model-training-documentation

August 19, 2021

1 1. Loading and preparing data

Data sources

Training data is constructed from two data sources: energy and weather data.

Energy data source

Yearly reports of an energy provider from Berlin (Berlin Stromnetz) which are available as Open Data. Yearly csv-reports are available for years 2011-2015 as well as 2018. They provide information on total energy consumption of small private and industrial clients measured in kW in 15-minute intervals.

Weather data source

To gather temperature data associated with the energy consumption data points, I use Meteostat API for open weather and climate data.

Dataset preparation

Dataset preparation consists of - loading each yearly historical data csv and formatting it correctly - adding temperature features to each timestep - connecting the full dataset

Time-series dataset split into train, validation and test sets

As data is used in time series forecasting models, train-test-validation split is not randomized. Instead, slices are made in a way that the time series structure is preserved.

[1]:
```
import pandas as pd
import datetime
import os
from meteostat import Point, Hourly
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
%matplotlib inline
import IPython
import IPython.display
```

[3]:
```
def read_energy_csv(path):
    """Reading yearly consumption csv, naming columns and formatting the data"""
    df = pd.read_csv(path, skiprows=8, sep=";")
```

```
    df.columns = ["Datum", "Zeit", "kW"]

    if df["kW"].dtype == "int64":
        df["kW"] = df["kW"].astype(float)
    elif df["kW"].dtype == "object":
        df["kW"] = df["kW"].str.replace('\.', '', regex=True)
        df["kW"] = df["kW"].str.replace(',', '.', regex=True).astype(float)

    return df

def add_time_features(df):
    """Computing  time features in correct format
    'Date' is datetime format for ordering time series.
    'Daytime' is a float version of daytime used as an input feature in␣
 ↪forecasting,
        i.e. 00:30 is represented by 0.3.
    """
    time_df = df.copy()

    time_df["Date"] = time_df["Datum"] + " " + time_df["Zeit"].str.slice(0,5)
    time_df["Date"] = pd.to_datetime(time_df["Date"], format='%d.%m.%Y %H:%M')

    time_df['Daytime'] = time_df['Date'].dt.time.astype(str).str.slice(0,5).str.
 ↪replace(":",".").astype(float)
    time_df['Hour'] = time_df["Date"].dt.round("H")

    time_df = time_df.drop(["Datum","Zeit"], axis=1)

    return time_df

def add_temperature_features(time_df, geopoint):
    """Connect to the Meteostat API, get hourly temperature and add it to each␣
 ↪timestep"""
    temperature_df = time_df.copy()

    start = temperature_df["Hour"].iloc[0].to_pydatetime()
    end = temperature_df["Hour"].iloc[-1].to_pydatetime()

    hourly = Hourly(geopoint, start, end)
    hourly = hourly.fetch()
    hourly = hourly.reset_index()
    hourly = hourly[["time", "temp"]]
    hourly.columns = ["Hour","Temperature"]

    temperature_df = temperature_df.merge(hourly, on='Hour', how='left')

    return temperature_df
```

```
def get_yearly_df(path, geopoint):
    """Wrapper function to process one yearly consumption csv"""
    df = read_energy_csv(path)
    time_df = add_time_features(df)
    temperature_df = add_temperature_features(time_df, geopoint)

    return temperature_df

def load_dataset(files, geopoint):
    """Load all historical data, sort it as time series and convert kW to mW"""
    dataset = pd.DataFrame()
    for file in files:
        year_df = get_yearly_df(file, geopoint)
        dataset = pd.concat([dataset,year_df])

    dataset = dataset.sort_values("Date")
    dataset["mW"] = dataset["kW"]/1000

    dataset = dataset[["Date", "mW", "Temperature", "Daytime"]]

    return dataset

def df_split(df):
    """Train-validation-test on time-series data (non-random slicing)"""
    column_indices = {name: i for i, name in enumerate(df.columns)}

    n = len(df)
    train_df = df[0:int(n*0.7)]
    val_df = df[int(n*0.7):int(n*0.9)]
    test_df = df[int(n*0.9):]

    return column_indices, train_df, val_df, test_df
```

