TSSL Lab 4 - Recurrent Neural Networks

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

In [1]:
```
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

In [2]:
```
np.random.seed(42)
tf.random.set_seed(42)
print(np.__version__)
print(tf.__version__)
```

```
1.26.4
2.16.1
```

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.

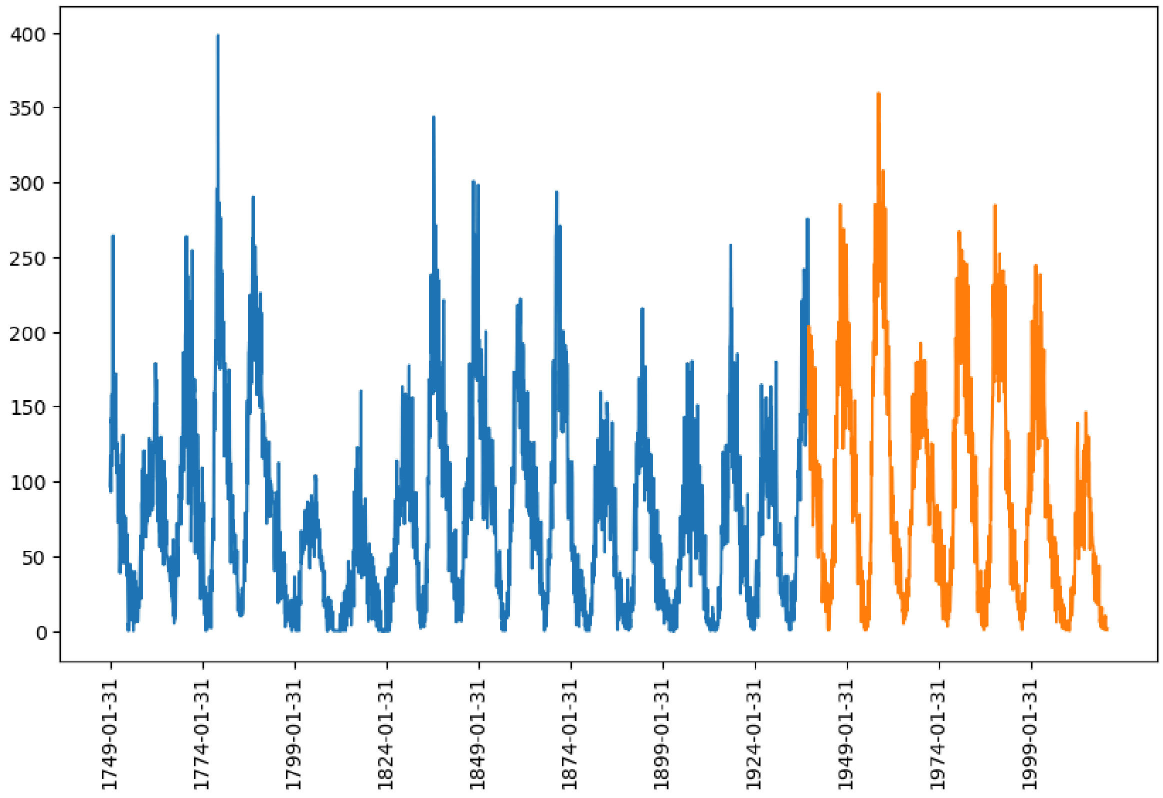
The data consists of the monthly mean total sunspot number, from 1749-01-01 to 2017-08-31.

In [3]:
```
# 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
```

In [4]:
```
plt.plot(dates[:ntrain], y[:ntrain])
plt.plot(dates[ntrain:], y[ntrain:])
plt.xticks(range(0, ndata, 300), dates[::300], rotation = 90);  # Show only one
```



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`.

In [5]:
```
MAX_VAL = 400
y = y/MAX_VAL
```

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.

In [6]:
```
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 traini
        The remaining elements, i.e. y_pred[split_time:] and y[split_time:] are
    """

    # 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
    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  Training MSE: {train_mse:.4f},   MAE: {train_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 $t = p + 1$).

