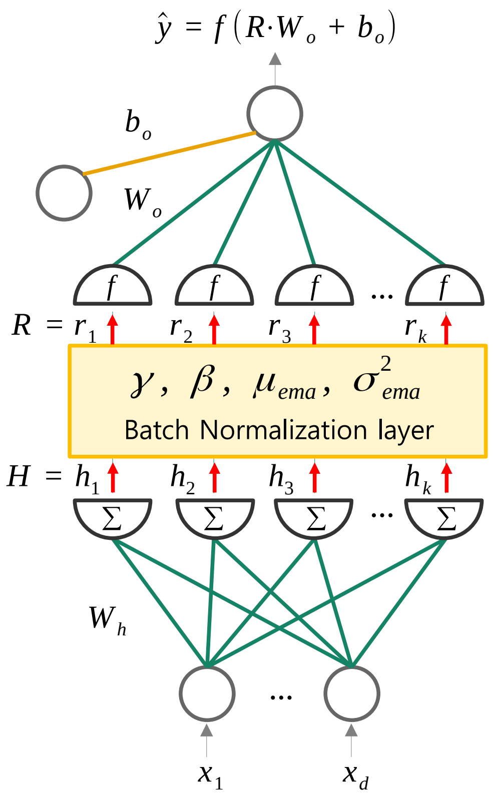


7. Batch Normalization

Part 1: Training and Inference stage

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

www.youtube.com/@meanxai

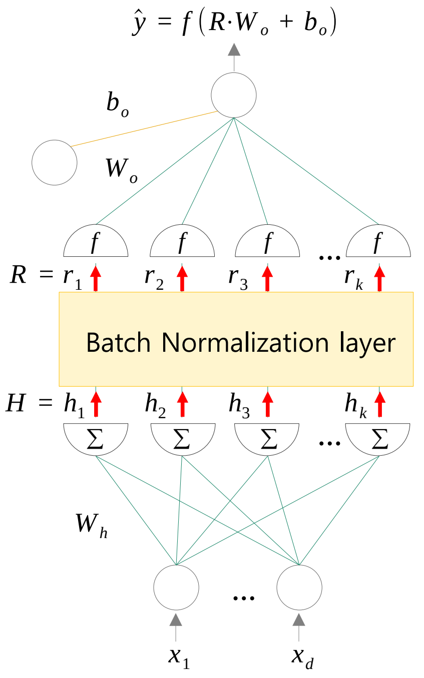
Batch Normalization

[MXDL-7-01]

1. Overview
2. Internal covariate shift and Normalization
3. Training stage
4. Inference (prediction) stage

[MXDL-7-02]

5. Observing the parameters of Keras BatchNormalization
6. Custom BatchNormalization layer
7. Implementing an ANN model with Batch Normalization using Keras



■ Batch Normalization paper (2015)

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., sioffe@google.com

Christian Szegedy
Google Inc., szegedy@google.com

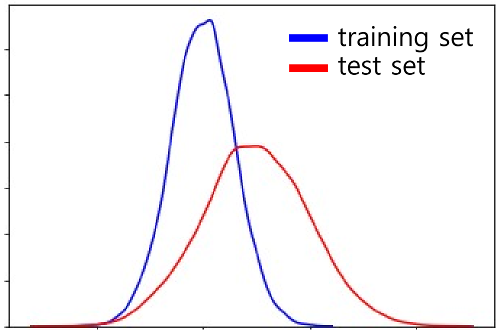
Abstract

Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as internal covariate shift, and address the problem by normalizing layer inputs. Our method draws its strength from making normalization a part of the model architecture and performing the normalization for each training mini-batch. Batch Normalization allows us to use much higher learning rates and be less careful about initialization. It also acts as a regularizer, in some cases eliminating the need for Dropout. Applied to a state-of-the-art image classification model, Batch Normalization achieves the same accuracy with 14 times fewer training steps, and beats the original model by a significant margin. Using an ensemble of batchnormalized networks, we improve upon the best published result on ImageNet classification: reaching 4.9% top-5 validation error (and 4.8% test error), exceeding the accuracy of human raters.