[9]:
```
files = os.listdir("data")
files = ["data/"+file for file in files]

munich = Point(48.1351, 11.5820, 519)
df = load_dataset(files, munich)
```

Saving historical data for later use by simulator. Only using data points before 2016 because of the subsequent gap of few years in the data

[5]:
```
early = (df['Date'] < '2018-01-01')
historical = df.loc[early]
historical["ts"] = pd.to_datetime(historical["Date"], unit="ns").astype(np.
 ↪int64) // 10**9
```

```
historical.to_csv("historical.csv", index=False)
```

```
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

Dividing the dataset into train, test and validation subsets.

[6]:
```
def df_split(df):
    """Slicing dataset into train-val-test split (70-20-10%) chronologically"""
    column_indices = {name: i for i, name in enumerate(df.columns)}

    n = len(df)
    train_df = df[0:int(n*0.7)]
    val_df = df[int(n*0.7):int(n*0.9)]
    test_df = df[int(n*0.9):]

    return column_indices, train_df, val_df, test_df
```

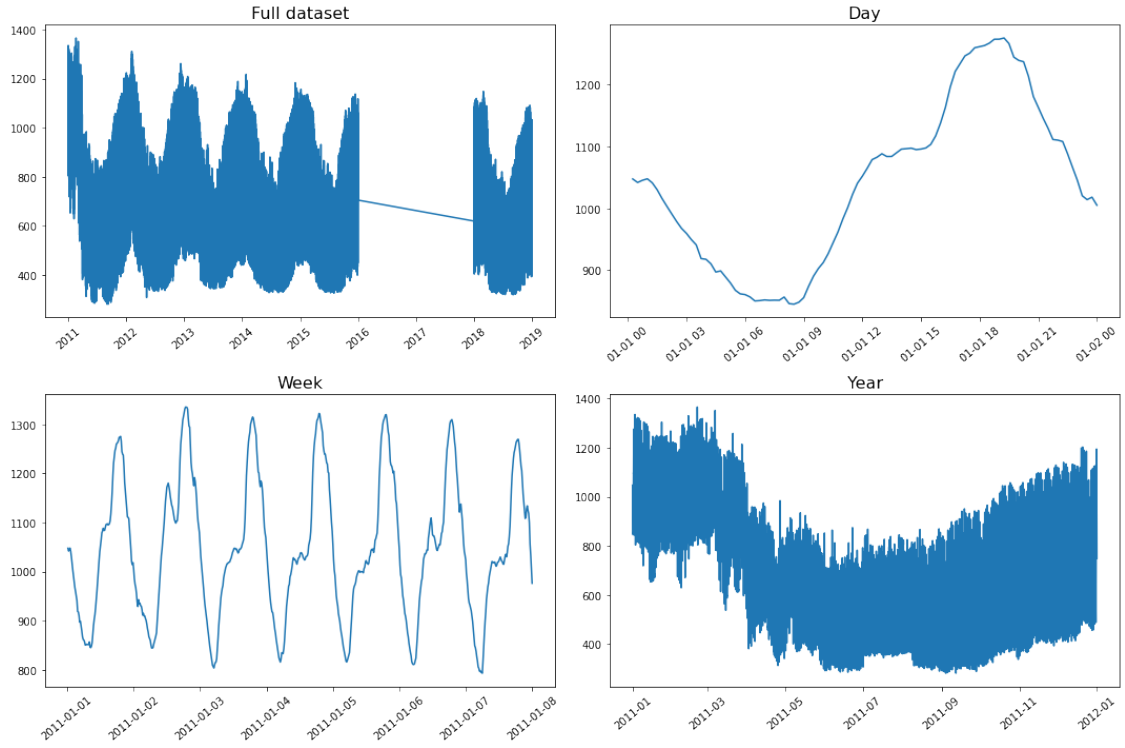
[10]:
```
dates = df.pop("Date")
column_indices, train_df, val_df, test_df = df_split(df)
dfs = [train_df, val_df, test_df]
```

2 2. High-level comparison of prediction models

Data at hand is a time series, which is a sequence of data points occurring in successive order over a period of time. Compared to predicting based on cross sectional data (one that has no coherent time component), this gives us an advantage of being able to analyse repetitive trends and cycles and use these trends either as sole predictor of the future values or alongside with other factors.

The following visualization shows the development of energy consumption in Berlin over years, which clearly shows cyclicity, as well as the temperature trends.

[11]:
```
fig,axes = plt.subplots(2,2, figsize=(15,10))
limits = [len(df),96,672,35040]
titles = ["Full dataset", "Day", "Week", "Year"]

