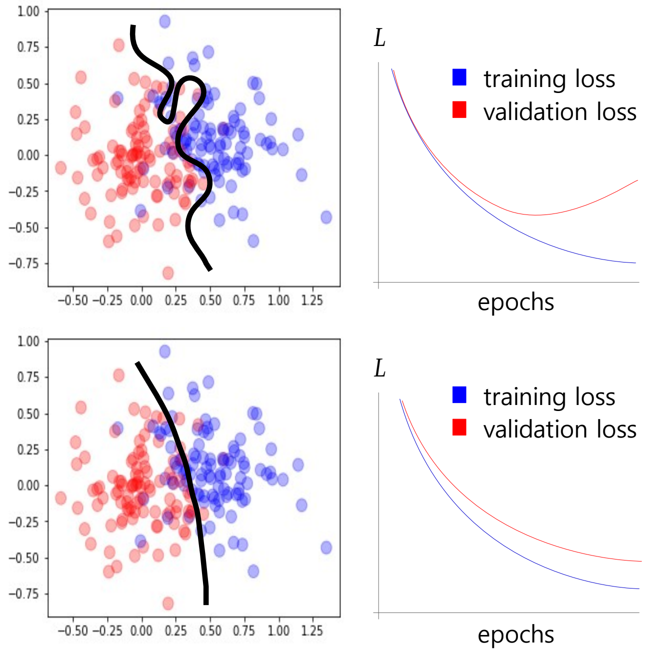


5. Regularization

Part 1: Weight and Bias Regularization

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

www.youtube.com/@meanxai

MX-AI

Regularization

[MXDL-5-01]

1. Accuracy and Validation loss
2. Validation loss and Overfitting
3. Weight and bias Regularization
   - Implement regularization by creating a regularized loss function.
   - Implement regularization by Keras Regularizer
   - Implement regularization using custom regularizer

[MXDL-5-02]

4. Activity (or Activation) regularization
   - Applying activity regularization
   - Applying activity regularization before activation function
   - Applying weight, bias, activity regularization together
5. Imposing a sparsity constraint on the hidden neurons (KL divergence regularization).
   - Imposing a sparsity constraint on the hidden neurons by activity regularization

Accuracy and Validation loss

Model (A)

y and y-hat of a validation dataset.

y = [1,   0,   0,   1,   0,   1,   1,   0,   1,   0]

$\hat{y}$ = [0.51, 0.49, 0.3, 0.7, 0.4, 0.6, 0.6, 0.4, 0.6, 0.3]

Accuracy = 100%

$$L = -\frac{1}{10}\sum_{i=1}^{10}\left[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)\right] = 0.50$$

Model A has higher accuracy than Model B, but has higher loss and unstable predictions. The accuracy may decrease when measuring with new validation data.

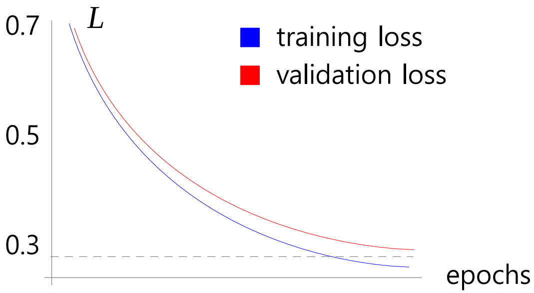
Model (B)

y = [1,   0,   0,   1,   0,   1,   1,   0,   1,   0]

$\hat{y}$ = [0.9, 0.6, 0.1, 0.9, 0.1, 0.8, 0.9, 0.2, 0.8, 0.1]

Accuracy = 90%

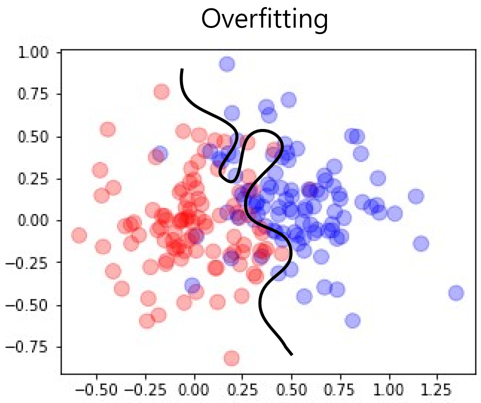
$$L = -\frac{1}{10}\sum_{i=1}^{10}\left[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)\right] = 0.22$$



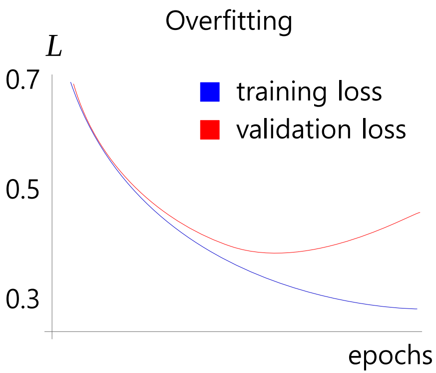
Model B has slightly lower accuracy than Model A, but has less loss and more stable predictions. It performs better overall than Model A. In general, models with lower loss on validation data perform better.