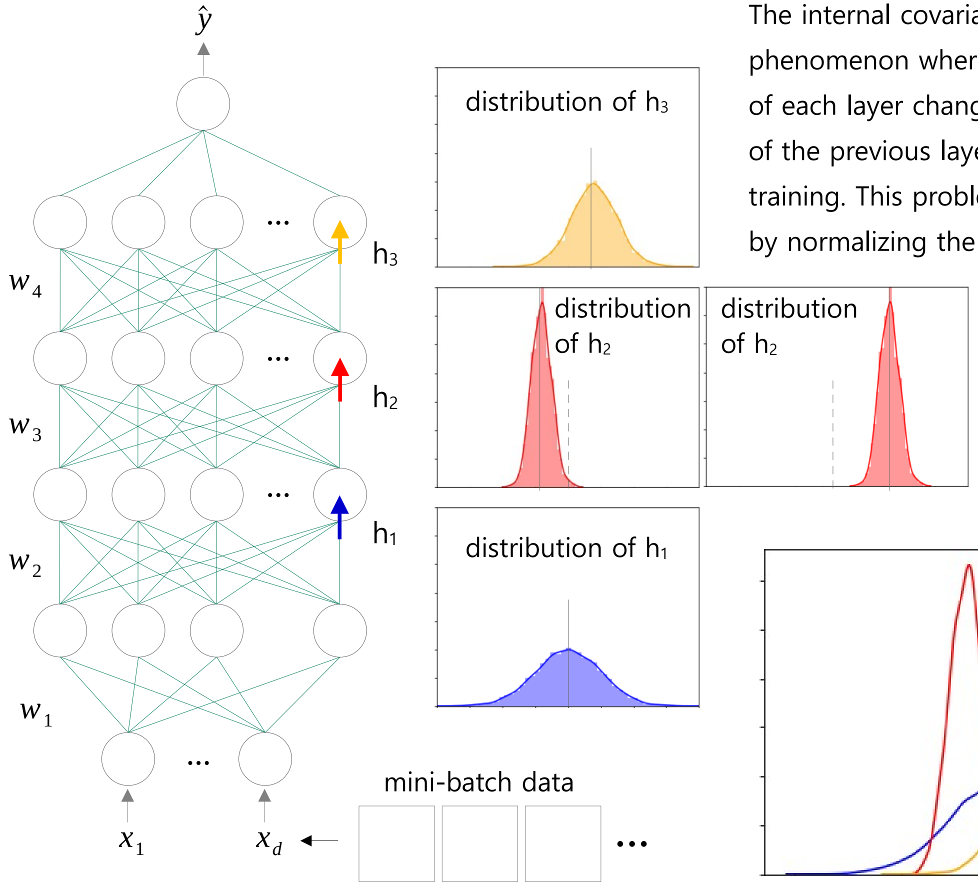
MX-AI

- Common covariate shift and internal covariate shift

Covariate shift



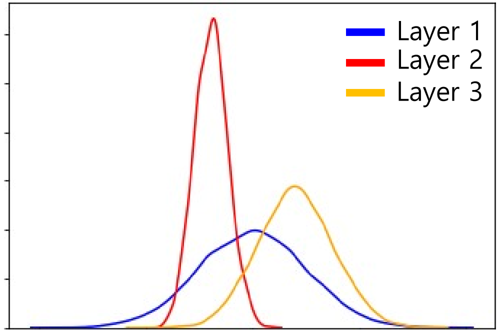
A common covariate shift problem is the difference in distribution between the training and test sets, which reduces the predictive performance of the model. These data sets can be processed through importance sampling, standardization, or whitening methods in the data preprocessing stage.



Internal covariate shift

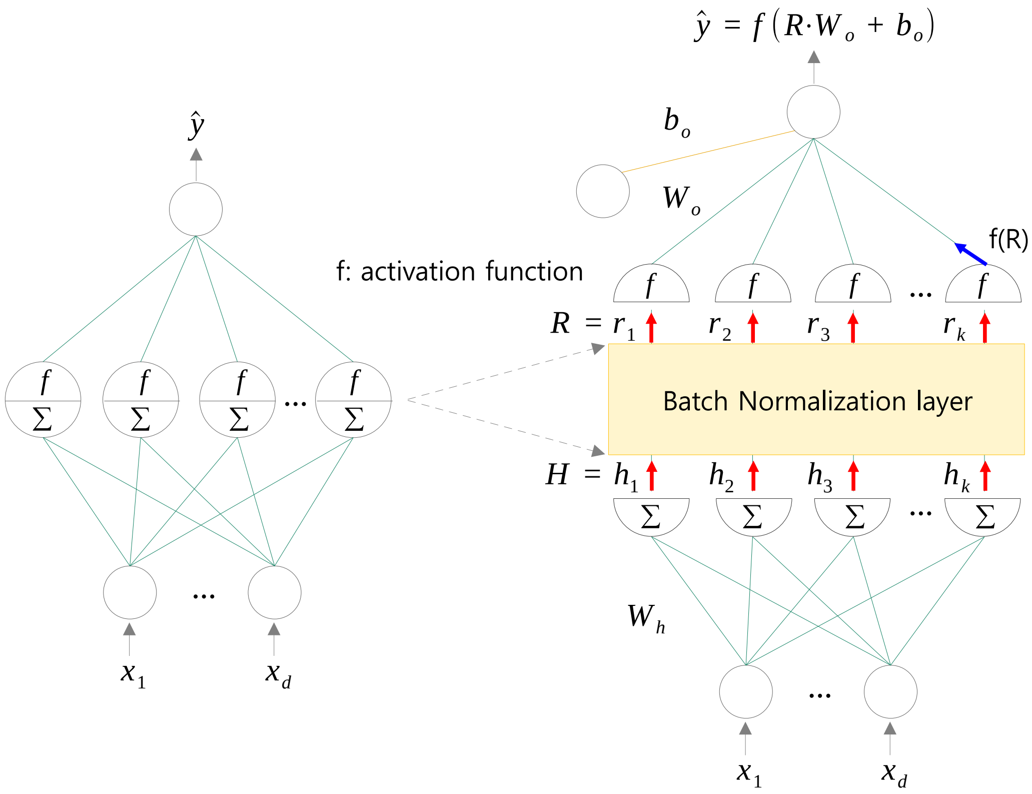
The internal covariate shift is the phenomenon where the input distribution of each layer changes as the parameters of the previous layer change during training. This problem can be addressed by normalizing the inputs of each layer.

