pm2-predicting-sin-wave

February 15, 2022

1 Predicting Sin Wave by using Simple RNN

In this notebook, you will understand how RNN can predict sin wave. We are going to use Keras, a high-level API for deep learning, which is a wrapper of TensorFlow, CNTK, or Theano. Keras was developed with a focus on enabling fast experimentation, which makes this framework easy to use and learn deep learning.

In order to have a clean notebook, some functions are implemented in the file *utils.py* (e.g., create_window). We are not going to discuss the implementation aspects of these functions here, but you can to explore and read the content of the functions later on.

Summary: - Creating sin data - Creating the Windowed Data Set - Building the Model - Data Pre-processing - Spliting the Data into Training and Test Dataset - Basic RNN Model - Using the history - Prediction and Performance Analysis - Model Understanding: What are the model weights and hidden units?

All the libraries used in this notebook are Open Source.

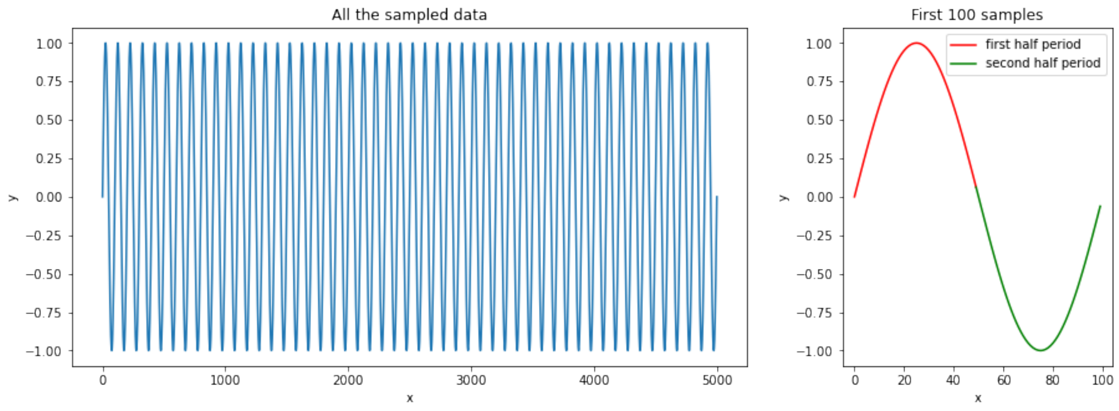
[1]:
```
# Standard libraries - no deep learning yet
import math # standard library, provides functions such as for trigonometry and␣
↪logarithms as well as constants.
import numpy as np # written in C, is faster and robust library for numerical␣
↪and matrix operations
import pandas as pd # data manipulation library, it is widely used for data␣
↪analysis and relies on numpy library.
import matplotlib.pyplot as plt # for plotting

from utils import * # this will load all the utils functions implements for␣
↪this tutorial

# the following to lines will tell to the python kernel to alway update the␣
↪kernel for every utils.py
# modification, without the need of restarting the kernel.
# Of course, for every motification in util.py, we need to reload this cell
%load_ext autoreload
%autoreload 2

# If you see some RuntimeWarning, just ignore them.
```

```
Using TensorFlow backend.
```

1.1 Creating sin data

The function **create_sin_data** (see **utils.py**) uses the function np.sin from numpy and store the data in a DataFrame format.

[2]:
```
# Creating the sin data, this returns a DataFrame
series = create_sin_data(samples=5000, period=50)

# Visualising the series
f, (ax0, ax1) = plt.subplots(1,2, figsize = (15, 5), gridspec_kw =␣
 ↪{'width_ratios':[5, 2]})

# Ploting the sampled data
    # series.values returns a numpy representation of the DataFrame
ax0.plot(series.values)
ax0.set_xlabel("x")
ax0.set_ylabel("y")
ax0.set_title("All the sampled data")

# Ploting the first PERIOD of the sampled data
  # series.values returns a numpy representation of the DataFrame, by doing␣
 ↪series.values[:PERIOD], we are
  # taking the first 50 samples in the numpy.array representation
ax1.plot(range(50),series.values[:50], 'r', label='first half period')
ax1.set_xlabel("x")
ax1.set_ylabel("y")


  # series.values returns a numpy representation of the DataFrame, by doing␣
 ↪series.values[:PERIOD], we are
  # taking the second 50 samples in the numpy.array representation
ax1.plot(range(49, 100), series.values[49:100], 'g', label='second half period')
ax1.set_xlabel("x")
ax1.set_ylabel("y")
ax1.set_title("First {} samples".format(100))
ax1.legend(loc="upper right")

plt.show()
```



