Assignment 5: Rainfall runoff modelling using ANN

PAN XINXIN STU_ID: A0283981N

1. Use suitable artificial neural network architecture to produce 10-, 20- and 60-minute forecast of flow rates at location Main Drain_04;
2. Present and discuss forecast accuracy as function of lead time;
3. Explain which neural network architecture did you use and why;
4. Discuss choice of training, cross-validation and testing data sets;
5. Present and discuss results using testing data set.

Contents

Solution: 2
1. Input features of discharge 2
2. Discuss choice of training and testing data sets 3
3. Train a simple neural network using 'neuralnet' 5
4. Train a MLP model using 'keras' 6
5. Train a LSTM model by .py 8
6. Produce 10-, 20- and 60-minute forecast 10
7. Discuss forecast accuracy as function of lead time 11
8. Choose a neural network architecture for the catchment evaluation 12
9. Appendix: Script in r 13
9.1. The neural network by 'neuralnet' function 13
9.2. MLP by keras() 14
9.3. LSTM in .py 17

Solution:

1. Input features of discharge

The first thing is to define the number of input discharge locations to forecast Q_MD04, aiming to the better model fitting and less compute cost, and it is necessary to consider comprehensively the rainfall-runoff paths at the whole catchment (layout seen in Fig 1) that may affect the flow rates Q_MD04.

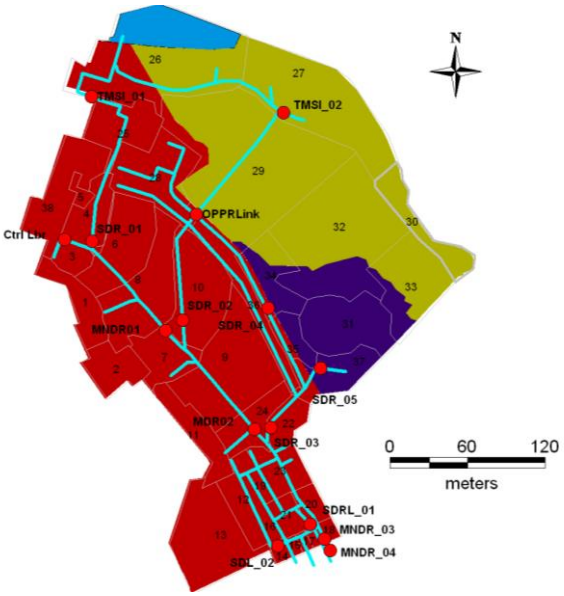


Fig 1. Discharge locations at the Kent Ridge Catchment

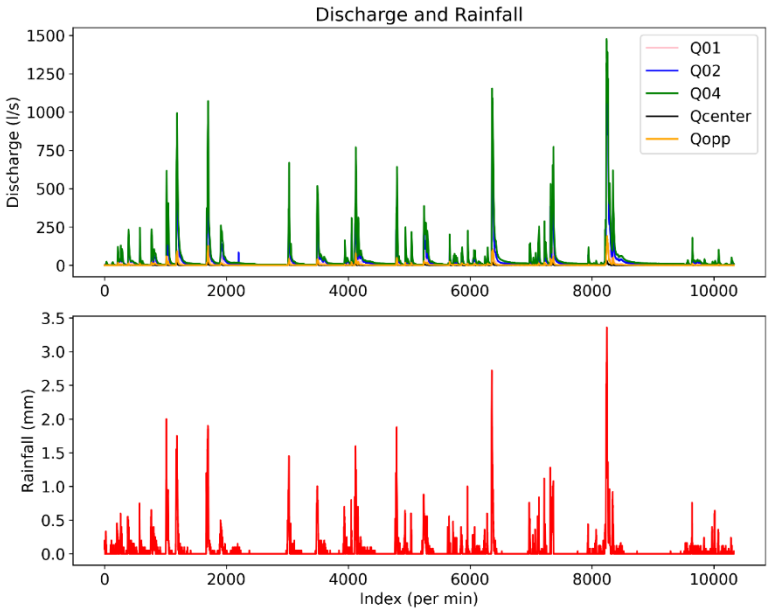


Fig 2. Rainfall-Runoff condition in one continue event

Thus, we compare their correlation and relative locations, and then use the model to simply verify the chose inputs.

1) Compare the higher correlation with Q_MD04

```
> cor_versue_Q04
Q_MD01      Q_MD02    Q_CNTRLIB    Q_OPPRLINK
 0.8702252  0.9789598  0.8496635    0.9534476
```

From the above correlations versus Q_MD04, the smallest one occurs in Q_ CNTRLIB with 0.85, although still shows relatively high correlation. Considering that Q_ CNTRLIB is far away from Q_MD04 and located on the boundary of this small catchment, we can ignore Q_CNTRLIB, also use the Q_opp located in the middle allows input discharge locations to cover a larger area for the rainfall.

2) Use MLP (RNNs) model to simply verify the choice of inputs

The MLP model set-up is seen in 4 Train a MLP model using ‘keras’, and the default split percentage is 80%(training)+20%(validation).