MX-AI

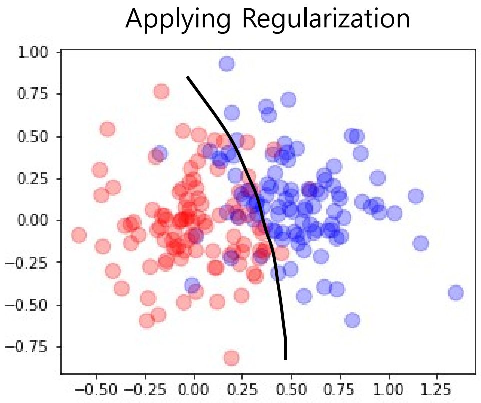
■ Validation loss and Overfitting



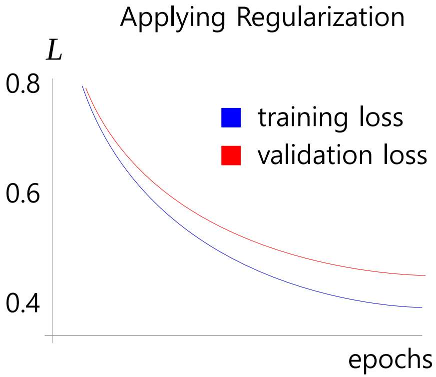
Overfitting occurs when you use a neural network that is too wide or too deep for simple data.



As training progresses, the training loss continues to decrease, but the validation loss decreases and then increases again. This means that the model explains the training data very well, but not the validation data.



Even when using complex models, regularization can help prevent overfitting.

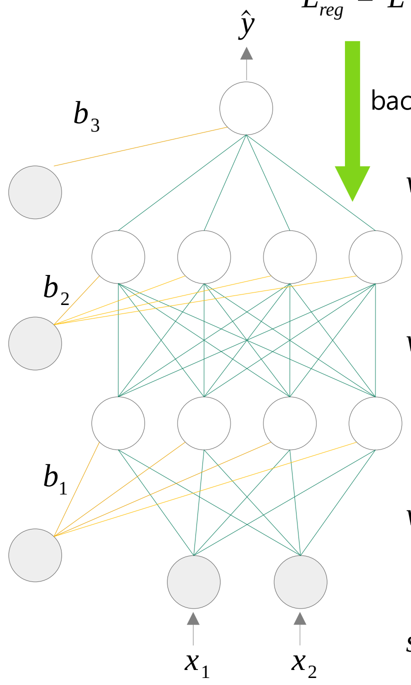


Compared to the above model, the overall loss may increase slightly, but both training and validation losses gradually decrease. This model can explain not only the training data but also the validation data well. It is likely to explain the new validation data well.

■ Weight (Bias) Regularization

LASSO	Ridge	Elastic-Net
(L_1 regularization)	(L_2)	(L_1 & L_2)

$$L_{reg} = L + \lambda_1 \sum_k |w_{1,k}| + \frac{\lambda_2}{2} \sum_k w_{2,k}^2 + \lambda_3 \sum_k |w_{3,k}| + \frac{\lambda_4}{2} \sum_k w_{3,k}^2$$



backpropagation

$$w_3 = w_3 - \alpha\left(\nabla_{w_3} L + \lambda_3 \sum_k sign(w_{3,k}) + \lambda_4 \sum_k w_{3,k}\right)$$

$$w_2 = w_2 - \alpha\left(\nabla_{w_2} J + \lambda_2 \sum_k w_{2,k}\right)$$

$$w_1 = w_1 - \alpha\left(\nabla_{w_1} L + \lambda_1 \sum_k sign(w_{1,k})\right)$$

$$sign(x) = +1 \ (x>0), \ -1 \ (x<0), \ 0 \ (x=0)$$

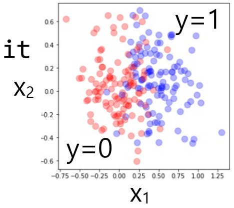
* Bias regularization can be performed by substituting b for w.

```
tf.keras.layers.Dense(
    units,
    activation = None,
    use_bias = True,
    kernel_initializer = 'glorot_uniform',
    bias_initializer = 'zeros',
    kernel_regularizer = None,
    bias_regularizer = None,
    activity_regularizer = None,
    kernel_constraint = None,
    bias_constraint=None,
    lora_rank=None,
    **kwargs
)

kernel_regularizer:
Regularizer function applied to the kernel
weights matrix.

bias_regularizer:
Regularizer function applied to the bias vector.
```

