Hydro powerplant constraints forecast - energy with LS

Introduction

In this experiment, we are building RNN models using GRU and LSTM cells, trying to beat MLP models that dont

For performance reasons, this is executed on google colab, to take advantage of their hardware accelaration ca

This is very largely inspired by the following tutorial: https://github.com/Hvass-Labs/TensorFlow-Tutorials/blob/

Imports

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import pandas as pd
import os
from sklearn.preprocessing import MinMaxScaler
from sklearn import metrics
from sklearn.metrics import mean_squared_error, mean_absolute_error
import statistics
import math


# from tf.keras.models import Sequential
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Input, Dense, GRU, Embedding, LSTM, Dropout
from tensorflow.python.keras.optimizers import RMSprop
from tensorflow.python.keras.callbacks import EarlyStopping, ModelCheckpoint, TensorBoard,
```

This was developed using google colab and package versions:

```
tf.__version__
```

```
'1.14.0'
```

```
tf.keras.__version__
```

```
'2.2.4-tf'
```

```
pd.__version__
```

```
'0.24.2'
```

Load Data

We load our clean data set.

```
from google.colab import files
uploaded = files.upload()
```

```
Sélect. fichiers  clean_dataframe.csv
```

- **clean_dataframe.csv**(application/vnd.ms-excel) - 386595 bytes, last modified: 16/09/2019 - 100% done

```
Saving clean_dataframe.csv to clean_dataframe.csv
```

```
dateparse = lambda x: pd.datetime.strptime(x, '%Y-%m-%d')

df = pd.read_csv("clean_dataframe.csv", parse_dates=['Date'], date_parser=dateparse, index
# rename date column
df.rename(columns={ df.columns[0]: "Date"}, inplace=True)
df.index = df["Date"]
df.rename(columns={ "Variante Prio": "Max prod"}, inplace=True)
```

The data imported has been previously cleaned.

These are the top rows of the data-set.

```
df.head()
```

	Date	Min prod	Inflow lake 1 [m3]	Inflow lake 2 [m3]	Inflow lake 3 [m3]	Inflow lake 4 [m3]	Vol lake 1 [%]	Max lake 1 [1000m3]	Availability plant 1 [%]
Date									
2014-04-01	2014-04-01	0.0	31.0	4.0	129.0	107.0	0.16467	30000.0	1.0
2014-04-02	2014-04-02	150.0	0.0	-14.0	148.0	116.0	0.15557	30000.0	1.0
2014-04-03	2014-04-03	150.0	10.0	6.0	132.0	118.0	0.14765	30000.0	1.0
2014-04-04	2014-04-04	150.0	19.0	6.0	150.0	118.0	0.13716	30000.0	1.0
2014-04-05	2014-04-05	180.0	41.0	15.0	148.0	124.0	0.13091	30000.0	1.0

Target Data for Prediction

We will try and predict the future maximum production, contained in column "Max prod"

```
target_names = ['Max prod']
```

```
df_targets = df[target_names]
df_targets.tail()
```

Date	Max prod
2019-06-26	314.759725
2019-06-27	326.450801
2019-06-28	283.883295
2019-06-29	253.906178
2019-06-30	253.906178

NumPy Arrays

We now convert the Pandas data-frames to NumPy arrays that can be input to the neural network.

These are the input-signals:

```
regressors = ['Inflow lake 1 [m3]', \
           'Inflow lake 2 [m3]', 'Inflow lake 3 [m3]', 'Inflow lake 4 [m3]', \
           'Vol lake 1 [%]', 'Max lake 1 [1000m3]', 'Availability plant 1 [%]', \
           'Availability plant 2 [%]', 'Availability plant 3 [%]', \
           'Availability plant 4 [%]']

x_data = df[regressors].values
```

```
print(type(x_data))
print("Shape:", x_data.shape)
```

```
<class 'numpy.ndarray'>
Shape: (1917, 10)
```

These are the output-signals (or target-signals):

```
y_data = df_targets.values
```

```
print(type(y_data))
print("Shape:", y_data.shape)
```

```
<class 'numpy.ndarray'>
Shape: (1917, 1)
```

This is the number of observations (aka. data-points or samples) in the data-set:

```
num_data = len(x_data)
num_data
```

```
1917
```

This is the fraction of the data-set that will be used for the training-set:

```
train_split = 0.9
```

This is the number of observations in the training-set:

```
num_train = int(train_split * num_data)
num_train
```

```
1725
```

This is the number of observations in the test-set:

```
num_test = num_data - num_train
num_test
```

```
192
```

These are the input-signals for the training- and test-sets:

```
x_train = x_data[0:num_train]
x_test = x_data[num_train:]
len(x_train) + len(x_test)
```

```
1917
```

These are the output-signals for the training- and test-sets:

```
y_train = y_data[0:num_train]
y_test = y_data[num_train:]
len(y_train) + len(y_test)
```

```
1917
```

This is the number of input-signals:

```
num_x_signals = x_data.shape[1]
num_x_signals
```

```
10
```

This is the number of output-signals:

```
num_y_signals = y_data.shape[1]
num_y_signals
```

```
1
```

Scaled Data

The data-set contains a wide range of values:

```
print("Min:", np.min(x_train))
print("Max:", np.max(x_train))
```

```
Min: -482.0
Max: 30000.0
```

The neural network works best on values roughly between -1 and 1, so we need to scale the data before it is beir `learn` for this.

We first create a scaler-object for the input-signals.

```
x_scaler = MinMaxScaler()
```

We then detect the range of values from the training-data and scale the training-data.

```
x_train_scaled = x_scaler.fit_transform(x_train)
```

Apart from a small rounding-error, the data has been scaled to be between 0 and 1.

```
print("Min:", np.min(x_train_scaled))
print("Max:", np.max(x_train_scaled))
```

```
Min: 0.0
Max: 1.0000000000000002
```

We use the same scaler-object for the input-signals in the test-set.

```
x_test_scaled = x_scaler.transform(x_test)
```

The target-data comes from the same data-set as the input-signals. But the target-data could be from a differen separate scaler-object for the target-data.

```
y_scaler = MinMaxScaler()
y_train_scaled = y_scaler.fit_transform(y_train)
y_test_scaled = y_scaler.transform(y_test)
```

Data Generator

The data-set has now been prepared as 2-dimensional numpy arrays. The training-data has 1725 observations, c

These are the array-shapes of the input and output data:

```
print(x_train_scaled.shape)
print(y_train_scaled.shape)
```

```
(1725, 10)
(1725, 1)
```

Instead of training the Recurrent Neural Network on the complete sequences 1725 observations, we will use the sequences picked at random from the training-data.

```
def batch_generator(batch_size, sequence_length):
    """
    Generator function for creating random batches of training-data.
    """

    # Infinite loop.
    while True:
        # Allocate a new array for the batch of input-signals.
        x_shape = (batch_size, sequence_length, num_x_signals)
        x_batch = np.zeros(shape=x_shape, dtype=np.float16)

        # Allocate a new array for the batch of output-signals.
        y_shape = (batch_size, sequence_length, num_y_signals)
        y_batch = np.zeros(shape=y_shape, dtype=np.float16)

        # Fill the batch with random sequences of data.
        for i in range(batch_size):
            # Get a random start-index.
            # This points somewhere into the training-data.
            idx = np.random.randint(num_train - sequence_length)

            # Copy the sequences of data starting at this index.
            x_batch[i] = x_train_scaled[idx:idx+sequence_length]
            y_batch[i] = y_train_scaled[idx:idx+sequence_length]

        yield (x_batch, y_batch)
```

We will use a large batch-size so as to keep the GPU near 100% work-load. You may have to adjust this number
`sequence_length` below.

```
batch_size = 256
```

We will use a sequence-length of 364, which means that each random sequence contains observations for 52 w

```
sequence_length = 1 * 7 * 52
sequence_length
```

```
364
```

We then create the batch-generator.

```
generator = batch_generator(batch_size=batch_size,
                            sequence_length=sequence_length)
```

We can then test the batch-generator to see if it works.