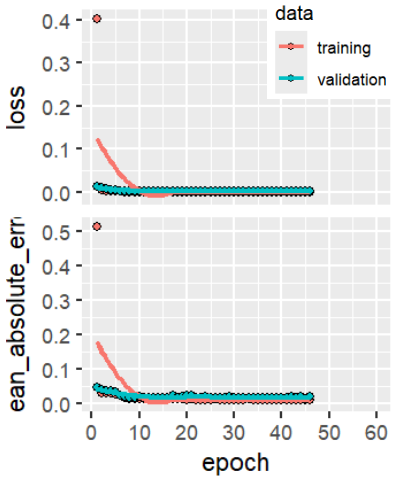


Fig 3. loss and error of T1

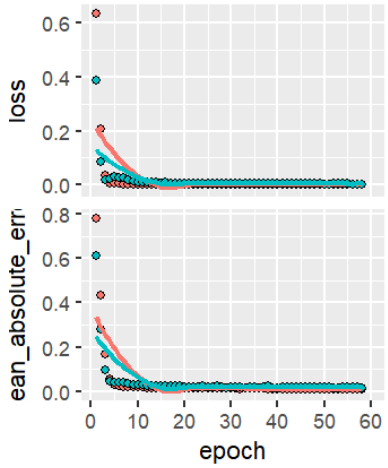


Fig 4. loss and error of T2

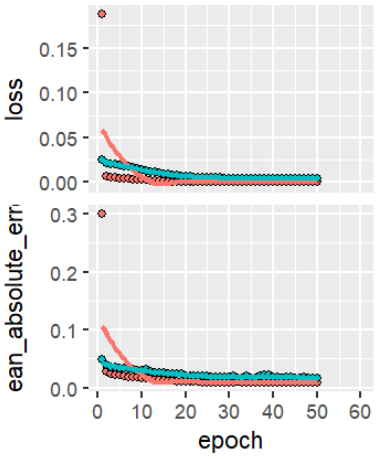


Fig 5. loss and error of T3

Table 1. 3 schemes of input features and statistical indicators

No.	Input features	MLP layer	accuracy	correlation	RMSE
T1	R+4Q	5+10	0.857	0.994	10.966
T2	R+3Q (without Q_CNTRLIB)	4+8	0.955	0.994	11.695
T3	R+2Q (Q_01+Q_02)	3+6	0.931	0.992	12.466

where the formula of accuracy: $accuracy = 1 - abs\_deviation = 1 - |\frac{actual - predicted}{actual}|$, and RMSE $= \sqrt{\frac{\sum(actual - predicted)^2}{n}}$.

From the training and validation loss and error (Fig 3~Fig 5), the difference is small, indicating that the MLP structure is suitable to avoid the overfitting or underfitting.

From the statistical indicators of the model, it shows that the accuracy and correlation in T2 is largest, while the RMSE is relatively large, indicating that the overall spread or dispersion of the predictions around the actual values is smaller, but some individual predictions might be further from the actual values (as indicated by the higher RMSE). Consider the discharge condition we focus on includes its overall trend and extreme discharge, T2 is a suitable choice.

Therefore, the input variables are **rainfall+Q01+Q02+Q_opp**.

2. Discuss choice of training and testing data sets

In deep learning model training, it's common to split the dataset into training and validation sets. The training set is used for updating model parameters, while the validation set is used to evaluate the modeling performance.

In this comparison, we use MLP model to compare and choose suitable splitting friction, and the model's set-up is seen in 4 Train a MLP model using 'keras'. The comparison is shown in Fig 6, and S2 and S3 avoid the test dataset ranging from the peak value of Q_MD04.

From the prediction (in Fig 7) and corresponding statistical indicators (in Table 2), the better prediction is based on S1. Although the RMSE is relatively high, showing an insufficient predictive performance in some cases, its prediction in the overall trend is

better with a higher accuracy. Besides, the test sets in the model commence with peak values, but it does not hinder the more correct prediction, demonstrating the model's strong robust adaptability.

In S2, for test sets with initially lower discharge, the prediction exhibits a slight underestimation compared to the actual values, resulting in an overall downward shift in predictions.

In S3, to avoid the overfitting, the smaller units are used. And its prediction in overall trend is worse than S1 and that in extreme peak is worse than S2.

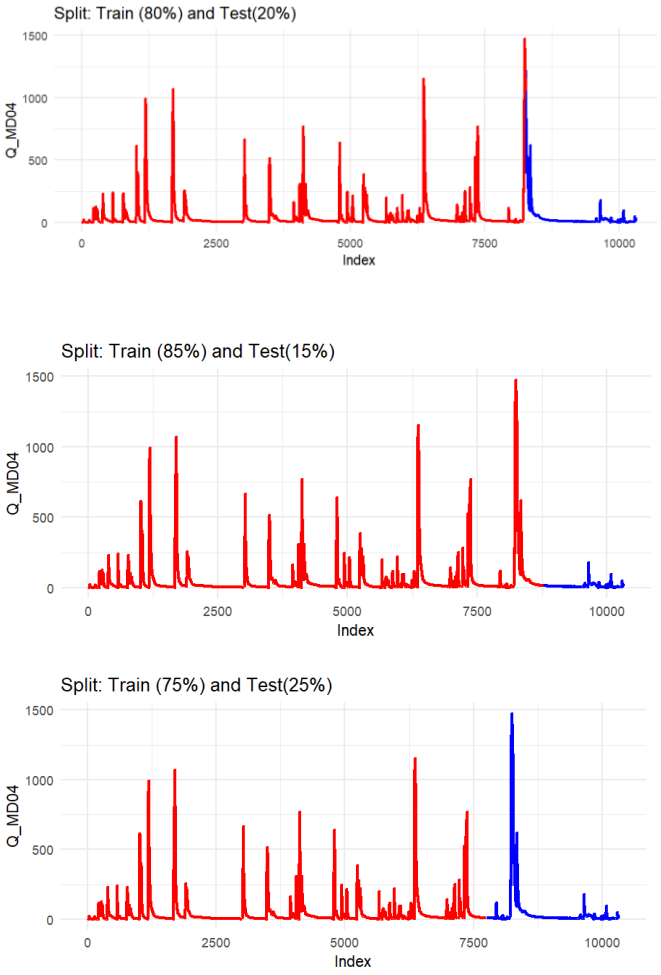


Fig 6. 3 split choices in dataset

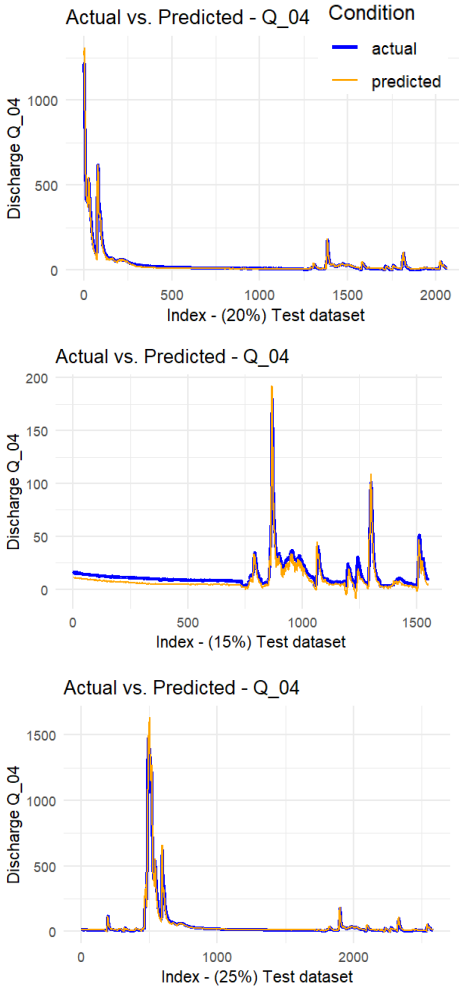


Fig 7. Comparison in test sets

Table 2. Comparison of 3 split choices and statistical indicators

No.	Split Choice	MLP layer	accuracy	correlation	RMSE
S1	80% + 20%	4+8	0.955	0.994	11.695
S2	85% + 15%	5+10	0.612	0.959	5.946
S3	75% + 25%	3+6	0.799	0.990	24.013

Thus, as we want to capture the overall trend, the split percentage as 80% training + 20% validation(test) is more suitable, because of the highest accuracy and its RMSE falling into the center ranges of 3 schemes.