for i,ax in enumerate(axes.flat):
    ax.plot(dates[:limits[i]], df["mW"][:limits[i]])
    ax.set_title(titles[i], size=16)
    ax.tick_params(axis='x', labelrotation=40)

fig.tight_layout()
```



For our prediction we use the following factors: 1. sequentionality of data itself, by choosing to use a time series forecasting model with multi-step input 2. temperature and daytime as additional factors influencing the consumption

We compared the baseline performance of several neural network models in forecasting multiple timesteps of energy consumption based on a sequence of input timesteps of energy consumption, temperature and daytime.

We then took the best performing model type and conducted further experimentation on it to find the best performing hyperparameters.

Let us first look at each model with more detail.

[12]:
```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
```

Models make predictions based on a window of consecutive samples from the time series.

Features of a window: - width (number of input + output timesteps) - time offset between input and output (predicting directly after the last timestep or with a shift) - which features are used as inputs and labels

WindowGenerator class can:

- Handle the indexes and offsets
- Split windows of features into a (features, labels) pairs

- Plot the content of the resulting windows.
- Generate batches of these windows from the training, evaluation, and test data, using tf.data.Datasets.

[13]:
```
class WindowGenerator():
    def __init__(self, input_width, label_width, shift,
                 train_df, val_df, test_df,
                 label_columns=None):
        # Store the raw data.
        self.train_df = train_df
        self.val_df = val_df
        self.test_df = test_df

        # Work out the label column indices.
        self.label_columns = label_columns
        if label_columns is not None:
            self.label_columns_indices = {name: i for i, name in
                                          enumerate(label_columns)}
        self.column_indices = {name: i for i, name in
                               enumerate(train_df.columns)}

        # Work out the window parameters.
        self.input_width = input_width
        self.label_width = label_width
        self.shift = shift

        self.total_window_size = input_width + shift

        self.input_slice = slice(0, input_width)
        self.input_indices = np.arange(self.total_window_size)[self.input_slice]

        self.label_start = self.total_window_size - self.label_width
        self.labels_slice = slice(self.label_start, None)
        self.label_indices = np.arange(self.total_window_size)[self.
↪labels_slice]

    def __repr__(self):
        return '\n'.join([
            f'Total window size: {self.total_window_size}',
            f'Input indices: {self.input_indices}',
            f'Label indices: {self.label_indices}',
            f'Label column name(s): {self.label_columns}'])

    def split_window(self, features):
        """Split list of consecutive inputs into a window of inputs and window␣
↪of (training) labels"""
        inputs = features[:, self.input_slice, :]
```

```
        labels = features[:, self.labels_slice, :]
        if self.label_columns is not None:
            labels = tf.stack(
                [labels[:, :, self.column_indices[name]] for name in self.
↪label_columns],
                axis=-1)

        # Slicing doesn't preserve static shape information, so set the shapes
        # manually. This way the `tf.data.Datasets` are easier to inspect.
        inputs.set_shape([None, self.input_width, None])
        labels.set_shape([None, self.label_width, None])

        return inputs, labels

    def plot(self, model=None, plot_col='mW', max_subplots=3):
        """Visualization of the split window"""
        inputs, labels = self.example
        plt.figure(figsize=(12, 8))
        plot_col_index = self.column_indices[plot_col]
        max_n = min(max_subplots, len(inputs))
        for n in range(max_n):
            plt.subplot(max_n, 1, n+1)
            plt.ylabel(f'{plot_col} [normed]')
            plt.plot(self.input_indices, inputs[n, :, plot_col_index],
                             label='Inputs', marker='.', zorder=-10)

            if self.label_columns:
                label_col_index = self.label_columns_indices.get(plot_col, None)
            else:
                label_col_index = plot_col_index