```
x_batch, y_batch = next(generator)
```

This gives us a random batch of 256 sequences, each sequence having 1344 observations, and each observatio

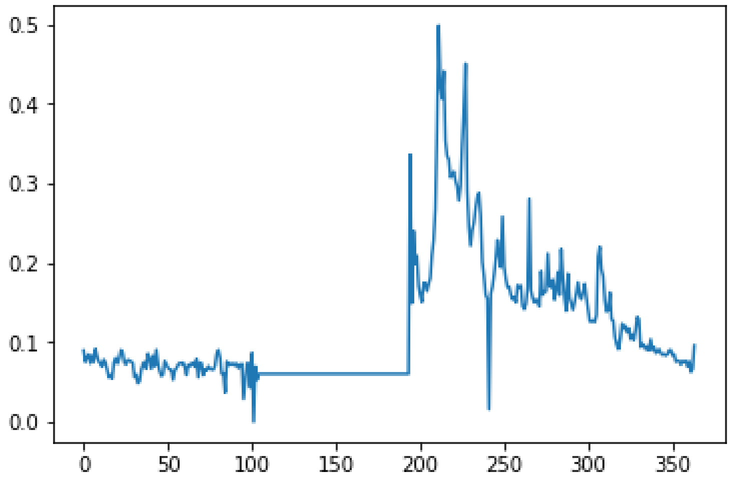
```
print(x_batch.shape)
print(y_batch.shape)
```

```
(256, 364, 10)
(256, 364, 1)
```

We can plot one of the 10 input-signals as an example.

```
batch = 0   # First sequence in the batch.
signal = 0  # First signal from the 20 input-signals.
seq = x_batch[batch, :, signal]
plt.plot(seq)
```

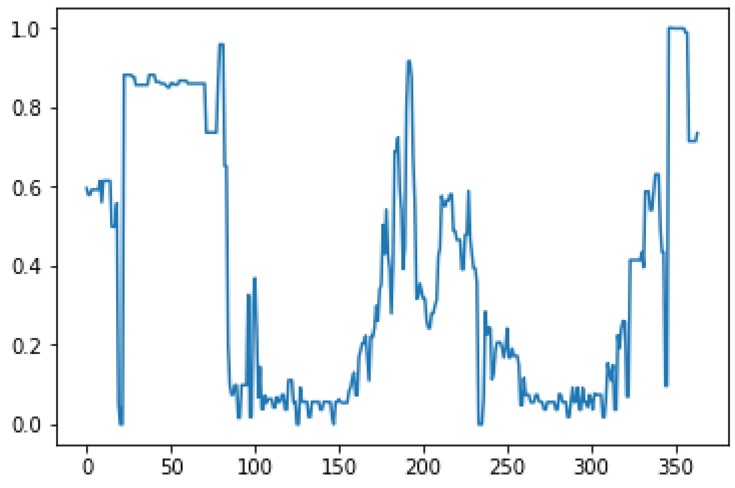
```
[<matplotlib.lines.Line2D at 0x7fa753789a20>]
```



We can also plot one of the output-signals that we want the model to learn how to predict given all those 10 inpu

```
seq = y_batch[batch, :, signal]
plt.plot(seq)
```

```
[<matplotlib.lines.Line2D at 0x7fa750f1e390>]
```



Validation Set

The neural network trains quickly so we can easily run many training epochs. But then there is a risk of overfittin generalize well to unseen data. We will therefore monitor the model's performance on the test-set after each epc performance is improved on the test-set.

The batch-generator randomly selects a batch of short sequences from the training-data and uses that during tra through the entire sequence from the test-set and measure the prediction accuracy on that entire sequence.

```
validation_data = (np.expand_dims(x_test_scaled, axis=0),
                   np.expand_dims(y_test_scaled, axis=0))
```

Create the Recurrent Neural Network

We are now ready to create the Recurrent Neural Network (RNN). We will use the Keras API for this because of it

```
model = Sequential()
```

We can now add 2 Gated Recurrent Unit (GRU) layers to the network. This will have 22 (15+7) outputs for each ti

We add drop out for normalization.

Note that because this is the first layer in the model, Keras needs to know the shape of its input, which is a batch None), where each observation has a number of input-signals (`num_x_signals`).

```
model.add(GRU(units=10,
              return_sequences=True,
              input_shape=(None, num_x_signals,)))
model.add(Dropout(0.1))
model.add(GRU(units=7,
              return_sequences=True))
model.add(Dropout(0.1))
```

The GRU outputs a batch of sequences of 7 values. We want to predict 1 output-signal, so we add a fully-connec only 1 value.

We can use a linear activation function on the output. This allows for the output to take on arbitrary values. It mi network architecture, but for more complicated network architectures e.g. with more layers, it might be necessa avoid `NaN` values during training.