Additionally, we can infer that for discharge datasets characterized by low overall trends and localized occurrences of peak, the corresponding RNNs model does not

expand prediction errors even that the training sets start from lower value and the test sets commence with larger values, referring to suitable splitting choice. However, if commencing the test sets with lower values, that may cause a decrease in overall prediction accuracy.

Also, it seems the friction (80%+20%) is a better balance of the sufficient data of training and validation.

3. Train a simple neural network using 'neuralnet'

There is a simple discussion about the model with multiple hidden layers using 'neuralnet()'.

```
# model1 - 2 hidden layers with 4~6 units in the 1st layer and 2~3 units in 2nd
# logistic sigmoid function; (rainfall, Q01, Q02, Qopp ~ Q04)
nn<- neuralnet(Q04 ~ rainfall+Q01+Q02+Qopp, data=train,
                    hidden=c(4,2), linear.output=FALSE, threshold=0.01)
plot(nn)
```

we can use the neural network plot showing weights and bias to roughly evaluate the model (in Fig 8). The dataset contains 4 input variables and 1 output variable.

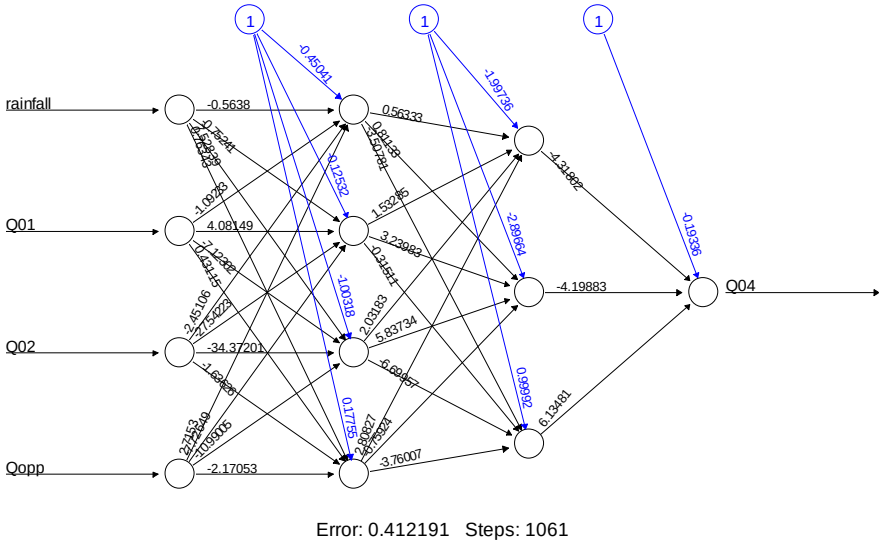


Fig 8. model architecture - by neuralnet function

Based on the model architecture, we modify the units, and then check the magnitude of weights and biases to ensure they are not too large or too small. Large weights indicate overfitting, while too small weights lead to vanishing gradients during training.

Then, we train several groups of multiple hidden layers (Table 3). Their accuracies are hard to exceed 90%, showing an obvious limitation of the predictive performance.

Table 3. Trials and tests for Model 1

No.	MLP layer	accuracy	correlation
Neur1	5+3	0.767	0.992
Neur2	**4+3**	**0.895**	**0.990**
Neur3	6+3	0.749	0.991
Neur4	5+2	0.752	0.991
Neur5	4+2	0.840	0.991

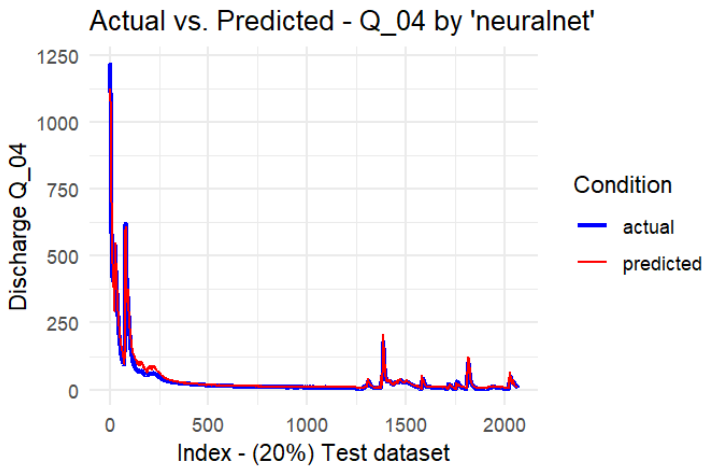


Fig 9. The better predictive performance in Neur2 model

4. Train a MLP model using 'keras'

Before creating a model based on time series datasets, it is necessary to check whether we need to detrend data. Script in r is seen in 9.2 MLP by keras().

As the results by undiff() and diff() are similar, showing no obvious trend, and we can use Q_MD04 to create model by an explicit method. Since every feature has values with varying ranges, we do normalization to confine feature values to a range of [0, 1] before training a neural network, by subtracting the mean and dividing by the standard deviation of each feature. This process is also taken in LSTM modeling.

In r, we use keras() function to create the MLP model with dense (fully connected) layers. MLP is suitable for handling non-sequential data, as it disregards the order or temporal relationships in the datasets, treating each data point as an independent input. Also, this function can use layer_lstm() to better capture long-term dependencies in datasets like rainfall-runoff features in LSTM model. In this case, we create the LSTM model using py in 5 Train a LSTM model by .py.

