Υπολογιστική Νοημοσύνη

Επίλυση προβλήματος παλινδρόμησης με χρήση RBF δικτύου

Θεόδωρος Κατζάλης
ΑΕΜ: 9282
katzalis@ece.auth.gr

Αριστοτέλειο Πανεπιστήμιο Θεσσαλονικής

March 6, 2023

Περιεχόμενα

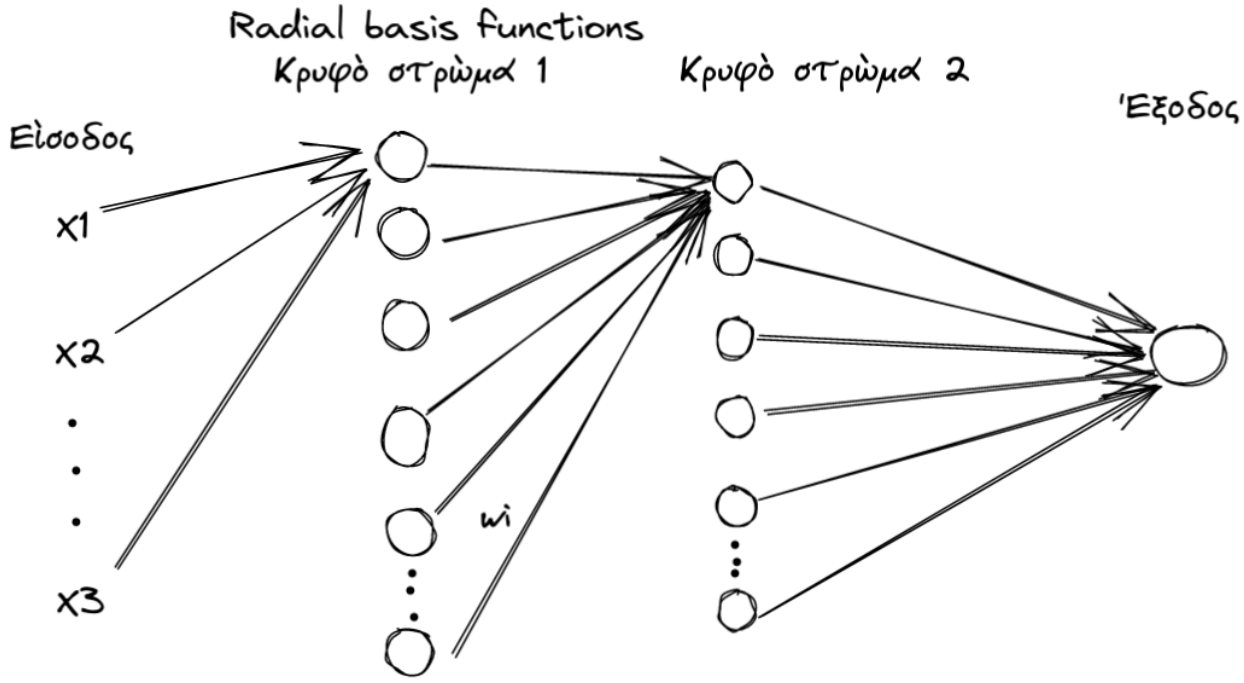
1 Εισαγωγή

Η συγκεκριμένη εργασία πραγματεύεται την επίλυση ενός προβλήματος παλινδρόμησης (regression) χρησιμοποιώντας ένα νευρωνικό δίκτυο με αρχιτεκτονική RBF. Το dataset στο οποίο έγινε η μελέτη είναι το λεγόμενο Boston housing, το οποίο αποτελείται απο 506 δείγματα σπιτιών, με 13 χαρακτηριστικά για το καθένα (έχοντας αφαιρέσει το B), και των τιμών τους. Στόχος είναι η πρόβλεψη της τιμής ενός σπιτιού με βάση αυτά τα 13 χαρακτηριστικά αλλά και η κατανόηση της επίδρασης των παραμέτρων του δικτύου για την βελτιστοποίηση της απόδοσης του. Η υλοποίηση της εργασίας έγινε με την βιβλιοθήκη `Keras`.

2 RBF KMeans

Τα νευρωνικά δίκτυα αρχιτεκτονικής RBF συνήθως αποτελούνται απο 3 επίπεδα. Το επίπεδο εισόδου, ένα κρυφό στρώμα και η έξοδος. Στην συγκεκριμένη ανάλυση θα χρησιμοποιήσουμε ένα επιπλέον κρυφό στρώμα με 128 νευρώνες. Τέλος, η έξοδος μας θα έχει έναν νευρώνα για να οδηγηθούμε σε μια βαθμωτή τιμή.

