 6, 32)	0
L #3	conv2d_2 (Conv2D)	(None, 4, 4, 64)	18496
	batch_normalization_2 (BatchNormalization)	(None, 4, 4, 64)	256
	activation_2 (Activation)	(None, 4, 4, 64)	0
	average_pooling2d_2 (AveragePooling2D)	(None, 2, 2, 64)	0
FFN	flatten (Flatten)	(None, 256)	0
	dense (Dense)	(None, 128)	32896
	dropout (Dropout)	(None, 128)	0
	dense_1 (Dense)	(None, 10)	1290

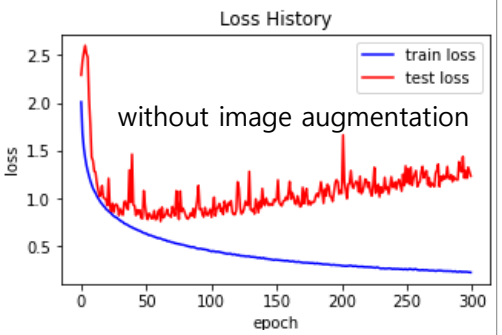
Total params: 58,218
Trainable params: 57,994
Non-trainable params: 224

MX-AI

- CIFAR-10 image classification using Keras' Conv2D and AvergePooling2D

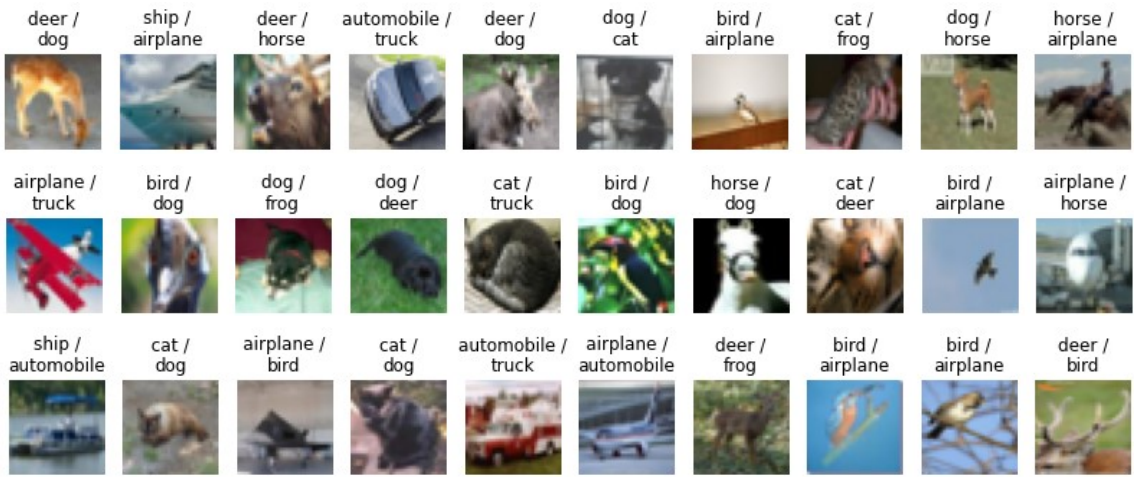
```
# Loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'], color='blue', label='train loss')
plt.plot(hist.history['val_loss'], color='red', label='test loss')
plt.legend()
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()

# Accuracy for test data
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 for the test data = {:.4f}'.format(acc))

# 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):
    ax[i].imshow(x_test[miss])
    ax[i].axis('off')
    s_test = categories[y_test[miss][0]]
    s_pred = categories[y_pred[miss][0]]
    ax[i].set_title(s_test + ' /\n' + s_pred)
plt.show()
```

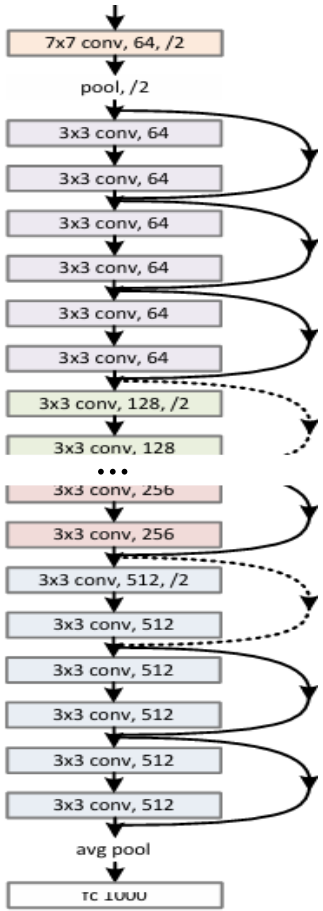




Accuracy for the test data = 0.7649

Accuracy = 0.7284





Shortcut connection

12. Convolutional Neural Networks (CNN)

Part 4: Residual Neural Network

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

www.youtube.com/@meanxai

MX-AI

■ Residual Network (2015)

Deep Residual Learning for Image Recognition

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

Abstract

Deeper neural networks are more difficult to train. We present a residual learning framework to ease the training of networks that are substantially deeper than those used previously. We explicitly reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions. We provide comprehensive empirical evidence showing that these residual networks are easier to optimize, and can gain accuracy from considerably increased depth. ...

1. Introduction

< ... >

When deeper networks are able to start converging, a degradation problem has been exposed: with the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly. Unexpectedly, such degradation is not caused by overfitting, and adding more layers to a suitably deep model leads to higher training error, as reported in [11, 42] and thoroughly verified by our experiments. Fig. 1 shows a typical example.

The degradation (of training accuracy) indicates that not all systems are similarly easy to optimize. Let us consider a shallower architecture and its deeper counterpart that adds more layers onto it. There exists a solution by construction to the deeper model: the added layers are identity mapping, and the other layers are copied from the learned shallower model.

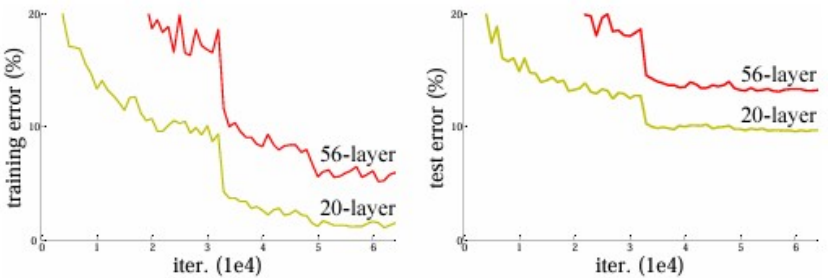


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

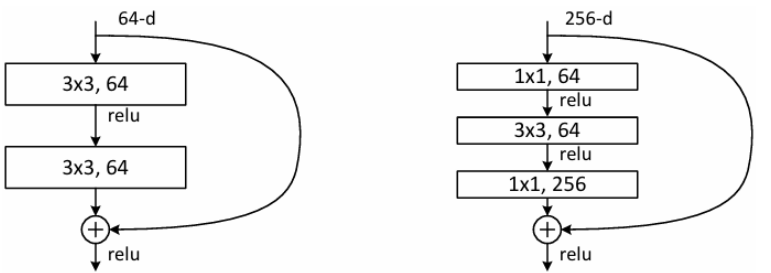


Figure5. A deeper residual function F for ImageNet. Left: a building block (on 56 x 56 feature maps)as in Fig.3 for ResNet34. Right: a "bottleneck" building block for ResNet-50/101/152.

MX-AI

- Architecture of Residual Network

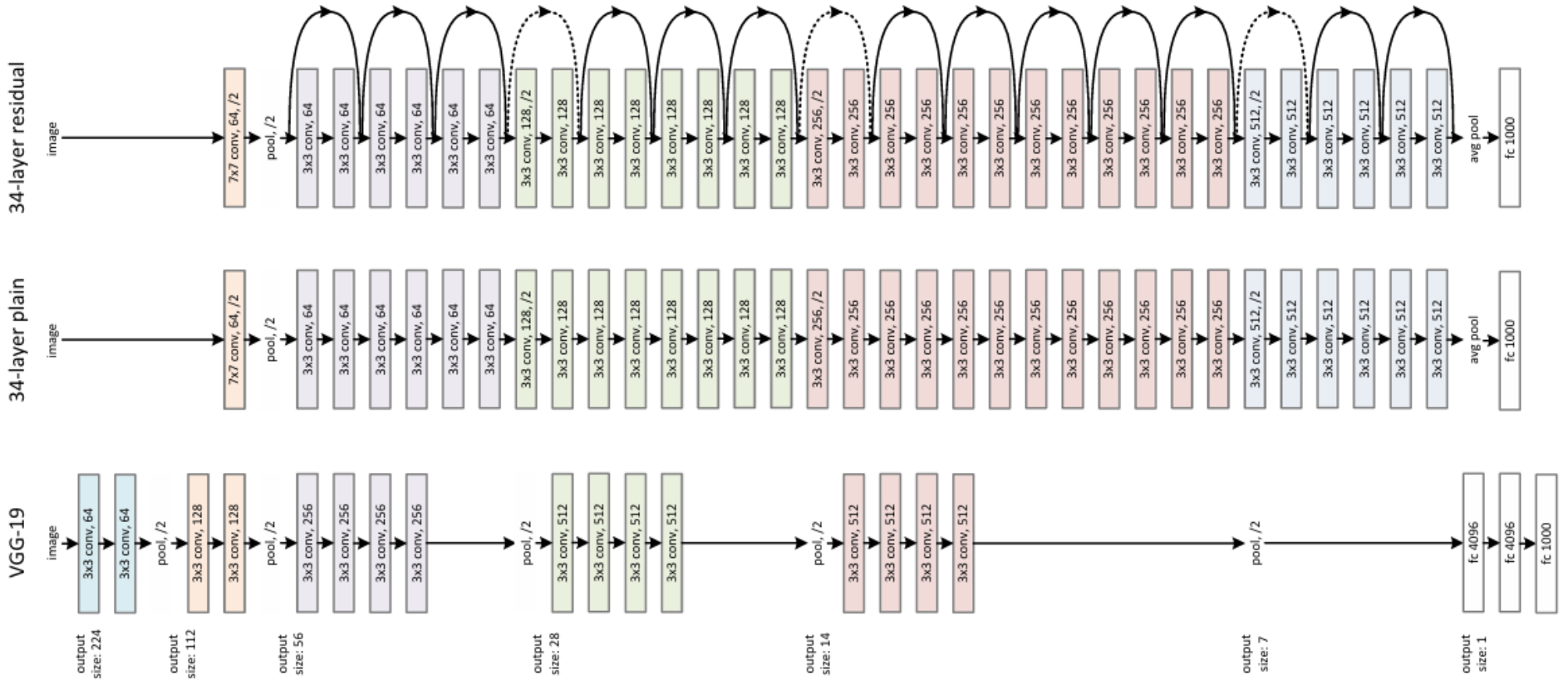


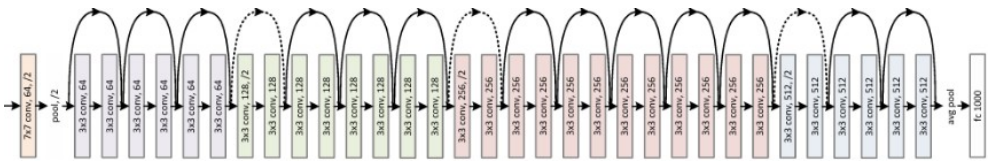
Figure 3. Example network architectures for ImageNet. Left: the VGG-19 model [41] (19.6 billion FLOPs) as a reference. Middle: a plain network with 34 parameter layers (3.6 billion FLOPs). Right: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. Table 1 shows more details and other variants.