1.2 Creating the Windowed Data Set

To window the dataset, we wrote the function **create_window** (see *utils.py*), this fixes *window_size* and uses pandas shift function. The function shifts an entire column by a given number. In our case, we shifted the column up by 1 and then concatenate that to the original data. For example, if window_size is 5, we will use the first 4 elements and predict the next one. Therefore, we each row is on sample traning.

```
[3]: # Performing the sliding window (window_size=5) with drop_nan = False.
     series_backup = series.copy()
     t = create_window_sin(series_backup, window_size = 5)
     # show the last 5 elements in the DataFrame (dataframe.tail can be used for␣
      ↪this propose)
     t.tail()
```

```
[3]:                  0             0             0             0             0    0
     4995 -2.487386e-01 -1.874184e-01 -1.253582e-01 -6.280306e-02  9.821934e-16 NaN
     4996 -1.874184e-01 -1.253582e-01 -6.280306e-02  9.821934e-16           NaN NaN
     4997 -1.253582e-01 -6.280306e-02  9.821934e-16           NaN           NaN NaN
     4998 -6.280306e-02  9.821934e-16           NaN           NaN           NaN NaN
     4999  9.821934e-16           NaN           NaN           NaN           NaN NaN
```

As you can see in the table above, a lot of NaN where included in the dataset, in order to remove these value we will set *drop_nan* = True. This will allow us to use pandas function dropna

```
[4]: # Performing the sliding window (window_size=5) with drop_nan = True.
     series_backup = series.copy()
     t = create_window_sin(series_backup, window_size = 5, drop_nan=True)
     # print the last 5 elements in the DataFrame (dataframe.tail can be used for␣
      ↪this propose)
     t.tail()
```

```
[4]:             0         0         0         0         0             0
     4990 -0.535922 -0.481842 -0.425859 -0.368195 -0.309077 -2.487386e-01
     4991 -0.481842 -0.425859 -0.368195 -0.309077 -0.248739 -1.874184e-01
     4992 -0.425859 -0.368195 -0.309077 -0.248739 -0.187418 -1.253582e-01
     4993 -0.368195 -0.309077 -0.248739 -0.187418 -0.125358 -6.280306e-02
     4994 -0.309077 -0.248739 -0.187418 -0.125358 -0.062803  9.821934e-16
```

For now, we can ignore the **column_names** (they are all zero). They will not play any role in this tutorial, however, if you would like to change them,| click in this link for more information.

```
[5]: # Print the first 5 elements by using dataframe.head()
     t.head()
```

```
[5]:             0         0         0         0         0         0
     0 -9.821934e-16  0.062803  0.125358  0.187418  0.248739  0.309077
     1  6.280306e-02  0.125358  0.187418  0.248739  0.309077  0.368195
     2  1.253582e-01  0.187418  0.248739  0.309077  0.368195  0.425859
     3  1.874184e-01  0.248739  0.309077  0.368195  0.425859  0.481842
     4  2.487386e-01  0.309077  0.368195  0.425859  0.481842  0.535922
```

2 Building the Model

So far, we just created the dataset and showed how we can use pandas for data manipulation. Now, we are going to preprocess the data and build a **simple RNN** model.

```
[6]: # Loading the libraries required for this step.
     # keras components, we will describe each one along this tutorial, so don't be␣
      ↪worry
     from keras.models import Sequential
     from keras.layers.recurrent import SimpleRNN
     from keras.layers.core import Dense, Activation

     # Scikit learn libraries
     from sklearn.preprocessing import MinMaxScaler # allows normalisation
     from sklearn.metrics import mean_squared_error # allows compute the mean square␣
      ↪error to performance analysis
```