Η κεντρική ιδέα αυτών των δικτύων είναι η χρήση radial basis functions ως συναρτήσεις ενεργοποίησης για το πρώτο κρυφό στρώμα του δικτύου. Αρχικά επιλέγουμε ορισμένα κέντρα του χώρου και υπολογίζουμε την απόσταση των σημείων απο αυτά τα κέντρα. Η τιμή αυτή, στη συνέχεια, αποτελεί είσοδο των radial basis functions και δηλώνει την ομοιότητα του σημείου με το κέντρο. Έτσι πετυχαίνουμε έναν μετασχηματισμό εισόδου. Πιο συγκεκριμένα έχουμε την ακόλουθη μαθηματική ανάλυση:



Σχήμα 1: Αρχιτεκτονική δικτύου

Η είσοδος μετασχηματίζεται με την χρήση radial basis functions ως:

$$f(x) = e^{\frac{-||x-c_i||^2}{2\sigma_i^2}}$$

όπου c_i, το κέντρο του i-οστού νευρώνα του πρώτου κρυφού στρώματος και x το διανύσμα είσόδου.

Ως σ_i ορίζουμε:

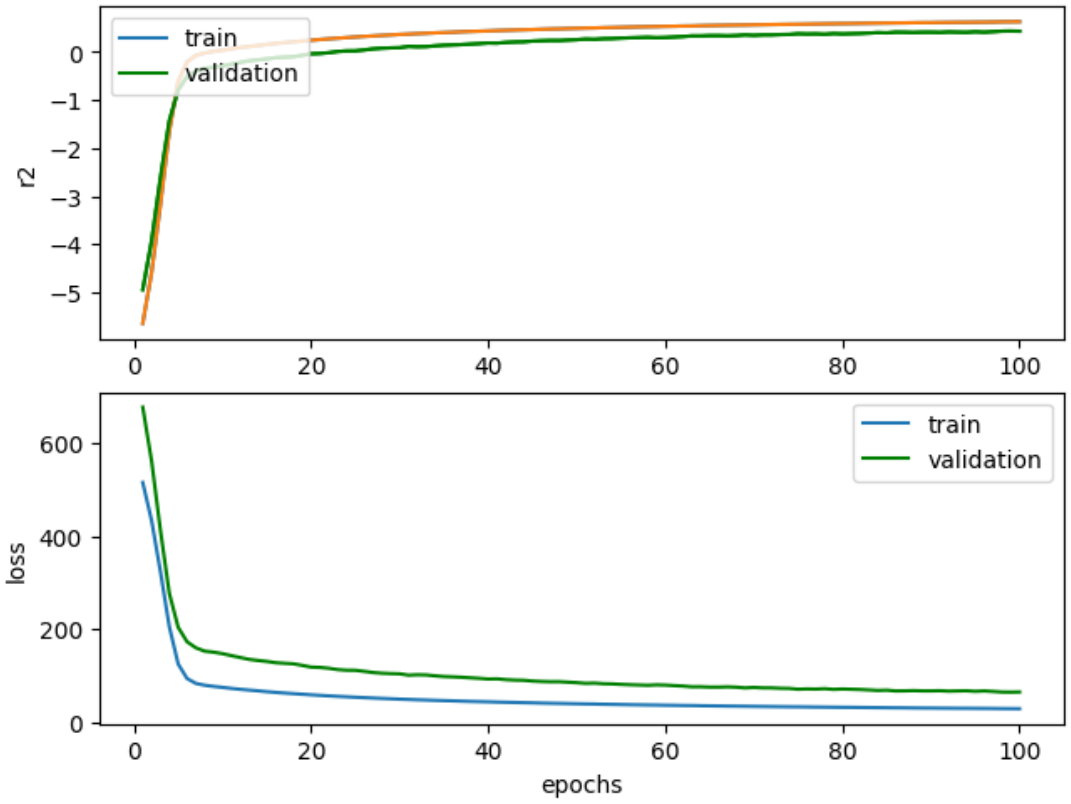
$$\sigma = \frac{d_{max}}{\sqrt{2P}}$$

όπου d_{max}, η μέγιστη απόσταση μεταξύ των κέντρων και P, ο αριθμός των κέντρων. Το υπόλοιπο του δικτύου, έπειτα απο το πρώτο κρυφό στρώμα, είναι ένα συμβατικό MLP δίκτυο.