MX-AI

Architecture of Residual Network

$$f=\frac{n+2p-k}{s}+1$$

(224, 224, 3)



$$\frac{224+2\times3-7}{2}+1=112 \qquad \frac{112+2\times1-3}{2}+1=56$$

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2 zeropadding: 3 x 3				
conv2_x	56×56	3×3 max pool, stride 2 zeropadding: 1 x 1				
		[3×3, 64; 3×3, 64] ×2	[3×3, 64; 3×3, 64] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3
conv3_x	28×28	[3×3, 128; 3×3, 128] ×2	[3×3, 128; 3×3, 128] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×8
conv4_x	14×14	[3×3, 256; 3×3, 256] ×2	[3×3, 256; 3×3, 256] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×23	[1×1, 256; 3×3, 256; 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512; 3×3, 512] ×2	[3×3, 512; 3×3, 512] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Table1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig.5), with the numbers of blocks stacked. Down sampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.

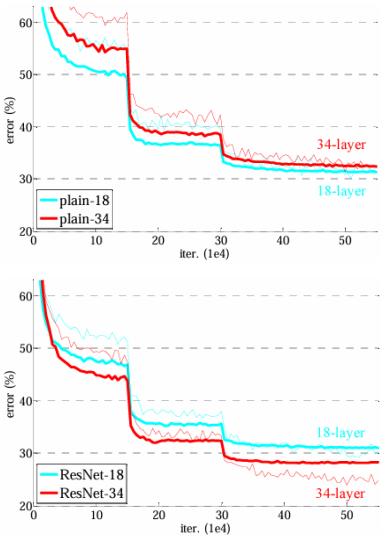


Figure4. Training on ImageNet. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

MX-AI

ResNet for CIFAR-10

4.2.CIFAR-10 and Analysis

We conducted more studies on the CIFAR-10 dataset[20], which consists of 50k training images and 10k testing images in 10 classes. We present experiments trained on the training set and evaluated on the test set. Our focus is on the behaviors of extremely deep networks, but not on pushing the state-of-the-art results, so we intentionally use simple architectures as follows.

The plain/residual architectures follow the form in Fig.3 (middle/right). The network inputs are 32 x 32 images, with the per-pixel mean subtracted. The first layer is 3 x 3 convolutions. Then we use a stack of 6n layers with 3 x 3 convolutions on the feature maps of sizes {32, 16, 8} respectively, with 2n layers for each feature map size. The numbers of filters are {16, 32, 64} respectively. The subsampling is performed by convolutions with a stride of 2. Then network ends with a global average pooling, a 10-way fully-connected layer, and softmax. There are totally 6n+2 stacked weighted layers. The following table summarizes the architecture:

output map size	32 x 32	16 x 16	8 x 8
# layers	1 + 2n	2n	2n
# filters	16	32	64

When shortcut connections are used, they are connected to the pairs of 3 x 3 layers (totally 3n shortcuts). On this dataset we use identity shortcuts in all cases (i.e., option A), so our residual models have exactly the same depth, width, and number of parameters as the plain counterparts.

We use a weight decay of 0.0001 and momentum of 0.9, and adopt the weight initialization in [13] and BN [16] but with no dropout. These models are trained with a mini batch size of 128 on two GPUs. We start with a learning rate of 0.1, divide it by 10 at 32k and 48k iterations, and terminate training at 64k iterations, which is determined on a 45k/5k train/val split. We follow the simple data augmentation in [24]for training: 4 pixels are padded on each side, and a 32 x 32 crop is randomly sampled from the padded image or its horizontal flip. For testing, we only evaluate the single view of the original 32 x 32 image.

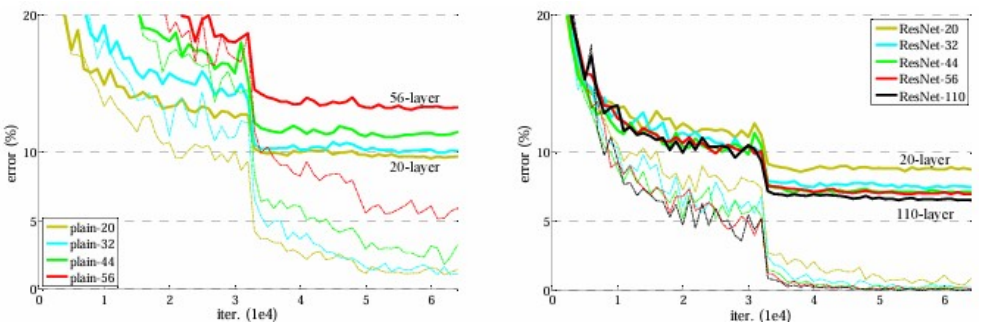


Figure6. Training on CIFAR-10. Dashed lines denote training error, and bold lines denote testing error. Left: plain networks. The error of plain-110 is higher than 60% and not displayed. Right:ResNets.

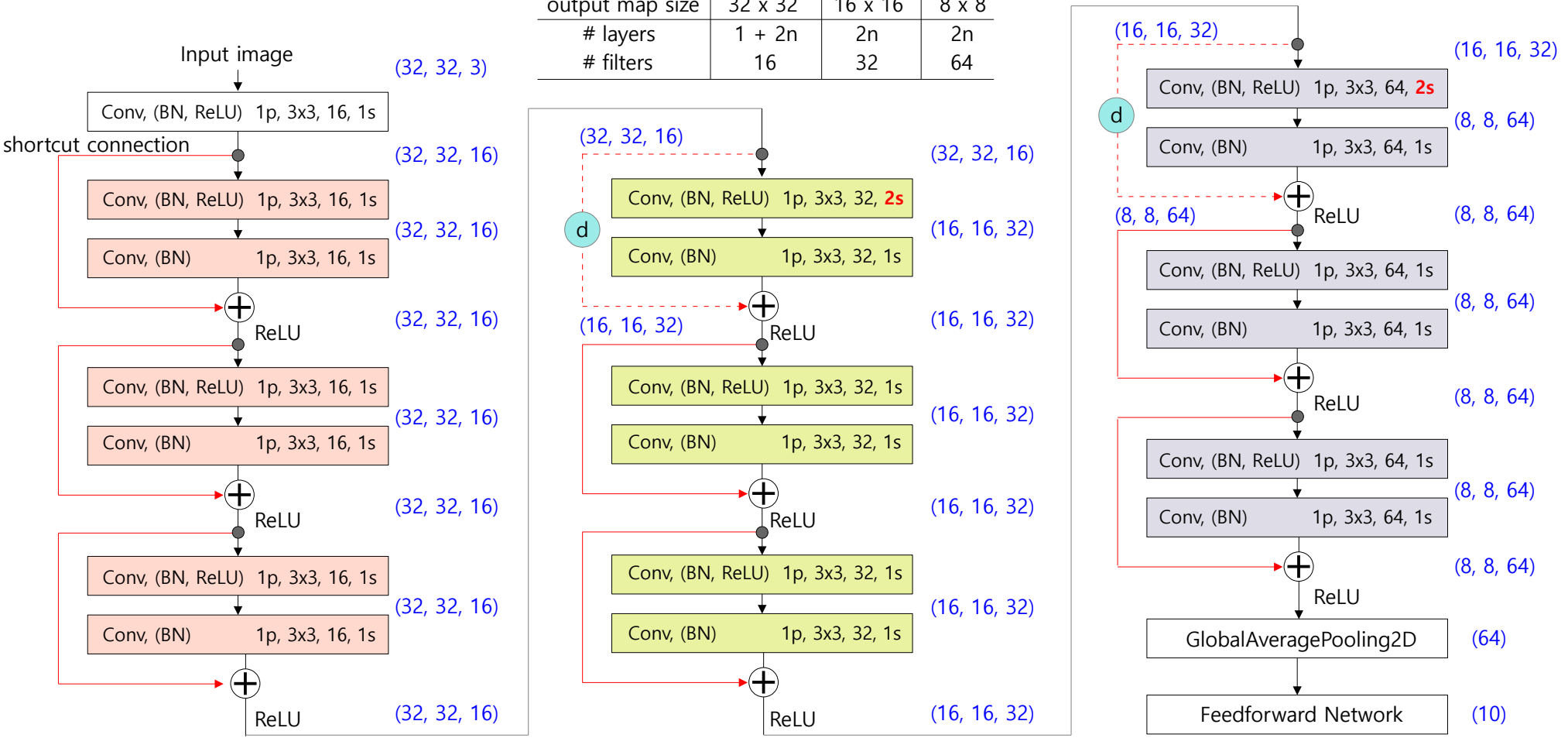
MX-AI

■ ResNet20 for CIFAR-10

(n=3)

output map size	32 x 32	16 x 16	8 x 8
# layers	1 + 2n	2n	2n
# filters	16	32	64

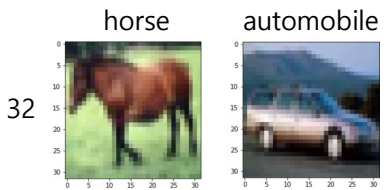
d - down sampling : Conv (1 x 1), no padding



MX-AI

- Implementation of a ResNet20 model for CIFAR10 classification

```
# [MXDL-12-04] 8.resnet20(cifar10).py
from tensorflow.keras.layers import Input, Dense, Conv2D, Activation
from tensorflow.keras.layers import BatchNormalization, ZeroPadding2D
from tensorflow.keras.layers import Add, GlobalAveragePooling2D
from tensorflow.keras.layers import RandomFlip, RandomRotation
from tensorflow.keras.layers import RandomZoom
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import matplotlib.pyplot as plt
import pickle

# Residual building block
def ResBlock(x, filters, stride):
    x1 = ZeroPadding2D((1,1))(x)
    x1 = Conv2D(filters, (3,3), stride[0])(x1)
    x1 = BatchNormalization()(x1)
    x1 = Activation('relu')(x1)
    x1 = ZeroPadding2D((1,1))(x1)
    x1 = Conv2D(filters, (3,3), stride[1])(x1)
    x1 = BatchNormalization()(x1)

    # down sampling the input data x to match dimensions
    if stride[0] == 2:
        x2 = Conv2D(filters, (1,1), strides=(2,2))(x)
        x2 = BatchNormalization()(x2)
    else:
        x2 = x
    x3 = Add()([x1, x2])        # shortcut connection
    x3 = Activation('relu')(x3)
    return x3

def ResNet20(x):
    x = ZeroPadding2D((1,1))(x)
    x = Conv2D(16, (3,3), (1,1))(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = ResBlock(x, 16, (1, 1))
    x = ResBlock(x, 16, (1, 1))
    x = ResBlock(x, 16, (1, 1))
    x = ResBlock(x, 32, (2, 1))
    x = ResBlock(x, 32, (1, 1))
    x = ResBlock(x, 32, (1, 1))
    x = ResBlock(x, 64, (2, 1))
    x = ResBlock(x, 64, (1, 1))
    x = ResBlock(x, 64, (1, 1))
    return x

# Read a CIFAR10 dataset
with open('data/cifar10.pkl', 'rb') as f:
      x_train, y_train, x_test, y_test = pickle.load(f)

categories = ['airplane', 'automobile', 'bird', 'cat', 'deer',
              'dog', 'frog', 'horse', 'ship', 'truck']
```