```
from tensorflow.python.keras.initializers import RandomUniform

# Maybe use lower init-ranges.
init = RandomUniform(minval=-0.05, maxval=0.05)

model.add(Dense(num_y_signals,
                activation='linear',
                kernel_initializer=init))
```

Loss Function

We will use Mean Squared Error (MSE) as the loss-function that will be minimized. This measures how closely th

However, at the beginning of a sequence, the model has only seen input-signals for a few time-steps, so its gene value for the early time-steps may cause the model to distort its later output. We therefore give the model a "wa accuracy in the loss-function, in hope of improving the accuracy for later time-steps.

```
warmup_steps = 10


def loss_mse_warmup(y_true, y_pred):
    """
    Calculate the Mean Squared Error between y_true and y_pred,
    but ignore the beginning "warmup" part of the sequences.

    y_true is the desired output.
    y_pred is the model's output.
    """

    # The shape of both input tensors are:
    # [batch_size, sequence_length, num_y_signals].

    # Ignore the "warmup" parts of the sequences
    # by taking slices of the tensors.
    y_true_slice = y_true[:, warmup_steps:, :]
    y_pred_slice = y_pred[:, warmup_steps:, :]

    # These sliced tensors both have this shape:
    # [batch_size, sequence_length - warmup_steps, num_y_signals]

    # Calculate the MSE loss for each value in these tensors.
    # This outputs a 3-rank tensor of the same shape.
    loss = tf.losses.mean_squared_error(labels=y_true_slice,
                                        predictions=y_pred_slice)

    # Keras may reduce this across the first axis (the batch)
    # but the semantics are unclear, so to be sure we use
    # the loss across the entire tensor, we reduce it to a
    # single scalar with the mean function.
    loss_mean = tf.reduce_mean(loss)

    return loss_mean
```

Compile Model

This is the optimizer and the beginning learning-rate that we will use.

```
optimizer = RMSprop(lr=1e-3)
```

We then compile the Keras model so it is ready for training.

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

This is a very small model with only two layers. The output shape of `(None, None, 3)` means that the model w
sequences, each of which has an arbitrary number of observations, and each observation has 3 signals. This co

```
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
gru_8 (GRU)                  (None, None, 10)          630       
_________________________________________________________________
dropout_8 (Dropout)          (None, None, 10)          0         
_________________________________________________________________
gru_9 (GRU)                  (None, None, 7)           378       
_________________________________________________________________
dropout_9 (Dropout)          (None, None, 7)           0         
_________________________________________________________________
dense_3 (Dense)              (None, None, 1)           8         
=================================================================
Total params: 1,016
Trainable params: 1,016
Non-trainable params: 0
_________________________________________________________________
```

Callback Functions

During training we want to save checkpoints and log the progress to TensorBoard so we create the appropriate c

This is the callback for writing checkpoints during training.

```
path_checkpoint = 'PowerPredictionLSTM.keras'
callback_checkpoint = ModelCheckpoint(filepath=path_checkpoint,
                                      monitor='val_loss',
                                      verbose=1,
                                      save_weights_only=True,
                                      save_best_only=True)
```

This is the callback for stopping the optimization when performance worsens on the validation-set.

```
callback_early_stopping = EarlyStopping(monitor='val_loss',
                                        patience=5, verbose=1)
```

This is the callback for writing the TensorBoard log during training.

```
callback_tensorboard = TensorBoard(log_dir='./PPPredLSTM_logs/',
                                   histogram_freq=0,
                                   write_graph=False)
```

This callback reduces the learning-rate for the optimizer if the validation-loss has not improved since the last ep
will be reduced by multiplying it with the given factor. We set a start learning-rate of 1e-3 above, so multiplying it
the learning-rate to go any lower than this.

