https://linuxtut.com/en/33867fbec742bda3f307/

If you position the model of machine learning and deep learning \*\* very roughly \*\*, it will look like this.

import numpy as np

import pandas as pds

from keras.models import Sequential

from keras.layers import Input, Dense, Dropout, BatchNormalization

from keras.wrappers.scikit\_learn import KerasRegressor

from sklearn.model\_selection import cross\_val\_score

from sklearn.model\_selection import KFold

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.pipeline import Pipeline

from sklearn.metrics import mean\_squared\_error

from sklearn.datasets import load\_diabetes

diabetes = load\_diabetes()

X = diabetes.data

Y = diabetes.target

![Table

Description automatically generated]()

It seems that it has already been normalized. It is a small sample data with 442 lines and 10 input variables.

![Graphical user interface, application, table

Description automatically generated]()

Click here for details of the data. http://web.stanford.edu/~hastie/Papers/LARS/LeastAngle\_2002.pdf

KerasRegressor Keras provides an API for regression analysis called Keras Regressor. https://keras.io/ja/scikit-learn-api/

Keras itself doesn't give much detail, but the point is that it seems to be a wrapper for scikit-learn's regression model. Perhaps Keras Regressor was created to work with scikit-learn's useful metric APIs for regression analysis (such as cross\_val\_score and mean\_squared\_error).

The way to write a neural network model is Keras itself. First, let's make a simple model (one layer each for the input layer, middle layer, and output layer).

# create regression model

def reg\_model():

model = Sequential()

model.add(Dense(10, input\_dim=10, activation='relu'))

model.add(Dense(16, activation='relu'))

model.add(Dense(1))

# compile model

model.compile(loss='mean\_squared\_error', optimizer='adam')

return model

![Table

Description automatically generated]()

There are roughly two ways to learn.

1. Learn by separating training data and test data
2. Learn with cross-validation

It seems that a general method can be used for regression analysis.

Example 1 Learning separately from training data and test data

# use data split and fit to run the model

x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.1, random\_state=0)

estimator = KerasRegressor(build\_fn=reg\_model, epochs=100, batch\_size=10, verbose=0)

estimator.fit(x\_train, y\_train)

y\_pred = estimator.predict(x\_test)

# show its root mean square error

mse = mean\_squared\_error(y\_test, y\_pred)

print("KERAS REG RMSE : %.2f" % (mse \*\* 0.5))

Finally, the standard output gives the square root of the mean squared error (root mean squared erro). The writing style is scikit-learn-like (but Keras is scikit-learn-like in the first place).

**Example 2 Learning by cross-validation**

Let's continue learning with cross-validation.

# use Kfold and cross validation to run the model

seed = 7

np.random.seed(seed)

estimator = KerasRegressor(build\_fn=reg\_model, epochs=100, batch\_size=10, verbose=0)

kfold = KFold(n\_splits=10, random\_state=seed)

# show its root mean square error

results = cross\_val\_score(estimator, X, Y, scoring='neg\_mean\_squared\_error', cv=kfold)

mse = -results.mean()

print("KERAS REG RMSE : %.2f" % (mse \*\* 0.5))

Here, too, the square root of the mean square error is given at the end. Let's arrange each result.

![Graphical user interface, text, application, email

Description automatically generated]()

Well, it doesn't make much difference.

Let's try deepening the network layer

I have done multiple regression analysis with a simple neural network so far. Now let's try deepening the network layer.

import numpy as np

import pandas as pds

from keras.models import Sequential

from keras.layers import Input, Dense, Dropout, BatchNormalization

from keras.wrappers.scikit\_learn import KerasRegressor

from sklearn.model\_selection import cross\_val\_score

from sklearn.model\_selection import KFold

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.pipeline import Pipeline

from sklearn.metrics import mean\_squared\_error

from sklearn.datasets import load\_diabetes

diabetes = load\_diabetes()

X = diabetes.data

Y = diabetes.target

# create deep learning like regression model

def deep\_reg\_model():

model = Sequential()

model.add(Dense(10, input\_dim=10, activation='relu'))

model.add(BatchNormalization())

model.add(Dropout(0.2))

model.add(Dense(256, activation='relu'))

model.add(BatchNormalization())

model.add(Dropout(0.2))

model.add(Dense(128, activation='relu'))

model.add(BatchNormalization())

model.add(Dropout(0.2))

model.add(Dense(64, activation='relu'))

model.add(BatchNormalization())

model.add(Dropout(0.2))

model.add(Dense(1))

# compile model

model.compile(loss='mean\_squared\_error', optimizer='adam')

return model

Since it's a big deal, I added Batch normalization and Dropout.

![Table

Description automatically generated]()

# use data split and fit to run the model

x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.1, random\_state=0)

estimator = KerasRegressor(build\_fn=deep\_reg\_model, epochs=100, batch\_size=10, verbose=0)

estimator.fit(x\_train, y\_train)

y\_pred = estimator.predict(x\_test)

# show its root mean square error

mse = mean\_squared\_error(y\_test, y\_pred)

print("KERAS REG RMSE : %.2f" % (mse \*\* 0.5))

OUTPUT

KERAS REG RMSE : 59.49

# use Kfold and cross validation to run the model

seed = 7

np.random.seed(seed)

estimator = KerasRegressor(build\_fn=deep\_reg\_model, epochs=100, batch\_size=10, verbose=0)

#kfold = KFold(n\_splits=10, random\_state=seed)

kfold = KFold(n\_splits=10)

# show its root mean square error

results = cross\_val\_score(estimator, X, Y, scoring='neg\_mean\_squared\_error', cv=kfold)

mse = -results.mean()

print("KERAS REG RMSE : %.2f" % (mse \*\* 0.5))

OUTPUT

KERAS REG RMSE : 56.67

It's not much different from a simple network. Considering the calculation time, there is no point in deepening it.

X = np.array([ ])

prediction = estimator.predict(X)