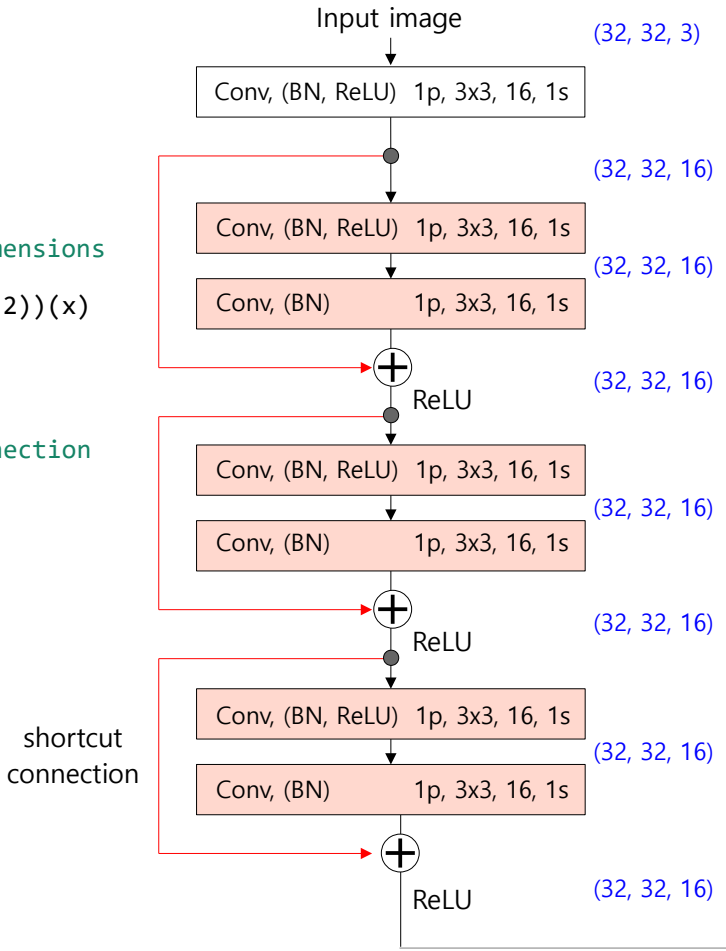
MX-AI

- Implementation of a ResNet20 model for CIFAR10 classification

```
# Build a ResNet20 model for CIFAR10
x_input = Input(batch_shape=(None, *x_train.shape[1:]))
x = RandomFlip(mode='horizontal_and_vertical')(x_input)
x = RandomRotation(0.2)(x)
x = RandomZoom(0.1)(x)
h = ResNet20(x)
h = GlobalAveragePooling2D()(h)
y_output = Dense(10, activation='softmax')(h)

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

# Training
hist = model.fit(x_train, y_train,
                 epochs = 1000,
                 batch_size = 256,
                 validation_data=(x_test, y_test),
                 shuffle = True)

# Loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'][0:], color='blue', label='train loss')
plt.plot(hist.history['val_loss'][0:], color='red', label='test loss')
plt.legend()
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()

# Accuracy for test data
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 for the test data = {:.4f}'.format(acc))

# 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):
    ax[i].imshow(x_test[miss])
    ax[i].axis('off')
    s_test = categories[y_test[miss][0]]
    s_pred = categories[y_pred[miss][0]]
    ax[i].set_title(s_test + ' /\n' + s_pred)
plt.show()
```

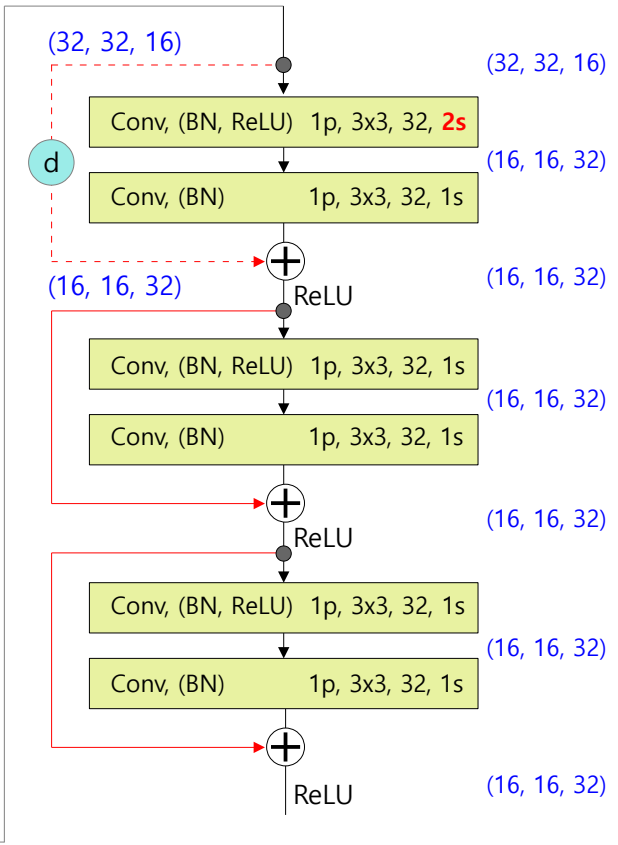
- Implementation of a ResNet20 model for CIFAR10 classification

```
def ResBlock(x, filters, stride):
    x1 = ZeroPadding2D((1,1))(x)
    x1 = Conv2D(filters, (3,3), stride[0])(x1)
    x1 = BatchNormalization()(x1)
    x1 = Activation('relu')(x1)
    x1 = ZeroPadding2D((1,1))(x1)
    x1 = Conv2D(filters, (3,3), stride[1])(x1)
    x1 = BatchNormalization()(x1)

    # down sampling the input data to match dimensions
    if stride[0] == 2:
        x2 = Conv2D(filters, (1,1), strides=(2,2))(x)
        x2 = BatchNormalization()(x2)
    else:
        x2 = x

    x3 = Add()([x1, x2])         # shortcut connection
    x3 = Activation('relu')(x3)
    return x3

def ResNet20(x):
    x = ZeroPadding2D((1,1))(x)
    x = Conv2D(16, (3,3), (1,1))(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = ResBlock(x, 16, (1, 1))
    x = ResBlock(x, 16, (1, 1))
    x = ResBlock(x, 16, (1, 1))
    x = ResBlock(x, 32, (2, 1))
    x = ResBlock(x, 32, (1, 1))
    x = ResBlock(x, 32, (1, 1))
    x = ResBlock(x, 64, (2, 1))
    ...
```



(n=3)

output map size	32 x 32	16 x 16	8 x 8
# layers	1 + 2n	2n	2n
# filters	16	32	64

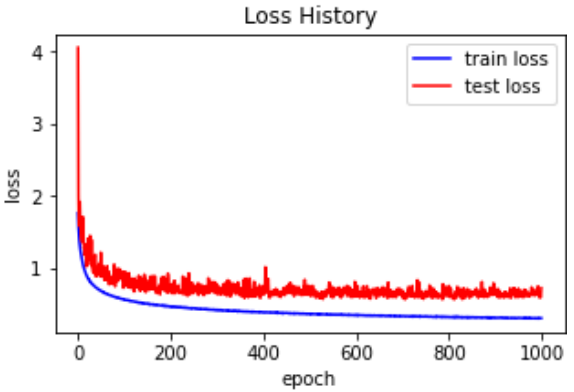


MX-AI

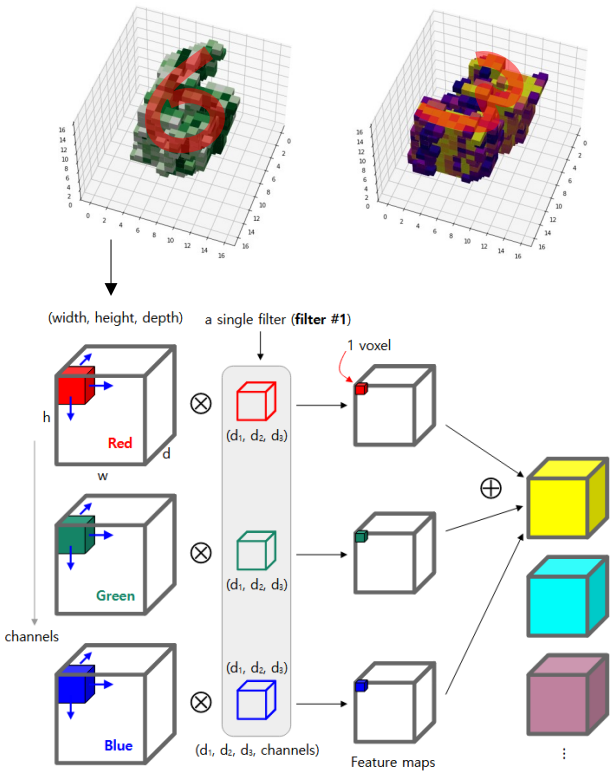
- Implementation of a ResNet20 model for CIFAR10 classification

```
Epoch 1/1000
196/196 [==========] - 19s 52ms/step - loss: 1.7786 - val_loss: 2.7941
Epoch 2/1000
196/196 [==========] - 8s 42ms/step - loss: 1.5664 - val_loss: 1.9773
Epoch 3/1000
196/196 [==========] - 8s 43ms/step - loss: 1.4416 - val_loss: 1.6360
Epoch 4/1000
196/196 [==========] - 8s 43ms/step - loss: 1.3471 - val_loss: 1.4293
Epoch 5/1000
196/196 [==========] - 8s 42ms/step - loss: 1.2871 - val_loss: 1.6675
...

Epoch 991/1000
196/196 [==========] - 8s 42ms/step - loss: 0.1947 - val_loss: 0.5820
Epoch 992/1000
196/196 [==========] - 8s 41ms/step - loss: 0.1962 - val_loss: 0.6007
Epoch 993/1000
196/196 [==========] - 8s 42ms/step - loss: 0.1965 - val_loss: 0.6318
Epoch 994/1000
196/196 [==========] - 8s 42ms/step - loss: 0.1936 - val_loss: 0.5971
Epoch 995/1000
196/196 [==========] - 8s 42ms/step - loss: 0.1964 - val_loss: 0.6267
Epoch 996/1000
196/196 [==========] - 8s 42ms/step - loss: 0.1917 - val_loss: 0.6895
Epoch 997/1000
196/196 [==========] - 8s 42ms/step - loss: 0.1998 - val_loss: 0.6190
Epoch 998/1000
196/196 [==========] - 8s 42ms/step - loss: 0.1955 - val_loss: 0.5656
Epoch 999/1000
196/196 [==========] - 8s 42ms/step - loss: 0.1933 - val_loss: 0.6705
Epoch 1000/1000
196/196 [==========] - 8s 42ms/step - loss: 0.1959 - val_loss: 0.6385
```

Accuracy for the test data = 0.8324



* Misclassified images



12. Convolutional Neural Networks (CNN)

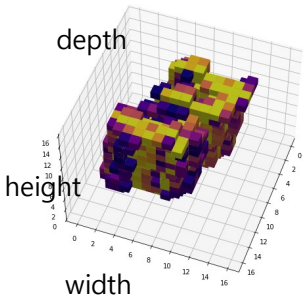
Part 5: 3D Convolution

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

www.youtube.com/@meanxai

MX-AI

■ 3D convolution



a single 4D RGB image
with 3 channels.