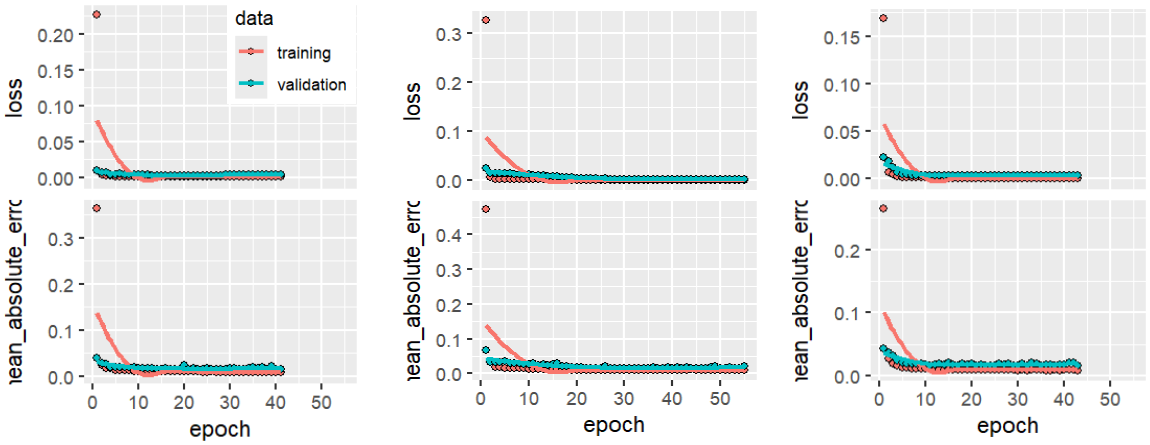
For MLP, it is important to do some trials and tests for the number of units in the 2nd hidden layers.

```
# keras model setup - MLP #####
# 1) Dense (fully connected) layers
model = keras_model_sequential() %>%
   layer_dense(units = 4, activation = 'relu', input_shape = dim(x_train)[2]) %>%
# Input layer
   layer_dense(units = 8, activation = 'relu') %>%   # Hidden layer
   layer_dense(units = 1)   # Output layer

#model compile
model %>% compile(loss = 'mse',
                  optimizer = 'adam', #optimizer_rmsprop(lr = 0.001)
                  metrics = list("mean_absolute_error")
```

```
)

early_stop <- callback_early_stopping(monitor = "val_loss", patience = 40,
min_delta=0.01)
# fit model
result<-model %>% fit(x_train, y_train, epochs=55, verbose=1,
                      validation_split = 0.20, callbacks = list(early_stop))
```



SubFig. a SubFig. b SubFig. c

Fig 10. Different MLP layer groups for 1). a) for DL.1, layers: 5+10, it fits within smaller epochs; b) for DL.2, layers: 3+6, it is underfitting as error does not reach delta, and needs to add more epochs; c) for DL.3, layers: 8+16, it shows a good-fit.

Table 4. Comparison of DL

No.	MLP layer	Evaluation from loss & error	accuracy	correlation	RMSE
DL.3	8+16	Good-fit	0.6111	0.9981	7.8209
DL.4 (S1)	4+8	Good-fit	0.9553	0.9940	11.6950
DL.4-2		Good-fit	0.9840	0.9940	11.5005

Here the training loss and validation loss both decrease and stabilize at a specific point of DL.3 and DL.4, showing a good-fit.

The different MLP groups show that the corresponding relationship between the dimensions of test sets and units of a layer is important, indicating that the DL.4 model shows a better accuracy within smaller compute cost for 4 inputs, while the DL.3 model is underfitting and needs more epochs to train the model. And the DL.4 performance is better showing the stable validation loss and error.

It is noted that the more units with good-fit performance do not lead to more accuracy, showing that DL.3 model with the 16 units in the 2nd layer with lower accuracy but lower RMSE, meaning its worse prediction in the overall trend but it can reduce significant derivation in some points.

Considering that we want to evaluate the overall trend, the DL.4 model is more suitable. And its prediction is following (Fig 11), showing the overall trend of the prediction (red line) and the actual values (blue line) closely align, especially for lower discharge

values, where these two results match closely. However, there is a noticeable deviation for peak values, showing a overestimating prediction at the peak.

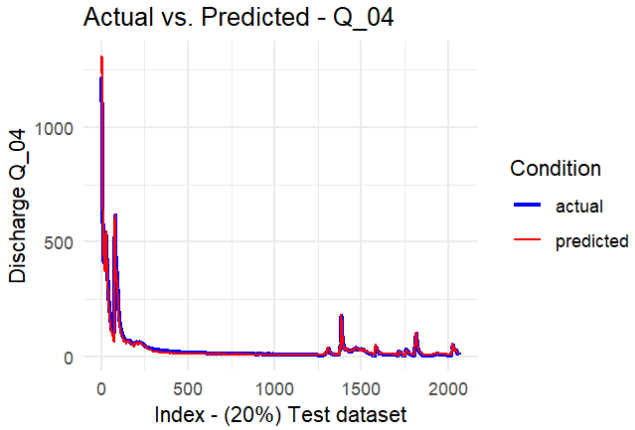


Fig 11. The better predictive performance in 1) DL.4 (S1)

5. Train a LSTM model by .py

LSTM is commonly employed in modeling and predicting time-series data. And it needs a wrapped function to reshape the dimension of original datasets, to ensure the input data be 3-dimensional, where dimension 1 again is the batch dimension, dimension 2 again corresponds to the number of timesteps (10-, 20-, 60- min), and dimension 3 is the size of the wrapped layer (4 features). That wrapper's task is to apply the same calculation (i.e., the same weight matrix) to every state input it receives.
Then we need to take trials and tests to train a model with suitable units in the lstm layer.

```
# Build up a LSTM model
inputs = layers.Input(x_train.shape[1:], name='input')
lstm   = layers.LSTM(units=5, name='lstm')(inputs)
output = layers.Dense(units=1, name='dense', activation='linear')(lstm)

model  = models.Model(inputs, output)
model.summary()
es       = callbacks.EarlyStopping(monitor='val_loss', mode='min', verbose=1,
patience=40, min_delta=0.01, restore_best_weights=True)
model.compile(loss='mse', metrics=['mean_absolute_error'], optimizer='adam')
history = model.fit(x_train_scaled, y_train_scaled, epochs=80,
validation_split=0.2, callbacks=[es, PlotLossesKeras()])
```