As w3 changes, the distribution of h2 may also change.



MX-AI

Batch Normalization: Training stage



m: batch size, d: feature dimension

k: number of hidden units

During training, exponential moving averages for E[h] and Var[h] are computed and stored. These are used to normalize h for the test data during prediction.

$$E_{ema}[h_j]_{t+1} = \rho E_{ema}[h_j]_t + (1-\rho) E[h_j]$$

$$Var_{ema}[h_j]_{t+1} = \rho Var_{ema}[h_j]_t + (1-\rho) Var[h_j]$$

$$R = \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & r_{1,4} & \dots & r_{1,k} \\ r_{2,1} & r_{2,2} & r_{2,3} & r_{2,4} & \dots & r_{2,k} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ r_{m,1} & r_{m,2} & r_{m,3} & r_{m,4} & \cdots & r_{m,k} \end{bmatrix}$$

γ and β are trainable parameters. These are determined through training and stored in the Batch Normalization layer.

$$r_{i,j} = \gamma_j \hat{h}_{i,j} + \beta_j$$

$$H = \begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} & h_{1,4} & \dots & h_{1,k} \\ h_{2,1} & h_{2,2} & h_{2,3} & h_{2,4} & \dots & h_{2,k} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ h_{m,1} & h_{m,2} & h_{m,3} & h_{m,4} & \cdots & h_{m,k} \end{bmatrix}$$

$$\hat{h}_{i,j} = \frac{h_{i,j} - E[h_j]}{\sqrt{Var[h_j] + \epsilon}}$$

$E[h_1]$ $\quad$ $E[h_k]$ ← np.mean(H, axis=0)

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} & \dots & x_{1,d} \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} & \dots & x_{2,d} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ x_{m,1} & x_{m,2} & x_{m,3} & x_{m,4} & \cdots & x_{m,d} \end{bmatrix}$$

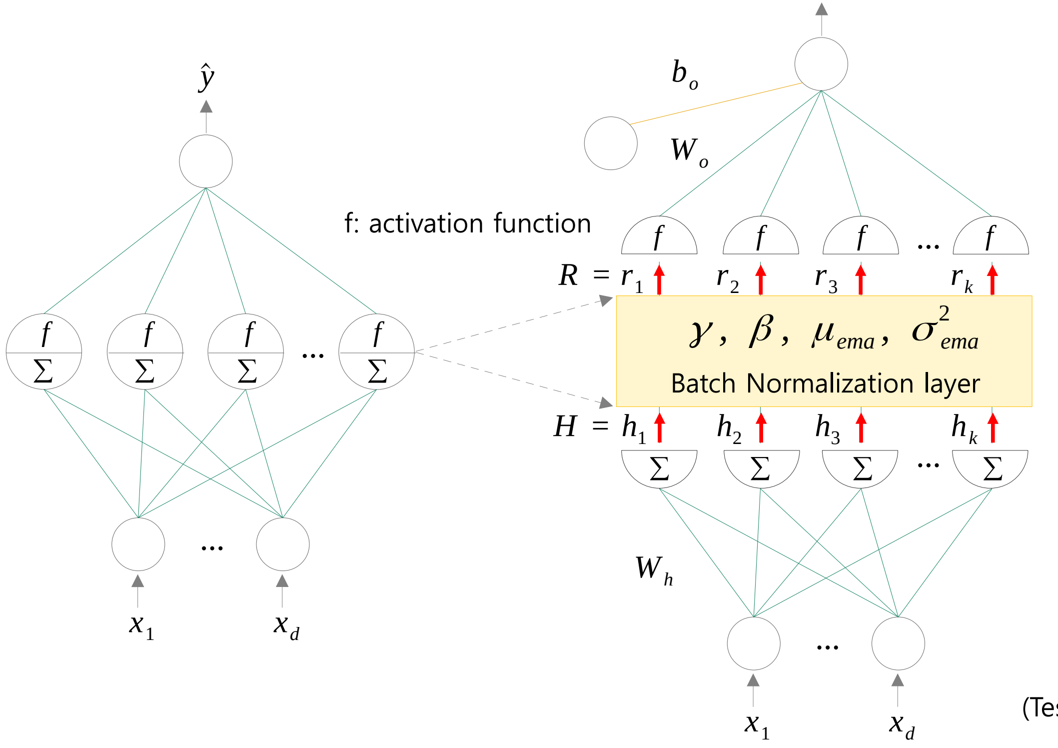
$$H = X \cdot W_h$$

The β above acts as a bias, so there is no need to use bias in this layer.