(width, height, depth, channels)

(width, height, depth)

a single filter (**filter #1**)

1 voxel

h

w

d

Red

Green

Blue

channels

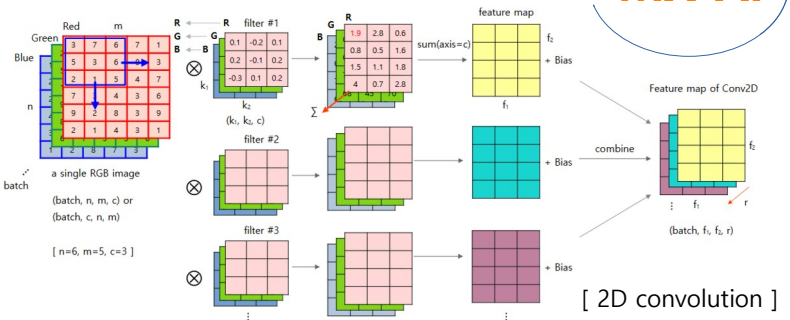
(d_1, d_2, d_3)

(d_1, d_2, d_3)

(d_1, d_2, d_3)

$(d_1, d_2, d_3, \text{channels})$

Feature maps



[2D convolution]

+ Bias → Feature map from the **filter #1**

+ Bias → Feature map from the **filter #2**

+ Bias → Feature map from the **filter #3**

filters

Final feature map of 3D convolution:
(None, dim1, dim2, dim3, filters)

Multiple 3D RGB images:
(None, width, height, depth, channels)
or (None, channels, width, height, depth)

MX-AI

■ 3D MNIST dataset

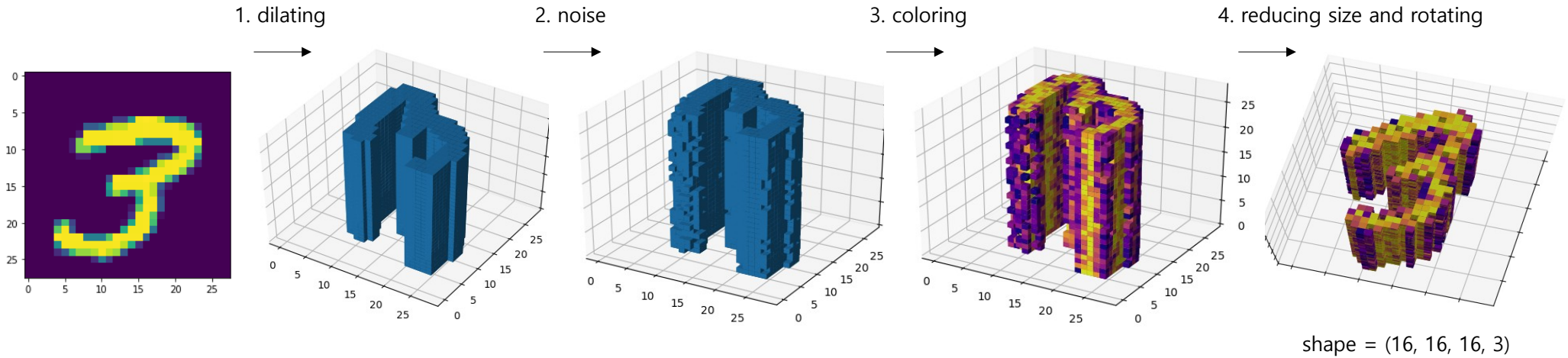
1) 3D MNIST: A 3D version of the MNIST database of handwritten digits (https://www.kaggle.com/datasets/daavoo/3d-mnist)

2) Augmented MNIST 3D: 3D version of MNIST with augmented features like rotation and colors (https://www.kaggle.com/datasets/doleron/augmented-mnist-3d)

This dataset consists of an augmented 3D version of MNIST handwritten digit dataset.

- GIthub repo: https://github.com/doleron/augmented_3d_mnist
- medium story: https://medium.com/@doleron/start-using-3d-images-today-845f959a6b0b

Augmentation: The following transformations are applied in order to generate a 3D digit from the a 2D image:



[source : https://www.kaggle.com/datasets/doleron/augmented-mnist-3d]

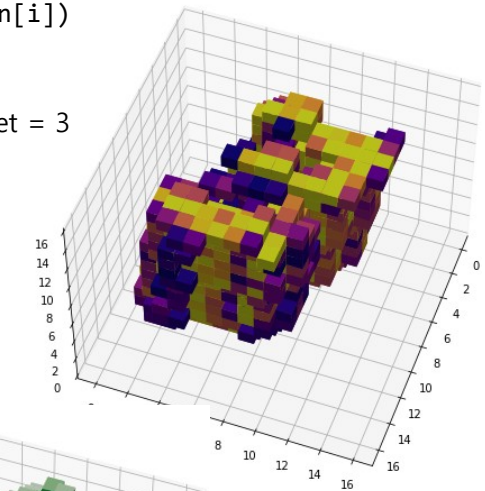
MX-AI

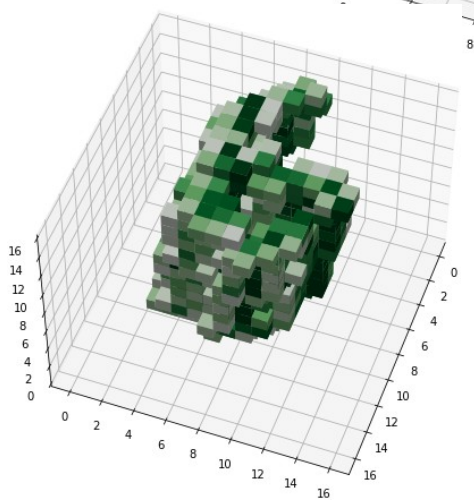
■ 3D MNIST dataset

```
# [MXDL-12-05] 9.3d_mnist(data).py
# Code source: https://github.com/doleron/augmented_3d_mnist/blob/main/mnist_3d.ipynb
# Data source: https://www.kaggle.com/datasets/doleron/augmented-mnist-3d
import numpy as np
import h5py
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D

df = h5py.File('data/3d-mnist-luiz.h5', 'r')
x_train = np.array(df["train_x"])  # (60000, 16, 16, 16, 3)
x_test  = np.array(df["test_x"])   # (10000, 16, 16, 16, 3)
y_train = np.array(df["train_y"])  # (60000,)
y_test  = np.array(df["test_y"])   # (10000,)

def print_grid(grid):
    grid_shape = grid.shape
    flattened = grid.reshape(((grid_shape[0] * grid_shape[1] * grid_shape[2]), 3))
    voxel_grid_array = np.zeros(len(flattened))

    for i in range(len(flattened)):
        temp = flattened[i]
        if temp[0] > 0 or temp[1] > 0 or temp[2] > 0:
            voxel_grid_array[i] = 1

    voxel_grid = voxel_grid_array.reshape((grid_shape[0], grid_shape[1],
                                           grid_shape[2]))

    fig = pyplot.figure(figsize=(8,8))
    ax = fig.add_subplot(projection='3d')
    ax.azim = 20
    ax.elev = 50
    ax.voxels(voxel_grid, facecolors=grid)
    pyplot.show()
```

```
# Visually examine one 3D image.
i = 7
print_grid(x_train[i])
print(y_train[i])
```

Target = 3





Target = 6

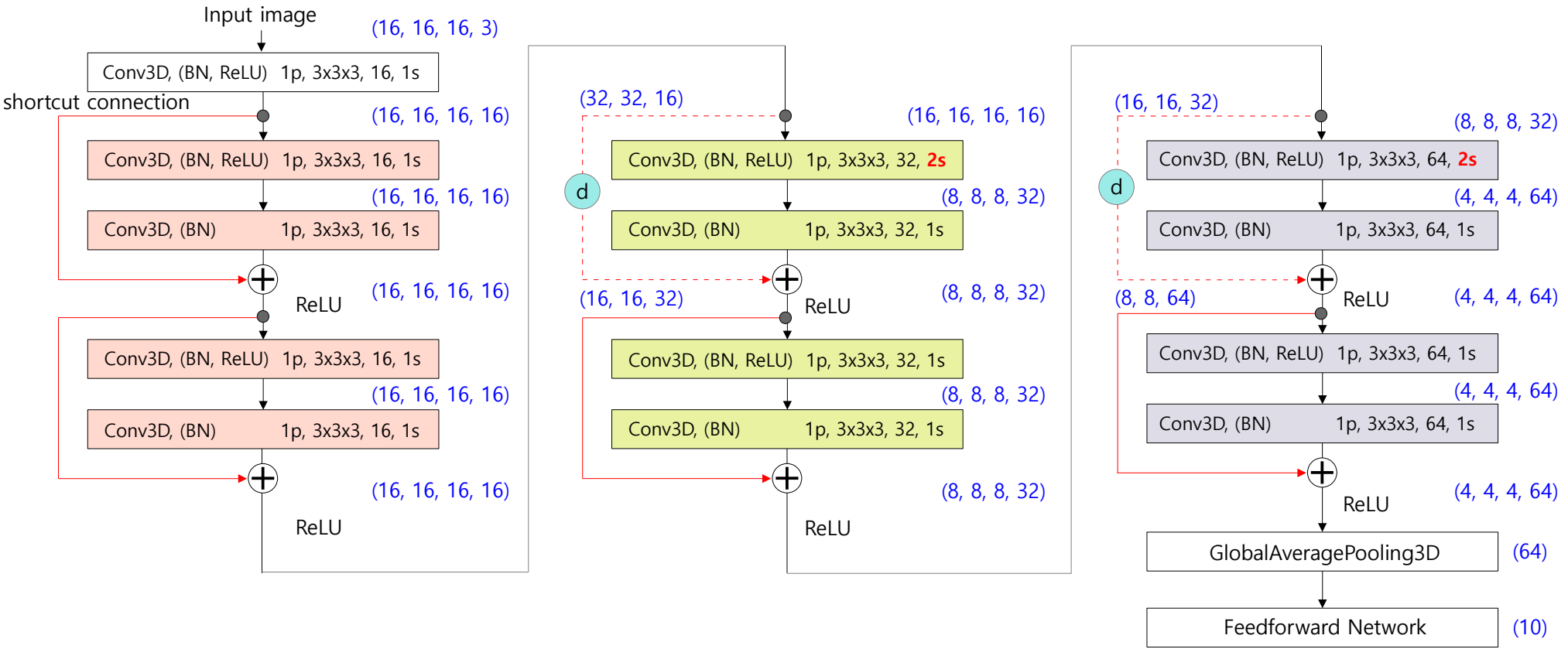
MX-AI

- ResNet14 for 3D MNIST

(n=2)