```
callback_reduce_lr = ReduceLROnPlateau(monitor='val_loss',
                                       factor=0.1,
                                       min_lr=1e-4,
                                       patience=0,
                                       verbose=1)
```

```
callbacks = [callback_early_stopping,
             callback_checkpoint,
             callback_tensorboard,
             callback_reduce_lr]
```

Train the Recurrent Neural Network

We can now train the neural network.

Note that a single "epoch" does not correspond to a single processing of the training-set, because of how the ba
the training-set. Instead we have selected `steps_per_epoch` so that one "epoch" is processed in a few minutes.

Also note that the loss sometimes becomes `NaN` (not-a-number). This is often resolved by restarting and running
your neural network architecture, learning-rate, batch-size, sequence-length, etc. in which case you may have to r

```
%%time
model.fit_generator(generator=generator,
                    epochs=20,
                    steps_per_epoch=100,
                    validation_data=validation_data,
                    callbacks=callbacks)
```

```
Epoch 1/20
 99/100 [============================>.] - ETA: 1s - loss: 0.0882
Epoch 00001: val_loss improved from inf to 0.03497, saving model to PowerPredictionLS
100/100 [==============================] - 121s 1s/step - loss: 0.0877 - val_loss: 0.
Epoch 2/20
 99/100 [============================>.] - ETA: 1s - loss: 0.0364
Epoch 00002: val_loss improved from 0.03497 to 0.02577, saving model to PowerPredicti
100/100 [==============================] - 117s 1s/step - loss: 0.0363 - val_loss: 0.
Epoch 3/20
 99/100 [============================>.] - ETA: 1s - loss: 0.0319
Epoch 00003: val_loss improved from 0.02577 to 0.02313, saving model to PowerPredicti
100/100 [==============================] - 117s 1s/step - loss: 0.0319 - val_loss: 0.
Epoch 4/20
 99/100 [============================>.] - ETA: 1s - loss: 0.0281
Epoch 00004: val_loss improved from 0.02313 to 0.02098, saving model to PowerPredicti
100/100 [==============================] - 116s 1s/step - loss: 0.0281 - val_loss: 0.
Epoch 5/20
 99/100 [============================>.] - ETA: 1s - loss: 0.0251
Epoch 00005: val_loss improved from 0.02098 to 0.01958, saving model to PowerPredicti
100/100 [==============================] - 119s 1s/step - loss: 0.0251 - val_loss: 0.
Epoch 6/20
 99/100 [============================>.] - ETA: 1s - loss: 0.0235
Epoch 00006: val_loss did not improve from 0.01958

Epoch 00006: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.
100/100 [==============================] - 118s 1s/step - loss: 0.0235 - val_loss: 0.
Epoch 7/20
 99/100 [============================>.] - ETA: 1s - loss: 0.0222
Epoch 00007: val_loss did not improve from 0.01958

Epoch 00007: ReduceLROnPlateau reducing learning rate to 0.0001.
100/100 [==============================] - 119s 1s/step - loss: 0.0222 - val_loss: 0.
Epoch 8/20
 99/100 [============================>.] - ETA: 1s - loss: 0.0218
Epoch 00008: val_loss did not improve from 0.01958
100/100 [==============================] - 118s 1s/step - loss: 0.0218 - val_loss: 0.
Epoch 9/20
 99/100 [============================>.] - ETA: 1s - loss: 0.0215
Epoch 00009: val_loss did not improve from 0.01958
100/100 [==============================] - 118s 1s/step - loss: 0.0215 - val_loss: 0.
Epoch 10/20
 99/100 [============================>.] - ETA: 1s - loss: 0.0211
Epoch 00010: val_loss did not improve from 0.01958
100/100 [==============================] - 120s 1s/step - loss: 0.0211 - val_loss: 0.
Epoch 00010: early stopping
CPU times: user 26min 47s, sys: 1min 36s, total: 28min 23s
Wall time: 19min 41s
<tensorflow.python.keras.callbacks.History at 0x7fa6fc970cc0>
```

Load Checkpoint

Because we use early-stopping when training the model, it is possible that the model's performance has worsen
was stopped. We therefore reload the last saved checkpoint, which should have the best performance on the tes

Double-cliquez (ou appuyez sur Entrée) pour modifier

```
try:
    model.load_weights(path_checkpoint)
except Exception as error:
    print("Error trying to load checkpoint.")
    print(error)
```

Performance on Test-Set

We can now evaluate the model's performance on the test-set. This function expects a batch of data, but we will just expand the array-dimensionality to create a batch with that one sequence.