MX-AI

■ Batch Normalization: Inference stage

$$\hat{y} = f(R \cdot W_o + b_o)$$



Exponential moving average and variance for h of the training data stored during training.

$$E_{ema}[h_j]_{t+1} = \rho E_{ema}[h_j]_t + (1-\rho) E[h_j] = \mu_{ema}$$

$$Var_{ema}[h_j]_{t+1} = \rho Var_{ema}[h_j]_t + (1-\rho) Var[h_j] = \sigma^2_{ema}$$

$$R = \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & r_{1,4} & \dots & r_{1,k} \\ r_{2,1} & r_{2,2} & r_{2,3} & r_{2,4} & \dots & r_{2,k} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ r_{n,1} & r_{n,2} & r_{n,3} & r_{n,4} & \dots & r_{n,k} \end{bmatrix}$$

γ and β are trained parameters.

$$r_{i,j} = \gamma_j \hat{h}_{i,j} + \beta_j$$

$$H = X \cdot W_h$$

$$H = \begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} & h_{1,4} & \dots & h_{1,k} \\ h_{2,1} & h_{2,2} & h_{2,3} & h_{2,4} & \dots & h_{2,k} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ h_{n,1} & h_{n,2} & h_{n,3} & h_{n,4} & \dots & h_{n,k} \end{bmatrix}$$

$$\hat{h}_{i,j} = \frac{h_{i,j} - \mu_{ema}}{\sqrt{\sigma^2_{ema} + \epsilon}}$$

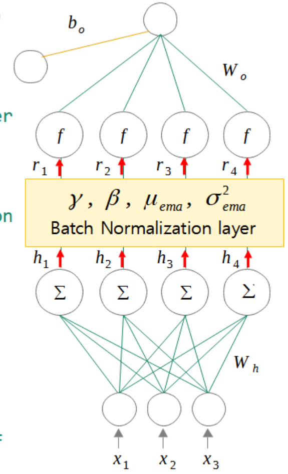
k: number of hidden units

The exponential moving average and variance of E[h] and Var[h] of the training data, calculated and stored during training, are used to normalize the h of the test data.

(Test data) $$X = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} & \dots & x_{1,d} \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} & \dots & x_{2,d} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ x_{n,1} & x_{n,2} & x_{n,3} & x_{n,4} & \dots & x_{n,d} \end{bmatrix}$$

n: the number of test data points, d: feature dimension


```
# Training. Gamma and beta are also learned, and moving mu and
# var are calculated and stored.
model.fit(x, y, epochs=10, batch_size=10, verbose=0)

# outputs of the hidden layer
print('After training: prediction stage')
h = model_h.predict(x, verbose=0)[:3]
print('h = '); print(h.round(3))

# outputs of the Batch Normalization layer
r = model_r.predict(x, verbose=0)[:3]
print('\nr = '); print(r.round(3))

# Parameters stored in Batch Normalization
# layer
gamma, beta, mu, var = \
    model.get_layer('BN').get_weights()
print('\n    γ =', gamma.round(3))
print('    β =', beta.round(3))
print('E[h] =', mu.round(3))
print('V[h] =', var.round(3))

# Let's manually calculate the outputs of
# the BatchNormalization layer.
bn = gamma * (h - mu) / np.sqrt(var + e) + beta
print('\nManual calculation (r)')
print(bn.round(3)) # This matches r above.
```



7. Batch Normalization

Part 2: Implementing Batch Normalization

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

www.youtube.com/@meanxai

MX-AI

- Observation of the internal parameters of Keras BatchNormalization

```python
# [MXDL-7-02] 1.obs_parameters.py
# Observing the parameters inside Batch Normalization layer
import numpy as np
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.layers import BatchNormalization, Activation
from tensorflow.keras.models import Model

x = np.random.normal(size=(100, 3))
y = np.random.choice([0,1], 100).reshape(-1,1)
e = 0.001; rho = 0.99

x_input = Input(batch_shape=(None, 3))
h = Dense(4, use_bias=False, name='HN')(x_input)
r = BatchNormalization(momentum=rho, epsilon=e, name='BN')(h)
h_act = Activation('relu')(r)
y_output = Dense(1, activation='sigmoid')(h_act)
model = Model(x_input, y_output)
model.compile(loss='mean_squared_error', optimizer = 'adam')
model_h = Model(x_input, h)
model_r = Model(x_input, r)
model.summary()

# Initial values of the parameters in Batch Normalization layer
gamma, beta, mu, var = model.get_layer('BN').get_weights()
print('\nInitial values:')
print('   γ =', gamma.round(3))
print('   β =', beta.round(3))
print('E[h] =', mu.round(3))
print('V[h] =', var.round(3))
```