output map size	16 x 16	8 x 8	4 x 4
# layers	1 + 2n	2n	2n
# filters	16	32	64

d - down sampling : Conv (1 x 1), no padding



MX-AI

- 3D MNIST image classification using 3D ResNet14

```
# [MXDL-12-05] 10.3d_mnist(resnet14).py
# Data source: https://www.kaggle.com/datasets/doleron/augmented-mnist-3d
from tensorflow.keras.layers import Input, Dense, Conv3D, Activation
from tensorflow.keras.layers import BatchNormalization, ZeroPadding3D
from tensorflow.keras.layers import Add, GlobalAveragePooling3D
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import numpy as np
import h5py
import matplotlib.pyplot as plt

def ResBlock(x, filters, stride):
    x1 = ZeroPadding3D((1,1,1))(x)
    x1 = Conv3D(filters, (3,3,3), stride[0])(x1)
    x1 = BatchNormalization()(x1)
    x1 = Activation('relu')(x1)
    x1 = ZeroPadding3D((1,1,1))(x1)
    x1 = Conv3D(filters, (3,3,3), stride[1])(x1)
    x1 = BatchNormalization()(x1)

    if stride[0] == 2:  # down sampling the input data to match dimensions
        x2 = Conv3D(filters, (1,1,1), strides=(2,2,2))(x)
        x2 = BatchNormalization()(x2)
    else:
        x2 = x

    x3 = Add()([x1, x2])        # shortcut connection
    x3 = Activation('relu')(x3)
    return x3
```

```
def ResNet14(x):
    x = ZeroPadding3D((1,1,1))(x)
    x = Conv3D(16, (3,3,3), (1,1,1))(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = ResBlock(x, 16, (1, 1))
    x = ResBlock(x, 16, (1, 1))
    x = ResBlock(x, 32, (2, 1))
    x = ResBlock(x, 32, (1, 1))
    x = ResBlock(x, 64, (2, 1))
    x = ResBlock(x, 64, (1, 1))
    return x

# Load the 3D MNIST dataset.
df = h5py.File('data/3d-mnist-luiz.h5', 'r')
x_train = np.array(df["train_x"])  # (60000, 16, 16, 16, 3)
x_test  = np.array(df["test_x"])   # (10000, 16, 16, 16, 3)
y_train = np.array(df["train_y"])  # (60000,)
y_test  = np.array(df["test_y"])   # (10000,)

# Build a ResNet14 model for 3D MNIST classification
x_input = Input(batch_shape=(None, *x_train.shape[1:]))
h = ResNet14(x_input)
h = GlobalAveragePooling3D()(h)
y_output = Dense(10, activation='softmax')(h)

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

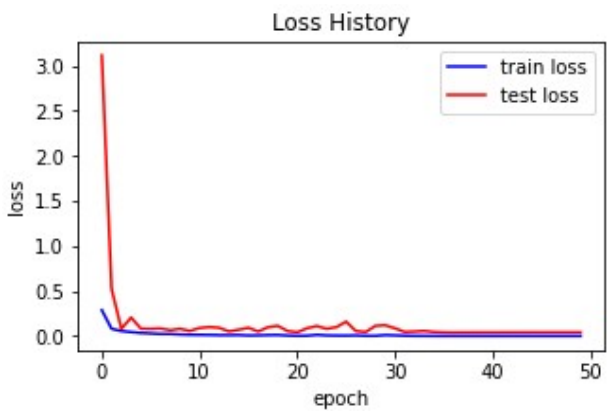
MX-AI

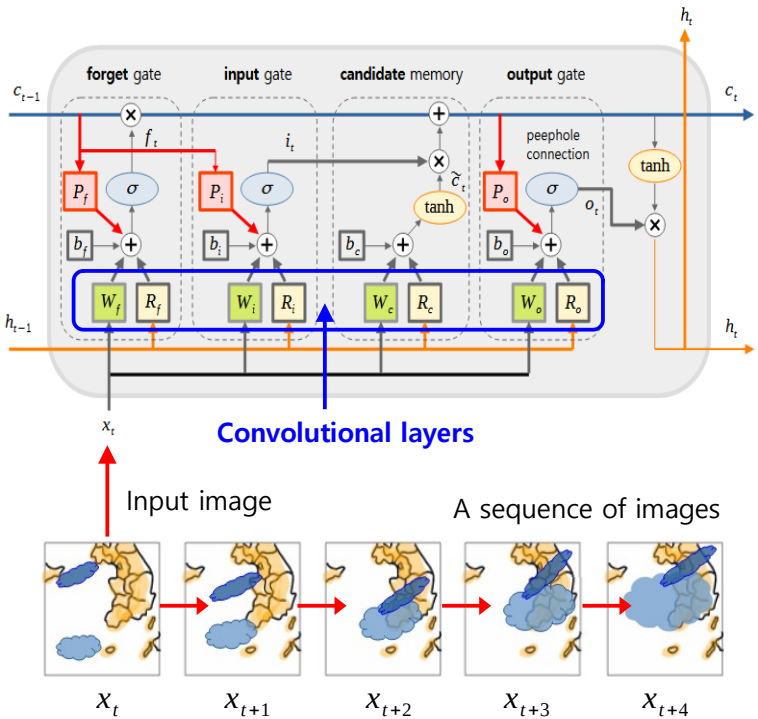
- 3D MNIST image classification using 3D ResNet14

```
# Training
hist = model.fit(x_train, y_train,
                 epochs = 50,
                 batch_size = 256,
                 validation_data=(x_test, y_test),
                 shuffle = True)

# Loss history
plt.figure(figsize=(5, 3))
plt.plot(hist.history['loss'][0:], color='blue',
         label='train loss')
plt.plot(hist.history['val_loss'][0:], color='red',
         label='test loss')
plt.legend()
plt.title("Loss History")
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()

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

```
Epoch 1/50
235/235 [==========] - 50s 167ms/step - loss: 0.2870 - val_loss: 3.1130
Epoch 2/50
235/235 [==========] - 36s 151ms/step - loss: 0.0802 - val_loss: 0.5210
Epoch 3/50
235/235 [==========] - 36s 151ms/step - loss: 0.0554 - val_loss: 0.0787
…
Epoch 46/50
235/235 [==========] - 35s 151ms/step - loss: 1.4385e-05 - val_loss: 0.0402
Epoch 47/50
235/235 [==========] - 35s 151ms/step - loss: 1.2839e-05 - val_loss: 0.0404
Epoch 48/50
235/235 [==========] - 35s 151ms/step - loss: 1.1213e-05 - val_loss: 0.0403
Epoch 49/50
235/235 [==========] - 35s 151ms/step - loss: 1.0902e-05 - val_loss: 0.0404
Epoch 50/50
235/235 [==========] - 35s 151ms/step - loss: 1.0606e-05 - val_loss: 0.0407
```

Accuracy of the test data = 0.9902





12. Convolutional Neural Networks (CNN)

Part 6: Convolutional LSTM

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

www.youtube.com/@meanxai

MX-AI

■ Convolutional LSTM: Paper

- The Convolutional LSTM was first introduced by Xingjian Shi et al. in their 2015 paper titled "Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting".

Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting

Xingjian Shi **Zhourong Chen** **Hao Wang** **Dit-Yan Yeung**
Department of Computer Science and Engineering, Hong Kong University of Science and Technology
{xshiab,zchenbb,hwangaz,dyyeung}@cse.ust.hk

Wai-kin Wong **Wang-chun Woo**
Hong Kong Observatory, Hong Kong, China
{wkwong,wcwoo}@hko.gov.hk

Abstract

The goal of precipitation nowcasting is to predict the future rainfall intensity in a local region over a relatively short period of time. Very few previous studies have examined this crucial and challenging weather forecasting problem from the machine learning perspective. In this paper, we formulate precipitation nowcasting as a spatiotemporal sequence forecasting problem in which both the input and the prediction target are spatiotemporal sequences. By extending the fully connected LSTM (FC-LSTM) to have convolutional structures in both the input-to-state and state-to-state transitions, we propose the **convolutional LSTM (ConvLSTM)** and use it to build an end-to-end trainable model for the precipitation nowcasting problem. Experiments show that our ConvLSTM network captures spatiotemporal correlations better and consistently outperforms FC-LSTM and the state-of-the-art operational ROVER algorithm for precipitation nowcasting.

2 Preliminaries

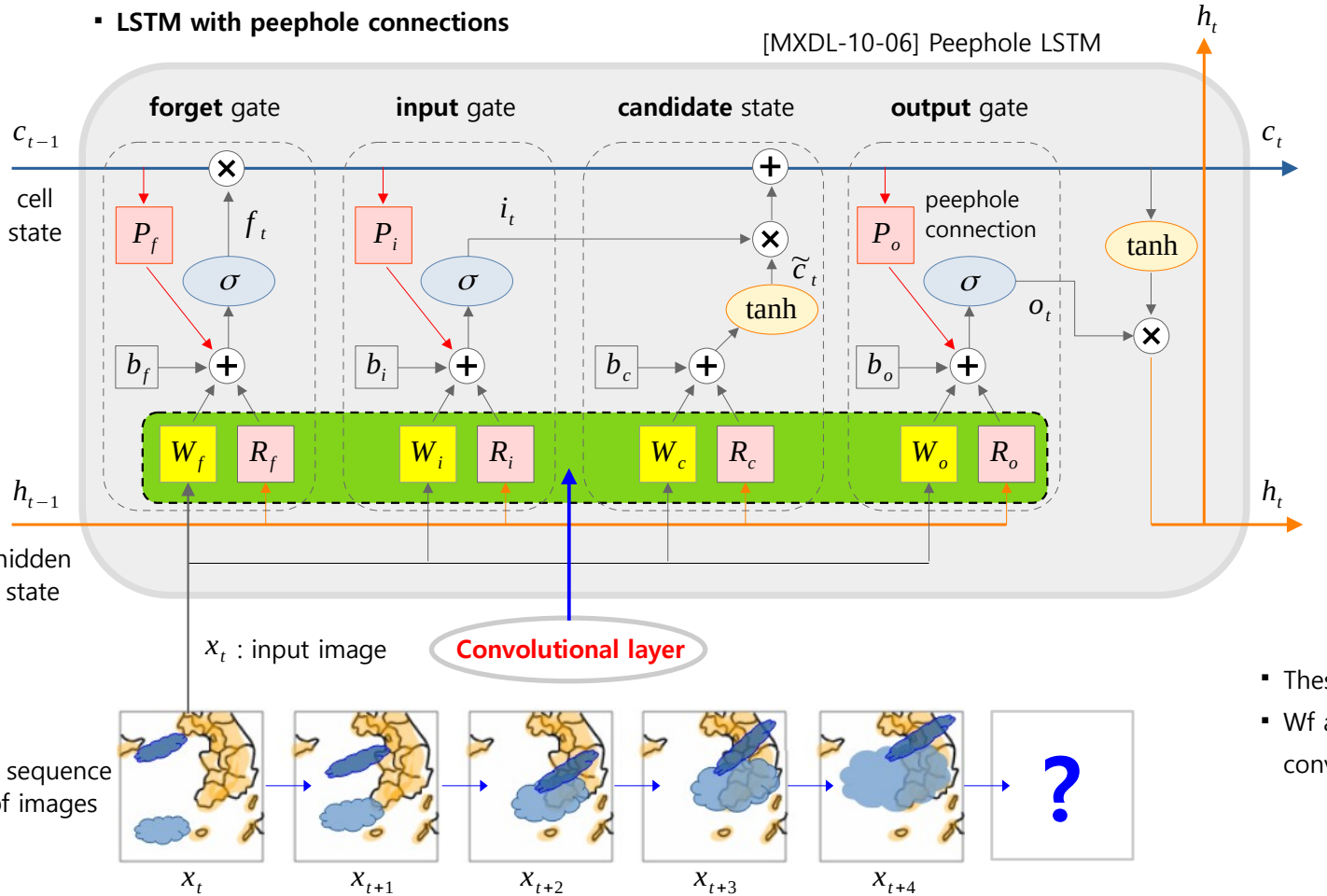
2.1 Formulation of Precipitation Nowcasting Problem

The goal of precipitation nowcasting is to use the previously observed radar echo sequence to forecast a fixed length of the future radar maps in a local region (e.g., Hong Kong, New York, or Tokyo). In real applications, the radar maps are usually taken from the weather radar every 6-10 minutes and nowcasting is done for the following 1-6 hours, i.e., to predict the 6-60 frames ahead. From the machine learning perspective, this problem can be regarded as a spatiotemporal sequence forecasting problem.

MX-AI

The Architecture of the Convolutional LSTM

▪ **LSTM with peephole connections**



▪ **Convolutional LSTM**

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

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

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

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

$$o_t = \sigma(x_t * W_o + h_{t-1} * R_o + c_{t-1} \odot P_o + b_o)$$

$*$: **convolution operator**

$\odot$: element-wise (or hadamard) product

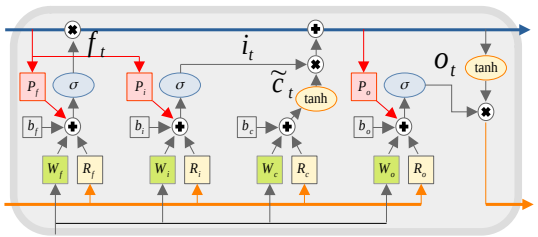
- These formulas are given in equation (3) of the paper.
- Wf and Rf correspond to the filters of the convolutional layer.

MX-AI

■ Build a 2D Convolutional LSTM class

[MXDL-10-06]

```
class PeepholeLSTM(Layer):
  def __init__(self, n_hidden):
    super().__init__()
    self.nh = n_hidden

    # weights and bias for data x
    self.Wf = Dense(n_hidden)  # forget
    self.Wi = Dense(n_hidden)  # input
    self.Wc = Dense(n_hidden)  # candidate state
    self.Wo = Dense(n_hidden)  # output

    self.Rf = Dense(n_hidden,
    self.Ri = Dense(n_hidden,
    self.Rc = Dense(n_hidden,
    self.Ro = Dense(n_hidden,

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

  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_gate= tf.math.tanh(self.Wc(x) + self.Rc(h))
    o_gate= tf.math.sigmoid(self.Wo(x)+self.Ro(h)+self.Po(c))
    c_stat = c * f_gate + c_gate * i_gate
    h_stat = tf.math.tanh(c_stat) * o_gate
    return h_stat, c_stat

  def call(self, x):
    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
```




```
class MyConvLSTM2D(Layer):
  def __init__(self, filters, kernel_size, pad, d):  # d: input dims
    super().__init__()
    self.nh = filters        # input dims = (batch, time step, height, width, color)

    # Convolutional layers
    self.Wf = Conv2D(filters, kernel_size, padding=pad)  # forget
    self.Wi = Conv2D(filters, kernel_size, padding=pad)  # input
    self.Wc = Conv2D(filters, kernel_size, padding=pad)  # candidate
    self.Wo = Conv2D(filters, kernel_size, padding=pad)  # output
    self.Rf = Conv2D(filters, kernel_size, padding=pad)  # forget
    self.Ri = Conv2D(filters, kernel_size, padding=pad)  # input
    self.Rc = Conv2D(filters, kernel_size, padding=pad)  # candidate
    self.Ro = Conv2D(filters, kernel_size, padding=pad)  # output

    # Peephole connections
    w_init = tf.random_normal_initializer()
    self.Pf = tf.Variable(w_init([1,d[2],d[3],filters]),trainable=True)
    self.Pi = tf.Variable(w_init([1,d[2],d[3],filters]),trainable=True)
    self.Po = tf.Variable(w_init([1,d[2],d[3],filters]),trainable=True)

  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_gate = tf.math.tanh(self.Wc(x) + self.Rc(h))
    o_gate = tf.math.sigmoid(self.Wo(x) + self.Ro(h) + self.Po * c)
    c_stat = c * f_gate + c_gate * i_gate
    h_stat = tf.math.tanh(c_stat) * o_gate
    return h_stat, c_stat

  def call(self, x):
    h=tf.zeros(shape=(tf.shape(x)[0], x.shape[2], x.shape[3], self.nh))
    c=tf.zeros(shape=(tf.shape(x)[0], x.shape[2], x.shape[3], 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
```

MX-AI

■ Moving MNIST dataset

- The moving MNIST dataset was introduced by Nitish Srivastava et al. in their 2015 paper "Unsupervised Learning of Video Representations Using LSTMs".

Unsupervised Learning of Video Representations using LSTMs

Nitish Srivastava NITISH@CS.TORONTO.EDU
Elman Mansimov EMANSIM@CS.TORONTO.EDU
Ruslan Salakhutdinov RSALAKHU@CS.TORONTO.EDU
University of Toronto, 6 Kings College Road, Toronto, ON M5S 3G4 CANADA

Abstract

We use Long Short Term Memory (LSTM) networks to learn representations of video sequences. Our model uses an encoder LSTM to map an input sequence into a fixed length representation. This representation is decoded using single or multiple decoder LSTMs to perform different tasks, such as reconstructing the input sequence, or predicting the future sequence. We experiment with two kinds of input sequences – patches of image pixels and high-level representations ("percepts") of video frames extracted using a pretrained convolutional net. We explore different design choices such as whether the decoder LSTMs should condition on the generated output. We analyze the outputs of the model qualitatively to see how well the model can extrapolate the learned video representation into the future and into the past. We further evaluate the representations by finetuning them for a supervised learning problem – human action recognition on the UCF-101 and HMDB-51 datasets. We show that the representations help improve classification accuracy, especially when there are only few training examples. Even models pretrained on unrelated datasets (300 hours of YouTube videos) can help action recognition performance.

http://www.cs.toronto.edu/~nitish/unsupervised_video/

Unsupervised Learning of Video Representations using LSTMs

A test set for evaluating sequence prediction/reconstruction

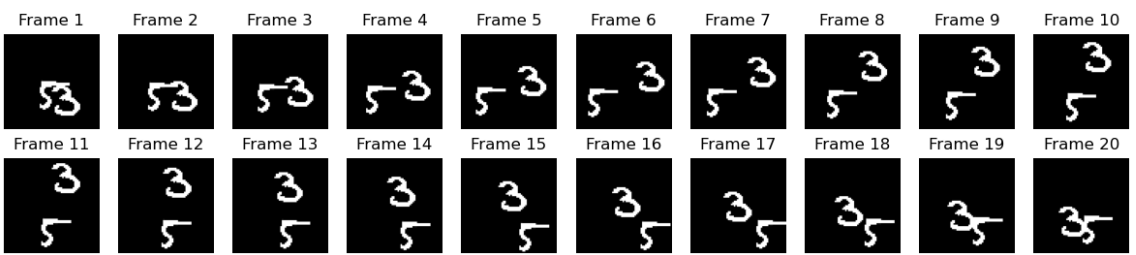
Moving MNIST [782Mb] contains 10,000 sequences each of length 20 showing 2 digits moving in a 64 x 64 frame. The results in the updated arxiv paper use this test set to report numbers. For future prediction, the metric is cross entropy loss for predicting the last 10 frames for each sequence conditioned on the first 10 frames.

Code

unsup_video_lstm.tar.gz [119Kb]

Papers

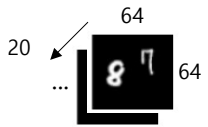
Unsupervised Learning of Video Representations using LSTMs [pdf]
Nitish Srivastava, Elman Mansimov, Ruslan Salakhutdinov, ICML 2015.



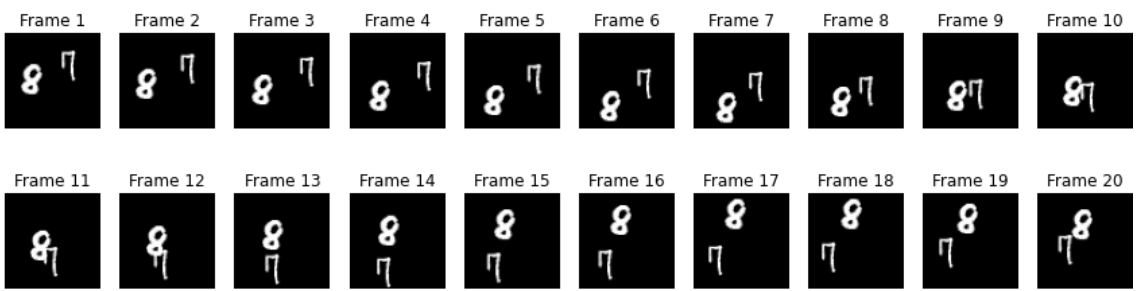
MX-AI

■ Pre-processing the moving MNIST dataset

```
# [MXDL-12-06] 11.moving_mnist.py
# Pre-processing the moving MNIST dataset
#
# data source: http://www.cs.toronto.edu/~nitish/unsupervised_video/
import numpy as np
import matplotlib.pyplot as plt
import pickle

# Load a moving MNIST dataset
data = np.load('data/mnist_test_seq.npy') # (20, 10000, 64, 64)
data = np.swapaxes(data, 0, 1)            # (10000, 20, 64, 64)
data = data[:5000, ...].astype('float32') # (5000, 20, 64, 64)
data /= 255.                              # normalization (0 ~ 1)
data = np.expand_dims(data, axis=-1)      # (5000, 20, 64, 64, 1)

# Split the dataset into training and test data
n_train = int(0.9 * data.shape[0])  # the number of training data
idx = np.arange(data.shape[0])
np.random.shuffle(idx)
d_train = data[idx[: n_train]]
d_test = data[idx[n_train :]]

def create_xy(x):
      t = x.shape[1]  # the number of time steps
      tx = x[:, 0:(t - 1), :, :, :]
      ty = x[:, 1:t, :, :, :]
      return tx, ty
```




```
# Create input and output data to feed into the model.
x_train, y_train = create_xy(d_train) # (4500, 19, 64, 64, 1)
x_test, y_test = create_xy(d_test)    # ( 500, 19, 64, 64, 1)

# Visualize a sequence of images.
fig, axes = plt.subplots(2, 10, figsize=(15, 4))
idx = np.random.choice(n_train, 1)[0]
for i, ax in enumerate(axes.flat):
      ax.imshow(np.squeeze(d_train[idx][i]), cmap="gray")
      ax.set_title(f"Frame {i + 1}")
      ax.axis("off")
plt.show()

# save the training and test data
with open('data/mv_mnist.pkl', 'wb') as f:
      pickle.dump([x_train, y_train, x_test, y_test, d_test], f)
```



MX-AI

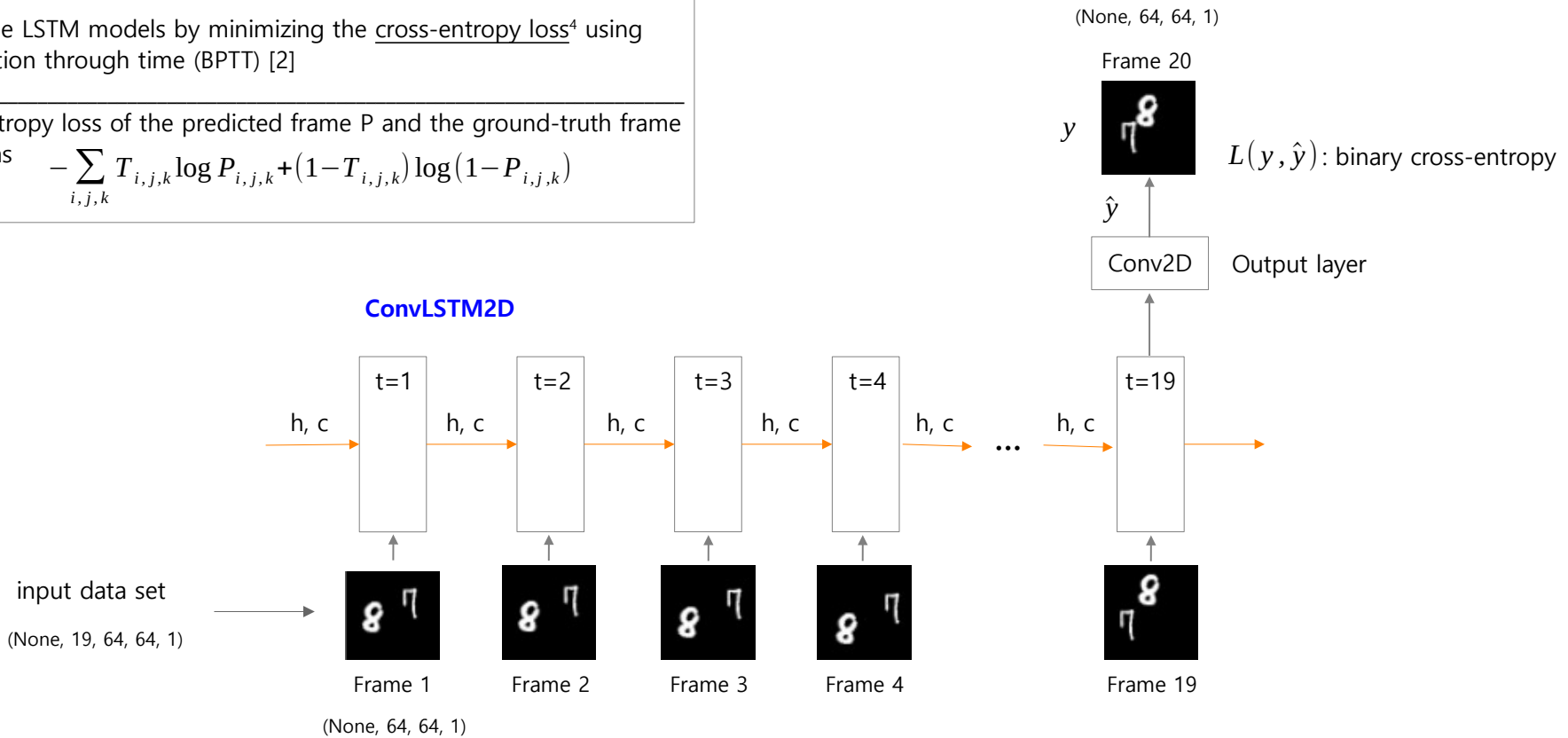
- Many-to-One Convolutional LSTM model for predicting moving MNIST image sequences

4.1 Moving-MNIST Dataset

< ... >

We train all the LSTM models by minimizing the cross-entropy loss[4] using back-propagation through time (BPTT) [2]

[4] The cross-entropy loss of the predicted frame P and the ground-truth frame T is defined as $-\sum_{i,j,k} T_{i,j,k} \log P_{i,j,k} + (1-T_{i,j,k}) \log (1-P_{i,j,k})$

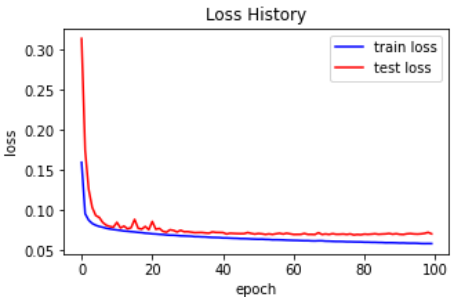


MX-AI

■ Many-to-One Convolutional LSTM model for predicting moving MNIST sequences: Training stage

```
# [MXDL-12-06] 12.conv_lstm_m2o(train).py - Many-to-One model
# Data: http://www.cs.toronto.edu/~nitish/unsupervised_video/
from tensorflow.keras.layers import Input, Conv2D, Activation
from tensorflow.keras.layers import BatchNormalization
from myConvLSTM2D import MyConvLSTM2D
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
import pickle

# Load a moving MNIST dataset
with open('data/mv_mnist.pkl', 'rb') as f:
    x_train, y_train, x_test, y_test, _ = pickle.load(f)

# the last frame of the sequence
y_train = y_train[:, -1, :, :, :] # (4500, 64, 64, 1)
y_test = y_test[:, -1, :, :, :]   # ( 500, 64, 64, 1)

# Build a Convolutional LSTM model
x = Input(shape=(x_train.shape[1], *x_train.shape[2:]))
h = MyConvLSTM2D(filters=32, kernel_size=(5, 5),
                 pad = 'same', d = x_train.shape)(x)
h = BatchNormalization()(h)
h = Activation('relu')(h)
y = Conv2D(filters=1, kernel_size=(3, 3), padding='same',
           activation='sigmoid')(h)

model = Model(x, y)
model.compile(loss='binary_crossentropy',
              optimizer=Adam(learning_rate=0.001))

hist = model.fit(x_train, y_train, batch_size=50,
                 epochs=100, validation_data=(x_test, y_test))
```

```
# Save the trained model
model.save_weights('data/conv_lstm_m2o/weights')

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



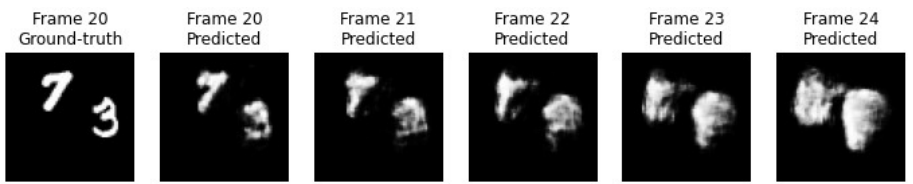

```
 Layer (type)                                 Output Shape              Param #
=================================================================================
 input_1 (InputLayer)                        [(None, 19, 64, 64, 1)]   0
 my_conv_lstm2d (MyConvLSTM2D)                (None, 64, 64, 32)        499072
 batch_normalization (BatchNormalization)     (None, 64, 64, 32)        128
 activation (Activation)                      (None, 64, 64, 32)        0
 conv2d_8 (Conv2D)                            (None, 64, 64, 1)         289
=================================================================================
Total params: 499,489
Trainable params: 499,425
Non-trainable params: 64
_________________________________________________________________________________
Epoch 1/100
90/90 [==============================] - 47s 461ms/step - loss: 0.1599 - val_loss: 0.3140
Epoch 2/100
90/90 [==============================] - 41s 454ms/step - loss: 0.0960 - val_loss: 0.1751
...
Epoch 99/100
90/90 [==============================] - 40s 448ms/step - loss: 0.0589 - val_loss: 0.0727
Epoch 100/100
90/90 [==============================] - 41s 461ms/step - loss: 0.0588 - val_loss: 0.0709
```

MX-AI

■ Many-to-One Convolutional LSTM model for predicting moving MNIST sequences: Prediction stage

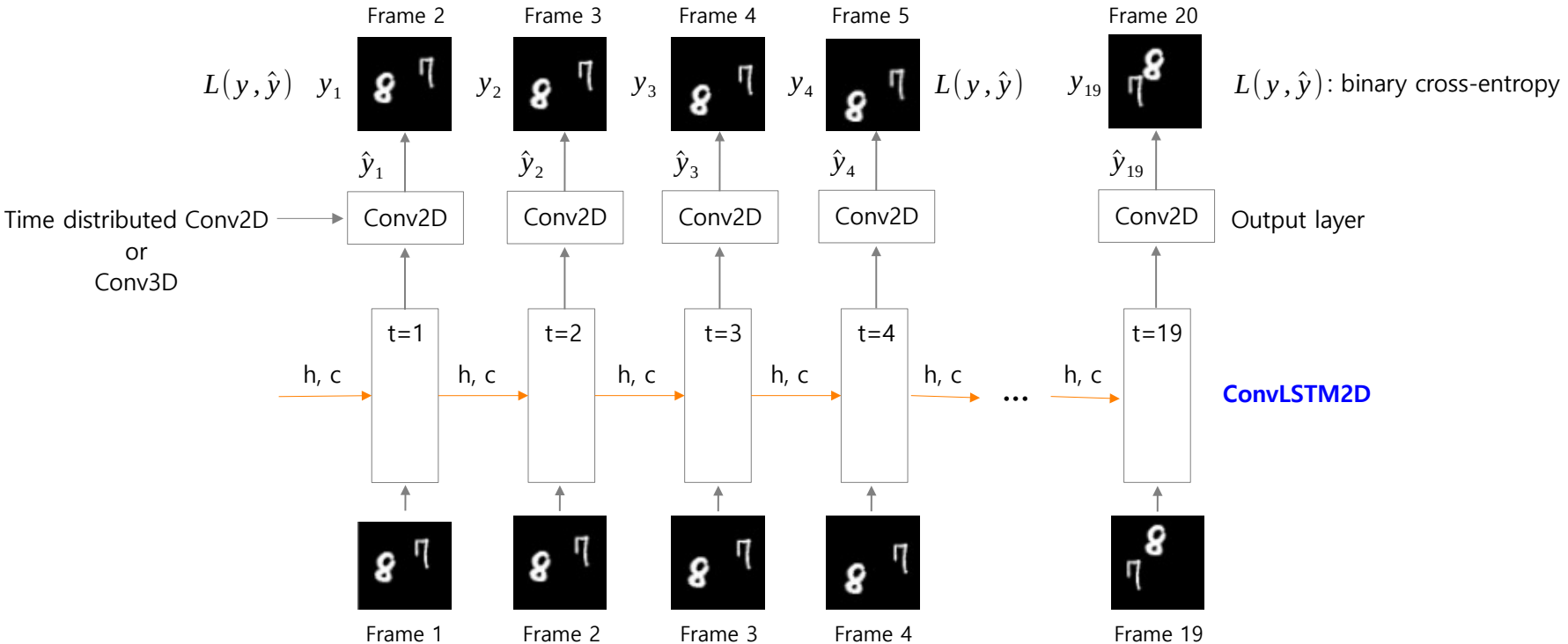
```
# [MXDL-12-06] 13.conv_lstm_m2o(predict).py
# data: http://www.cs.toronto.edu/~nitish/unsupervised_video/
import numpy as np
import pickle
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Input, Conv2D, Activation
from tensorflow.keras.layers import BatchNormalization,
from tensorflow.keras.models import Model
from myConvLSTM2D import MyConvLSTM2D