            if label_col_index is None:
                continue

            plt.scatter(self.label_indices, labels[n, :, label_col_index],
                                    edgecolors='k', label='Labels',␣
↪c='#2ca02c', s=64)
            if model is not None:
                predictions = model(inputs)
                plt.scatter(self.label_indices, predictions[n, :,␣
↪label_col_index],
                                            marker='X', edgecolors='k',␣
↪label='Predictions',
                                            c='#ff7f0e', s=64)

            if n == 0:
                plt.legend()
```

```
        plt.xlabel('Time [h]')

    def make_dataset(self, data):
        """Create batchable tf.data.Datasets for iteration over data"""
        data = np.array(data, dtype=np.float32)
        ds = tf.keras.preprocessing.timeseries_dataset_from_array(
                data=data,
                targets=None,
                sequence_length=self.total_window_size,
                sequence_stride=1,
                shuffle=True,
                batch_size=32,)

        ds = ds.map(self.split_window)

        return ds

    @property
    def train(self):
        return self.make_dataset(self.train_df)

    @property
    def val(self):
        return self.make_dataset(self.val_df)

    @property
    def test(self):
        return self.make_dataset(self.test_df)

    @property
    def example(self):
        """Get and cache an example batch of `inputs, labels` for plotting."""
        result = getattr(self, '_example', None)
        if result is None:
            # No example batch was found, so get one from the `.train` dataset
            result = next(iter(self.train))
            # And cache it for next time
            self._example = result
        return result
```

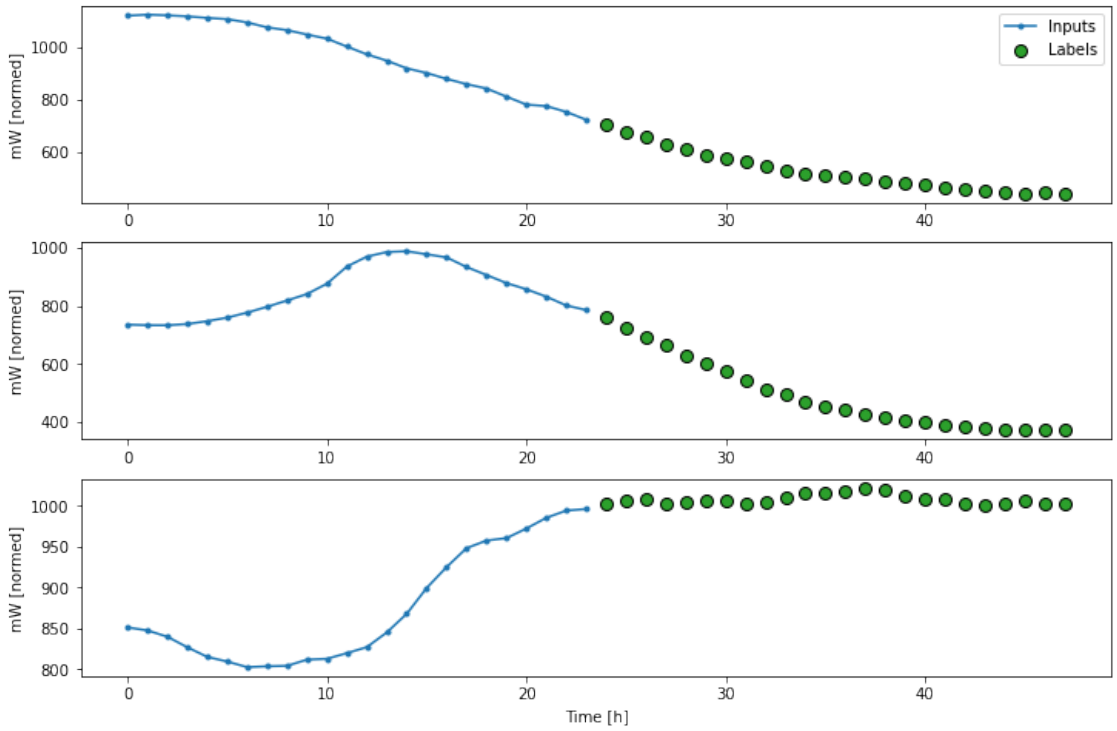
[14]:
```
def create_window(dfs, input_timesteps, output_timesteps):
    """Create a window generator with multiple-timestep input and output """
    train_df = dfs[0]
    val_df = dfs[1]
    test_df = dfs[2]
```