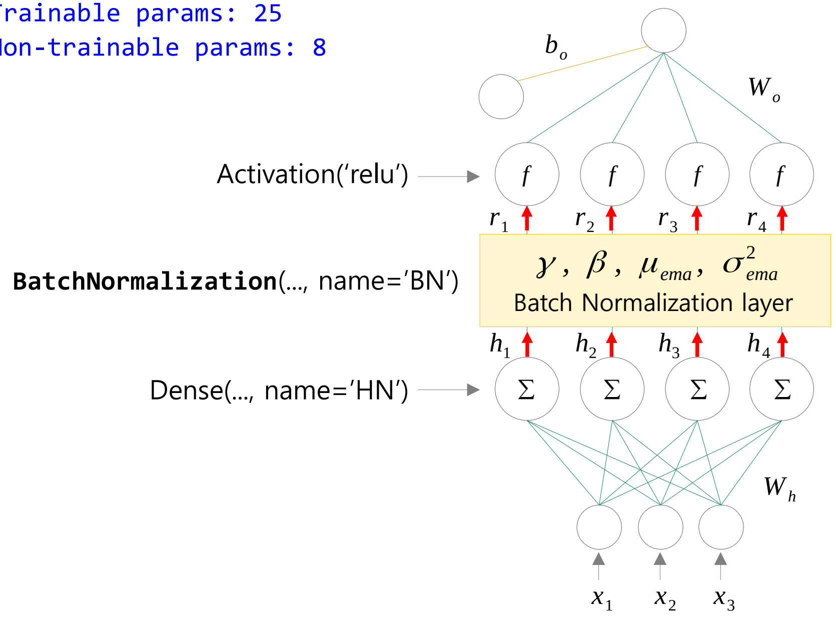
$$E_{ema}[h_j]_{t+1} = \rho E_{ema}[h_j]_t + (1-\rho) E[h_j]$$

$$Var_{ema}[h_j]_{t+1} = \rho Var_{ema}[h_j]_t + (1-\rho) Var[h_j]$$

$$\hat{h}_{i,j} = \frac{h_{i,j} - E[h_j]}{\sqrt{Var[h_j] + \epsilon}}$$

$$r_{i,j} = \gamma_j \hat{h}_{i,j} + \beta_j$$

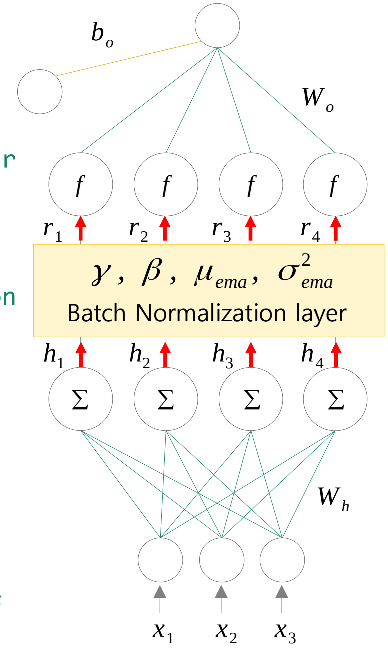
```
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 3)]               0
 HN (Dense)                   (None, 4)                12
 BN (BatchNormalization)      (None, 4)                16
 activation (Activation)      (None, 4)                0
 dense (Dense)                (None, 1)                5
=================================================================
Total params: 33
Trainable params: 25
Non-trainable params: 8
```



MX-AI

■ Observation of the internal parameters of Keras BatchNormalization

```
# Training. Gamma and beta are also learned, and moving mu and
# var are calculated and stored.
model.fit(x, y, epochs=10, batch_size=10, verbose=0)

# outputs of the hidden layer
print('After training: prediction stage')
ho = model_h.predict(x, verbose=0)[:3]
print('h = '); print(ho.round(3))

# outputs of the Batch Normalization layer
ro = model_r.predict(x, verbose=0)[:3]
print('\nr = '); print(ro.round(3))

# Parameters stored in Batch Normalization
# layer
gamma, beta, mu, var = \
    model.get_layer('BN').get_weights()
print('\n    γ =', gamma.round(3))
print('    β =', beta.round(3))
print('E[h] =', mu.round(3))
print('V[h] =', var.round(3))

# Let's manually calculate the outputs of
# the BatchNormalization layer.
rm = gamma * (ho - mu) / np.sqrt(var + e) + beta
print('\nManual calculation (r)')
print(rm.round(3)) # This matches r above.
```




```
Initial values:
    γ = [1. 1. 1. 1.]
    β = [0. 0. 0. 0.]
 E[h] = [0. 0. 0. 0.]
 V[h] = [1. 1. 1. 1.]


After training: inference stage
h = [[ 2.231  0.996  0.742  1.156]
     [ 0.094 -3.605 -2.613  2.943]
     [ 1.894 -1.353 -1.513  3.84 ]

r = [[ 2.326  0.897  0.655  0.723]
     [ 0.302 -2.276 -2.051  1.946]
     [ 2.007 -0.723 -1.164  2.56 ]

    γ = [1.049 1.074 0.923 0.938]
    β = [ 0.053  0.063 -0.063 -0.06 ]
 E[h] = [-0.169 -0.214 -0.148  0.012]
 V[h] = [1.225 2.426 1.309 1.878]


Manual calculation (r)
r = [[ 2.326  0.897  0.655  0.723]
     [ 0.302 -2.276 -2.051  1.946]
     [ 2.007 -0.723 -1.164  2.56 ]
```