```
result = model.evaluate(x=np.expand_dims(x_test_scaled, axis=0),
                        y=np.expand_dims(y_test_scaled, axis=0))
```

```
1/1 [==============================] - 0s 138ms/sample - loss: 0.0196
```

```
print("loss (test-set):", result)
```

```
loss (test-set): 0.01958109624683857
```

```
# If you have several metrics you can use this instead.
if False:
    for res, metric in zip(result, model.metrics_names):
        print("{0}: {1:.3e}".format(metric, res))
```

Generate Predictions

This helper-function plots the predicted and true output-signals.

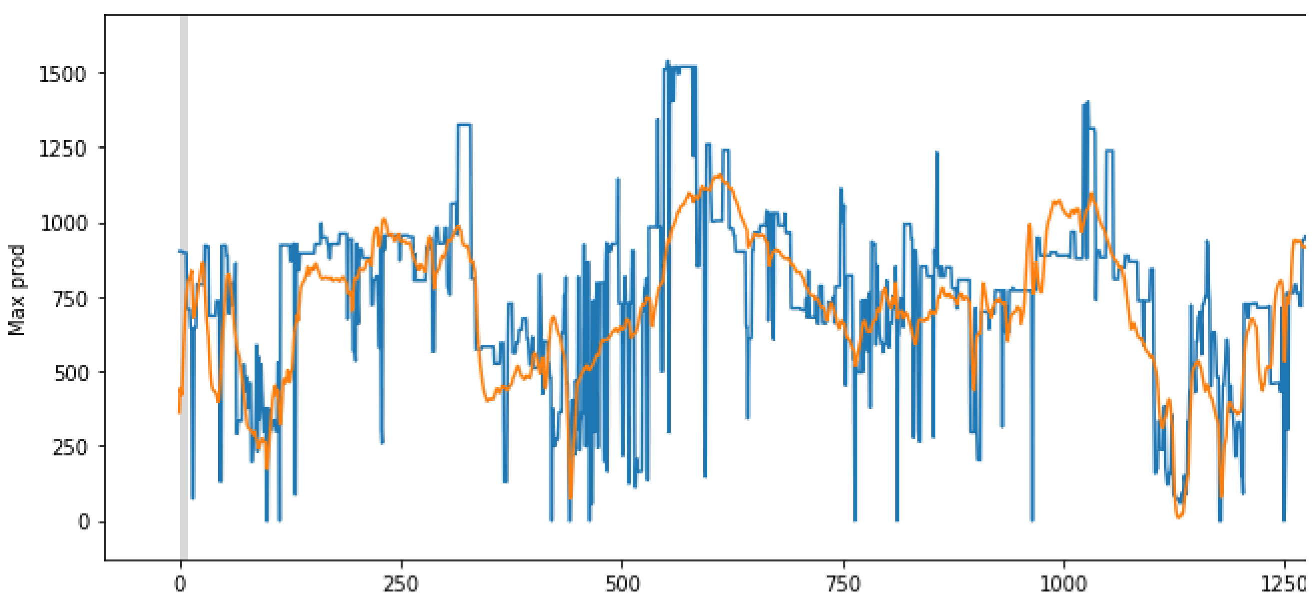
```
def plot_comparison(start_idx, length=100, train=True):
    """
    Plot the predicted and true output-signals.

    :param start_idx: Start-index for the time-series.
    :param length: Sequence-length to process and plot.
    :param train: Boolean whether to use training- or test-set.
    """

    if train:
        # Use training-data.
        x = x_train_scaled
        y_true = y_train
    else:
        # Use test-data.
        x = x_test_scaled
        y_true = y_test

    # End-index for the sequences.
    end_idx = start_idx + length

    # Select the sequences from the given start-index and
    # of the given length.
    x = x[start_idx:end_idx]
    y_true = y_true[start_idx:end_idx]

    # Input-signals for the model.
    x = np.expand_dims(x, axis=0)

    # Use the model to predict the output-signals.
    y_pred = model.predict(x)

    # The output of the model is between 0 and 1.
    # Do an inverse map to get it back to the scale
    # of the original data-set.
    y_pred_rescaled = y_scaler.inverse_transform(y_pred[0])

    # For each output-signal.
    for signal in range(len(target_names)):
        # Get the output-signal predicted by the model.
        signal_pred = y_pred_rescaled[:, signal]

        # Get the true output-signal from the data-set.
        signal_true = y_true[:, signal]

        # Make the plotting-canvas bigger.
        plt.figure(figsize=(15,5))

        # Plot and compare the two signals.
        plt.plot(signal_true, label='true')
        plt.plot(signal_pred, label='pred')