```
    multistep_window = WindowGenerator(
        input_width=input_timesteps, # n of timestamps that are input for␣
↪prediction
        label_width=output_timesteps, # how many to predict
        shift=output_timesteps,
        train_df=train_df, val_df=val_df, test_df=test_df)

    return multistep_window
```

[15]:
```
def compile_model(multistep_model, lr):
    """Compile model with given learning rate"""

    multistep_model.compile(loss=tf.losses.MeanSquaredError(),
                optimizer=tf.optimizers.Adam(learning_rate=lr),
                metrics=[tf.metrics.MeanAbsoluteError()])

    return multistep_model
```

[16]:
```
def fit_model(multistep_model, window, epochs=20):
    """Traint a model on the generator window during given number of epochs"""

    history = multistep_model.fit(window.train, epochs=epochs,
                          validation_data=window.val)

    return history
```

[17]:
```
input_steps = 24
output_steps = 24
num_features = 1

multi_window = WindowGenerator(input_width=input_steps,
                               label_width=output_steps,
                               shift=output_steps,
                               label_columns=["mW"],
                               train_df=train_df,
                               val_df=val_df,
                               test_df=test_df)

multi_window.plot()
```



[46]:
```
# Define dictionaries for performance comparison
multi_val_performance = {}
multi_performance = {}
```

We will compare three types of models: - Multilayer Dense (simple Feed Forward Neural Network) - Convolutional Neural Network with several convolutions - Long Short Term Memory Network

2.1 2.0. Baseline

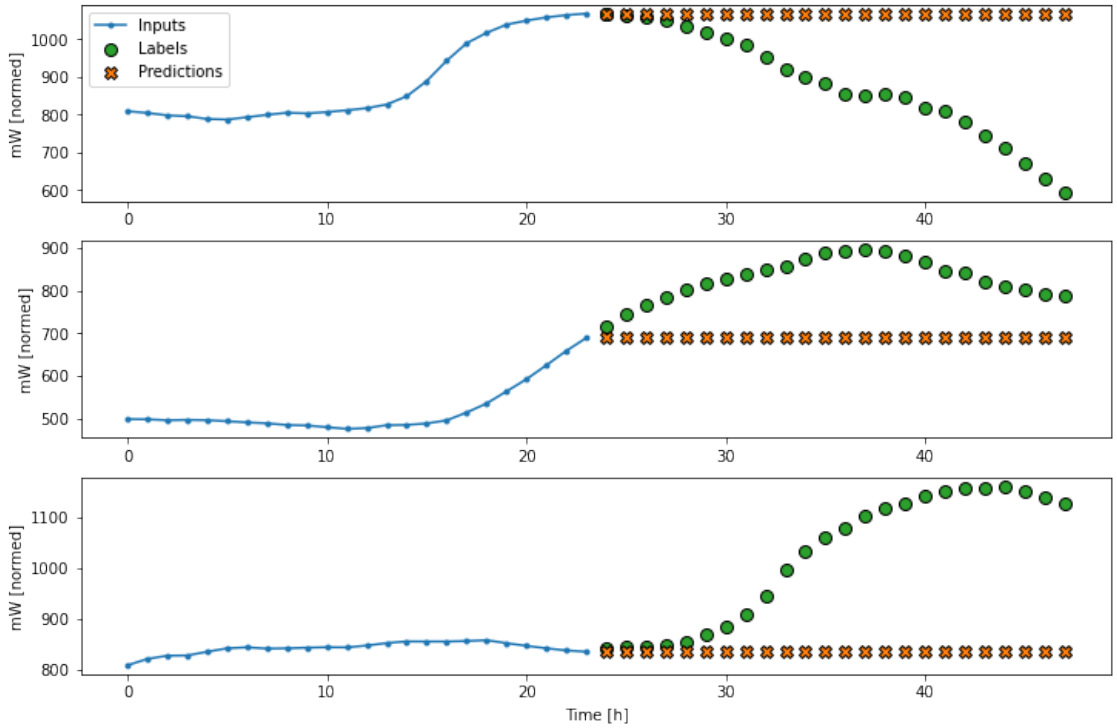
First, setting a baseline for comparison with actual models performance.

Two models are used as baseline: - repeating the last timestep for all predictions - repeating the input timestep sequence as a prediction

[48]:
```
class MultiStepLastBaseline(tf.keras.Model):
    def call(self, inputs):
        return tf.tile(inputs[:, -1:, :], [1, output_steps, 1])