$$E_{ema}[h_j]_{t+1} = \rho E_{ema}[h_j]_t + (1-\rho) E[h_j]$$

$$Var_{ema}[h_j]_{t+1} = \rho Var_{ema}[h_j]_t + (1-\rho) Var[h_j]$$

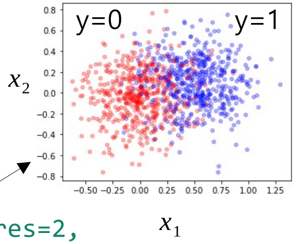
$$\hat{h}_{i,j} = \frac{h_{i,j} - E[h_j]}{\sqrt{Var[h_j] + \epsilon}}$$

$$r_{i,j} = \gamma_j \hat{h}_{i,j} + \beta_j$$

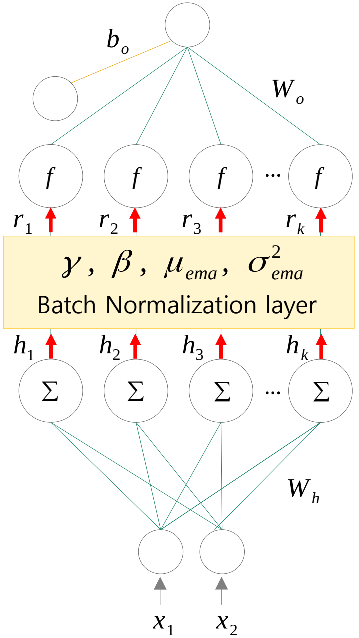
MX-AI

■ Custom BatchNormalization layer

```python
# [MXDL-7-02] 2.MyBatchNorm.py
# Custom BatchNormalization layer
import numpy as np
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras.layers import Layer,Input,Dense,Activation
from tensorflow.keras.models import Model
from tensorflow.keras import optimizers
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
import pickle

# Generate a dataset
# x, y = make_blobs(n_samples=1000, n_features=2,
#                   centers=[[0., 0.], [0.5, 0.1]],
#                   cluster_std=0.25, center_box=(-1., 1.))
# y = y.reshape(-1, 1).astype('float32')
# x_train, x_test, y_train, y_test = train_test_split(x, y)
# with open('data/blobs.pkl', 'wb') as f:
#     pickle.dump([x_train, x_test, y_train, y_test], f)
with open('data/blobs.pkl', 'rb') as f:
    x_train, x_test, y_train, y_test = pickle.load(f)

# Visually see the data distribution
plt.figure(figsize=(5,4))
color = [['red', 'blue'][a] for a in y_train.reshape(-1,)]
plt.scatter(x_train[:, 0], x_train[:, 1],s=20,c=color,alpha=0.3)
plt.show()
```




```python
# Custom Batch Normalization layer
class MyBatchNorm(Layer):
    def __init__(self, units=32, rho=0.99):
        super(MyBatchNorm, self).__init__()
        self.units = units
        self.rho = rho

    def build(self, input_shape):
        dim = (input_shape[-1],)
        self.gamma = self.add_weight(
            shape=dim, name='gamma',
            initializer='ones',
            trainable=True)

        self.beta = self.add_weight(
            shape=dim, name='beta',
            initializer='zeros',
            trainable=True)

        self.mean_ema = self.add_weight(
            shape=dim, name='mean_ema',
            initializer='zeros',
            trainable=False)

        self.var_ema = self.add_weight(
            shape=dim, name='var_ema',
            initializer='zeros',
            trainable=False)
```



$$E_{ema}[h_j]_{t+1} = \rho E_{ema}[h_j]_t + (1-\rho) E[h_j]$$

$$Var_{ema}[h_j]_{t+1} = \rho Var_{ema}[h_j]_t + (1-\rho) Var[h_j]$$

$$\hat{h}_{i,j} = \frac{h_{i,j} - E[h_j]}{\sqrt{Var[h_j] + \epsilon}}$$

$$r_{i,j} = \gamma_j \hat{h}_{i,j} + \beta_j$$

[MXDL-7-02] Deep Learning / Batch Normalization – Custom layer

■ Custom BatchNormalization layer

```python
    def call(self, inputs, training=True):
        if training:
            mean= tf.reduce_mean(inputs, axis=0)
            var= tf.math.reduce_variance(inputs, axis=0)

        else:
            mean= self.mean_ema
            var= self.var_ema
            print("\n===========> training=False")

        h_hat = (inputs-mean) / tf.math.sqrt(var+EPSILON)
        self.mean_ema.assign(self.rho * self.mean_ema + \
                             (1 - self.rho) * mean)
        self.var_ema.assign(self.rho * self.var_ema + \
                            (1 - self.rho) * var)

        return self.gamma * h_hat + self.beta
```

$$\hat{h}_{i,j} = \frac{h_{i,j} - E[h_j]}{\sqrt{Var[h_j] + \epsilon}} \qquad \hat{h}_{i,j} = \frac{h_{i,j} - \mu_{ema}}{\sqrt{\sigma^2_{ema} + \epsilon}}$$