# Load a moving MNIST dataset
with open('data/mv_mnist.pkl', 'rb') as f:
    x_train, y_train, x_test, y_test, _ = pickle.load(f)

# The last frame of the sequence
y_train = y_train[:, -1, :, :, :]   # (4500, 64, 64, 1)
y_test = y_test[:, -1, :, :, :]     # ( 500, 64, 64, 1)

# Build a Convolutional LSTM model
x = Input(shape=(x_train.shape[1], *x_train.shape[2:]))
h = MyConvLSTM2D(filters=32, kernel_size=(5, 5),
                 pad='same', d=x_train.shape)(x)
h = BatchNormalization()(h)
h = Activation('relu')(h)
y = Conv2D(filters=1, kernel_size=(3, 3),
           padding='same', activation='sigmoid')(h)
model = Model(x, y)

# Load the trained model
model.load_weights('data/conv_lstm_m2o/weights')
```

```
# Predict a sequence of 5 images.
idx = np.random.choice(x_test.shape[0], 1)[0]
x_sample = np.expand_dims(x_test[idx], axis = 0) # (1, 19, 64, 64, 1)

p_result = []
for i in range(5):
    y_pred = model.predict(x_sample, verbose=0)  # (1, 64, 64, 1)
    p_result.append(y_pred)

    y_pred = np.expand_dims(y_pred, axis = 1)    # (1, 1, 64, 64, 1)
    x_sample = np.append(x_sample, y_pred, axis=1)[:,-19:,:,:]

# Plot the ground-truth image and a sequence of 5 predicted images.
fig, axes = plt.subplots(1, 6, figsize=(12, 5))
for i, ax in enumerate(axes.flat):
    if i == 0:
        ax.imshow(np.squeeze(y_test[idx]), cmap="gray")
        ax.set_title(f"Frame 20\nGround-truth")
    else:
        ax.imshow(np.squeeze(p_result[i-1]), cmap="gray")
        ax.set_title(f"Frame {20 + i - 1}\nPredicted")
    ax.axis("off")
plt.show()
```



MX-AI

- Many-to-Many Convolutional LSTM model for predicting moving MNIST sequences



MX-AI

■ Many-to-Many Convolutional LSTM model for predicting moving MNIST sequences: Training stage

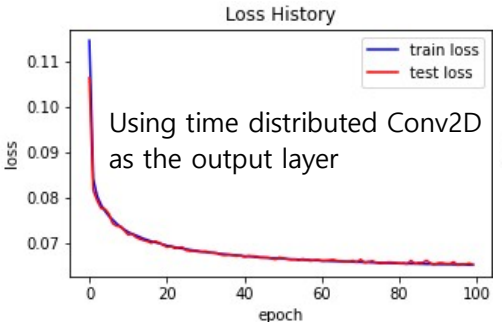
```
# [MXDL-12-06] 14.conv_lstm_m2m(train).py - Many-to-Many
# data: http://www.cs.toronto.edu/~nitish/unsupervised_video/
from tensorflow.keras.layers import Input, ConvLSTM2D, Conv2D
from tensorflow.keras.layers import BatchNormalization, Conv3D
from tensorflow.keras.layers import TimeDistributed, Activation
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
import pickle

# Load a moving MNIST dataset
with open('data/mv_mnist.pkl', 'rb') as f:
    x_train, y_train, x_test, y_test, _ = pickle.load(f)

# Build a Convolutional LSTM model
x = Input(shape=(None, *x_train.shape[2:])) # (None, None, 64, 64,1)
h = ConvLSTM2D(filters=32, kernel_size=(5, 5),
               padding='same',
               return_sequences=True)(x)
h = BatchNormalization()(h)
h = Activation('relu')(h)
y = TimeDistributed(Conv2D(filters=1, kernel_size=(3, 3),
                           padding='same',
                           activation='sigmoid'))(h)

# Instead of using TimeDistributed, you can use a 3D
# convolutional layer in the output layer as follows.
# y = Conv3D(filters=1, kernel_size=(3,3,3),
#                 padding='same', activation='sigmoid')(h)
```

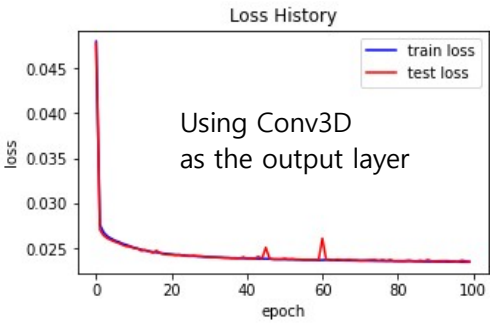
```
model = Model(x, y)
model.compile(loss='binary_crossentropy',
              optimizer=Adam(learning_rate=0.001))
model.summary()

# Fit the model to the training data.
hist = model.fit(x_train, y_train, batch_size = 10, epochs = 20,
                 validation_data=(x_test, y_test))

# Save the model
model.save('data/conv_lstm_m2m.h5', save_format="h5")

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

MX-AI

- Many-to-Many Convolutional LSTM model for predicting moving MNIST sequences: Training stage

```
 Layer (type)                 Output Shape              Param #
=================================================================
 input_1 (InputLayer)         [(None, None, 64, 64, 1)]     0
 conv_lstm2d (ConvLSTM2D)     (None, None, 64, 64, 32)      105728
 batch_normalization (BatchNormalization)  (None, None, 64, 64, 32)      128
 activation (Activation)      (None, None, 64, 64, 32)      0
 conv_lstm2d_1 (ConvLSTM2D)   (None, None, 64, 64, 32)      73856
 batch_normalization_1 (BatchNormalization)(None, None, 64, 64, 32)      128
 activation_1 (Activation)    (None, None, 64, 64, 32)      0
 time_distributed (TimeDistributed)        (None, None, 64, 64, 1)       289
=================================================================
Total params: 180,129
Trainable params: 180,001
Non-trainable params:128
```





* The case of using a time distributed 2D Convolutional layer as the output layer.

```
Epoch 1/100
450/450 [==============================] - 59s 126ms/step - loss: 0.1146 - val_loss: 0.1063
Epoch 2/100
450/450 [==============================] - 56s 125ms/step - loss: 0.0840 - val_loss: 0.0816
Epoch 3/100
450/450 [==============================] - 56s 125ms/step - loss: 0.0803 - val_loss: 0.0794
Epoch 4/100
450/450 [==============================] - 56s 124ms/step - loss: 0.0784 - val_loss: 0.0777
...
Epoch 98/100
450/450 [==============================] - 57s 127ms/step - loss: 0.0652 - val_loss: 0.0653
Epoch 99/100
450/450 [==============================] - 57s 126ms/step - loss: 0.0652 - val_loss: 0.0655
Epoch 100/100
450/450 [==============================] - 57s 126ms/step - loss: 0.0652 - val_loss: 0.0652
```

* The case of using a 3D Convolutional layer as the output layer.

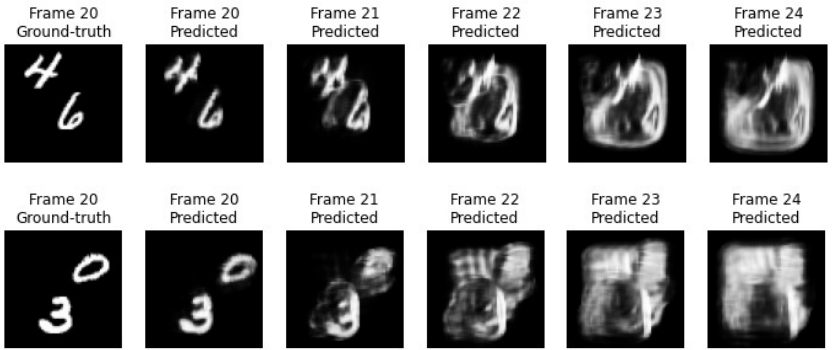
```
Epoch 1/100
450/450 [==============================] - 58s 124ms/step - loss: 0.0480 - val_loss: 0.0478
Epoch 2/100
450/450 [==============================] - 55s 123ms/step - loss: 0.0276 - val_loss: 0.0271
Epoch 3/100
450/450 [==============================] - 55s 123ms/step - loss: 0.0268 - val_loss: 0.0264
Epoch 4/100
450/450 [==============================] - 55s 123ms/step - loss: 0.0263 - val_loss: 0.0261
...
Epoch 98/100
450/450 [==============================] - 56s 124ms/step - loss: 0.0235 - val_loss: 0.0237
Epoch 99/100
450/450 [==============================] - 56s 124ms/step - loss: 0.0235 - val_loss: 0.0236
Epoch 100/100
450/450 [==============================] - 55s 123ms/step - loss: 0.0235 - val_loss: 0.0236
```

MX-AI

■ Many-to-Many Convolutional LSTM model for predicting moving MNIST sequences: Prediction stage

```
# [MXDL-12-06] 15.conv_lstm_m2m(predict).py
# data : http://www.cs.toronto.edu/~nitish/unsupervised_video/
import numpy as np
import pickle
import matplotlib.pyplot as plt
from tensorflow.keras.models import load_model

# Load the moving MNIST dataset and load the trained model.
with open('data/mv_mnist.pkl', 'rb') as f:
    x_train, y_train, x_test, y_test, _ = pickle.load(f)

model = load_model("data/conv_lstm_m2m.h5")

# Predict a sequence of 5 images.
idx = np.random.choice(x_test.shape[0], 1)[0]
x_sample = np.expand_dims(x_test[idx], axis = 0) # (1, 19, 64, 64, 1)

p_result = []
for i in range(5):
    y_pred = model.predict(x_sample, verbose=0)[:,-1,:,:,:]  # (1, 64, 64, 1)
    p_result.append(y_pred)
    y_pred = np.expand_dims(y_pred, axis = 1)    # (1, 1, 64, 64, 1)
    x_sample = np.append(x_sample, y_pred, axis=1)[:,-19:,:,:]

# Plot the ground-truth image and a sequence of 5 predicted images.
fig, axes = plt.subplots(1, 6, figsize=(12, 5))
for i, ax in enumerate(axes.flat):
    if i == 0:
        ax.imshow(np.squeeze(y_test[idx][-1]), cmap="gray")
        ax.set_title(f"Frame 20\nGround-truth")
    else:
        ax.imshow(np.squeeze(p_result[i-1]), cmap="gray")
        ax.set_title(f"Frame {20 + i - 1}\nPredicted")
    ax.axis("off")
plt.show()
```

* The case of using a time distributed 2D Convolutional layer as the output layer.



* The case of using a 3D Convolutional layer as the output layer.