last_baseline = MultiStepLastBaseline()
last_baseline.compile(loss=tf.losses.MeanSquaredError(),
                      metrics=[tf.metrics.MeanAbsoluteError()])

multi_val_performance['Baseline_Last'] = last_baseline.evaluate(multi_window.
 ↪val)
```

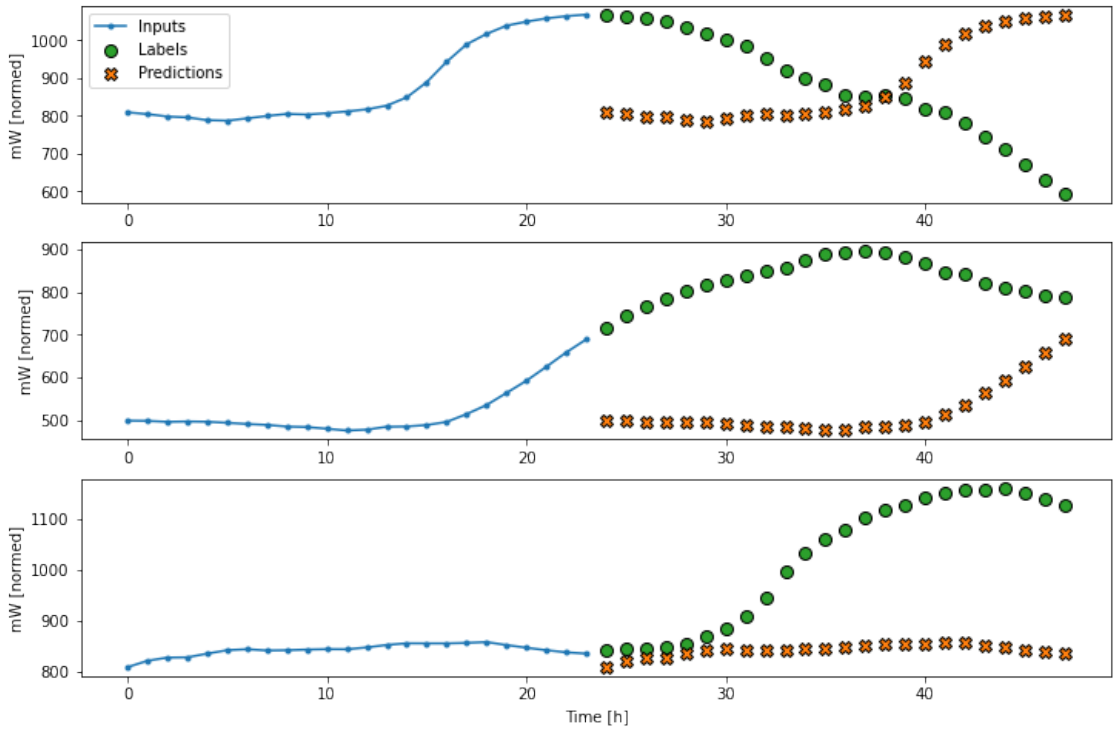
```
multi_performance['Baseline_Last'] = last_baseline.evaluate(multi_window.test,␣
↪verbose=0)
multi_window.plot(last_baseline)
```