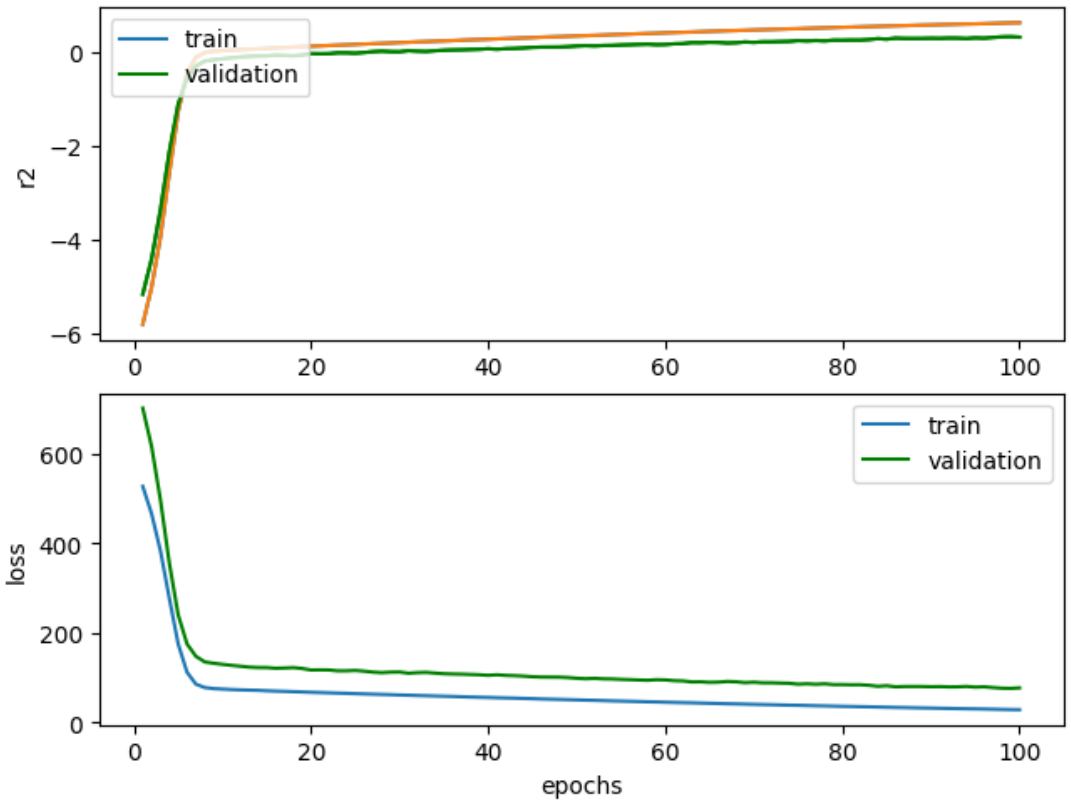
Στη συνέχεια θα μας απασχολήσει η εκπαίδευση αυτού του δικτύου για μεταβαλλόμενο αριθμό νευρώνων στο πρώτο κρυφό στρώμα. Για την επιλογή των κέντρων θα χρησιμοποιήσουμε τον αλγόριθμο ομαδοποίησης KMeans. Αξίζει να σημειωθεί ότι ο υπολογισμός των κέντρων δεν αποτελεί αντικείμενο προπόνησης. Τα κέντρα υπολογίζονται μια φορά και παραμένουν σταθερά καθόλη την διάρκειά της. Για να υλοποίησουμε ένα τέτοιο στρώμα με την χρήση `keras`, δημιουργήσαμε την κλάση `RBFWithKMeansLayer`, κληρονομώντας απο την `Layer`. Προσθέτουμε τα κέντρα ως βάρη του στρώματος (`add_weight()`), αρχικοποιώντας τα με την χρήση του `initializer = InitCentersKMeans()` και επιλέγουμε `trainable=False`. Τέλος, υπολογίζουμε την έξοδο του στρώματος, υλοποιώντας την συνάρτηση `build()`. Η ακριβής υλοποίηση φαίνεται στην ενότητα 4.

