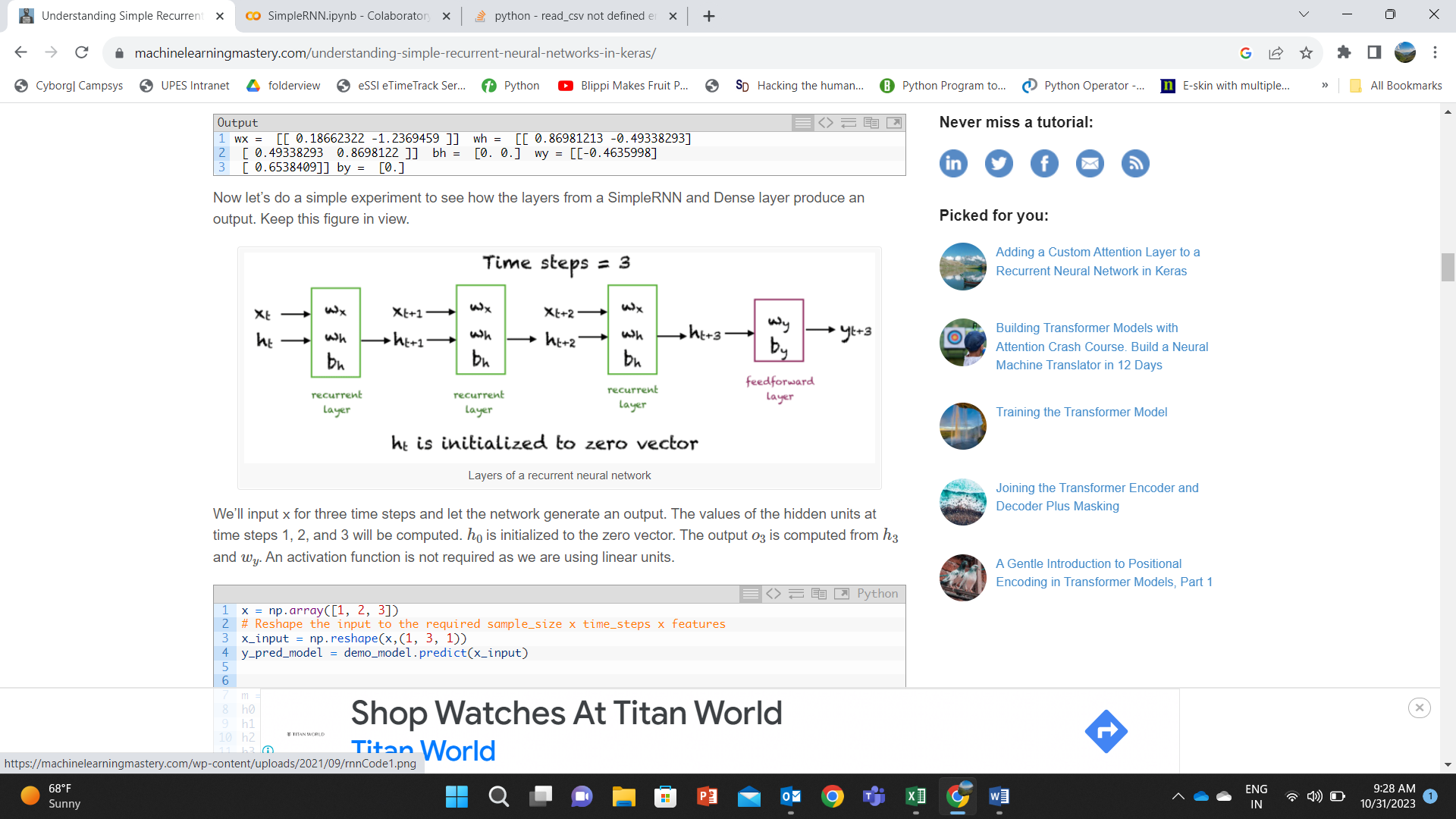
**Simple RNN Implementation**



## Consolidated Code

from pandas import read\_csv

import numpy as np

from keras.models import Sequential

from keras.layers import Dense, SimpleRNN

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean\_squared\_error

import math

import matplotlib.pyplot as plt

# Parameter split\_percent defines the ratio of training examples

def get\_train\_test(url, split\_percent=0.8):

    df = read\_csv(url, usecols=[1], engine='python')

    data = np.array(df.values.astype('float32'))

    scaler = MinMaxScaler(feature\_range=(0, 1))

    data = scaler.fit\_transform(data).flatten()

    n = len(data)

    # Point for splitting data into train and test

    split = int(n\*split\_percent)

    train\_data = data[range(split)]

    test\_data = data[split:]

    return train\_data, test\_data, data

# Prepare the input X and target Y

def get\_XY(dat, time\_steps):