In [7]:
```
# Store the predictions in an array of length ndata-1. Note that there is a shif
# between the prediction and the observation sequence, since there is no predict
# Specifically, y_pred_naive[t] is a prediction of y[t+1], so the first element
# second element of y, and so on. We will use the same "bookeeping convention" t
# 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 a
                      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.

In [8]:
```
# 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)
```

In [9]:
```
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 arra
                      name='AR')
```

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

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:
Q => The number of sequences => 1
M => The length of the sequences (number of time steps) => 3252
P => The number of features (at each time step), i.e. the dimension of each time series => 1

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:

- The meaning of setting `units = d` is dimensionality of the output space and takes positive integer d.

- The dimension of the output is (1, n, d).
- The answer would be (1, 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.

In [10]:
```python
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 t
model0.summary()
init_weights = model0.get_weights().copy()
```

```
Model: "sequential"
```

Layer (type)	Output Shape
simple_rnn (SimpleRNN)	(None, None, 10)
dense (Dense)	(None, None, 1)

```
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:
W => (P,d)=(1,10) => P x d => 1 x 10 => 10 total number of parameters
U => (d,d)=(10,10) => d x d => 10 x 10 => 100 total number of parameters
b => (d,) = (10,) => 10 total number of parameters

C => (d,1) = (10,1) => 10 x 1 => 10

c => (1,) => 1

In [11]:
```
model0.weights
```

Out[11]:
```
[<KerasVariable shape=(1, 10), dtype=float32, path=sequential/simple_rnn/simple
_rnn_cell/kernel>,
 <KerasVariable shape=(10, 10), dtype=float32, path=sequential/simple_rnn/simpl
e_rnn_cell/recurrent_kernel>,
 <KerasVariable shape=(10,), dtype=float32, path=sequential/simple_rnn/simple_r
nn_cell/bias>,
 <KerasVariable shape=(10, 1), dtype=float32, path=sequential/dense/kernel>,
 <KerasVariable shape=(1,), dtype=float32, path=sequential/dense/bias>]
```

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 $y_{1:n}$.

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

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

# Reshape the data
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))
```

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.

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

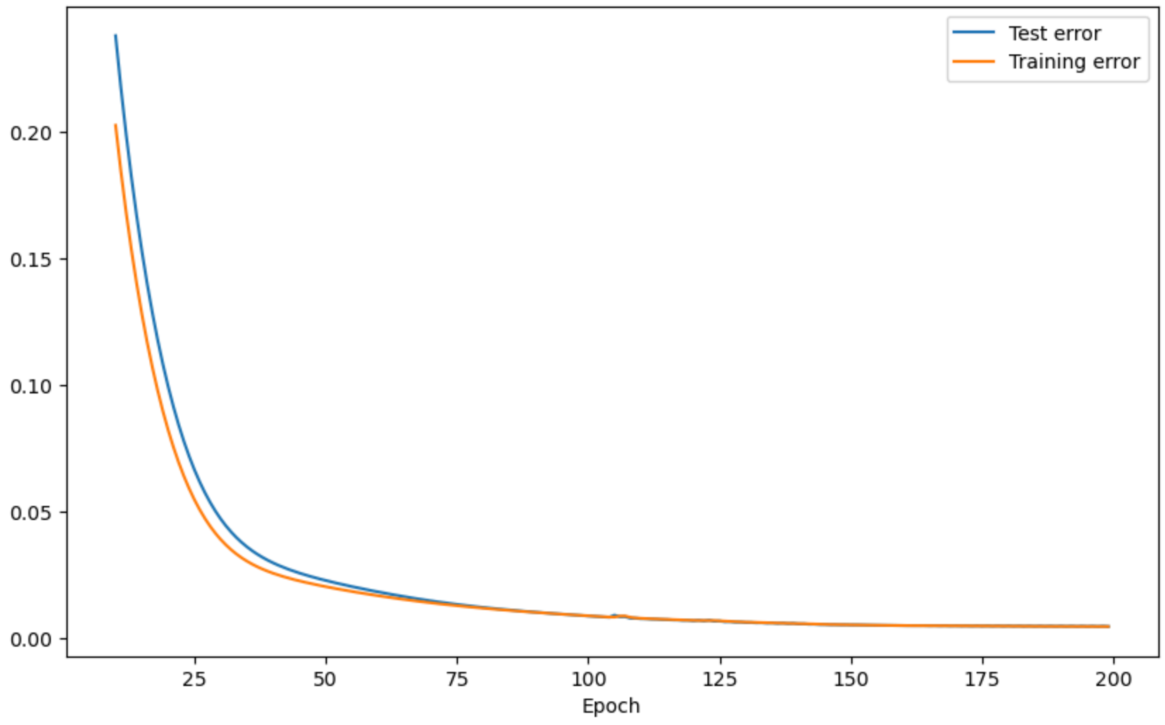
Q5: What should we set the *batch size* to, in order to compute the gradient based on the complete training data sequnce at each iteration? Complete the code below!

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.

In [14]:
```
model1.compile(loss='mse', optimizer='rmsprop', metrics=['mse'])
history = model1.fit(x_train, yt_train,
                     epochs = 200,
                     batch_size = 1,
                     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.

In [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)!

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

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

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 150ms/step
```

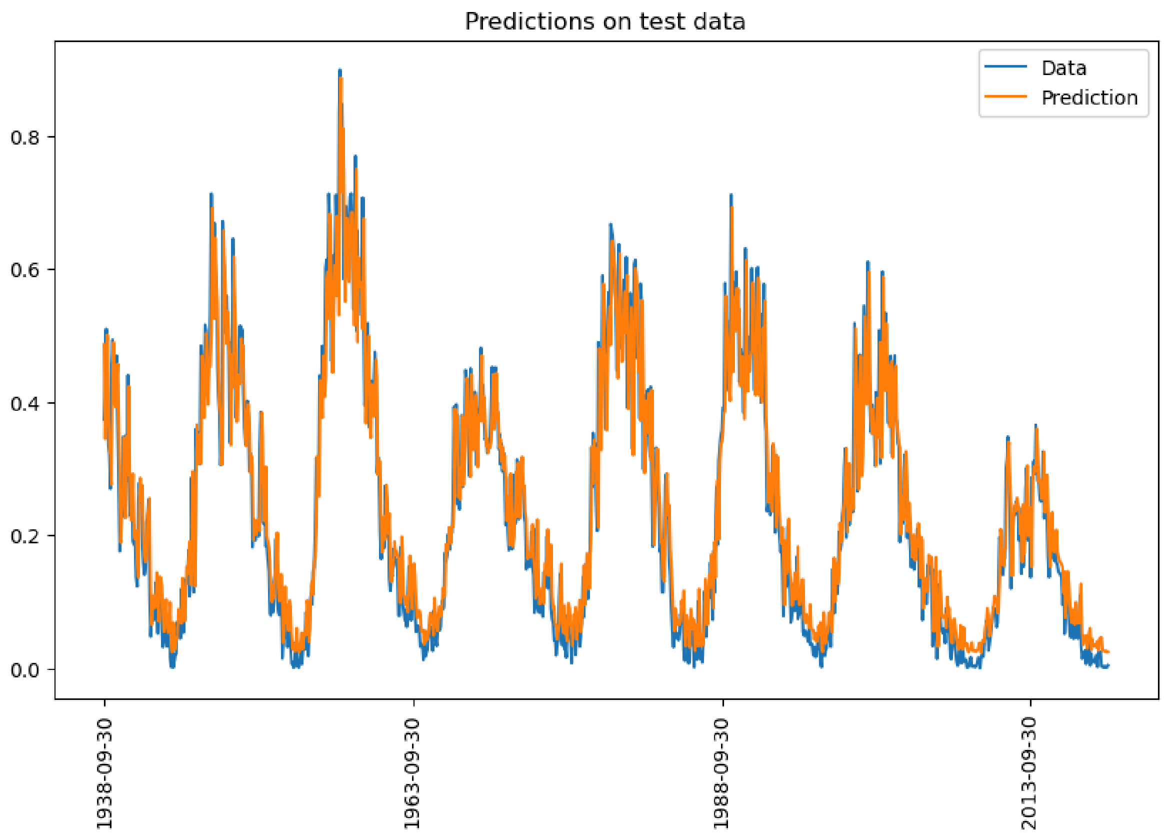
Using the prediction computed above we can plot them and evaluate the performance of the model in terms of MSE and MAE.

In [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:])
    plt.xticks(range(0, ntest, 300), dates[ntrain::300], rotation = 90);  # Show
    plt.legend(['Data','Prediction'])
    plt.title('Predictions on test data')
```

In [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: 777.8972,   MAE: 20.0962
  Testing MSE:  713.4347,   MAE: 19.9598
```



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.

In [19]:
```
def generator_train(window_size):
    while True:
        """The upper value is excluded in randint, so the maximum value that we
        Hence, the maximum end point of a window is ntrain-1, in agreement with
        when working with one-step-ahead prediction."""
        start_of_window = np.random.randint(0, ntrain - window_size)  # First ti
        end_of_window = start_of_window + window_size  # Last time index of wind
        yield x_train[:,start_of_window:end_of_window,:], yt_train[:,start_of_wi
```

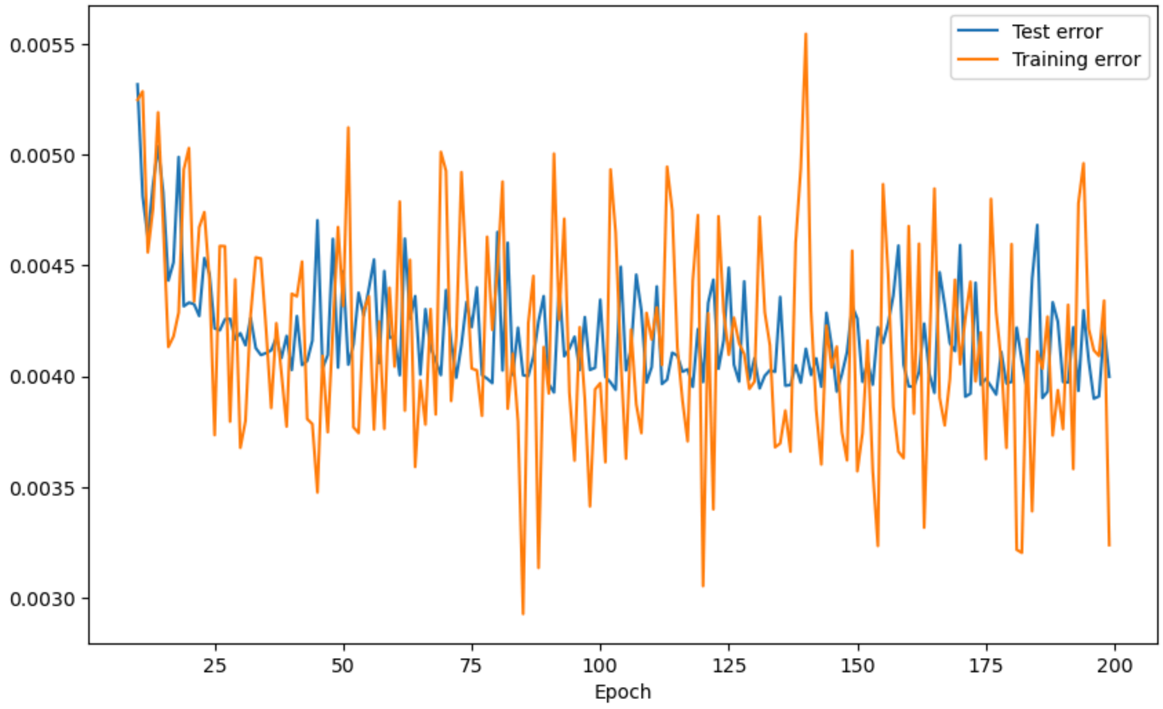
In [20]:
```
model2 = keras.models.clone_model(model0)  # This creates a new instance of the
model2.set_weights(init_weights)  # We set the initial weights to be the same fo
```

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.

In [21]:
```
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 = ntrain//window_size ,
           validation_data = (x_test, yt_test))
```

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

In [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 this error plot to the one we got for training Option 1, we can see that in Option 1, both the training error and test error decrease smoothly and converge fastly. However for option2, test error and training error has a oscillating pattern and they cannot converge and the variance is really high so there is instability. This difference is seen because of the the variation in data processing during traning. In the first option we use all data for each gradient update but in option 2 we use random windowing which means we use only a small randomly selected subset of the data within a chosen window for each gradient update.

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

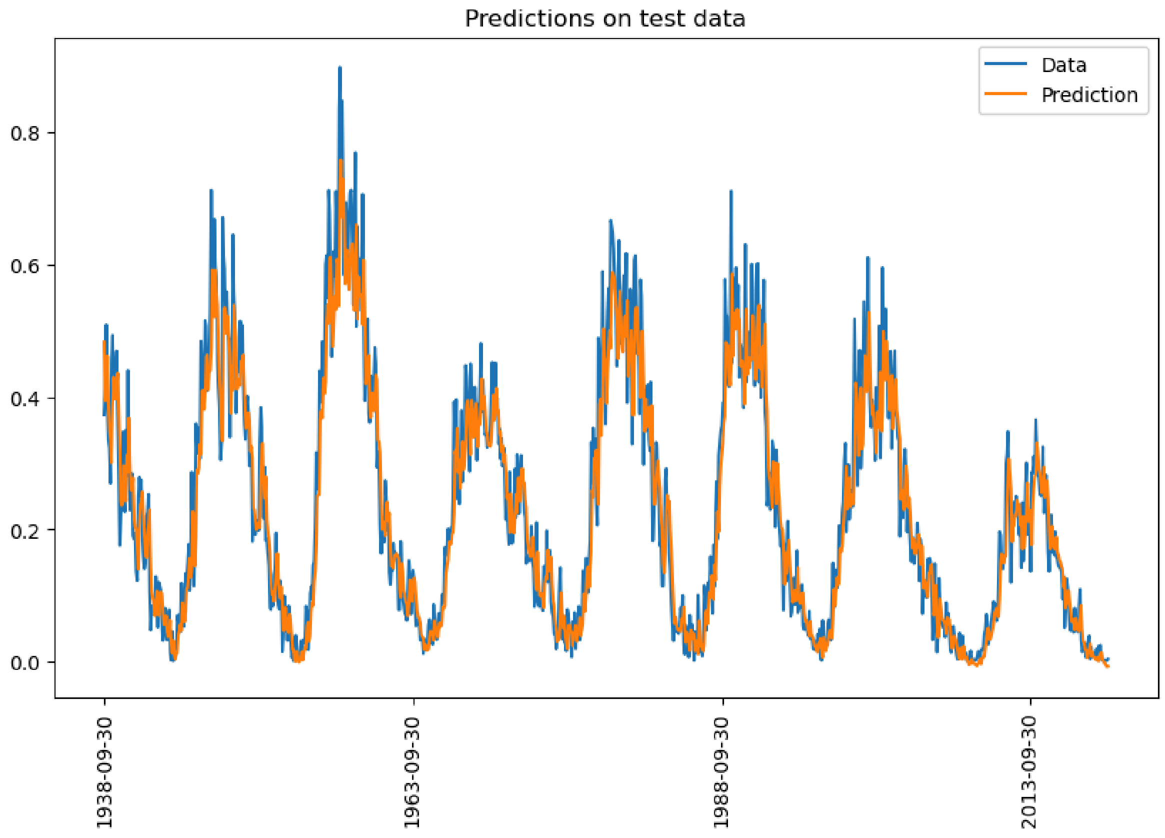
In [23]:
```
# Predict on all data using the final model.
# We predict using y_1,...,y_{n-1} as inputs, resulting in predictions of the va
y_pred2 = model2.predict(y[:ndata-1].reshape((1,ndata-1,1))).flatten() + y[:ndat
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 149ms/step
```

In [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: 647.8971,   MAE: 17.9317
  Testing MSE:  638.1297,   MAE: 18.0909
```



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.

In [25]:
```
# To enable stateful training, we need to create model where we set stateful=Tru
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
    # 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.

In [26]:
```
# Calculate number of windows
number_of_windows = int(ntrain/window_size)

# We might have to discard some samples
x_train_stateful = x_train[0,0:number_of_windows*window_size,0].reshape([-1,wind
yt_train_stateful = yt_train[0,0:number_of_windows*window_size,0].reshape([-1,wi
```

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).

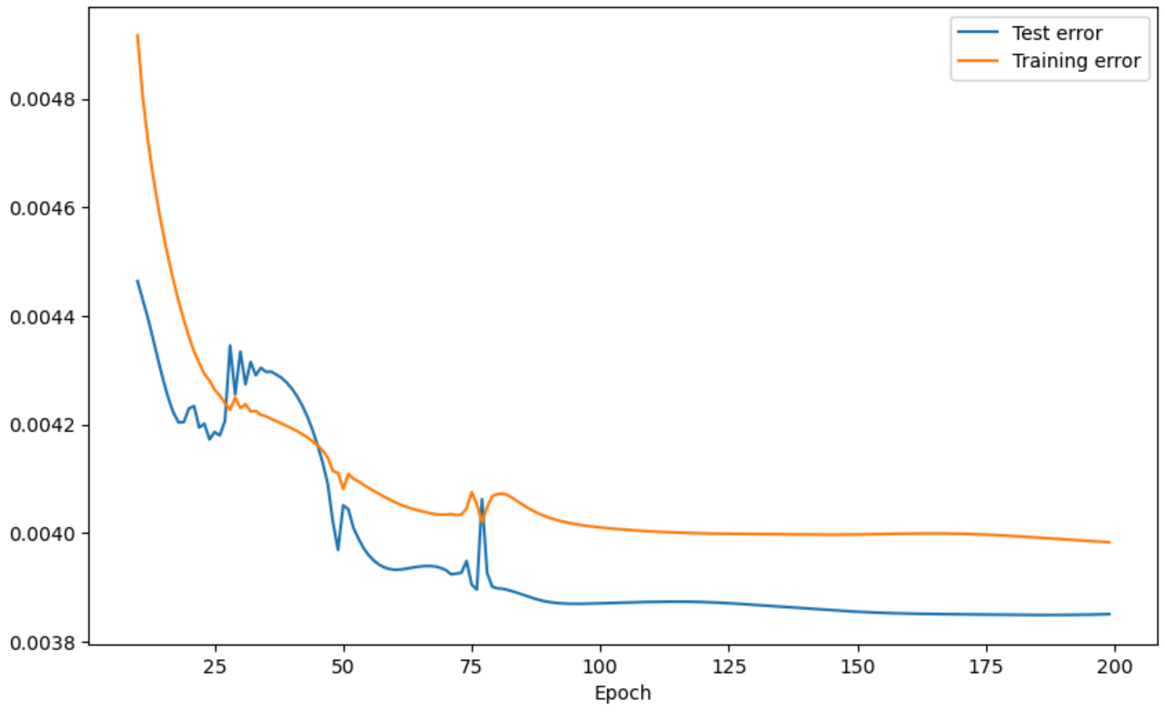
Because of this we need to save the data in our own structure between the epochs and save the results.

In [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)
    model3.layers[0].reset_states() # Resets the RNN state at the end of each ep
    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.

In [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: Comparing this error plot to the ones we got for training Options 1 and 2, we can observe that the test and training errors converge after a certain number of epochs, although not as quickly as in Option 1. However, before they stabilize, there are some fluctuations, but these are fewer and less severe compared to Option 2. The differences could be because the Options 3 uses a combination of both statefullness and windowing.

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

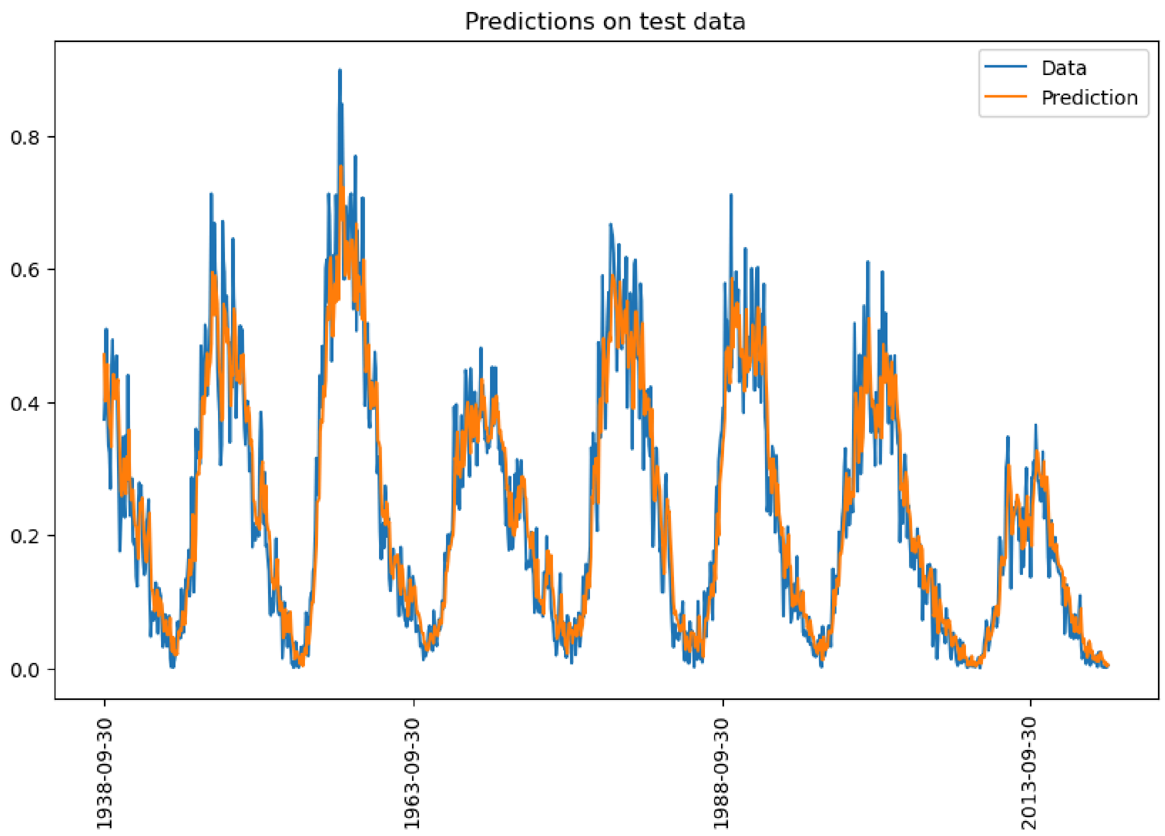
In [29]:
```