metrics \ Αριθμός νευρώνων	37	189	341
RMSE	5.88	7.09	7.71
R^2	0.54	0.33	0.21

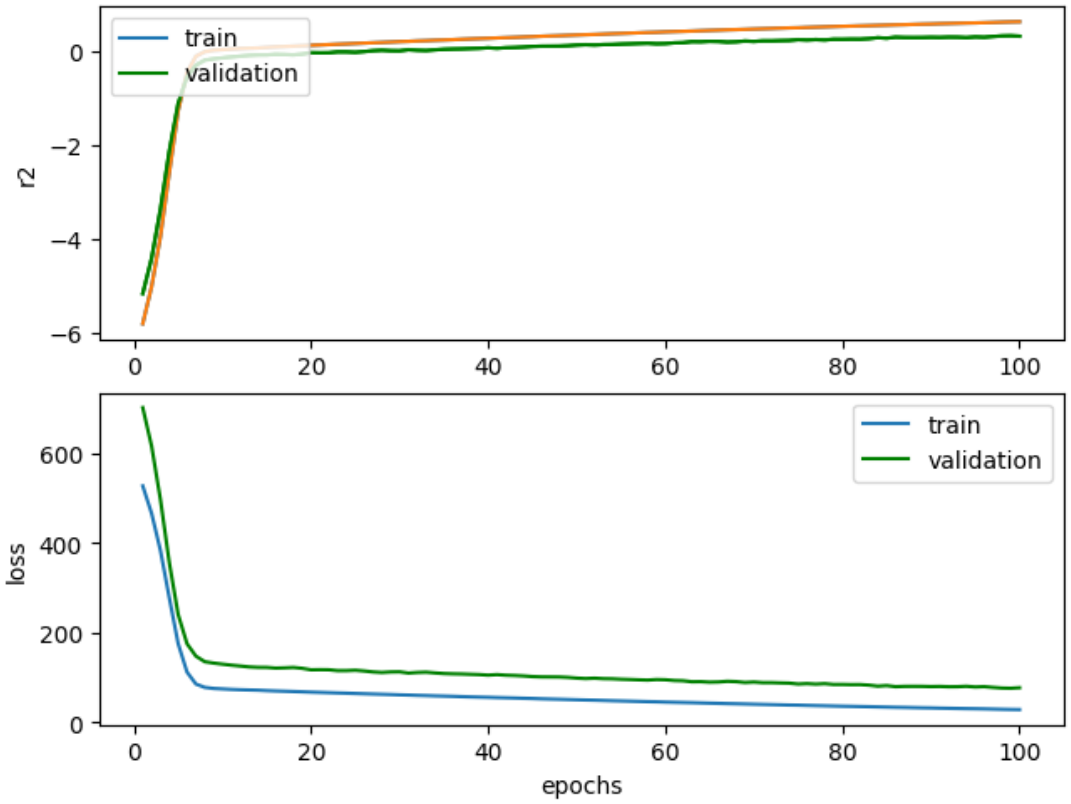
Πίνακας 1: Τιμές RMSE και R^2 συναρτήσει του αριθμού νευρώνων



Σχήμα 2: Αριθμός νευρώνων = 37



Σχήμα 3: Αριθμός νευρώνων = 189



Σχήμα 4: Αριθμός νευρώνων = 341

Παρατηρούμε ότι η απόδοση του μοντέλου είναι καλύτερη με μικρότερο αριθμό νευρώνων. Ο αριθμός των νευρώνων έχει άμεση συσχέτιση με την πολυπλοκότητα του μοντέλου. Στόχος του δικτύου θα πρέπει να είναι η εύρεση της τιμής αυτών των νευρώνων για την πολυπλοκότητα του dataset που μελετάται.