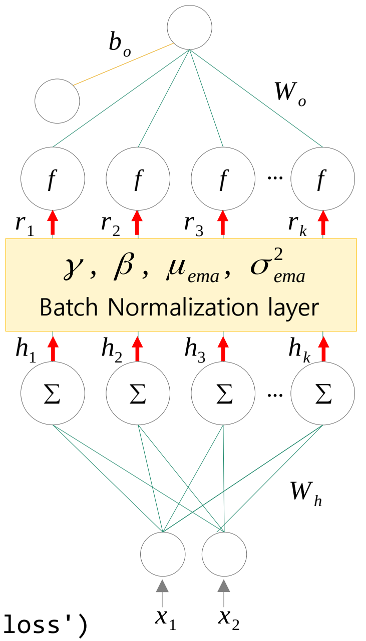
$$E_{ema}[h_j]_{t+1} = \rho E_{ema}[h_j]_t + (1-\rho) E[h_j]$$

$$Var_{ema}[h_j]_{t+1} = \rho Var_{ema}[h_j]_t + (1-\rho) Var[h_j]$$

$$r_{i,j} = \gamma_j \hat{h}_{i,j} + \beta_j$$

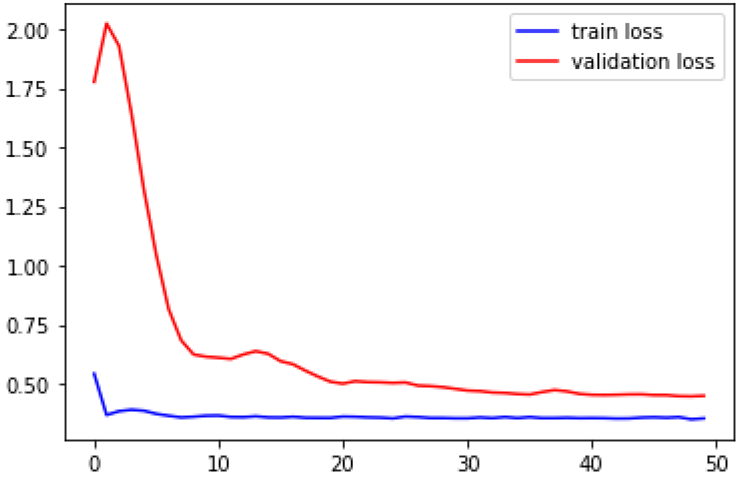
```python
# Create an ANN model with Batch Normalization
n_input = x_train.shape[1] # number of input neurons
n_output = 1               # number of output neurons
n_hidden = 128             # number of hidden neurons
EPSILON = 1e-5
```

```python
BATCH_SIZE = 300; RHO = 0.99
adam = optimizers.Adam(learning_rate=0.01)

BatchNorm = MyBatchNorm(n_hidden, RHO)
x_input = Input(batch_shape=(None, n_input))
h = Dense(n_hidden, use_bias=False)(x_input)
h = BatchNorm(h)
h = Activation('relu')(h)

y_output = Dense(n_output,
                 activation='sigmoid')(h)
model = Model(x_input, y_output)
model.compile(loss='binary_crossentropy',
              optimizer=adam)

h = model.fit(x_train, y_train,
              validation_data=(x_test, y_test),
              epochs=50, batch_size=300)

# Visually see the loss history
plt.plot(h.history['loss'], c='blue', label='train loss')
plt.plot(h.history['val_loss'], c='red', label='validation loss')
plt.legend()
plt.show()

# Check the accuracy of test data
y_hat = model(x_test, training=False).numpy()
y_pred = (y_hat > 0.5) * 1
acc = (y_pred == y_test).mean()
print("The accuracy of test data = {:4f}".format(acc))

BatchNorm.gamma; BatchNorm.beta; BatchNorm.mean_ema; BatchNorm.var_ema
```



Custom BatchNormalization layer

```
Epoch 46/50
3/3 [========= ] - 0s 10ms/step - loss: 0.3571 - val_loss: 0.4510
Epoch 47/50
3/3 [========= ] - 0s 10ms/step - loss: 0.3551 - val_loss: 0.4508
Epoch 48/50
3/3 [========= ] - 0s 10ms/step - loss: 0.3577 - val_loss: 0.4466
Epoch 49/50
3/3 [========= ] - 0s 12ms/step - loss: 0.3482 - val_loss: 0.4457
Epoch 50/50
3/3 [========= ] - 0s 10ms/step - loss: 0.3523 - val_loss: 0.4481
```