    Y\_ind = np.arange(time\_steps, len(dat), time\_steps)

    Y = dat[Y\_ind]

    rows\_x = len(Y)

    X = dat[range(time\_steps\*rows\_x)]

    X = np.reshape(X, (rows\_x, time\_steps, 1))

    return X, Y

def create\_RNN(hidden\_units, dense\_units, input\_shape, activation):

    model = Sequential()

    model.add(SimpleRNN(hidden\_units, input\_shape=input\_shape, activation=activation[0]))

    model.add(Dense(units=dense\_units, activation=activation[1]))

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

    return model

def print\_error(trainY, testY, train\_predict, test\_predict):

    # Error of predictions

    train\_rmse = math.sqrt(mean\_squared\_error(trainY, train\_predict))

    test\_rmse = math.sqrt(mean\_squared\_error(testY, test\_predict))

    # Print RMSE

    print('Train RMSE: %.3f RMSE' % (train\_rmse))

    print('Test RMSE: %.3f RMSE' % (test\_rmse))

# Plot the result

def plot\_result(trainY, testY, train\_predict, test\_predict):

    actual = np.append(trainY, testY)

    predictions = np.append(train\_predict, test\_predict)

    rows = len(actual)

    plt.figure(figsize=(15, 6), dpi=80)

    plt.plot(range(rows), actual)

    plt.plot(range(rows), predictions)

    plt.axvline(x=len(trainY), color='r')

    plt.legend(['Actual', 'Predictions'])

    plt.xlabel('Observation number after given time steps')

    plt.ylabel('Sunspots scaled')

    plt.title('Actual and Predicted Values. The Red Line Separates The Training And Test Examples')

a = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-sunspots.csv'

time\_steps = 12

train\_data, test\_data, data = get\_train\_test(a)

trainX, trainY = get\_XY(train\_data, time\_steps)

testX, testY = get\_XY(test\_data, time\_steps)

# Create model and train

model = create\_RNN(hidden\_units=3, dense\_units=1, input\_shape=(time\_steps,1),

                   activation=['tanh', 'tanh'])

model.fit(trainX, trainY, epochs=20, batch\_size=1, verbose=2)

# make predictions

train\_predict = model.predict(trainX)

test\_predict = model.predict(testX)

# Print error

print\_error(trainY, testY, train\_predict, test\_predict)

#Plot result

plot\_result(trainY, testY, train\_predict, test\_predict)