3 Fine tuning

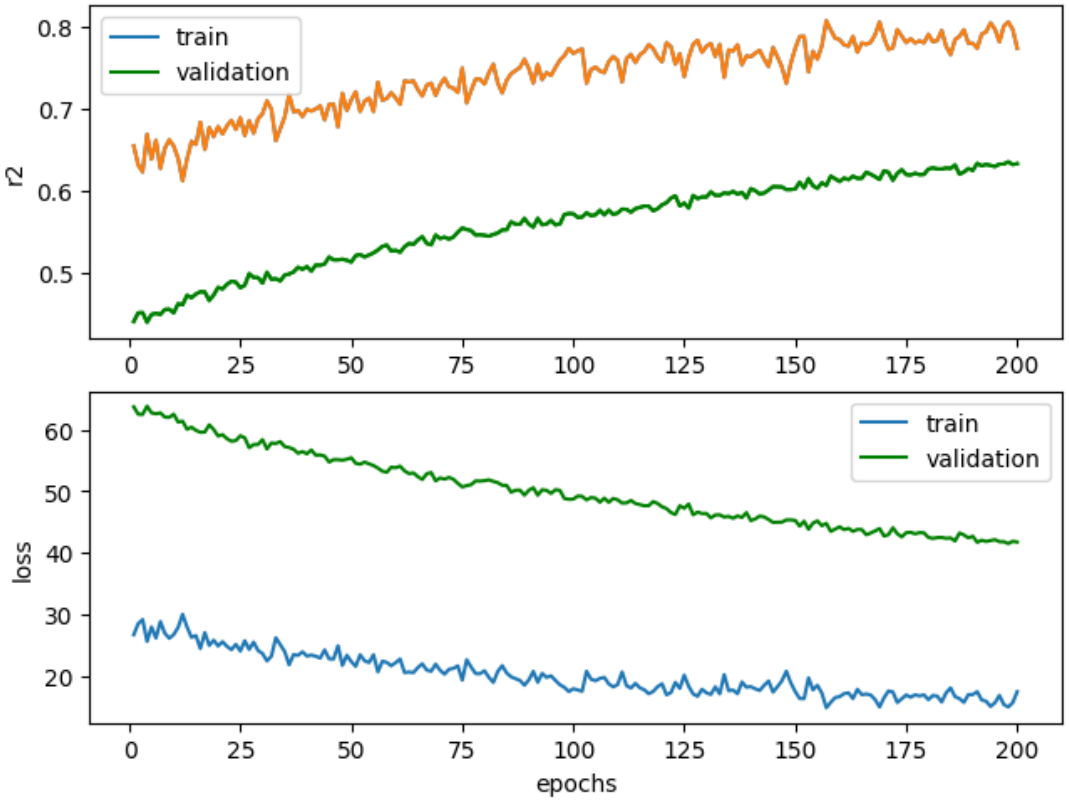
Σε αυτό το κομμάτι της εργασίας, θα πειραματιστούμε με την εύρεση βέλτιστων υπερπαραμέτρων. Θα μας απασχολήσουν, ο αριθμός των νευρώνων RBF και του δεύτερου κρυφού στρώματος αλλά και η πιθανότητα dropout. Έπειτα απο την εξερεύνηση των τιμών με κριτήριο αξιολόγησης το RMSE, έχουμε τις ακόλουθες βέλτιστες υπερπαραμέτρους:

$$\text{Νευρώνες RBF} = 56$$

$$\text{Νευρώνες 2ου κρυφού στρώματος} = 128$$

$$\text{Πιθανότητα dropout} = 0.35$$

Με βάση τις παραπάνω τιμές, προπονήσαμε το βέλτιστο μοντέλο:



Σχήμα 5: Καμπύλες εκμάθησης βέλτιστου μοντέλου $RMSE = 5.34$, $R^2 = 0.62$

Παρατηρούμε ότι έχουμε καλύτερη απόδοση, μικρότερη τιμή RMSE και υψηλότερη R^2, σε σχέση με τις προηγούμενες αναλύσεις.

4 Python

In [ ]:
```
import numpy as np
import os
import random as python_random
from keras.datasets import boston_housing
import tensorflow as tf
from sklearn.preprocessing import StandardScaler
from keras import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import SGD
import keras_tuner as kt
from keras_tuner import Objective
import tensorflow as tf
import tensorflow_addons as tfa
from keras.metrics import RootMeanSquaredError
from tensorflow_addons.metrics import RSquare
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Layer
from scipy.spatial.distance import pdist
from keras.initializers import Initializer
from sklearn.cluster import KMeans
import tensorflow.keras.backend as K



def set_seed(seed):
    os.environ["PYTHONHASHSEED"] = str(seed)
    os.environ["TF_CUDNN_DETERMINISTIC"] = str(seed)

    # source: https://keras.io/getting_started/faq/#how-can-i-obtain-reproducible-res
    # source: https://github.com/keras-team/keras/issues/2743
    np.random.seed(seed)
    python_random.seed(seed)
    tf.random.set_seed(seed)
```

In [ ]:
```
# Helpers
# plot learning curves
def plot_history(history):
    epochs = len(history.history["accuracy"])
    x = np.arange(1, epochs + 1)
    plt.figure(constrained_layout=True)
    plt.subplot(211)
    plt.plot(x, history.history["r_square"])
    plt.plot(x, history.history["val_r_square"], color="green")
    plt.legend(["train", "validation"], loc="upper left")

    plt.subplot(211)
    plt.plot(x, history.history["r_square"])
    plt.plot(x, history.history["val_r_square"], color="green")
    plt.ylabel("r2")
    plt.legend(["train", "validation"], loc="upper left")


    plt.subplot(212)
    plt.plot(x, history.history["loss"])
    plt.plot(x, history.history["val_loss"], color="green")
    plt.xlabel("epochs")
    plt.ylabel("loss")
    plt.legend(["train", "validation"], loc="upper right")
```