MX-AI

- Implement regularization by creating a regularized loss function.

```
# [MXDL-5-01] 1.regularized_loss.py
# Regularization can be easily implemented using Keras Dense's
# kernel_regularizer and bias_regularizer, but to better
# understand how regularization works, we implement it by
# creating a regularized loss function.
import numpy as np
import tensorflow as tf
from sklearn.datasets import make_blobs
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras import optimizers
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import pickle

# Generate a dataset
# x, y = make_blobs(n_samples=300, 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 distribution of the data points
plt.figure(figsize=(5, 5))
color = [['red', 'blue'][int(a)] for a in y_train.reshape(-1,)]
plt.scatter(x_train[:, 0], x_train[:, 1], s=100, c=color,
            alpha=0.3)
plt.show()

# Create an ANN model.
n_input = x_train.shape[1]  # number of input neurons
n_output = 1                # number of output neurons
```



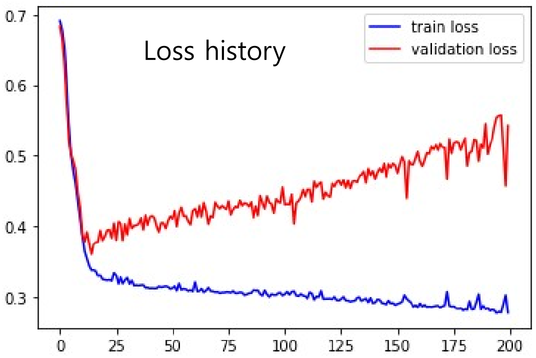

```
n_hidden = 32              # number of hidden neurons
R = 0.001                  # Regularization constant
e = 1e-6                   # small value to avoid log(0)
adam = optimizers.Adam(learning_rate=0.001)

# The data is simple, but we intentionally added many hidden
# layers to the model to demonstrate the effect of regularization.
x_input = Input(batch_shape=(None, n_input))
h = Dense(n_hidden, activation = 'relu')(x_input)

# 4 more hidden layers
for i in range(4):
    h = Dense(n_hidden, activation = 'relu')(h)

y_output = Dense(n_output, activation='sigmoid')(h)
model = Model(x_input, y_output)
model.summary()

# 0: no regularization, 1: L1, 2: L2, 12: L1 & L2
params = model.trainable_variables
r_list = [1, -1, 2, 2, 12, 2, 1, 2, -1, 2]

# Layer (type)                Output Shape             Param #
#=================================================================
# input_1 (InputLayer)       [(None, 2)]              0
# dense   (Dense)             (None, 32)              96
# dense_1 (Dense)             (None, 32)              1056
# dense_2 (Dense)             (None, 32)              1056
# dense_3 (Dense)             (None, 32)              1056
# dense_4 (Dense)             (None, 32)              1056
# dense_5 (Dense)             (None, 1)               33
#=================================================================
# Total params: 4,353
# Trainable params: 4,353
# Non-trainable params: 0
```

MX-AI

■ Implement regularization by creating a regularized loss function.

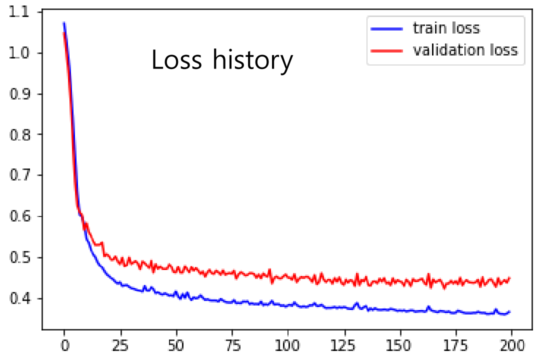
```
# Custom loss function: regularized loss
class reg_loss(tf.keras.losses.Loss):
    def __init__(self, reg_lambda, params, r_list, **kwargs):
        super().__init__(**kwargs)
        self.R = reg_lambda
        self.params = params
        self.r_list = r_list

    def call(self, y, y_pred):
        bce = -tf.reduce_mean(
                  y * tf.math.log(y_pred + e) + \
                  (1. - y) * tf.math.log(1. - y_pred + e))

        reg_terms = 0
        for i, p in zip(self.r_list, self.params):
            if i == 1:  # L1 regularization
                reg_terms += tf.reduce_sum(tf.math.abs(p))

            if i == 2:  # L2 regularization
                reg_terms += tf.reduce_sum(tf.square(p))

            if i == 12:  # L1 & L2 regularization
                reg_terms += tf.reduce_sum(tf.math.abs(p))
                reg_terms += tf.reduce_sum(tf.square(p))

        return bce + self.R * reg_terms

model.compile(loss=reg_loss(R, params, r_list), optimizer = adam)
f = model.fit(x_train, y_train,
              validation_data=[x_test, y_test],
              epochs=200, batch_size=20)

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