        # Plot grey box for warmup-period.
        p = plt.axvspan(0, warmup_steps, facecolor='black', alpha=0.15)

        # Plot labels etc.
        plt.ylabel(target_names[signal])
        plt.legend()
        plt.show()

        #compute errors
        RMSE = round(math.sqrt(metrics.mean_squared_error(signal_true,signal_pred)),2)
        MAE = round(metrics.mean_absolute_error(signal_true,signal_pred),2)
        R2 = round(metrics.r2_score(signal_true,signal_pred)*100, 2)
        print("RMSE: ",RMSE)
        print("MAE: ",MAE)
        print("R2: ",R2)
```

We can now plot an example of predicted output-signals. It is important to understand what these plots show, as might think.

The prediction is not very accurate for the first time-steps because the model has seen very little input-data at th output data for each time-step of the input-data, so when the model has only run for a few time-steps, it knows v cannot make an accurate prediction. The model needs to "warm up" by processing perhaps about 10 time-steps

That is why we ignore this "warmup-period" of 10 time-steps when calculating the mean-squared-error in the los
box in these plots.

```
plot_comparison(start_idx=0, length=10000, train=True)
```

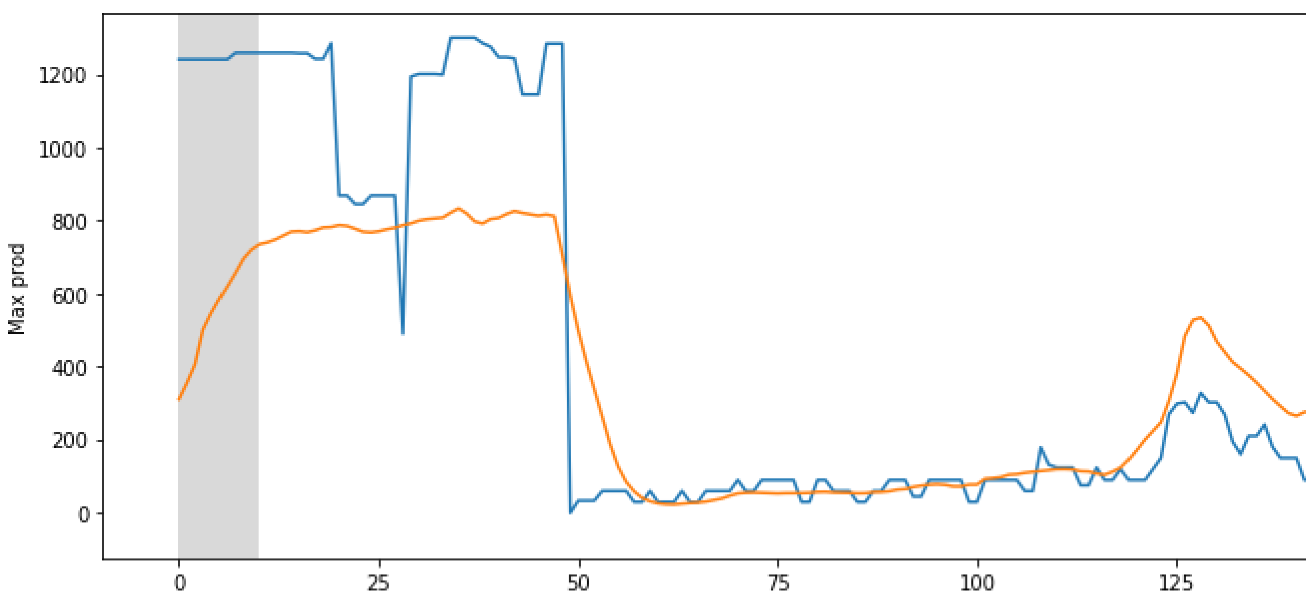


```
RMSE:  247.51
MAE:  188.56
R2:  54.86
```

Example from Test-Set

Now consider an example from the test-set. The model has not seen this data during training.

```
plot_comparison(start_idx=0, length=192, train=False)
```



```
RMSE:  274.09
MAE:  184.51
R2:  64.81
```

Conclusion

This experiment showed how to use a Recurrent Neural Network to predict a given time series from a number of