In [ ]:
```
# the feature B isn't included to avoid ethnical problems
# source: https://keras.io/api/datasets/boston_housing/
(train_x, train_y), (test_x, test_y) = boston_housing.load_data(test_split=0.25)
```

```
num_samples, num_features = train_x.shape

# z-score normalization
scaler = StandardScaler()
train_x = scaler.fit_transform(train_x)
test_x = scaler.transform(test_x)
```

In [ ]:

```
class InitCentersKMeans(Initializer):
    def __init__(self, X, max_iter=100):
        self.X = X
        self.max_iter = max_iter

    def __call__(self, shape, dtype=None):
        assert shape[1] == self.X.shape[1]

        n_centers = shape[0]
        km = KMeans(n_clusters=n_centers, max_iter=self.max_iter, verbose=0)
        km.fit(self.X)
        return km.cluster_centers_

class RBFWithKMeansLayer(Layer):
    def __init__(self, output_dim, train_data, **kwargs):
        self.output_dim = output_dim
        self.initializer = InitCentersKMeans(train_data)
        super(RBFWithKMeansLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        self.centers = self.add_weight(name='centers',
                                       shape=(self.output_dim, input_shape[1]),
                                       initializer=self.initializer,
                                       trainable=False)
        max_dist = max(pdist(self.centers))
        sigma = max_dist / np.sqrt(2 * self.output_dim)
        self.gamma = 1/(2*(sigma**2))
        super(RBFWithKMeansLayer, self).build(input_shape)

    def call(self, x):
        C = K.expand_dims(self.centers)
        H = K.transpose(C-K.transpose(x))
        return K.exp(-self.gamma * K.sum(H**2, axis=1))


def rbf_model(num_neurons):
    model = Sequential()
    model.add(RBFWithKMeansLayer(num_neurons, train_data=train_x, input_shape=(num_fe
    model.add(Dense(128))
    model.add(Dense(1))
    model.compile(optimizer=SGD(learning_rate=0.001),
                  loss="mse",
                  metrics=["accuracy", RootMeanSquaredError(), RSquare()])
    return model

def fit(model):
    return model.fit(train_x, train_y, batch_size=32, epochs=100, validation_split=0.


hidden_neurons = [int(0.1*num_samples), int(0.5*num_samples), int(0.9*num_samples)]
for neurons in hidden_neurons:
    set_seed(1)
    print("Neurons", neurons)
    model = rbf_model(neurons)
    plot_history(fit(model))
    print("Evaluation", model.evaluate(test_x, test_y))
```

In [ ]:

```
#Fine tuning
```

```
def build_model(hp):
    rbf_neurons = hp.Choice("rbf_neurons", values=[int(0.05*num_samples), int(0.15*nu
    hidden_layer_nodes = hp.Choice("hidden_layer_nodes", values=[32, 64, 128, 256])
    dropout_prob = hp.Choice("dropout_prob", values=[0.2, 0.35, 0.5])
    model = Sequential()
    model.add(RBFWithKMeansLayer(rbf_neurons, train_data=train_x, input_shape=(num_fe
    model.add(Dense(hidden_layer_nodes))
    model.add(Dropout(dropout_prob))
    model.add(Dense(1))
    model.compile(optimizer=SGD(learning_rate=0.001),
                  loss="mse",
                  metrics=["accuracy", RootMeanSquaredError(), RSquare()])
    return model


build_model(kt.HyperParameters())

tuner = kt.Hyperband(hypermodel=build_model, objective=Objective("val_root_mean_squar
tuner.search(
    train_x,
    train_y,
    validation_split=0.2,
    epochs=100,
)
tuner.results_summary()

best_model = tuner.get_best_models(num_models=1)[0]
history = best_model.fit(train_x, train_y, epochs=1000, validation_split=0.2)
loss_val, accuracy_val, rmse, r2 = best_model.evaluate(test_x, test_y)

print("loss:" + str(loss_val))
print("accuracy:" + str(accuracy_val))
print("rmse:" + str(rmse))
print("r2:" + str(r2))

# learning curves
plot_history(history)
```