```
# Check the accuracy of the test data
y_pred = (model.predict(x_test) > 0.5) * 1
acc = (y_pred == y_test).mean()
print("\nAccuracy of the test data = {:.2f}".format(acc))
```



Regularization was not applied.
(R = 0.0)

Accuracy of the test data = 0.85



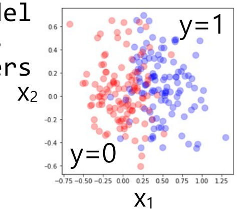
Regularization was applied.
(R = 0.001)

Accuracy of the test data = 0.87

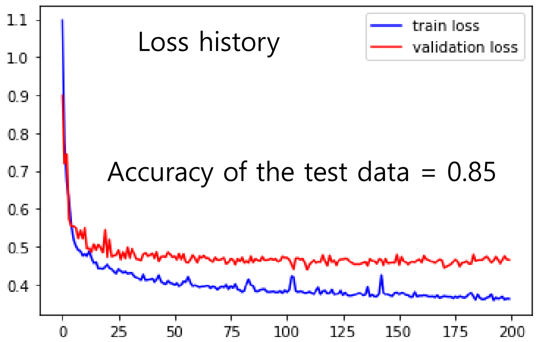
MX-AI

Implement regularization by Keras Regularizer

```python
# [MXDL-5-01] 2.wb_regularizer.py - Keras regularizer
import numpy as np
from sklearn.model_selection import train_test_split
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras import optimizers
from tensorflow.keras import regularizers
import matplotlib.pyplot as plt
import pickle

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

# Create an ANN model.
n_input = x_train.shape[1]  # number of input neurons
n_output = 1                # number of output neurons
n_hidden = 32               # number of hidden neurons

# Create Adam optimizer and L1, L2 regularizer
adam = optimizers.Adam(learning_rate=0.005)
L1 = regularizers.L1(0.001)
L2 = regularizers.L2(0.001)
L12 = regularizers.L1L2(0.001)

# The data is simple, but we intentionally added many
# hidden layers to the model to demonstrate the effect
# of regularization.
x_input = Input(batch_shape=(None, n_input))
h1 = Dense(n_hidden, activation = 'relu',
           kernel_regularizer = L1,
           bias_regularizer = L1)(x_input)

h2 = Dense(n_hidden, activation = 'relu',
           kernel_regularizer = L2,
           bias_regularizer = L2)(h1)
```

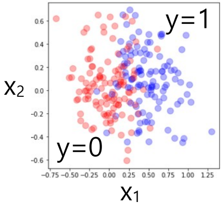



```python
h3 = Dense(n_hidden, activation = 'relu',
           kernel_regularizer = L12,
           bias_regularizer = L12)(h2)

h4 = Dense(n_hidden, activation = 'relu',
           kernel_regularizer = L1,
           bias_regularizer = L12)(h3)

h5 = Dense(n_hidden, activation = 'relu',
           kernel_regularizer = L12,
           bias_regularizer = L2)(h4)

y_output = Dense(n_output, activation='sigmoid',
                 kernel_regularizer = L12,
                 bias_regularizer = L2)(h5)
model = Model(x_input, y_output)
model.compile(loss='binary_crossentropy', optimizer=adam)

# training
f = model.fit(x_train, y_train,
              validation_data=(x_test, y_test),
              epochs=200, batch_size=20)

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

# Check the accuracy of the test data
y_pred = (model.predict(x_test) > 0.5) * 1
acc = (y_pred == y_test).mean()
print("\nAccuracy of the test data = {:.2f}".format(acc))
```



MX-AI

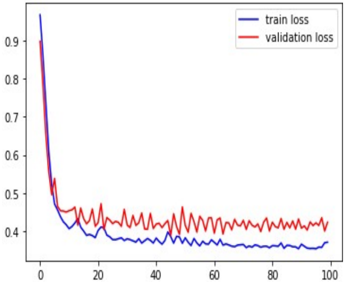
Implement regularization using custom regularizer

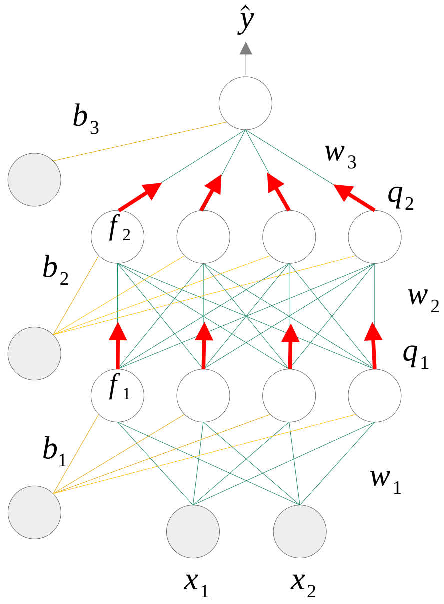



```
# [MXDL-5-01] 3.wb_custom_regularizer.py
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras import optimizers
from tensorflow.keras import regularizers
import matplotlib.pyplot as plt
import pickle

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

# Create an ANN model.
n_input = x_train.shape[1]  # number of input neurons
n_output = 1                # number of output neurons
n_hidden = 32               # number of hidden neurons
R = 0.01                    # regularization constant
adam = optimizers.Adam(learning_rate=0.005)

# Custom regularizer for L3 regularization
# L3 regularization is rarely used, but if you want to use
# it for some reason, you can implement it using a custom
# regularizer like this.
class reg_L3(regularizers.Regularizer):
    def __init__(self, reg_lambda):
        self.R = reg_lambda

    def __call__(self, x):
        # The w or b of a layer is passed to x.
        return self.R*tf.reduce_sum(tf.math.pow(tf.math.abs(x),3))

# The data is simple, but we intentionally added many hidden
# layers to the model to demonstrate the effect of regularization.
```

```
x_input = Input(batch_shape=(None, n_input))
h = Dense(n_hidden, activation = 'relu',
          kernel_regularizer=reg_L3(R),
          bias_regularizer=reg_L3(R))(x_input)

# 4 more hidden layers
for i in range(4):
    h = Dense(n_hidden, activation = 'relu',
              kernel_regularizer=reg_L3(R),
              bias_regularizer=reg_L3(R))(h)

y_output = Dense(n_output, activation='sigmoid',
                 kernel_regularizer=reg_L3(R),
                 bias_regularizer=reg_L3(R))(h)

model = Model(x_input, y_output)
model.compile(loss='binary_crossentropy',
              optimizer = adam)
h = model.fit(x_train, y_train, epochs=100, batch_size=50,
              validation_data=[x_test, y_test])

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

Accuracy of the test data = 0.84





$$L_{reg} = L + \lambda_1 \sum_k q_{1,k}^2 + \lambda_2 \sum_k |q_{2,k}|$$

$$w_3 = w_3 - \alpha \nabla_{w_3} L$$

$$w_2 = w_2 - \alpha \left(\nabla_{w_2} L + \lambda_2 \sum_k \nabla_{w_2} |q_{2,k}| \right)$$

5. Regularization

Part 2: Activity Regularization

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

www.youtube.com/@meanxai

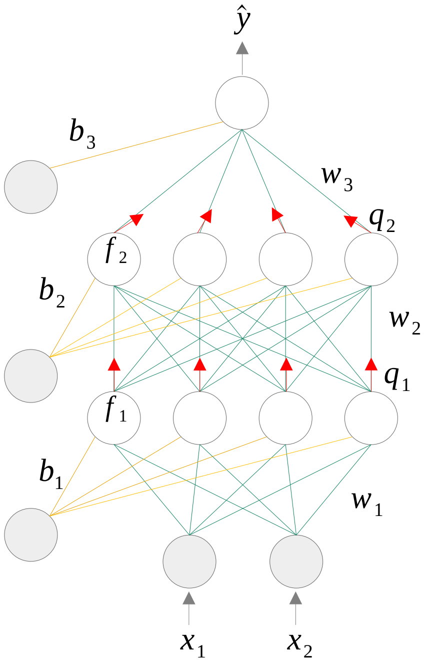
MX-AI

Activity (or Activation) Regularization

- Activity regularization prevents overfitting by imposing a penalty on the outputs of each layer to prevent them from becoming too large.
- This is more effective when there are many hidden neurons. This is because even if the weights are small, the output can increase if a lot of dot(q, w) are accumulated in a neuron.
- L1 activity regularization makes the output of some neurons zero, while L2 makes the output of neurons smaller overall.
- L1 activity regularization is most often used in sparse autoencoders to encourage sparse latent features that are the output of the encoder. This is to better capture salient features of the input data.

$$q_2 = f_2(q_1 \cdot w_2 + b_2)$$
$$= f_2(f_1(w_1 \cdot x + b_1) \cdot w_2 + b_2)$$

$$q_1 = f_1(w_1 \cdot x + b_1)$$



$$L_{reg} = L + \underbrace{\lambda_1 \sum_k q_{1,k}^2}_{} + \lambda_2 \sum_k |q_{2,k}|$$

$$(L_2) \qquad (L_1)$$

$$w_3 = w_3 - \alpha \nabla_{w_3} L$$

$$w_2 = w_2 - \alpha\left(\nabla_{w_2} L + \lambda_2 \sum_k \nabla_{w_2} |q_{2,k}|\right)$$

$$w_1 = w_1 - \alpha\left(\nabla_{w_1} L + \lambda_1 \sum_k \nabla_{w_1} q_{1,k}^2 + \lambda_2 \sum_k \nabla_{w_1} |q_{2,k}|\right)$$

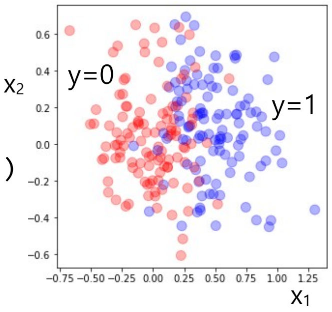
MX-AI

Applying activity regularization

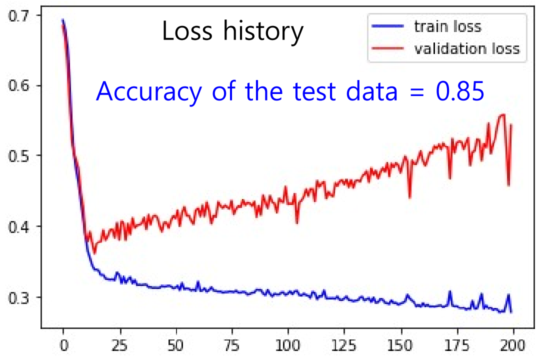
```python
# [MXDL-5-02] 4.act_regularizer(1).py
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras import optimizers, regularizers
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import pickle

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

# Create an ANN model.
n_input = x_train.shape[1]  # number of input neurons
n_output = 1                # number of output neurons
n_hidden = 32               # number of hidden neurons
adam = optimizers.Adam(learning_rate=0.001)
reg_L1 = regularizers.L1(0.01)

# The data is simple, but we intentionally added many
# hidden layers to the model to demonstrate the effect
# of regularization.
x_input = Input(batch_shape=(None, n_input))
h = Dense(n_hidden, activation = 'relu',
          activity_regularizer=reg_L1)(x_input)

# 4 more hidden layers
for i in range(4):
    h = Dense(n_hidden, activation = 'relu',
              activity_regularizer=reg_L1)(h)

y_output = Dense(n_output, activation='sigmoid')(h)

model = Model(x_input, y_output)
model.compile(loss='binary_crossentropy', optimizer=adam)
```

```python
f = model.fit(x_train, y_train, epochs=200, batch_size=20,
              validation_data=[x_test, y_test])

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

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

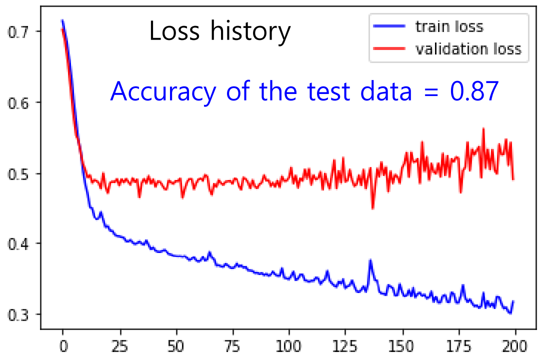
dataset



Before applying regularization.



After applying activity regularization.



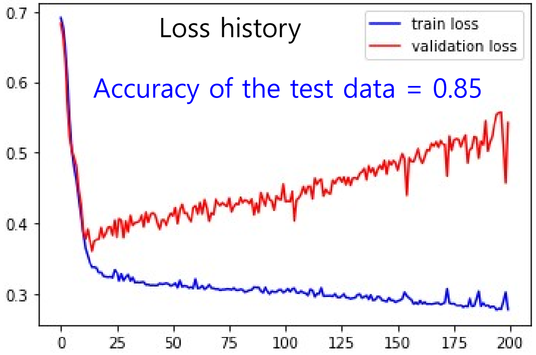
MX-AI

Applying activity regularization before activation function

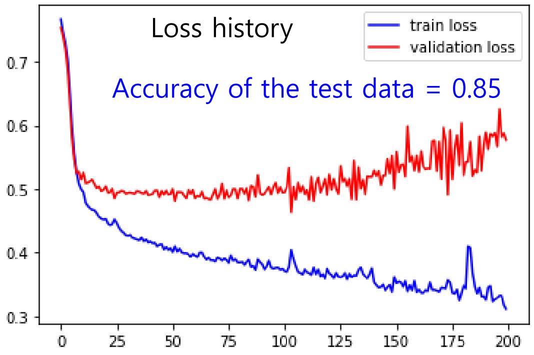
```
# [MXDL-5-02] 5.act_regularizer(2).py
# Activity regularization before activation function
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.layers import Activation
from tensorflow.keras.models import Model
from tensorflow.keras import optimizers, regularizers
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import pickle

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

# Create an ANN model.
n_input = x_train.shape[1]  # number of input neurons
n_output = 1                # number of output neurons
n_hidden = 32               # number of hidden neurons
adam = optimizers.Adam(learning_rate=0.001)
L1 = regularizers.L1(0.01)

# The data is simple, but we intentionally added many
# hidden layers to the model to demonstrate the effect
# of regularization.
x_input = Input(batch_shape=(None, n_input))
h = Dense(n_hidden, activity_regularizer=L1)(x_input)
h = Activation('relu')(h)

# 4 more hidden layers
for i in range(4):
    h = Dense(n_hidden, activity_regularizer=L1)(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)

f = model.fit(x_train, y_train, epochs=200, batch_size=20,
              validation_data=[x_test, y_test])

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

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

Before applying regularization.



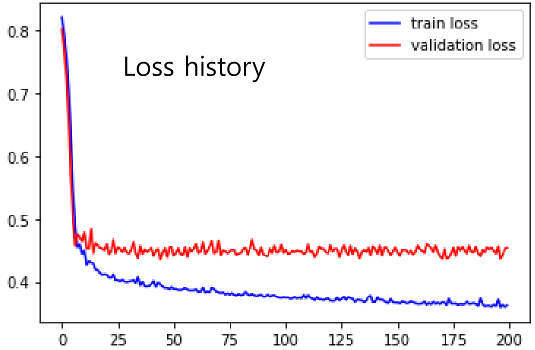
After applying activity regularization.



MX-AI

■ Applying weight, bias, activity regularization together

```
# [MXDL-5-02] 6.regularizer_all.py
# Applying weight, bias, activity regularization together
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras import optimizers, regularizers
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import pickle

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

# Create an ANN model.
n_input = x_train.shape[1]  # number of input neurons
n_output = 1                # number of output neurons
n_hidden = 32               # number of hidden neurons
adam = optimizers.Adam(learning_rate=0.001)
L1 = regularizers.L1(0.001)
L2 = regularizers.L2(0.001)

# The data is simple, but we intentionally added many hidden
# layers to the model to demonstrate the effect of regularization.
x_input = Input(batch_shape=(None, n_input))
h = Dense(n_hidden, activation='relu',
          kernel_regularizer = L2,
          bias_regularizer = L2,
          activity_regularizer = L1)(x_input)

# 4 more hidden layers
for i in range(4):
    h = Dense(n_hidden, activation='relu',
              kernel_regularizer = L2,
              bias_regularizer = L2,
              activity_regularizer = L1)(h)
y_output = Dense(n_output, activation='sigmoid',
                 kernel_regularizer = L2,
                 bias_regularizer = L2)(h)

model = Model(x_input, y_output)
model.compile(loss='binary_crossentropy',
              optimizer = adam)

h = model.fit(x_train, y_train, epochs=200, batch_size=20,
              validation_data=[x_test, y_test])

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



Accuracy of the test data = 0.87

MX-AI

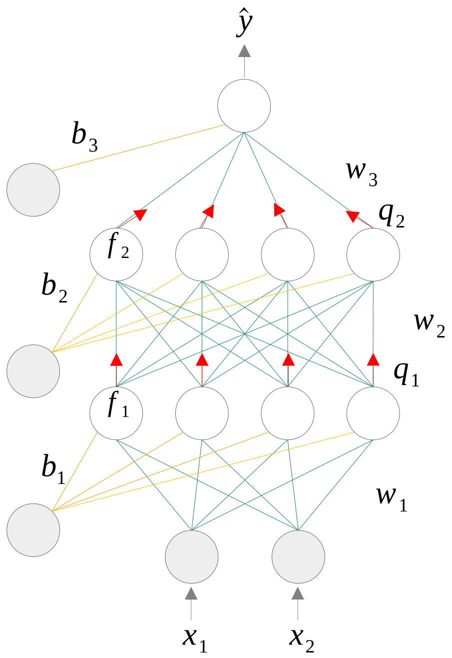
■ Imposing a sparsity constraint on the hidden neurons (KL divergence regularization).

- Imposing a sparsity constraint on hidden units can be done through activity regularization.
- For example, in the neural network below, we would like to constrain the average of q_2 to be around 0.05.
- This is primarily used in sparse autoencoders.

$$q_2 = f_2(q_1 \cdot w_2 + b_2)$$

$$= f_2(f_1(w_1 \cdot x + b_1) \cdot w_2 + b_2)$$

$$q_1 = f_1(w_1 \cdot x + b_1)$$



A neuron is assumed to be "active" if its output is close to 1, and "inactive" if its output is close to 0. The output q is a Bernoulli random variable.

$$\hat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m} q_{2,j}(x_i)$$

: Average activation value of hidden unit j for data x. Where m is the batch size of the data.

For example, we want $\hat{\rho}_j$ to be around ρ = 0.05, where ρ is a sparsity parameter, typically a small value close to zero. In other words, we would like the average activation of each hidden neuron j to be close to 0.05. To satisfy this constraint, the hidden unit's activations must mostly be near 0.

To achieve this, we will add a penalty term to our loss function. This penalizes $\hat{\rho}_j$ that deviates significantly from ρ. The following KL divergence can be used as a penalty term.

$$\sum_{j=1}^{n} KL(\rho \| \hat{\rho}_j) = \sum_{j=1}^{n} [\rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho)\log\frac{1-\rho}{1-\hat{\rho}_j}]$$

KL is the Kullback-Leibler divergence between a Bernoulli random variable with mean ρ and a Bernoulli random variable with mean $\hat{\rho}_j$.

The regularized loss function is:

$$L_{reg} = L + \lambda \sum_{j=1}^{n} KL(\rho \| \hat{\rho}_j)$$

* Source: CS294A Lecture notes by Andrew Ng (page 14)

MX-AI

- Imposing a sparsity constraint on the hidden neurons by activity regularization

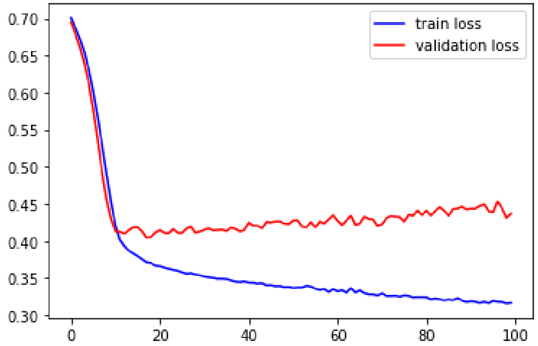
```python
# [MXDL-5-02] 7.act_custom_regularizer.py
# Imposing a sparsity constraint on the hidden neurons
# by activity regularization
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras import optimizers, regularizers
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import pickle

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

# Create an ANN model.
n_input = x_train.shape[1]  # number of input neurons
n_output = 1                # number of output neurons
n_hidden = 32               # number of hidden neurons
rho = 0.05
adam = optimizers.Adam(learning_rate=0.001)
L2 = regularizers.L2(0.01)

# Custom regularizer for KL divergence regularization
class KL_loss(regularizers.Regularizer):
    def __init__(self, reg_lambda, rho):
        self.R = reg_lambda
        self.rho = rho

    def __call__(self, q):
        rho_hat = tf.reduce_mean(q, axis=0)
        rho_hat = tf.clip_by_value(rho_hat, 1e-6, 0.99999)
        kl = self.rho * tf.math.log(self.rho / rho_hat) + \
            (1-self.rho) * tf.math.log((1-self.rho) / (1-rho_hat))
        return self.R * tf.reduce_sum(kl)
```

```python
# The data is simple, but we intentionally added many hidden
# layers to the model to demonstrate the effect of regularization.
x_input = Input(batch_shape=(None, n_input))
h = Dense(n_hidden, activation='relu',
          activity_regularizer=L2)(x_input)

# 3 more hidden layers
for i in range(3):
    h = Dense(n_hidden, activation='relu',
              activity_regularizer=L2)(h)

# last hidden layer
h_last = Dense(n_hidden, activation='relu',
               activity_regularizer=KL_loss(0.1, rho))(h)

y_output = Dense(n_output, activation='sigmoid')(h_last)

model = Model(x_input, y_output)
model.compile(loss='binary_crossentropy', optimizer = adam)

h_model = Model(x_input, h_last)

f = model.fit(x_train, y_train, epochs=100, batch_size=50,
              validation_data=[x_test, y_test])

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

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

MX-AI

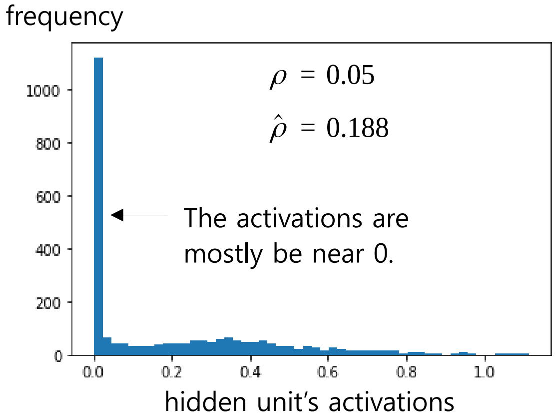
■ Imposing a sparsity constraint on the hidden neurons by activity regularization

```
# Check the distribution of the last hidden unit's activations.
h_act = h_model.predict(x_test).reshape(-1,)
plt.hist(h_act, bins=50)
plt.show()
print("Mean of the activations =", h_act.mean())
```

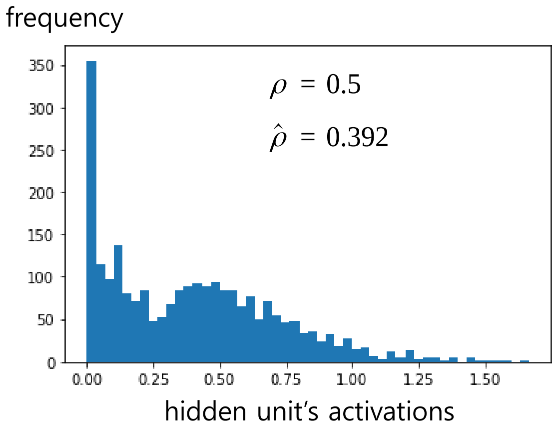


Accuracy of the test data = 0.85

Distribution of the last hidden unit's activations.



Mean of the activations ($\hat{\rho}$) = 0.188



Mean of the activations ($\hat{\rho}$) = 0.392