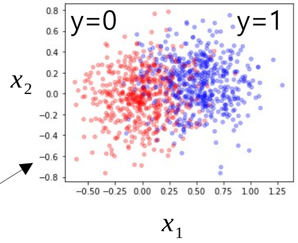

```
===========> training=False
The accuracy of test data = 0.812000
```

```
BatchNorm.gamma
<tf.Variable 'my_batch_norm/gamma:0' shape=(128,) dtype=float32, numpy=
array([1.0067441 , 0.89920324, 0.9865765 , 1.0484002 , 0.85001016,
       1.0272682 , 0.91234386, 0.9194974 , 0.9624974 , 0.9162404 , ...

BatchNorm.beta
<tf.Variable 'my_batch_norm/beta:0' shape=(128,) dtype=float32, numpy=
array([-0.21658531, -0.1587426 ,  0.17253388, -0.12346091, -0.23472252,
       -0.04440194, -0.07524271, -0.03106111, -0.3125096 , -0.0473272 ...

BatchNorm.mean_ema
<tf.Variable 'my_batch_norm/mean_ema:0' shape=(128,) dtype=float32, numpy=
array([-0.02217293, -0.033124   ,  0.0444601 , -0.03982214,  0.00502331,
       -0.058402  , -0.05558247, -0.02492878, -0.00193717, -0.02047776, ...

BatchNorm.var_ema
<tf.Variable 'my_batch_norm/var_ema:0' shape=(128,) dtype=float32, numpy=
array([0.00281983, 0.00334556, 0.00549125, 0.00417261, 0.00343633,
       0.00917257, 0.0085239 , 0.00288554, 0.00298871, 0.00193051, ...
```

MX-AI

■ Verifying that BatchNormalization prevents overfitting.

```python
# [MXDL-7-02] 3.BatchNorm(Keras).py
# Verify that BatchNormalization prevents overfitting.
import numpy as np
from sklearn.model_selection import train_test_split
from tensorflow.keras.layers import Input, Dense, \
     BatchNormalization, Activation
from tensorflow.keras.models import Model
from tensorflow.keras import optimizers
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
import pickle

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

# Visually see the data
plt.figure(figsize=(5,4))
color = [['red', 'blue'][a] for a in y_train.reshape(-1,)]
plt.scatter(x_train[:, 0], x_train[:, 1], s=20, c=color, alpha=0.3)
plt.show()

# Create an ANN model with Batch Normalization layer
n_input = x_train.shape[1] # number of input neurons
n_output = 1               # number of output neurons
n_hidden = 128             # number of hidden neurons
adam = optimizers.Adam(learning_rate=0.01)
```

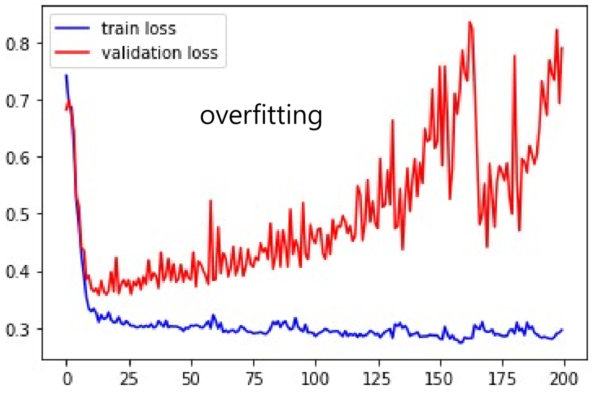



```python
x_input = Input(batch_shape=(None, n_input))
h = Dense(n_hidden, use_bias=False)(x_input)
h = BatchNormalization()(h)
h = Activation('relu')(h)

# Additional 10 hidden layers
# The data is simple, but we intentionally added many hidden
# layers to the model to verity the effect of Batch Normalization.
for i in range(10):
    h = Dense(n_hidden, use_bias=False)(h)
    h = BatchNormalization()(h)
    h = Activation('relu')(h)

y_output = Dense(n_output, activation='sigmoid')(h)
model = Model(x_input, y_output)
model.compile(loss='binary_crossentropy', optimizer=adam)

# training
h = model.fit(x_train, y_train,
              validation_data=(x_test, y_test),
              epochs=100, batch_size=300, shuffle=True)

# Visually see the loss history
plt.plot(h.history['loss'], c='blue', label='train loss')
plt.plot(h.history['val_loss'], c='red', label='validation loss')
plt.legend()
plt.show()
```

■ Verifying that BatchNormalization prevents overfitting.

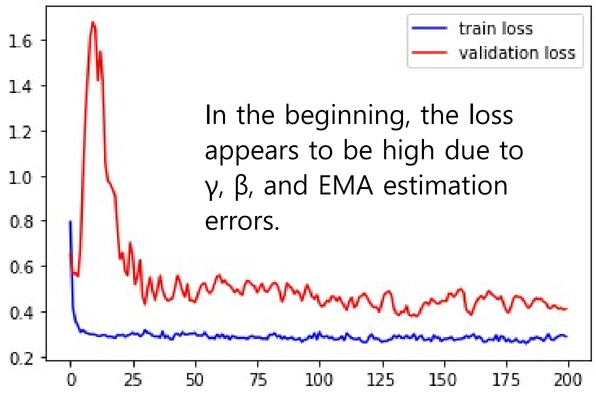
```
# Check the accuracy of the test data
y_pred = (model.predict(x_test) > 0.5) * 1
acc = (y_pred == y_test).mean()
print("\nThe accuracy of the test data = {:4f}".format(acc))
```

Before applying Batch Normalization



The accuracy of the test data = 0.86

After applying Batch Normalization



The accuracy of the test data = 0.84