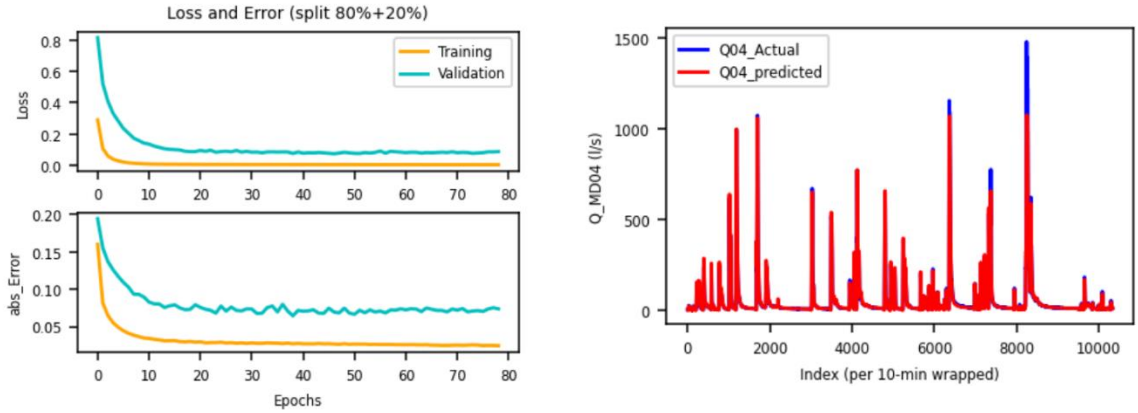


Fig 12. loss and error plot and prediction of LSTM.1

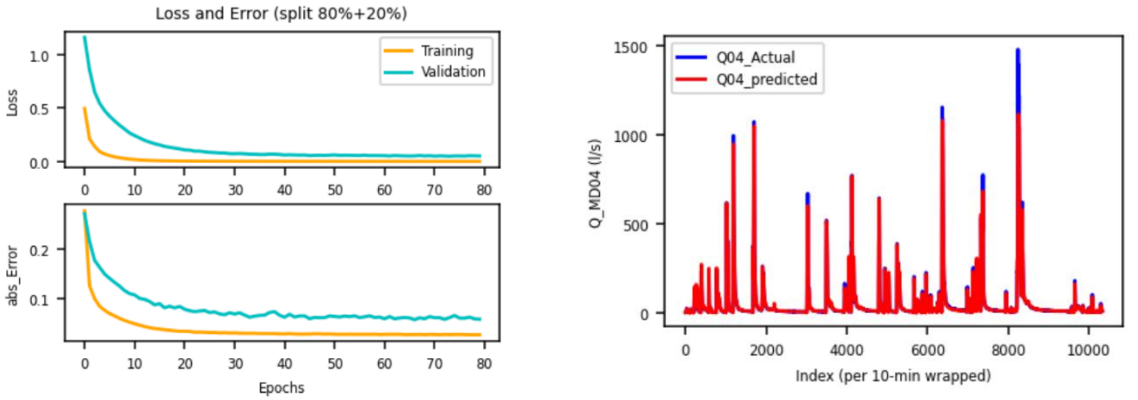


Fig 13. loss and error plot and prediction of LSTM.2

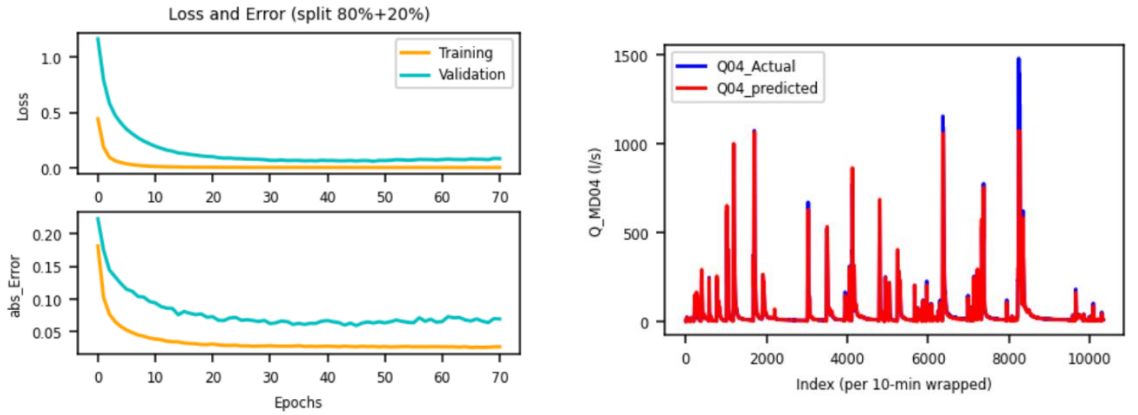


Fig 14. loss and error plot and prediction of LSTM.3

Table 5. Trials and tests indicators for LSTM (split-friction 80%+20%)

No.	Wrapped	Input/ Lstm	epochs	accuracy	RMSE
LSTM.1	10	(10, 4)/ 8	80	0.9960	9.9919
LSTM.2	10	(10, 4)/ 4	80	0.9562	6.9175
LSTM.3	**10**	**(10, 4)/ 5**	**70**	**0.9934**	**5.3363**

From the Table 5, it shows the units=5 is suitable for this model, as the smaller epochs needed and higher accuracy with lower RMSE.

6. Produce 10-, 20- and 60-minute forecast

We use the following two model to test the performance of different minutes forecast by MLP model and LSTM model. The corresponding data is extracted from the last several minutes.

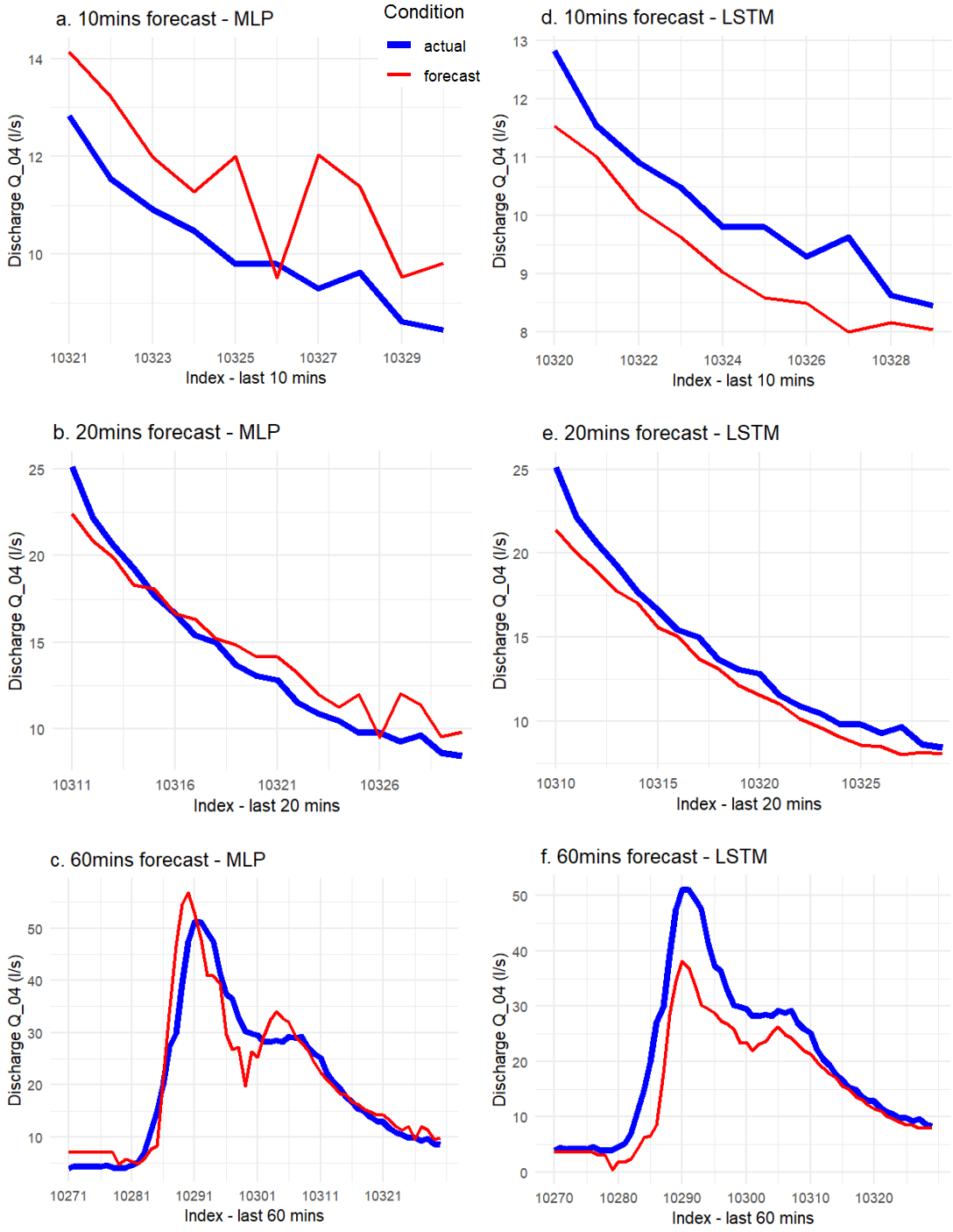


Fig 15. Comparison of Different Minutes Forecast by MLP model and LSTM model

From the Fig 15, the forecast by MLP obviously overestimates, showing a significant upwards deviation from the actual value in the overall series. And that by LSTM significantly underestimates, especially for the larger discharge scenario, the

underestimation is more obvious.

Besides, the comparison of statistical indicators is in Table 6. For the last 10- and 20-min forecast, the LSTM model shows a better prediction, as higher accuracy and lower RMSE. But for the last 60-min forecast, its performance is significantly worse than MLP model.

Table 6. Indicators of Different Minutes Forecast by MLP model (DL.4)

No.	Forecast	MLP - DL.4			LSTM - LSTM.3		
		accuracy	correlation	RMSE	accuracy	correlation	RMSE
1	10min	0.8639	0.8455	1.5641	0.9135	0.9566	0.9559
2	20min	0.931	0.9842	1.3909	0.9204	0.9921	1.3692
3	60min	0.887	0.9409	4.7447	0.7751	0.9572	6.6944

In generally, the LSTM represents better performance for time series forecast compared to MLP. From the model work frame, MLPs do not have inherent memory capabilities. They process each input independently without considering any temporal dependencies between them. Therefore, MLPs can respond to the peak actual values as a physiological effect of changing inputs.

As a result, if we want to obtain a more aggressive estimation showing a higher prediction, MLP model can exhibit optimistic projections instead of the conservatism. However, if want to consider the rainfall-runoff time series effect, LSTM shows a more meaningful prediction.

7. Discuss forecast accuracy as function of lead time

In LSTM model, we want to discuss the impact of different wrappers with 10-, 20-, 60-mins. Firstly, we consider its impact on the overall trend in Table 7.

Table 7. Indicators in the Overall trend of LSTM (in the whole dataset)

Wrapped	No.	Trial 1		Trial 2		Trial 3		average	
		accuracy	RMSE	accuracy	RMSE	accuracy	RMSE	accuracy	RMSE
10	LSTM.4	0.9641	7.0274	0.9658	9.4024	0.9931	6.2844	0.9743	7.5714
20	LSTM.5	0.9770	8.7198	0.9696	8.8683	0.9823	6.8119	0.9763	8.1333
60	LSTM.6	0.9662	7.1721	0.9605	5.6516	0.9503	5.4183	0.9590	6.0807

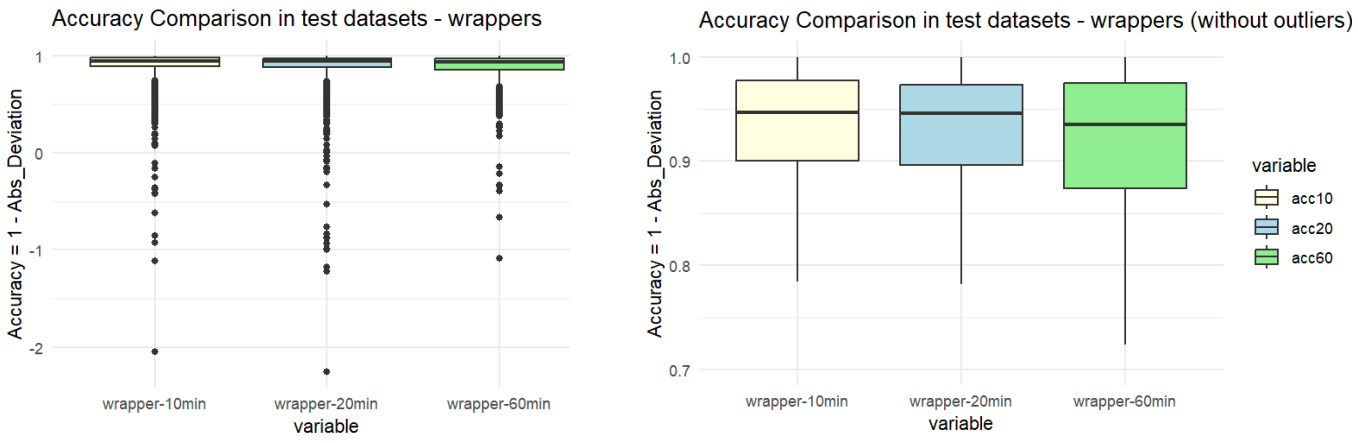


Fig 16. Boxplot for Accuracy Comparison in test datasets

After training, the model with 10-min and 20-min can generate a more accurate prediction but with some significant underestimation in some points (as higher RMSE in the whole datasets). If we only evaluate the model performance in test sets in Fig 16, it shows around 10% outliers in test sets, considering the significant underestimation in peak discharge, and the model with 60min leading time shows a better adjustability in peak value.

The wrapper's task is to apply the same calculation (i.e., the same weight matrix) to every state input it receives, and it works as adding or removing features iterative. In the catchment evaluation, the leading time shows the rainfall-runoff concentration condition in this catchment.

To get a comprehensive prediction, we can use the shorter leading time to evaluate the lower discharge scenario, and use a longer leading time to obtain the peak value with a higher accuracy.

8. Choose a neural network architecture for the catchment evaluation

In the hydrological theory, we know that the rainfall-runoff events have significant time series properties. And generally, the LSTM represents better performance for time series forecast compared to MLP.

From the above analysis, we can find that LSTM can forecast with a stable and higher accuracy. Besides, if generate a shorter time prediction, LSTM can train a model in a high-efficiency and provide a better performance both in the overall trend and capturing peak values.

9. Appendix: Script in r

9.1. The neural network by 'neuralnet' function

```
krevents <- read.xlsx("kentridgerrdata-55events.xlsx", sheet="Selected Events")  # read the 2nd sheet
colnames(krevents)[2:7] <- c("Rainfall", "Q_MD01", "Q_MD02", "Q_MD04", "Q_CNTRLIB", "Q_OPPRLINK")
# unit: Q [L/s]; Rainfall[mm]
rainfall <- krevents[, 2]
Q01 <- krevents[, 3]
Q02 <- krevents[, 4]
Q04 <- krevents[, 5]
# ignore: Qcenter <- krevents[, 6]
Qopp <- krevents[, 7]
df=data.frame(rainfall, Q01, Q02, Qopp, Q04)  # chosen 4 variables vs Q4

library(neuralnet)
# splitting #####
# sample size = 80% test + 20% train [default]
xnum=length(krevents[,1])
train_size <- round(0.8 * xnum)
# with time, the data is not random
train <- df[0:train_size,1:5]
test <- df[(train_size + 1):xnum,1:5]
pre_de_test <- df[(train_size + 1):xnum,1:5]
dim(train)
dim(test)

#normalisation
min_max <- function(x){
  y<-(x - min(x, na.rm=TRUE))/(max(x,na.rm=TRUE) - min(x, na.rm=TRUE))
  return (y)
}
train=as.data.frame(lapply(train, min_max))
test=as.data.frame(lapply(test, min_max))

# model1 - 2 hidden layers with 5 and 3 neurons (trials)
# logistic sigmoid function; (rainfall, Q01, Q02, Qopp, Q04)
nn<- neuralnet(Q04 ~ rainfall+Q01+Q02+Qopp, data=train,
                    hidden=c(4,2), linear.output=FALSE, threshold=0.01)
plot(nn)

# prediction ####
# testing
```

```
temp_test<- subset(test, select=c("rainfall", "Q01", "Q02", "Qopp"))
nn.results <- compute(nn, temp_test)
actual=test$Q04
prediction=nn.results$net.result

# denormalise
denormalize <- function(y, min_value, max_value){
   x <- y * (max_value - min_value) + min_value
   return(x)
}

prediction_de <- denormalize(prediction, min_value =min(pre_de_test$Q04), max_value = max(pre_de_test$Q04))
actual_de <- denormalize(actual, min_value =min(pre_de_test$Q04), max_value = max(pre_de_test$Q04))

#accuracy
deviation=(actual_de - prediction_de)/actual_de
accuracy_rate=1-abs(mean(deviation))
accuracy_rate
rmse=(sum((actual-prediction)^2)/nrow(test))^0.5
rmse
cor(actual,prediction)
```

9.2. MLP by keras()

```
# differencing is a way to make a non stationary time series #####
# unit: Q [L/s]; Rainfall[mm]
d_Q04 = diff(krevents$Q_MD04, differences = 1)

undiff_Q04 <- numeric(length(d_Q04) + 1)
undiff_Q04[1] <- krevents$Q_MD04[1]
for (i in 2:length(undiff_Q04)) {
   undiff_Q04 [i] <- undiff_Q04 [i - 1] + d_Q04[i - 1]
}
# undiff_Q04 is the same as diff() results d_Q04
# a tiny discrepancy occurs between the original series and its inverse first differences due to rounding.
# thus, the diff() is not necessary for this case, and discharge is analyzed directly by an explicit way.

## data for ANN (specifically for RNNs)#####
rainfall <- krevents[, 2]
Q01 <- krevents[ 3]
Q02 <- krevents[, 4]
```

```
Q04 <- krevents[, 5]
Qcenter <- krevents[, 6]
Qopp <- krevents[, 7]
df=data.frame(rainfall, Q01, Q02, Qopp, Q04)    # chosen 4 variables vs Q4
# data set-> df

#splitting #####
# sample size = 80% test + 20% train
N = nrow(krevents) # 10330
n = round(N *0.8, digits = 0) # 8264
train = df[0:n,]
dim(train)
test = df[seq(n+1,N,1),] # 2066
dim(test)

# normalize; ranging into [0,1]
normalize <- function(train, test, feature_range = c(0, 1)) {
   num_features = ncol(train)
   scaled_train = train
   scaled_test = test

   for (i in 1:num_features) {
      x = train[, i]
      fr_min = feature_range[1]
      fr_max = feature_range[2]
      std_train = ((x - min(x)) / (max(x) - min(x)))
      std_test = ((test[, i] - min(x)) / (max(x) - min(x)))

      scaled_train[, i] = std_train * (fr_max - fr_min) + fr_min
      scaled_test[, i] = std_test * (fr_max - fr_min) + fr_min
   }
   return(list(scaled_train = scaled_train, scaled_test = scaled_test, scaler = list(min =
apply(train, 2, min), max = apply(train, 2, max))))
}

Scaled = normalize(train, test, c(-1, 1))
# df (rainfall, Q01, Q02, Qopp, Q04)
y_train = Scaled$scaled_train[, 5]
x_train = Scaled$scaled_train[, seq(1:4)]
y_test = Scaled$scaled_test[, 5]
x_test = Scaled$scaled_test[, seq(1:4)]

# reshaping #####
x_train=as.matrix(x_train)
```

```
y_train=as.matrix(y_train)

x_test=as.matrix(x_test)
y_test=as.matrix(y_test)
class(x_train)
dim(x_train)

# keras model setup - MLP #####
# 1) dense (fully connected) layers
model = keras_model_sequential() %>%
  layer_dense(units = 5, activation = 'relu', input_shape = dim(x_train)[2]) %>%  # Input layer
#https://keras.io/activations/
  layer_dense(units = 10, activation = 'relu') %>%  # Hidden layer
  layer_dense(units = 1)  # Output layer

#model compile
model %>% compile(loss = 'mse',
                  optimizer = 'adam', #optimizer_rmsprop(lr = 0.001)
                  metrics = list("mean_absolute_error")
)

summary(model)
early_stop <- callback_early_stopping(monitor = "val_loss", patience = 40, min_delta=0.01)

# fit model
result<-model %>% fit(x_train, y_train, epochs=55, verbose=1,
                      validation_split = 0.15, callbacks = list(early_stop))
# evaluate #####
scores = model %>% evaluate(x_test, y_test, verbose = 0)
print(scores)
predictions <- model %>% predict(x_test)

denormalize <- function(scaled_data, min_val, max_val, feature_range = c(0, 1)) {
  fr_min = feature_range[1]
  fr_max = feature_range[2]
  # Denormalize the data
  original_data = (scaled_data - fr_min) / (fr_max - fr_min) * (max_val - min_val) + min_val
  return(original_data)
}

denormalized_predictions <- denormalize(predictions, Scaled$scaler$min[5],
Scaled$scaler$max[5], c(-1, 1))
actual_data <- test[, 5]
```

```
# accuracy #####
deviation=(actual_data - denormalized_predictions)/actual_data
# deviation
accuracy_rate=1-abs(mean(deviation)) # total
plot(1:length(deviation), deviation, type="l", col="red", xlab="index",ylab="deviation")

rmse<- sqrt(sum((actual_data-denormalized_predictions)^2)/nrow(test))
cor(actual_data, denormalized_predictions)
```

9.3. LSTM in .py

```
# define a function to reshape the dataset to meet LSTM model
def get_wrapped_data(dataset, wrap_length=10):
    data_x, data_y = [], []

    for i in range(len(dataset) - wrap_length):
        # dataset (rainfall, Q01, Q02, Q04, Qopp)
        data_x.append(dataset.iloc[i:i+wrap_length, [0, 1, 2, 4]].to_numpy(dtype='float64'))
        data_y.append(dataset.iloc[i+wrap_length, [3]].astype('float64'))

    return np.array(data_x), np.array(data_y)

# Wrap the data
data_x, data_y = get_wrapped_data(dataset, wrap_length=10)
data_x.shape
split_index = int(len(data_x) * 0.8) # splitting = 80% test + 20% train

# Split the data into training and testing sets
x_train = data_x[:split_index]
y_train = data_y[:split_index]
x_test = data_x[split_index:]
y_test = data_y[split_index:]

# Initialize scale parameters
scale_params = {
    "train_x_mean": 0,
    "train_x_std": 1,
    "train_y_mean": 0,
    "train_y_std": 1
}

# Calculate mean and standard deviation for scaling
scale_params["train_x_mean"] = np.mean(x_train, axis=(0, 1))
scale_params["train_x_std"] = np.std(x_train, axis=(0, 1))
scale_params["train_y_mean"] = np.mean(y_train, axis=0)
```

```
scale_params["train_y_std"] = np.std(y_train, axis=0)

# Normalize training and testing data
x_train_scaled = (x_train - scale_params["train_x_mean"][None, None, :]) / scale_params["train_x_std"][None, None, :]
y_train_scaled = (y_train - scale_params["train_y_mean"][None, :]) / scale_params["train_y_std"][None, :]
x_test_scaled = (x_test - scale_params["train_x_mean"][None, None, :]) / scale_params["train_x_std"][None, None, :]
y_test_scaled = (y_test - scale_params["train_y_mean"][None, :]) / scale_params["train_y_std"][None, :]

# Build up a LSTM model
inputs = layers.Input(x_train.shape[1:], name='input')
lstm   = layers.LSTM(units=5, name='lstm')(inputs)
output = layers.Dense(units=1, name='dense', activation='linear')(lstm)

model  = models.Model(inputs, output)
model.summary()
es       = callbacks.EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=40,
                                   min_delta=0.01, restore_best_weights=True)
model.compile(loss='mse', metrics=['mean_absolute_error'], optimizer='adam')
history = model.fit(x_train_scaled, y_train_scaled, epochs=80, validation_split=0.2,
                    callbacks=[es, PlotLossesKeras()])

# Prediction
pred_train = model.predict(x_train_scaled)
pred_test  = model.predict(x_test_scaled)
Training_result = pred_train * scale_params["train_y_std"] + scale_params["train_y_mean"]
testing_result = pred_test* scale_params["train_y_std"] + scale_params["train_y_mean"]
dataset['flow_pred'] = None
dataset.iloc[10:len(Training_result)+10, dataset.columns.get_loc('flow_pred')] = Training_result
dataset.iloc[len(Training_result)+10:, dataset.columns.get_loc('flow_pred')] = testing_result

# assessment and accuracy
actual_data = dataset['Q04'].values[len(Training_result)+10:]
denormalized_predictions = dataset['flow_pred'].values[len(Training_result)+10:]
# accuracy
deviation = (actual_data - denormalized_predictions) / actual_data
accuracy_rate = 1 - np.abs(np.mean(deviation))

# RMSE
RMSE = np.sqrt(np.mean((actual_data - denormalized_predictions) ** 2))
```