**EXPLANATION**

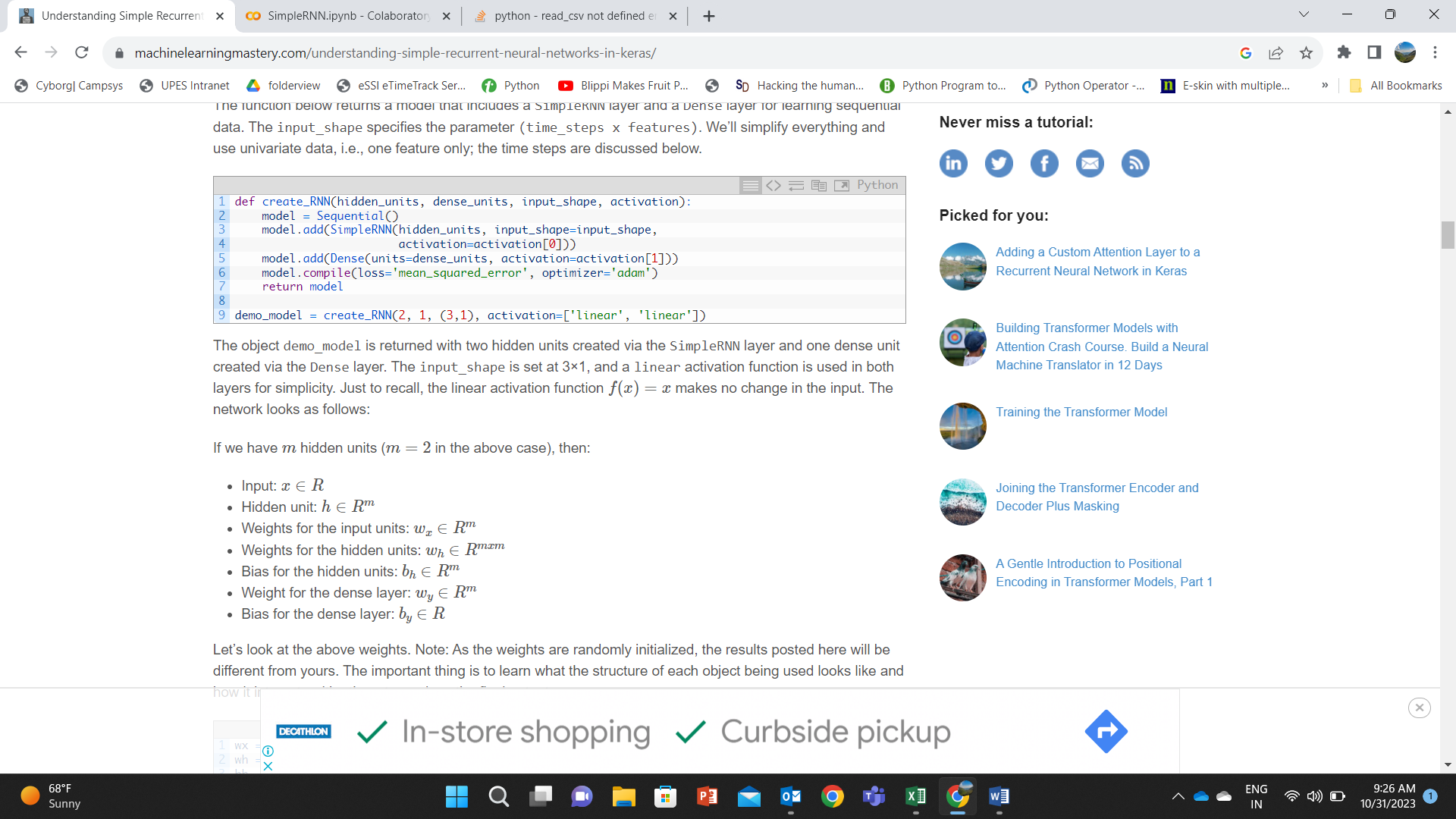
**Keras SimpleRNN**

The function below returns a model that includes a SimpleRNN layer and a Dense layer for learning sequential data. The input\_shape specifies the parameter (time\_steps x features). We’ll simplify everything and use univariate data, i.e., one feature only; the time steps are discussed below.

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def create\_RNN(hidden\_units, dense\_units, input\_shape, activation):      model = Sequential()      model.add(SimpleRNN(hidden\_units, input\_shape=input\_shape,                          activation=activation[0]))      model.add(Dense(units=dense\_units, activation=activation[1]))      model.compile(loss='mean\_squared\_error', optimizer='adam')      return model    demo\_model = create\_RNN(2, 1, (3,1), activation=['linear', 'linear']) |

The object demo\_model is returned with two hidden units created via the SimpleRNN layer and one dense unit created via the Dense layer. The input\_shape is set at 3×1, and a linear activation function is used in both layers for simplicity. Just to recall, the linear activation function f(x)=x makes no change in the input. The network looks as follows:



Let’s look at the above weights. Note: As the weights are randomly initialized, the results posted here will be different for everyone. The important thing is to learn what the structure of each object being used looks like and how it interacts with others to produce the final output.

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | wx = demo\_model.get\_weights()[0]  wh = demo\_model.get\_weights()[1]  bh = demo\_model.get\_weights()[2]  wy = demo\_model.get\_weights()[3]  by = demo\_model.get\_weights()[4]    print('wx = ', wx, ' wh = ', wh, ' bh = ', bh, ' wy =', wy, 'by = ', by) |

Output

|  |  |
| --- | --- |
| 1  2  3 | wx =  [[ 0.18662322 -1.2369459 ]]  wh =  [[ 0.86981213 -0.49338293]  [ 0.49338293  0.8698122 ]]  bh =  [0. 0.]  wy = [[-0.4635998]  [ 0.6538409]] by =  [0.] |

We’ll input x for three time steps and let the network generate an output. The values of the hidden units at time steps 1, 2, and 3 will be computed. ℎ0 is initialized to the zero vector. The output o3 is computed from ℎ3 and wy. An activation function is not required as we are using linear units.

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | x = np.array([1, 2, 3])  # Reshape the input to the required sample\_size x time\_steps x features  x\_input = np.reshape(x,(1, 3, 1))  y\_pred\_model = demo\_model.predict(x\_input)      m = 2  h0 = np.zeros(m)  h1 = np.dot(x[0], wx) + h0 + bh  h2 = np.dot(x[1], wx) + np.dot(h1,wh) + bh  h3 = np.dot(x[2], wx) + np.dot(h2,wh) + bh  o3 = np.dot(h3, wy) + by    print('h1 = ', h1,'h2 = ', h2,'h3 = ', h3)    print("Prediction from network ", y\_pred\_model)  print("Prediction from our computation ", o3) |

Output

|  |  |
| --- | --- |
| 1  2  3 | h1 =  [[ 0.18662322 -1.23694587]] h2 =  [[-0.07471441 -3.64187904]] h3 =  [[-1.30195881 -6.84172557]]  Prediction from network  [[-3.8698118]]  Prediction from our computation  [[-3.86981216]] |

**Running the RNN on Sunspots Dataset**

Now that we understand how the SimpleRNN and Dense layers are put together. Let’s run a complete RNN on a simple time series dataset. We’ll need to follow these steps:

1. Read the dataset from a given URL
2. Split the data into training and test sets
3. Prepare the input to the required Keras format
4. Create an RNN model and train it
5. Make the predictions on training and test sets and print the root mean square error on both sets
6. View the result

### **Step 1, 2: Reading Data and Splitting Into Train and Test**

The following function reads the train and test data from a given URL and splits it into a given percentage of train and test data. It returns single-dimensional arrays for train and test data after scaling the data between 0 and 1 using MinMaxScaler from scikit-learn.