# Predict on all data using the final model.
# We predict using y_1,...,y_{n-1} as inputs, resulting in predictions of the va
y_pred3 = model3.predict(y[:ndata-1].reshape((1,ndata-1,1)))[0,:,:].flatten() +
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 149ms/step
```

In [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/stat
```

```
Model Simple RNN, windowing/stateful
  Training MSE: 629.0992,   MAE: 17.8367
  Testing MSE:  609.8030,   MAE: 17.9990
```



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:

- Model Naive
  Training MSE: 776.5437, MAE: 19.3285
  Testing MSE: 708.6360, MAE: 19.2256
- Model AR
  Training MSE: 603.8656, MAE: 17.3420
  Testing MSE: 590.3732, MAE: 17.6221
- Model Simple RNN, "do nothing"
  Training MSE: 777.8972, MAE: 20.0962
  Testing MSE: 713.4347, MAE: 19.9598
- Model Simple RNN, windowing
  Training MSE: 647.8971, MAE: 17.9317
  Testing MSE: 638.1297, MAE: 18.0909
- Model Simple RNN, windowing/stateful
  Training MSE: 629.0992, MAE: 17.8367
  Testing MSE: 609.8030, MAE: 17.9990

When we compared the the results of test error for all models, the model AR performed best because this model has the lowest MSE and MAE. We can say that for using RNN models, "windowing/statefulness" model show improvment over the "do nothing" and "windowing" models and also naive model because the testing error rate is very low. For this data we can say that the basic model like AR model is more efficient and and effective, making the use of complex models like RNNs unnecessary, however in this lab we also see the beneficial sides within RNN models, particularly in capturing sequential dependencies, which could be valuable in more complex scenarios.