2.1 Data Pre-processing

Normalising the data on the (-1,1) interval by using the function MinMaxScaler.

```
[7]: # creating the dataset
     series = create_sin_data(samples=5000, period=50)

     # normalising the data
     scaler = MinMaxScaler(feature_range = (-1, 1))
```

```
# Note that below we are normalising the series.values (numpy array data) and␣
 ↪not
# series (DataFrame data)
scaled = scaler.fit_transform(series.values)

# Convert scaled (numpy.array) to DataFrame.
series = pd.DataFrame(scaled)
```

[8]:
```
# Windowing the dataset
window_size = 50

# you can also try drop_nan = False just to see the series_w.tail() result.
series_w = create_window_sin(series, window_size, drop_nan = True)

# Visualising the first 5 samples
series_w.head()
```

[8]:
```
              0         0         0         0         0         0         0  \
0 -9.821934e-16  0.062803  0.125358  0.187418  0.248739  0.309077  0.368195
1  6.280307e-02  0.125358  0.187418  0.248739  0.309077  0.368195  0.425859
2  1.253582e-01  0.187418  0.248739  0.309077  0.368195  0.425859  0.481842
3  1.874184e-01  0.248739  0.309077  0.368195  0.425859  0.481842  0.535922
4  2.487386e-01  0.309077  0.368195  0.425859  0.481842  0.535922  0.587887

          0         0         0  …        0         0         0         0  \
0  0.425859  0.481842  0.535922  …  0.535392  0.481291  0.425290  0.367610
1  0.481842  0.535922  0.587887  …  0.481291  0.425290  0.367610  0.308479
2  0.535922  0.587887  0.637531  …  0.425290  0.367610  0.308479  0.248130
3  0.587887  0.637531  0.684657  …  0.367610  0.308479  0.248130  0.186801
4  0.637531  0.684657  0.729081  …  0.308479  0.248130  0.186801  0.124735

          0         0         0         0         0         0
0  0.308479  0.248130  0.186801  0.124735  0.062176 -0.000628
1  0.248130  0.186801  0.124735  0.062176 -0.000628 -0.063430
2  0.186801  0.124735  0.062176 -0.000628 -0.063430 -0.125982
3  0.124735  0.062176 -0.000628 -0.063430 -0.125982 -0.188036
4  0.062176 -0.000628 -0.063430 -0.125982 -0.188036 -0.249347

[5 rows x 51 columns]
```

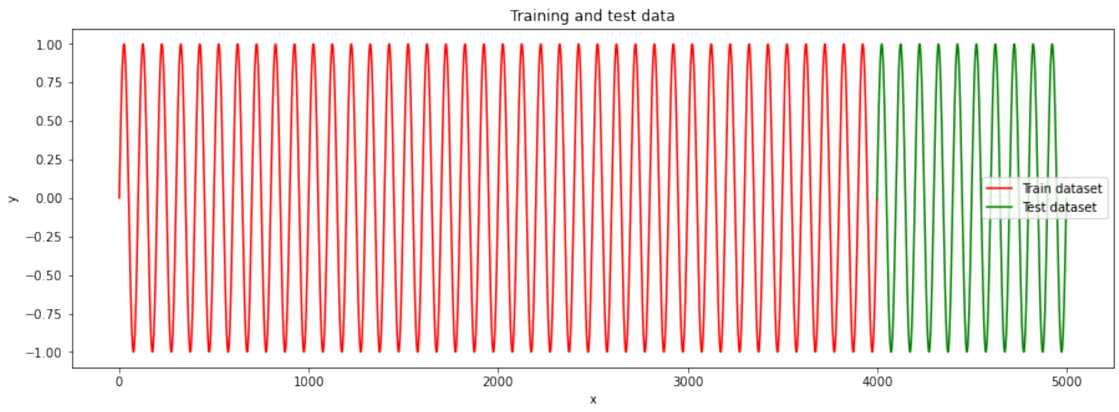
[9]:
```
print("Shape of input data: {}".format(series_w.shape))
```

```
Shape of input data: (4950, 51)
```

2.2 Spliting the Data into Training and Test Dataset

First we need to split our dataset into training and dataset, as described below.

[10]:
```
# First, let's visualise how our splitted dataset will like. For our better␣
↪understading,
# here we will plot the non-windowed data.

# we are using the function implemented in utils.py. We defined 80% to be used␣
↪for training and 20%
# for testing
plot_training_test_data(series, train_size=0.8)
```



[11]:
```
# Spliting the data into 80% of data for training and 20% of data for testing
train_X, train_Y, test_X, test_Y = train_test_split(series_w, train_size=0.8)

print("Training set shape: :")
print("  - X (inputs)", train_X.shape)
print("  - Y (output):", train_Y.shape)
print("\nTresting set shape: :")
print("  - X (inputs)", test_X.shape)
print("  - Y (output):", test_Y.shape)
```

```
Training set shape: :
  - X (inputs) (3960, 50)
  - Y (output): (3960,)

Tresting set shape: :
  - X (inputs) (990, 50)
  - Y (output): (990,)
```

[12]:
```
# The function train_test_split return 2D arrays, however, we will use tensor␣
↪representation,
# in which the third dimension is the number of the batch. To fix the shape, we␣
↪use
```

```
# the numpy.reshape (https://docs.scipy.org/doc/numpy/reference/generated/numpy.
↪reshape.html)
# function, as follows
train_X = np.reshape(train_X.values, (train_X.shape[0], train_X.shape[1], 1))
test_X = np.reshape(test_X.values, (test_X.shape[0], test_X.shape[1], 1))

print("Fixed shape: :")
print("  - Training dataset: ", train_X.shape)
print("  - Testing dataset: ", test_X.shape)
```

```
Fixed shape: :
  - Training dataset:  (3960, 50, 1)
  - Testing dataset:  (990, 50, 1)
```

Note that now *train_X* and *text_X* are numpy.array.

2.3 Basic RNN Model

You are going to build a model based on a simple (basic) RNN. This model consists in a **many_to_one** architecture, in which our input is the **window_size** and our output is the predicted value (dimension 1).

To build the model, you are going to use the following components from Keras:

- Sequencial: allows us to create models layer-by-layer.
- SimpleRNN: provides a Basic (simple) RNN architecture
- Dense: provides a regular fully-connected layer
- Activation: defines the activation function to be used

Basically, we will defined the sequence of our model by using *Sequential()*, include the layers:

```
model = Sequential()
model.add(SimpleRNN(...))
...
```

once created the model we can configure the model for training by using the method compile. Here we need to define the loss function (mean squared error, mean absolute error, cosine proximity, among others.), the optimizer (Stochastic gradient descent, RMSprop, adam, among others) and the metric to define the evaluation metric to be used to evaluate the performance of the model in the training step, as follows:

```
model.compile(loss = "...",
              optimizer = "...")
```

Also, we have the option to see a summary representation of the model by using the function summary. This function summarise the model and tell us the number of parameters that we need to tune.

[13]:
```
# Define the model.
model_rnn = Sequential()

# The input shape is the number of inputs per windows size (windows_size,1)
# we will need to define the number of units.

# The activation function for each cell, by default, is 'tanh'␣
↪(activation='tanh').
# Finally, we need to set return_sequences = False (by default is True). This␣
↪parameter
# is quite important because determines whether to return the last output in␣
↪the output sequence,
# or the full sequence. As we are using a many_to_one architecture, we are␣
↪interested in the last
# output. Note that the output will be the same shape of the units, for␣
↪instance if  units
# is equal 4, we will have an output array with four elements that represent␣
↪the weights to be used
# by the fully-connect (dense) layer that will generate the predicted output␣
↪(y_hat), see the code below.
model_rnn.add(SimpleRNN(input_shape = (window_size, 1),
                        units = 64, # try here with 2 or 4 units and check the␣
↪performance of the model
                        return_sequences = False))
model_rnn.add(Dense(units=1)) # in which units is dimensionality of the output␣
↪space, our case 1
                              # (we will predict the next value). By default,␣
↪the activation function is
                              # linear

# just print the model
model_rnn.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_1 (SimpleRNN)     (None, 64)                4224
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 65
=================================================================
Total params: 4,289
Trainable params: 4,289
Non-trainable params: 0
_________________________________________________________________
```

Exercise: Define optimiser (try ‘rmsprop’, ‘sgd’, ‘adagrad’ or ‘adadelta’ if you wich) and loss. As we are doing a regression task, the loss functions ‘mae’ (mean average error) and ‘mse’ (mean squared error) might be the choices.

For example:

```
model_rnn.compile(loss = "mse",
                  optimizer = "adam")
```

```
[14]: model_rnn.compile(loss = "mse",
                        optimizer = "adam")
```

Exercise: As you already know, once defined the model, we need to train it by using the function fit. This function performs the optmisation step. Hence, we can define the following parameters such as:

- batch size: defines the number of samples that will be propagated through the network
- epochs: defines the number of times in which all the training set (x_train_scaled) are used once to update the weights
- validation split: defines the percentage of training data to be using for validation
- among others (click here for more information)

For this exercise, define the number of epochs, validation_split and batch_size that best fit for you model and call the function fit to train the model.

For example:

```
# training the model
history = model_rnn.fit(train_X,
                        train_Y,
                        batch_size = 512,
                        epochs = 10,
                        validation_split = 0.1,
                        shuffle=False)
```

```
[15]: history = model_rnn.fit(train_X,
                              train_Y,
                              batch_size = 512,
                              epochs = 10,
                              validation_split = 0.1,
                              shuffle=False)
```

```
WARNING:tensorflow:From /home/leo/anaconda3/envs/day09/lib/python3.7/site-
packages/keras/backend/tensorflow_backend.py:422: The name tf.global_variables
is deprecated. Please use tf.compat.v1.global_variables instead.


2022-02-15 19:31:22.938191: I tensorflow/core/platform/cpu_feature_guard.cc:142]
Your CPU supports instructions that this TensorFlow binary was not compiled to
use: SSE4.1 SSE4.2 AVX AVX2 AVX512F FMA
2022-02-15 19:31:22.967120: I
tensorflow/core/platform/profile_utils/cpu_utils.cc:94] CPU Frequency:
2599990000 Hz
2022-02-15 19:31:22.969078: I tensorflow/compiler/xla/service/service.cc:168]
XLA service 0x5595458fb170 executing computations on platform Host. Devices:
2022-02-15 19:31:22.969118: I tensorflow/compiler/xla/service/service.cc:175]
StreamExecutor device (0): <undefined>, <undefined>
```

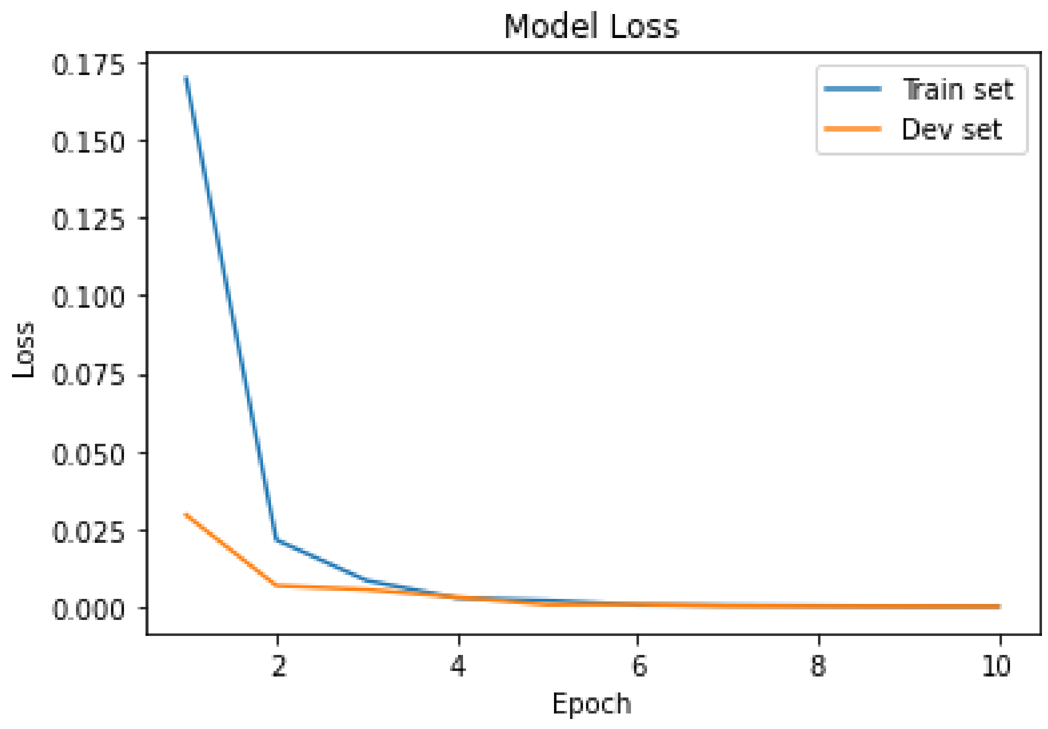
```
2022-02-15 19:31:23.014456: W
tensorflow/compiler/jit/mark_for_compilation_pass.cc:1412] (One-time warning):
Not using XLA:CPU for cluster because envvar
TF_XLA_FLAGS=--tf_xla_cpu_global_jit was not set.  If you want XLA:CPU, either
set that envvar, or use experimental_jit_scope to enable XLA:CPU.  To confirm
that XLA is active, pass --vmodule=xla_compilation_cache=1 (as a proper command-
line flag, not via TF_XLA_FLAGS) or set the envvar XLA_FLAGS=--xla_hlo_profile.
```

```
Train on 3564 samples, validate on 396 samples
Epoch 1/10
3564/3564 [==============================] - 1s 155us/step - loss: 0.1697 -
val_loss: 0.0295
Epoch 2/10
3564/3564 [==============================] - 0s 88us/step - loss: 0.0215 -
val_loss: 0.0068
Epoch 3/10
3564/3564 [==============================] - 0s 75us/step - loss: 0.0084 -
val_loss: 0.0055
Epoch 4/10
3564/3564 [==============================] - 0s 73us/step - loss: 0.0028 -
val_loss: 0.0031
Epoch 5/10
3564/3564 [==============================] - 0s 80us/step - loss: 0.0020 -
val_loss: 6.6844e-04
Epoch 6/10
3564/3564 [==============================] - 0s 81us/step - loss: 5.8694e-04 -
val_loss: 7.9299e-04
Epoch 7/10
3564/3564 [==============================] - 0s 83us/step - loss: 4.2061e-04 -
val_loss: 2.3089e-04
Epoch 8/10
3564/3564 [==============================] - 0s 89us/step - loss: 2.7516e-04 -
val_loss: 1.3926e-04
Epoch 9/10
3564/3564 [==============================] - 0s 89us/step - loss: 1.0795e-04 -
val_loss: 1.2225e-04
Epoch 10/10
3564/3564 [==============================] - 0s 89us/step - loss: 7.6166e-05 -
val_loss: 5.5553e-05
```

2.3.1 Using the history

Here we can see if the model overfits or underfits.

[16]:
```
plot_loss(history)
```



2.3.2 Prediction and Performance Analysis

Once the model was trained, we can use the function predict for prediction tasks.

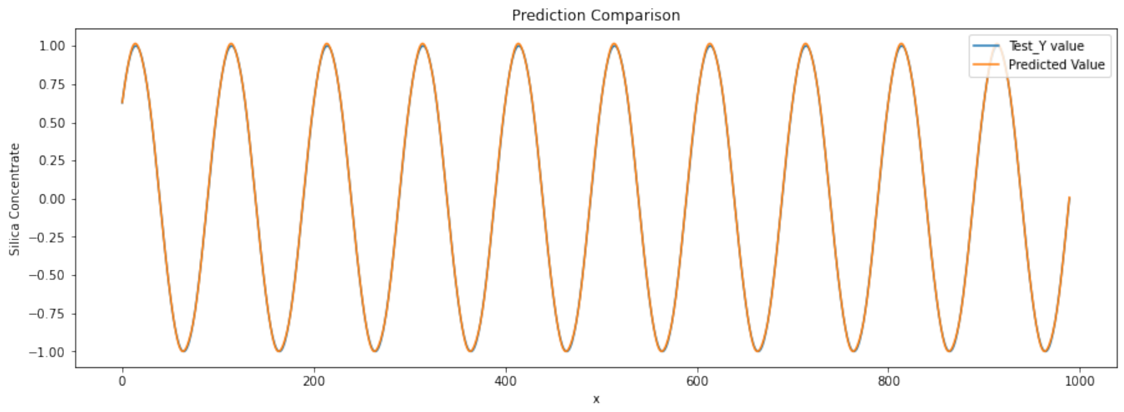
[17]:
```
# Prediction on test data.
pred_test = model_rnn.predict(test_X)

# In order to use to right data, first we need to use the inverse␣
↪transformation to get back true values.
#We do this to converte the predictions back into their original scale.
# This can be done by using the method inverse_transform provided by the␣
↪component MinMaxScaler
# (http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.
↪MinMaxScaler.html)
test_y_actual = scaler.inverse_transform(test_Y.values.reshape(test_Y.shape[0],␣
↪1))

# Computing the mean squared error of of the test data against the predicted␣
↪data
model_rnn_mse = mean_squared_error(test_y_actual, pred_test)
print("MSE for predicted test set: %2f" % model_rnn_mse)
```

```
MSE for predicted test set: 0.000054
```

2.3.3 Visualising the predicted Data

Here, we are going to plot the test data against the the predicted data.

[18]:
```
plot_comparison([test_y_actual, pred_test],
                ['Test_Y value','Predicted Value'],
                title='Prediction Comparison')
```



2.4 Model Understanding: What are the model weights and hidden units?

In order to understand the model, let's have a look the shapes of the weights and the hidden units.

For the SimpleRNN: - weights[0] is the input matrix and has a shape [input_dim, output_dim] - weights[1] is the weight matrix and has a shape [output_dim, output_dim] - weights[2] is the bias matrix and has a shape [output_dim]

For the fully-connected layer (Dense): - weights[0] is the weight matrix and has shape [output_dim, output_dim] - weights[1] is the bias matrix and has a shape [output_dim]

For the hidden units: - The shape must be the same shape of the test set, i.e., if the shape of the test set it 100 we will have 100 hidden units in the prediction stage

[19]:
```
print('Number of layers: ', len(model_rnn.layers))

weights = {}
layers = []
for layer in model_rnn.layers:
    weights[layer.name] = layer.get_weights()
    layers.append(layer.name)

hidden_units = model_rnn.predict(test_X).flatten()

print('Weights shape:')

# we know that we have just two layers: SimpleRNN and Dense
```

```
print(" ", layers[0])
print("    Input: ",weights[layers[0]][0].shape)
print("    Weights (W_hh): ",weights[layers[0]][1].shape)
print("    Bias (b_h): ",weights[layers[0]][2].shape)

print(" ", layers[1])
print("    Weights (W_yh): ",weights[layers[1]][0].shape)
print("    Bias (b_y): ",weights[layers[1]][1].shape)

print('\nHidden units shape: ', hidden_units.shape, ' and the shape of the␣
 ↪test_X is ', test_X.shape[0])
```

```
Number of layers:  2
Weights shape:
  simple_rnn_1
    Input:  (1, 64)
    Weights (W_hh):  (64, 64)
    Bias (b_h):  (64,)
  dense_1
    Weights (W_yh):  (64, 1)
    Bias (b_y):  (1,)

Hidden units shape:  (990,)  and the shape of the test_X is  990
```

3 Another Way to Create Models

So far we see how create a RNN model in Keras by using the **Sequential** API. However, this API is limited, it does not allow us to create models that share layers or have multiple inputs or outputs. For example, it is not easy to define models that may have:

- multiple different input sources,
- produce multiple output destinations, or
- models that re-use layers.

Another way to create a model is by using the **Functional** API. This API allows us to create a more robust model, which more flexibility and fredoom do define and re-use layers. This allow us do define complex models, such as multi-output models, directed acyclic graphs, or models with shared layers.

The use of **Functional API** is beyond this tutorial, however it is well documented can be found here.

[ ]:

```
```

[ ]:

```
```