# Parameter split\_percent defines the ratio of training examples

def get\_train\_test(url, split\_percent=0.8):

    df = read\_csv(url, usecols=[1], engine='python')

    data = np.array(df.values.astype('float32'))

    scaler = MinMaxScaler(feature\_range=(0, 1))

    data = scaler.fit\_transform(data).flatten()

    n = len(data)

    # Point for splitting data into train and test

    split = int(n\*split\_percent)

    train\_data = data[range(split)]

    test\_data = data[split:]

    return train\_data, test\_data, data

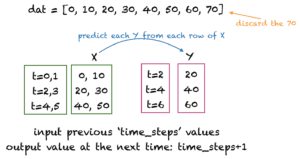
sunspots\_url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-sunspots.csv'

train\_data, test\_data, data = get\_train\_test(sunspots\_url)

### **Step 3: Reshaping Data for Keras**

The next step is to prepare the data for Keras model training. The input array should be shaped as: total\_samples x time\_steps x features.

There are many ways of preparing time series data for training. We’ll create input rows with non-overlapping time steps. An example for time steps = 2 is shown in the figure below. Here, time steps denotes the number of previous time steps to use for predicting the next value of the time series data.

[](https://machinelearningmastery.com/wp-content/uploads/2021/09/rnnCode2.png)

The following function get\_XY() takes a one-dimensional array as input and converts it to the required input X and target Y arrays. We’ll use 12 time\_steps for the sunspots dataset as the sunspots generally have a cycle of 12 months. You can experiment with other values of time\_steps.

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | # Prepare the input X and target Y  def get\_XY(dat, time\_steps):      # Indices of target array      Y\_ind = np.arange(time\_steps, len(dat), time\_steps)      Y = dat[Y\_ind]      # Prepare X      rows\_x = len(Y)      X = dat[range(time\_steps\*rows\_x)]      X = np.reshape(X, (rows\_x, time\_steps, 1))      return X, Y    time\_steps = 12  trainX, trainY = get\_XY(train\_data, time\_steps)  testX, testY = get\_XY(test\_data, time\_steps) |

### **Step 4: Create RNN Model and Train**

For this step, you can reuse your create\_RNN() function that was defined above.

Python

|  |  |
| --- | --- |
| 1  2  3 | model = create\_RNN(hidden\_units=3, dense\_units=1, input\_shape=(time\_steps,1),                     activation=['tanh', 'tanh'])  model.fit(trainX, trainY, epochs=20, batch\_size=1, verbose=2) |

### **Step 5: Compute and Print the Root Mean Square Error**

The function print\_error() computes the mean square error between the actual and predicted values.

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | def print\_error(trainY, testY, train\_predict, test\_predict):      # Error of predictions      train\_rmse = math.sqrt(mean\_squared\_error(trainY, train\_predict))      test\_rmse = math.sqrt(mean\_squared\_error(testY, test\_predict))      # Print RMSE      print('Train RMSE: %.3f RMSE' % (train\_rmse))      print('Test RMSE: %.3f RMSE' % (test\_rmse))    # make predictions  train\_predict = model.predict(trainX)  test\_predict = model.predict(testX)  # Mean square error  print\_error(trainY, testY, train\_predict, test\_predict) |

Output

|  |  |
| --- | --- |
| 1  2 | Train RMSE: 0.058 RMSE  Test RMSE: 0.077 RMSE |

### **Step 6: View the Result**

The following function plots the actual target values and the predicted values. The red line separates the training and test data points.

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | # Plot the result  def plot\_result(trainY, testY, train\_predict, test\_predict):      actual = np.append(trainY, testY)      predictions = np.append(train\_predict, test\_predict)      rows = len(actual)      plt.figure(figsize=(15, 6), dpi=80)      plt.plot(range(rows), actual)      plt.plot(range(rows), predictions)      plt.axvline(x=len(trainY), color='r')      plt.legend(['Actual', 'Predictions'])      plt.xlabel('Observation number after given time steps')      plt.ylabel('Sunspots scaled')      plt.title('Actual and Predicted Values. The Red Line Separates The Training And Test Examples')  plot\_result(trainY, testY, train\_predict, test\_predict) |

The following plot is generated:

