tssl_lab4

October 25, 2024

1 TSSL Lab 4 - Recurrent Neural Networks

- Qinyuan Qi (qinqi464)

In this lab we will explore different RNN models and training procedures for a problem in time series prediction.

1.1 0: Init script to load contents if running on Google CoLab

[1]:
```
# check the code is running in colab env or not, and then load data from google drive
import sys
IN_COLAB = 'google.colab' in sys.modules
if IN_COLAB:
    # mount google drive
    from google.colab import drive
    import sys
    # copy data from google drive to local disk
    drive.mount('/content/drive')
    !cp -r drive/MyDrive/TSSL/Lab4/* .
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

1.2 Lab Start Here

[2]:
```
import numpy as np
import tensorflow as tf
import keras
from keras import layers
import pandas
import matplotlib.pyplot as plt

plt.rcParams["figure.figsize"] = (10,6)  # Increase default size of plots
```

Set the random seed, for reproducibility

[3]:
```
np.random.seed(42)
tf.random.set_seed(42)
```

```
print(np.__version__)
print(tf.__version__)
```

```
1.26.4
2.17.0
```

1.3 1. Load and prepare the data

We will build a model for predicting the number of sunspots. We work with a data set that has been published on Kaggle, with the description:

Sunspots are temporary phenomena on the Sun's photosphere that appear as spots darker than the surrounding areas. They are regions of reduced surface temperature caused by concentrations of magnetic field flux that inhibit convection. Sunspots usually appear in pairs of opposite magnetic polarity. Their number varies according to the approximately 11-year solar cycle.

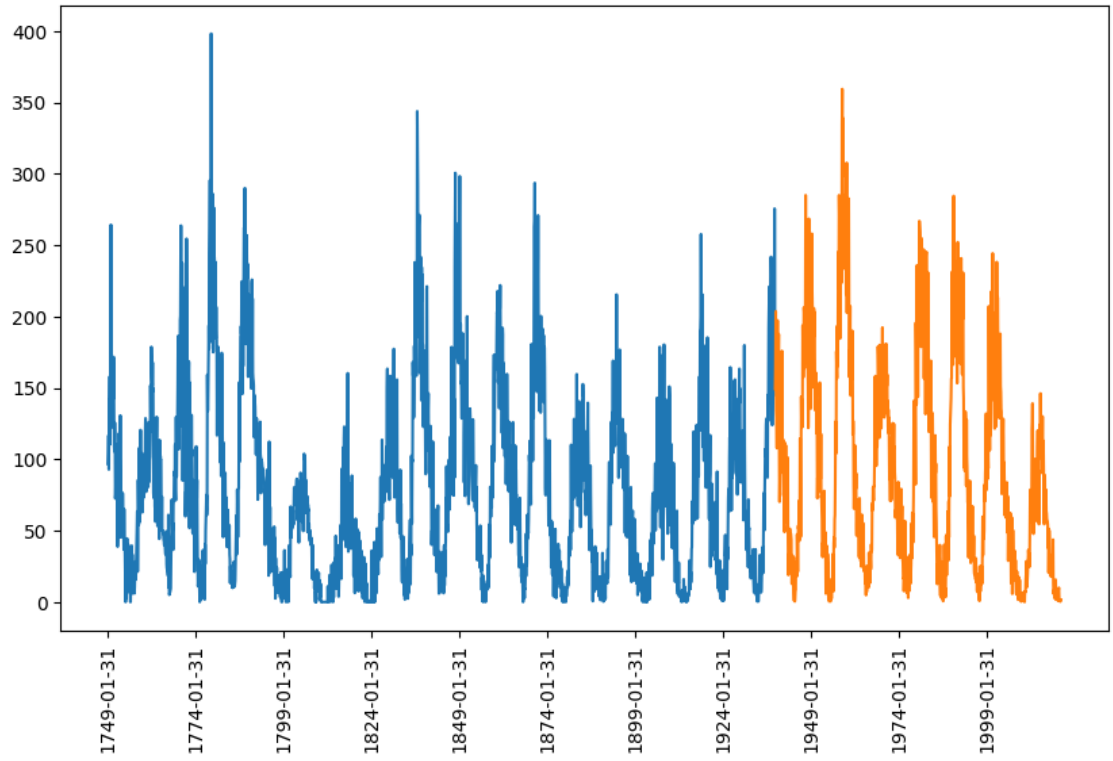
The data consists of the monthly mean total sunspot number, from 1749-01-01 to 2017-08-31.

[4]:
```
# Read the data
data = pandas.read_csv('Sunspots.csv',header = 0)
dates = data['Date'].values
y = data['Monthly Mean Total Sunspot Number'].values
ndata = len(y)
print(f'Total number of data points: {ndata}')

# We define a train/test split, here with 70 % training data
ntrain = int(ndata*0.7)
ntest = ndata - ntrain
print(f'Number of training data points: {ntrain}')
```

```
Total number of data points: 3252
Number of training data points: 2276
```

[5]:
```
plt.plot(dates[:ntrain], y[:ntrain])
plt.plot(dates[ntrain:], y[ntrain:])
# Show only one tick every 25th year for clarity
plt.xticks(range(0, ndata, 300), dates[::300], rotation = 90);
```



There is a clear seasonality to the data, but the amplitude of the peaks very quite a lot. Also, we note that the data is nonnegative, which is natural since it consists of counts of sunspots. However, for simplicity we will not take this constraint into account in this lab assignment and allow ourselves to model the data using a Gaussian likelihood (i.e. using MSE as a loss function).

From the plot we see that the range of the data is roughly [0,400] so as a simple normalization we divide by the constant `MAX_VAL=400`.

[6]:
```
MAX_VAL = 400
y = y/MAX_VAL
```

1.4 2. Baseline methods

Before constructing any sophosticated models using RNNs, let's consider two baseline methods,

1. The first baseline is a "naive" method which simply predicts $y_t = y_{t-1}$.
2. The second baseline is an AR(p) model (based on the implementation used for lab 1).

We evaluate the performance of these method in terms of mean-squared-error and mean-absolute-error, to compare the more advanced models with later on.

[7]:
```
def evalutate_performance(y_pred, y, split_time, name=None):
    """This function evaluates and prints the MSE and MAE of the prediction.

    Parameters
```

```
    ----------
    y_pred : ndarrary
        Array of size (n,) with predictions.
    y : ndarray
        Array of size (n,) with target values.
    split_time : int
        The leading number of elements in y_pred and y that belong to the
        training data set.
        The remaining elements, i.e. y_pred[split_time:] and y[split_time:]
        are treated as test data.
    """

    # Compute error in prediction
    resid = y - y_pred

    # We evaluate the MSE and MAE in the original scale of the data, i.e.
    # we add back MAX_VAL
    train_mse = np.mean(resid[:split_time]**2)*MAX_VAL**2
    test_mse = np.mean(resid[split_time:]**2)*MAX_VAL**2
    train_mae = np.mean(np.abs(resid[:split_time]))*MAX_VAL
    test_mae = np.mean(np.abs(resid[split_time:]))*MAX_VAL

    # Print
    print(f'Model {name}\n'
          f'Training MSE: {train_mse:.4f}, MAE: {train_mae:.4f}\n'
          f'Testing MSE:  {test_mse:.4f},  MAE: {test_mae:.4f}')
```

Q1: Implement the naive baseline method which predicts according to $\hat{y}_{t|t-1} = y_{t-1}$. Since the previous value is needed for the prediction we do not get a prediction at $t = 1$. Hence, we evaluate the method by predicting values at $t = 2, \ldots, n$ (cf. an AR(p) model where we start predicting at $t = p + 1$).

[8]:
```
# Store the predictions in an array of length ndata-1. Note that there is a
# shift in the indices between the prediction and the observation sequence,
# since there is no prediction available for the first observation.
# Specifically, y_pred_naive[t] is a prediction of y[t+1], so the first
# element of y_pred_naive is a prediction of the second element of y,
# and so on. We will use the same "bookeeping convention" throughout the lab,
# so it is important that you understand it!
y_pred_naive = y[:ndata-1]

evalutate_performance(y_pred_naive,   # Predictions
                      y[1:],          # Correspondsing target values
                      ntrain-1,       # Number of leading elements in the
                                      # input arrays corrsponding to training
                                      # data points
                      name='Naive')
```

```
Model Naive
Training MSE: 776.5437, MAE: 19.3285
Testing MSE:  708.6360,  MAE: 19.2256
```

Next, we consider a slightly more advanced baseline method, namely an AR(p) model.

[9]:
```
# We import two functions that were written as part of lab 1
from tssltools_lab4 import fit_ar, predict_ar_1step

p=30  # Order of the AR model (set by a few manual trials)
ar_coef = fit_ar(y[:ntrain], p)  # Fit the model to the training data

# Predict. Note that y contains both training and validation data,
# and the prediction is for the values y_{p+1}, ..., y_{n}.
y_pred_ar = predict_ar_1step(ar_coef, y)
```

[10]:
```
evalutate_performance(y_pred_ar,  # The prediction array is of length n-p
                      y[p:],      # Corresponding target values
                      ntrain-p,   # Number of leading elements in the input
                                  # arrays corrsponding to training data points
                      name='AR')
```

```
Model AR
Training MSE: 603.8656, MAE: 17.3420
Testing MSE:  590.3732,  MAE: 17.6221
```

1.5 3. Simple RNN

We will now construct a model based on a recurrent neural network. We will initially use the `SimpleRNN` class from *Keras*, which correspond to the basic Jordan-Elman network presented in the lectures.

Q2: Assume that we construct an "RNN cell" using the call `layers.SimpleRNN(units = d, return_sequences=True)`. Now, assume that an array `X` with the dimensions `[Q,M,P]` is fed as the input to the above object. We know that `X` contains a set of sequences (time series) with equal lengths. Specify which of the symbols Q, M, P that corresponds to each of the items below:
- The length of the sequences (number of time steps) - The number of features (at each time step), i.e. the dimension of each time series - The number of sequences

Furthermore, specify the values of Q, M, P for the data at hand (treated as a single time series).

Hint: Read the documentation for SimpleRNN to find the answer.

A2:

According to the Tensor flow documents, we know that [Q,M,P] is defined as a 3D tensor, with shape [batch, timesteps, feature].

So Q,M,P has the following meaning:

Q: The number of sequences

M: The length of the sequences (number of time steps)

P: The number of features (at each time step), i.e. the dimension of each time series

Q3: Continuing the question above, answer the following:

- What is the meaning of setting `units = d`?
- Assume that we pass a single time series of length n as input to the layer. Then what is the dimension of the *output*?
- If we would had set the parameter `return_sequences=False` when constructing the layer, then what would be the answer to the previous question?

A3:

According to the TF's document, we know that setting units = d means set the dimensionality of the output space (d > 0).

If we pass a single time series of length n as input to the layer, Q = 1, M = n, so the dimmension of output is [1,d]

Parameter "return_sequences" means whether to return the last output in the output sequence, if we set the parameter `return_sequences=False` when constructing the layer, dimmension of output is [1,n, d]

In *Keras*, each layer is created separately and are then joined by a `Sequential` object. It is very easy to construct stacked models in this way. The code below corresponds to a simple Jordan-Elman Network on the form,

$$\begin{aligned} \mathbf{h}_t &= \sigma(W\mathbf{h}_{t-1} + Uy_{t-1} + b), \\ \hat{y}_{t|t-1} &= C\mathbf{h_t} + c, \end{aligned}$$

Note: It is not necessary to explicitly specify the input shape, since this can be inferred from the input on the first call. However, for the `summary` function to work we need to tell the model what the dimension of the input is so that it can infer the correct sizes of the involved matrices. Also note that in *Keras* you can sometimes use `None` when some dimensions are not known in advance.

[11]:
```
d = 10  # hidden state dimension

model0=keras.Sequential([
    keras.Input(shape=(None,1)),
    # Simple RNN layer
    layers.SimpleRNN(units = d, return_sequences=True, activation='tanh'),
    # A linear output layer
    layers.Dense(units = 1, activation='linear')
])

# We store the initial weights in order to get an exact copy of the model
# when trying different training procedures
model0.summary()
init_weights = model0.get_weights().copy()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
simple_rnn (SimpleRNN)	(None, None, 10)	120
dense (Dense)	(None, None, 1)	11

```
Total params: 131 (524.00 B)

Trainable params: 131 (524.00 B)

Non-trainable params: 0 (0.00 B)
```

Q4: From the model summary we can see the number of paramters associated with each layer. Relate these numbers to the dimensions of the weight matrices and bias vectors $\{W, U, b, C, c\}$ in the mathematical model definition above.

A4:

Dimmensions as follows:

W:hidden to hidden with dimension [10 x 10]

U:input to hidden with dimension, but we need tranpose,so dim is [10 x 1]

b:[10]

C hidden to outputm, also need transpose, so dim is [1 x 10]

c:[1]

1.6 4. Training the RNN model

In this section we will consider a few different ways of handling the data when training the simple RNN model constructed above. As a first step, however, we construct explicit input and target (output) arrays for the training and test data, which will simplify the calls to the training procedures below.

The task that we consider in this lab is one-step prediction, i.e. at each time step we compute a prediction $\hat{y}_{t|t-1} \approx y_t$ which depend on the previous observations $y_{1:t-1}$. However, when working with RNNs, the information contained in previous observations is aggregated in the *state* of the RNN, and we will only use y_{t-1} as the *explicit input* at time step t.

Furthermore, when addressing a problem of time series prediction it is often a good idea to introduce an explicit skip connection from the input y_{t-1} to the prediction $\hat{y}_{t|t-1}$. Equivalently, we can *define the target value* at time step t to be the residual $\tilde{y}_t := y_t - y_{t-1}$. Indeed, if the model can predict the value of the residual, then we can simply add back y_{t-1} to get a prediction of y_t.

Taking this into consideration, we define explicit input and output arrays as shifted versions of the data series $y_{1:n}$.

[12]:
```
# Training data
# Input is denoted by x, training inputs are x[0]=y[0], ...,
# x[ntrain-1]=y[ntrain-1]
x_train = y[:ntrain-1]
# Output is denoted by yt, training outputs are yt[0]=y[1]-y[0], ...,
# yt[ntrain-1] = y[ntrain]-y[ntrain-1]
yt_train = y[1:ntrain] - x_train

# Test data
x_test = y[ntrain-1:-1]  # Test inputs are x_test[0] = y[ntrain-1], ...,
                         # x_test[ntest] = y[n-1]
yt_test = y[ntrain:] - x_test  # Test outputs are yt_test[0] = y[ntrain]-
                               # y[ntrain-1], ..., yt_test[ntest] = y[n]-y[n-1]

# Reshape the data
# Ensure compatibility with the input requirements of the RNN model[Q,M,P]
x_train = x_train.reshape((1,ntrain-1,1))
yt_train = yt_train.reshape((1,ntrain-1,1))
x_test = x_test.reshape((1,ntest,1))
yt_test = yt_test.reshape((1,ntest,1))
```

1.6.1 Option 1. Process all data in each gradient computation ("do nothing")

The first option is to process all data at each iteration of the gradient descent method.

[13]:
```
# This creates a new instance of the same model
model1 = keras.models.clone_model(model0)
# We set the initial weights to be the same for all models
model1.set_weights(init_weights)
```

Q5: What should we set the *batch size* to, in order to compute the gradient based on the complete training data sequnce at each iteration? Complete the code below!

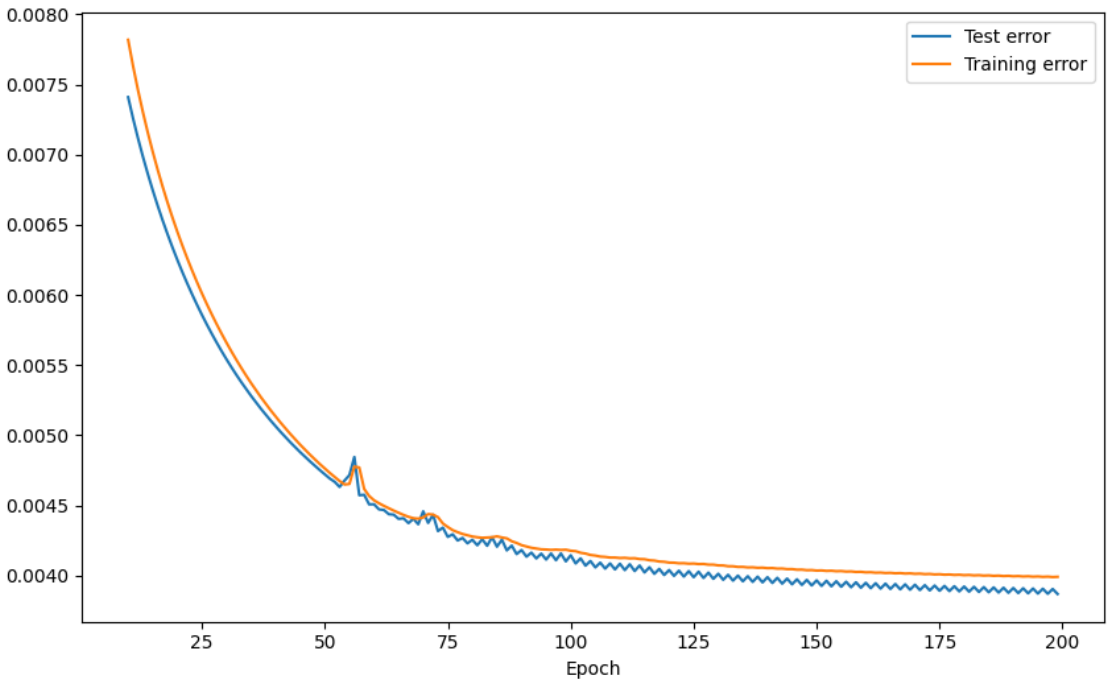
Note: You can set `verbose=1` if you want to monitor the training progress, but if you do, please **clear the output of the cell** before generating a pdf with your solutions, so that we don't get multiple pages with training errors in the submitted reports.

[14]:
```
model1.compile(loss='mse', optimizer='rmsprop', metrics=['mse'])
history = model1.fit(x_train, yt_train,
                     epochs = 200,
                     batch_size = ntrain, # one batch
```

```
                    verbose = 0,
                    validation_data = (x_test, yt_test))
```

We plot the training and test error vs the iteration (epoch) number, using a helper function from the `tssltools_lab4` module.

[15]:
```
from tssltools_lab4 import plot_history
start_at = 10  # Skip the first few epochs for clarity
plot_history(history, start_at)
```



Q6: Finally we compute the predictions of $\{y_t\}$ for both the training and test data uning the model's `predict` function. Complete the code below to compute the predictions.

Hint: You need to reshape the data when passing it to the `predict` to comply with the input shape used in *Keras* (cf. above).

Hint: Since the model is trained on the residuals $\tilde{y}_t$, don't forget to add back y_{t-1} when predicting y_t. However, make sure that you dont "cheat" by using a non-causal predictor (i.e. using y_t when predicting y_t)!

[16]:
```
# Predict on all data using the final model.

# We predict using y_1,...,y_{n-1} as inputs, resulting in predictions
# of the values y_2, ..., y_n. That is, y_pred1 should be an (n-1,)
# array where element y_pred[t] is based only on values y[:t]
y_pred1 = model1.predict(y[:ndata-1].reshape(1, ndata-1, 1) ).flatten() \
```

```
            + y[:ndata-1]
```

```
1/1             0s 288ms/step
```

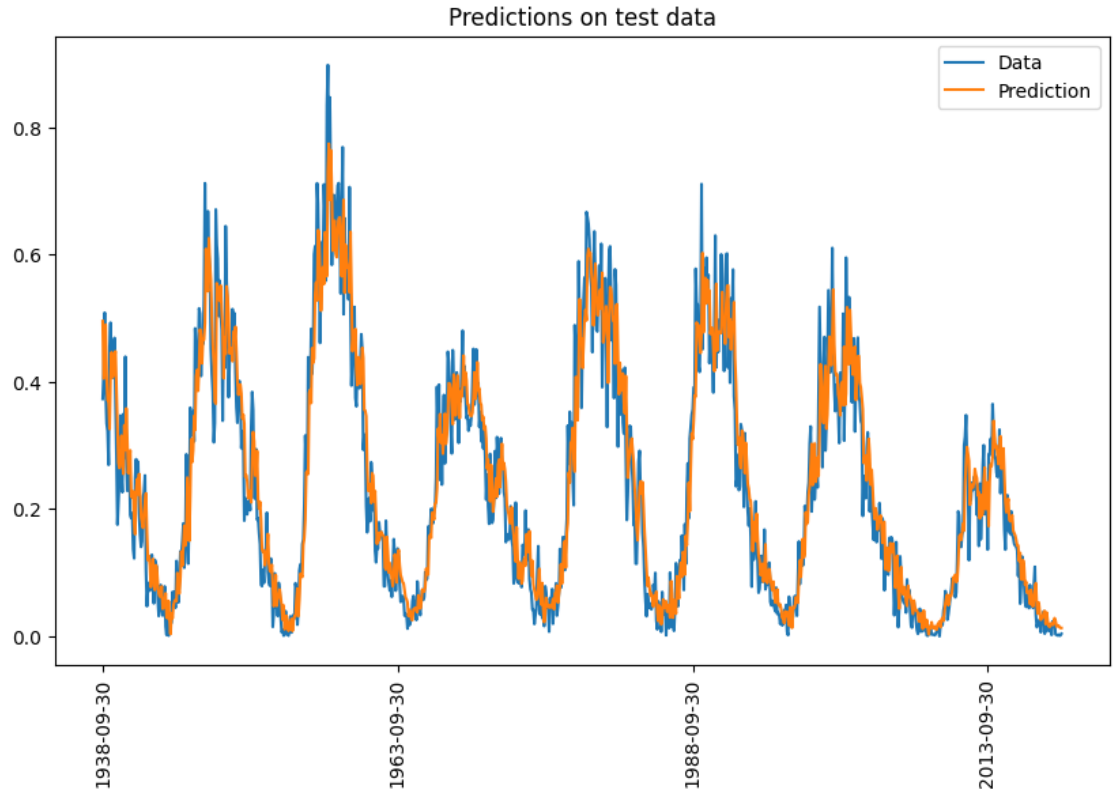
Using the prediction computed above we can plot them and evaluate the performance of the model in terms of MSE and MAE.

[17]:
```
def plot_prediction(y_pred):
    # Plot prediction on test data
    plt.plot(dates[ntrain:], y[ntrain:])
    plt.plot(dates[ntrain:], y_pred[ntrain-1:])
    # Show only one tick every 25th year for clarity
    plt.xticks(range(0, ntest, 300), dates[ntrain::300], rotation = 90);
    plt.legend(['Data','Prediction'])
    plt.title('Predictions on test data')
```

[18]:
```
# Plot prediction
plot_prediction(y_pred1)

# Evaluate MSE and MAE (both training and test data)
evalutate_performance(y_pred1, y[1:], ntrain-1, name='Simple RNN, "do nothing"')
```

```
Model Simple RNN, "do nothing"
Training MSE: 638.1165, MAE: 18.1304
Testing MSE:  605.6443,  MAE: 18.1051
```



1.6.2 Option 2. Random windowing

Instead of using all the training data when computing the gradient for the numerical optimizer, we can speed it up by restricting the gradient computation to a smaller window of consecutive time steps. Here, we sample a random window within the traing data and "pretend" that this window is independent from the observations outside the window. Specifically, when processing the observations within each window the hidden state of the RNN is initialized to zero at the first time point in the window.

To implement this method in Python, we will make use of a *generator function*. A generator is a function that can be paused, return an intermediate value, and then resumed to continue its execution. An intermediate return value is produces using the `yield` keyword.

Generators are used in *Keras* to implement inifinite loops that feed the training procedure with training data. Specifically, the `yield` statement of the generator should return a pair `x, y` with inputs and corresponding targets from the training data. Each epoch of the training procedure will then call the generator for a total of `steps_per_epoch` such `yield` statements.

[19]:
```
def generator_train(window_size):
    while True:
        """The upper value is excluded in randint, so the maximum value that we
        can get is tt = ntrain-window_size-1.
        Hence, the maximum end point of a window is ntrain-1, in agreement with
        the fact that the size of input/output is ntrain-1 when working with
        one-step-ahead prediction."""

        # First time index of window (inclusive)
        start_of_window = np.random.randint(0, ntrain - window_size)
        # Last time index of window (exclusive, i.e. this is really the first
        # index _after_ the window)
        end_of_window = start_of_window + window_size
        yield x_train[:,start_of_window:end_of_window,:], \
              yt_train[:,start_of_window:end_of_window,:]
```

[20]:
```
# This creates a new instance of the same model
model2 = keras.models.clone_model(model0)
# We set the initial weights to be the same for all models
model2.set_weights(init_weights)
```

Q7: Assume that we process a window of observations of length `window_size` at each iteration. Then, how many gradient steps per epoch can we afford, for computational cost per epoch to be comparable to the method considered in Option 1? Set the `steps_per_epoch` parameter of the fitting function based on your answer.

A7:

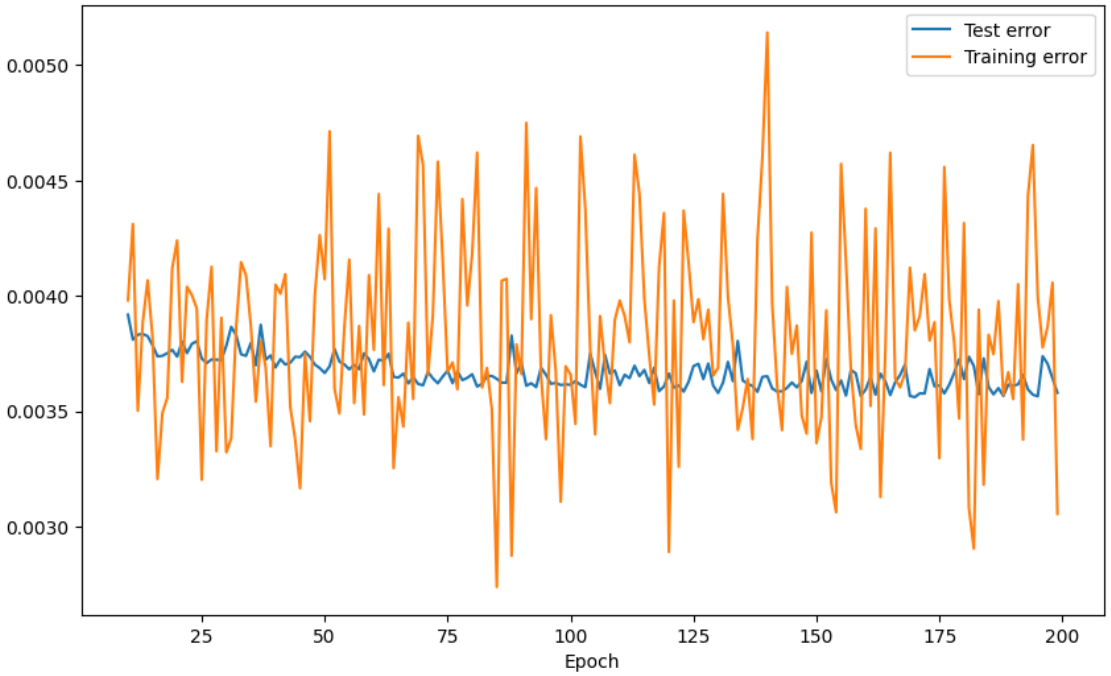
Since we use batch_size = ntrain in Option 1, so we can afford ntrain / window_size steps per epoch.

[21]:
```
from math import floor

window_size = 100
model2.compile(loss='mse', optimizer='rmsprop', metrics=['mse'])
history = model2.fit(generator_train(window_size),
           epochs = 200,
           verbose = 0,
           steps_per_epoch = floor(ntrain/window_size) ,
           validation_data = (x_test, yt_test))
```

Similarly to above we plot the error curves vs the iteration (epoch) number.

[22]:
```
plot_history(history, start_at)
```



Q8: Comparing this error plot to the one you got for training Option 1, can you see any *qualitative* differences? Explain the reason for the difference.

A8:

Comparing error plot in option 2 to that in Option 1, we can find that the error plot in option 1 drop significantly at the begining before it starts to converge. while the error plot 2 does not drop significantly at the begining but fluctuates a lot all the time(training data).

The reason is that in option 1, model see the whole data set in each epoch, but each epoch has less chances to update the weights of model, so it will drop sharply at the begining.

While in option 2, each epoch only uses a small window of data, it has more chance to update weight in each epoch, make the MSE relatively stable at the begining, but since the model only see

a small window of data, it will fluctuate a lot when training data.

Q9: Compute a prediction for all values of $\{y_2, \ldots, y_n\}$ analogously to **Q6**.

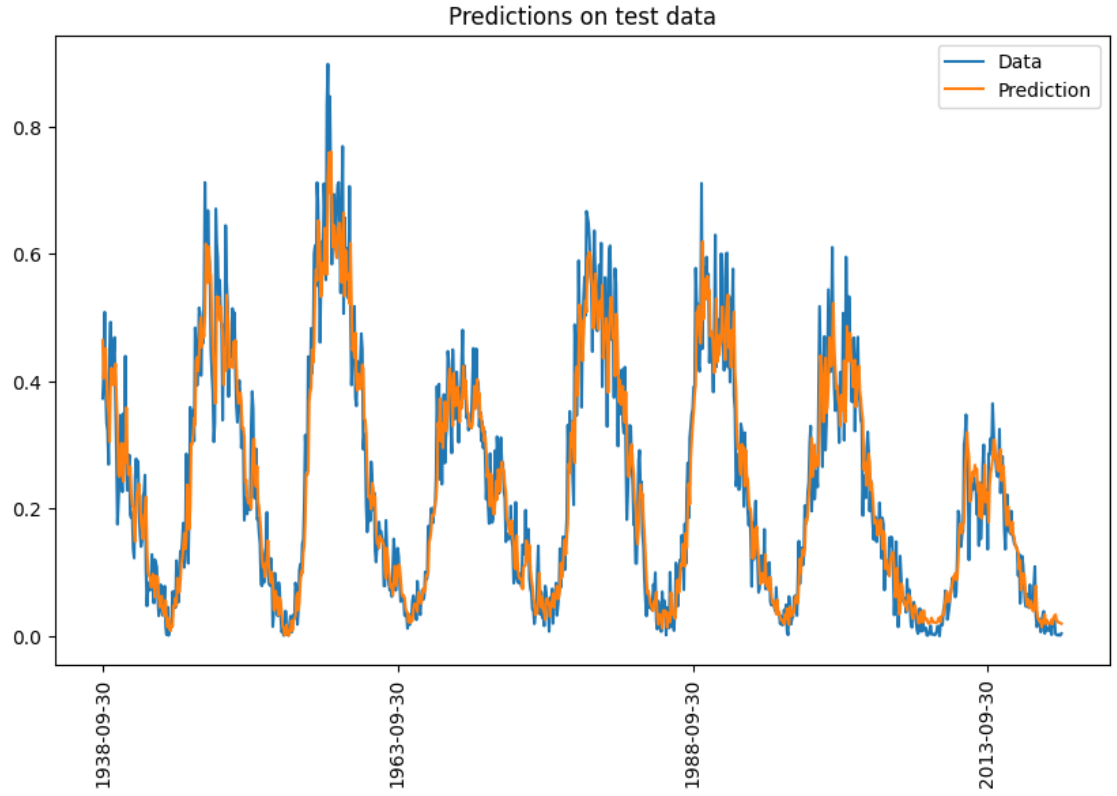
[23]:
```
# Predict on all data using the final model.
# We predict using y_1,...,y_{n-1} as inputs,
# resulting in predictions of the values y_2, ..., y_n
y_pred2 = model2.predict(y[:ndata-1].reshape(1, ndata-1, 1) ).flatten() \
          + y[:ndata-1]
```

```
1/1             0s 361ms/step
```

[24]:
```
# Plot prediction on test data
plot_prediction(y_pred2)

# Evaluate MSE and MAE (both training and test data)
evalutate_performance(y_pred2, y[1:], ntrain-1, name='Simple RNN, windowing')
```

```
Model Simple RNN, windowing
Training MSE: 605.3834, MAE: 17.5237
Testing MSE:  571.2137,  MAE: 17.2618
```



1.6.3 Option 3. Sequential windowing with stateful training

As a final option we consider a model aimed at better respecting the temporal dependencies between consequtive windows. This is based on "statefulness" which simply means that the RNN remembers its hidden state between calls. That is, if model is in stateful mode and is used to process two sequences of inputs after each other, then the final state from the first sequence is used as the initial state for the second sequence.

[25]:
```
# To enable stateful training, we need to create model where we
# set stateful=True in the RNN layer
model3=keras.Sequential([
    keras.Input(batch_shape=(1, None, 1)),
    # Simple RNN layer with stateful=True
    layers.SimpleRNN(units = d, return_sequences=True,
                     stateful=True, activation='tanh'),
    # A linear output layer
    layers.Dense(1, activation='linear')
])
model3.set_weights(init_weights)
```

Q10: When working with stateful training we need to make some adjustments to the training data generator. In this case we will not create a generator but instead reshape the data into an array of samples corresponding to consecutive windows from the time series.

Calculate the number of windows that we will fit in the data to complete the code below.

[26]:
```
from pathlib import WindowsPath
# Calculate number of windows
number_of_windows = int(floor(ntrain/window_size) )

# We might have to discard some samples,
# -1 is the placeholder, will set automatically
x_train_stateful = x_train[0,0:number_of_windows*window_size,0].\
                   reshape([-1,window_size,1])
yt_train_stateful = yt_train[0,0:number_of_windows*window_size,0].\
                    reshape([-1,window_size,1])
```

With the data defined we need to train the model, in this case we need to complete one epoch at a time and then when each new epoch starts (process the data once again) we need to reset the model (we should only keep the state within one run of the data).

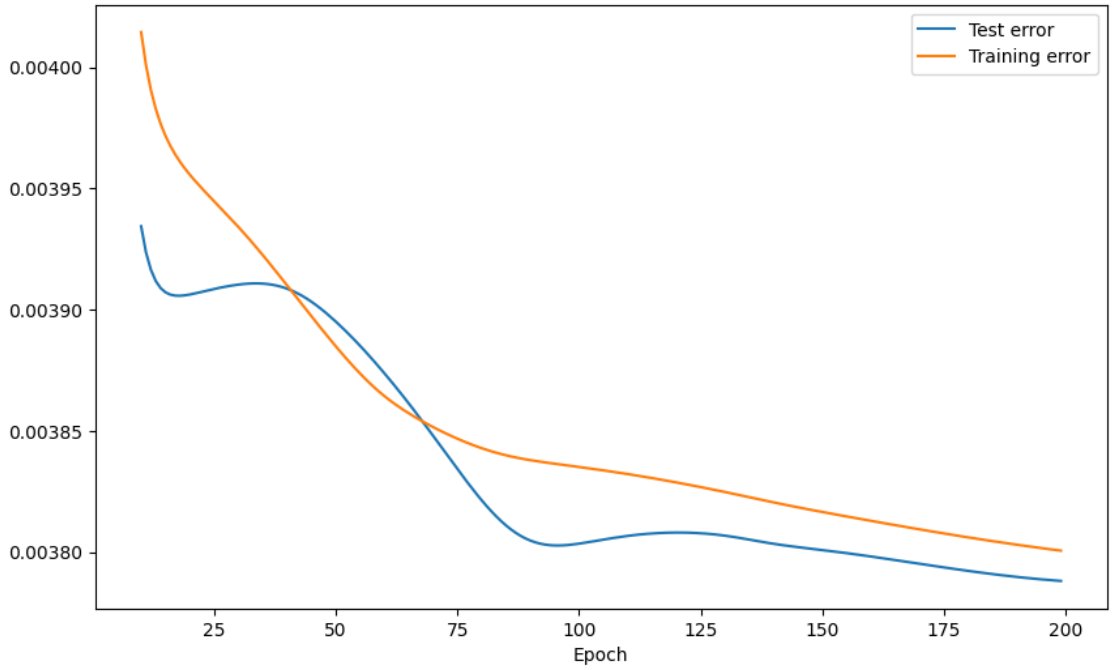
Because of this we need to save the data in our own structure between the epochs and save the results.

[27]:
```
model3.compile(loss='mse', optimizer='rmsprop', metrics=['mse'])
epochs = 200
history = keras.callbacks.History()
history.history = {'loss':[], 'val_loss':[]}
history.epoch = []
```

```
for i in range(epochs):
    h = model3.fit(x_train_stateful,yt_train_stateful,
        batch_size = 1,
        verbose = 0,
        validation_data = (x_test, yt_test),
        shuffle=False)
    # Resets the RNN state at the end of each epoch
    model3.layers[0].reset_states()
    history.history['loss'].extend(h.history['loss'])
    history.history['val_loss'].extend(h.history['val_loss'])
    history.epoch.extend([i])
```

Similarly to above we plot the error curves vs the iteration (epoch) number.

[28]:
```
plot_history(history, start_at)
```



Q11: Comparing this error plot to the one you got for training Options 1 and 2, can you see any *qualitative* differences?

Optional: If you have a theory regarding the reason for the observed differences, feel free to explain!

A11:

Compare the "window" model and "window/stateful" model, we can find that the error plot in "window" model fluctuates a lot, while the error plot in "window/stateful" model's curve is more smooth.

From the above plot(window/stateful), we can find that test error drop steadly when epoch increase,

which is what we expected, but if we fit the model several times, sometimes the test error is not so smooth and will fluctuate a lot in some point(still better than "window" model).

The reason maybe because of the small data size and the batch size in each epoch, which makes the model do not have enough chance to update the weights in each epoch.

Q12: Compute a prediction for all values of $\{y_2, \ldots, y_n\}$ analogously to **Q6**.

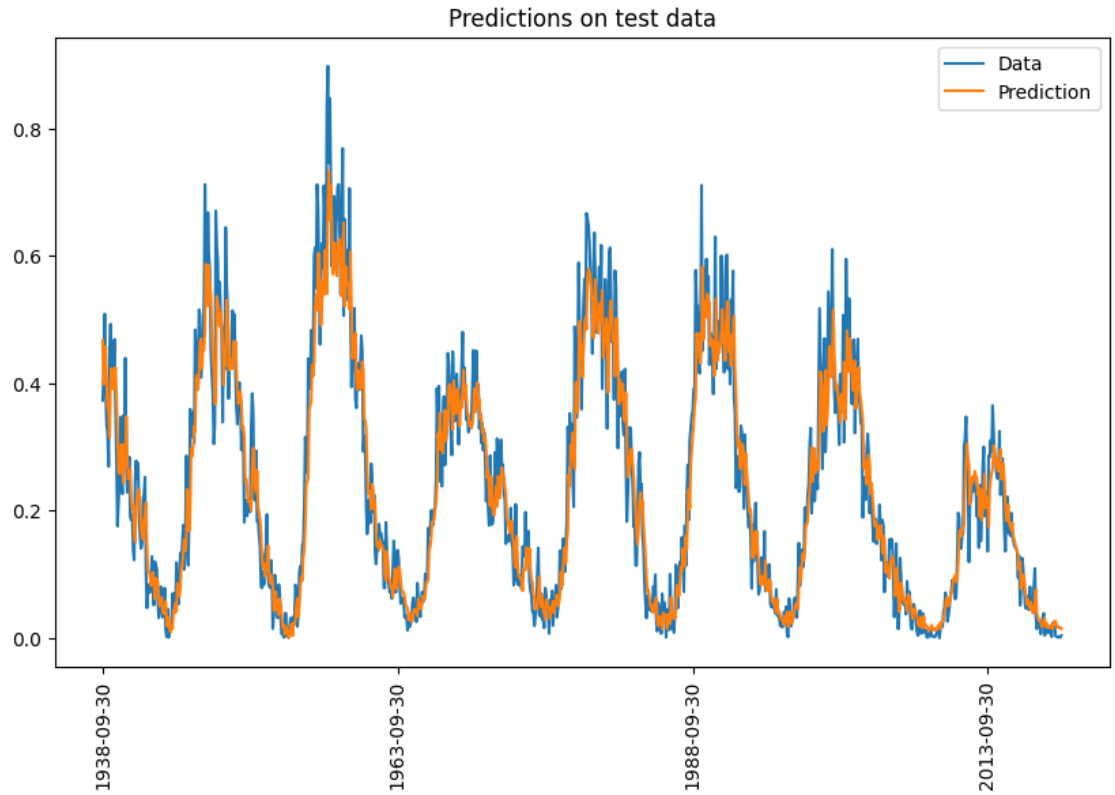
[29]:
```
# Predict on all data using the final model.
# We predict using y_1,...,y_{n-1} as inputs,
# resulting in predictions of the values y_2, ..., y_n
y_pred3 = model3.predict( y[:ndata-1].reshape(1,ndata-1,1) ).flatten() \
          + y[:ndata-1]
```

```
1/1             1s 760ms/step
```

[30]:
```
# Plot prediction on test data
plot_prediction(y_pred3)

# Evaluate MSE and MAE (both training and test data)
evalutate_performance(y_pred3, y[1:], ntrain-1,
                      name='Simple RNN, windowing/stateful')
```

```
Model Simple RNN, windowing/stateful
Training MSE: 608.9876, MAE: 17.4253
Testing MSE:  594.9701,  MAE: 17.4775
```



1.7 5. Reflection

Q13: Which model performed best? Did you manage to improve the prediction compared to the two baseline methods? Did the RNN models live up to your expectations? Why/why not? Please reflect on the lab using a few sentences.

A13:

Compare the MSE and MAE on the test and training data for different models, we can find that Simple RNN, windowing model has the lowest MSE and MAE on the test data and training data, which is the best performance model.

Since the data we use is temporal data which has strong temporal dependency, the RNN model can capture the temporal dependency may become a considerable advantage for the RNN model we should choose.

In the "windowing" model, we choose samples randomly without considering the temporal dependency, but the "stateful" model, considers the temporal dependency between consecutive windows, so the "stateful" model should have better performance than the "windowing" model.

However, the "stateful" model is worse than the random windowing model in this lab, which may be caused by the small data set / small epochs(only 200), so the model can not capture all the temporal dependencies between consecutive windows.

Considering the small gap between the random window model and the stateful model, windowing/stateful model is a good choice for this data set.

[31]:
```
# Print MSE MAE of 4 models
evalutate_performance(y_pred_naive,  # Predictions
                      y[1:],         # Correspondsing target values
                      ntrain-1,      # Number of leading elements in the
                                     # input arrays corrsponding to training
                                     # data points
                      name='Naive')
print("======================================================================")
evalutate_performance(y_pred1, y[1:], ntrain-1, name='Simple RNN, "do nothing"')
print("======================================================================")
evalutate_performance(y_pred2, y[1:], ntrain-1, name='Simple RNN, windowing')
print("======================================================================")
evalutate_performance(y_pred3, y[1:], ntrain-1, name='Simple RNN, windowing/
 ↪stateful')
```

```
Model Naive
Training MSE: 776.5437, MAE: 19.3285
Testing MSE:  708.6360,  MAE: 19.2256
======================================================================
Model Simple RNN, "do nothing"
Training MSE: 638.1165, MAE: 18.1304
```

```
Testing MSE:  605.6443,  MAE: 18.1051
======================================================================
Model Simple RNN, windowing
Training MSE: 605.3834, MAE: 17.5237
Testing MSE:  571.2137,  MAE: 17.2618
======================================================================
Model Simple RNN, windowing/stateful
Training MSE: 608.9876, MAE: 17.4253
Testing MSE:  594.9701,  MAE: 17.4775
```