```
1314/1314 [==============================] - 3s 2ms/step - loss: 327052.7188 -
mean_absolute_error: 484.9252
```



[49]:
```
class RepeatBaseline(tf.keras.Model):
    def call(self, inputs):
        return inputs

repeat_baseline = RepeatBaseline()
repeat_baseline.compile(loss=tf.losses.MeanSquaredError(),
                        metrics=[tf.metrics.MeanAbsoluteError()])

multi_val_performance['Baseline_Repeat'] = repeat_baseline.
↪evaluate(multi_window.val)
multi_performance['Baseline_Repeat'] = repeat_baseline.evaluate(multi_window.
↪test, verbose=0)
multi_window.plot(repeat_baseline)
```

```
1314/1314 [==============================] - 3s 2ms/step - loss: 340701.4062 -
mean_absolute_error: 513.9308
```

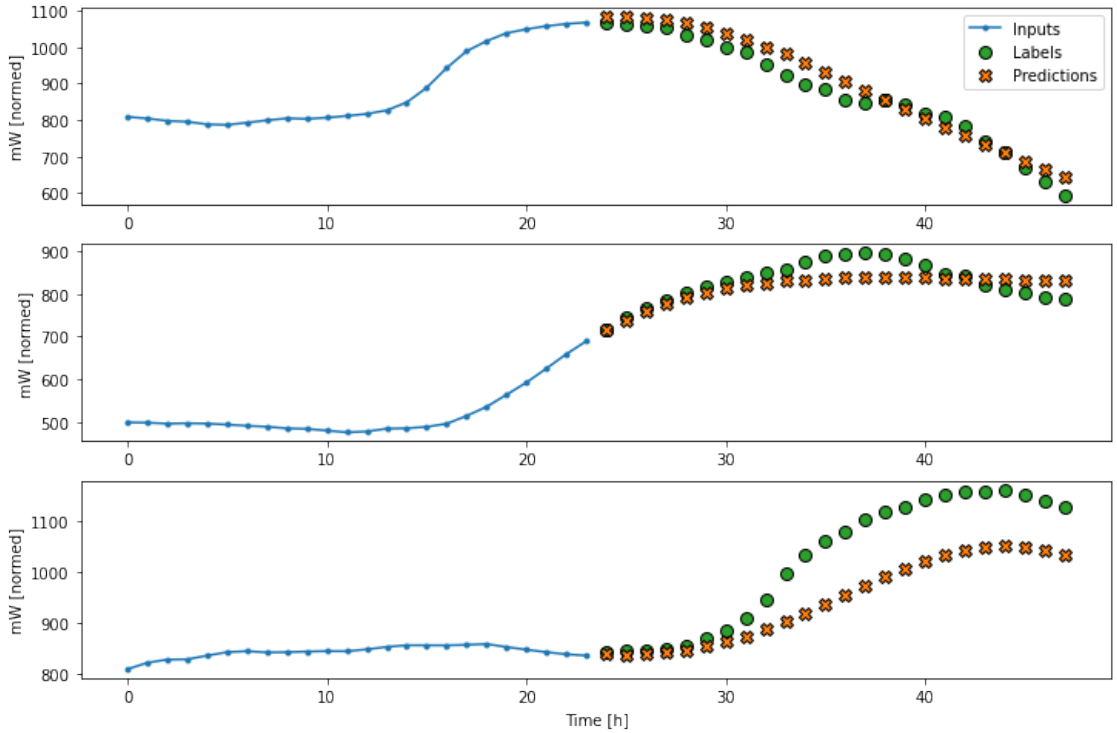


2.2 2.1. Multilayer dense

Predicts output timesteps from a single timestep (last in the input sequence) with a single dense inner layer model. (This type of model is underpowered and only captures a low-dimensional slice of the behavior).

[75]:
```
multi_dense_model = tf.keras.Sequential([
    tf.keras.layers.Lambda(lambda x: x[:, -1:, :]),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(output_steps*num_features,
                          kernel_initializer=tf.initializers.zeros()),
    tf.keras.layers.Reshape([output_steps, num_features])
])

multi_dense_model = compile_model(multi_dense_model, lr=0.001)
history = fit_model(multi_dense_model, multi_window,20)

IPython.display.clear_output()
multi_val_performance['Dense'] = multi_dense_model.evaluate(multi_window.val)
multi_performance['Dense'] = multi_dense_model.evaluate(multi_window.test,␣
 ↪verbose=0)
multi_window.plot(multi_dense_model)
```

```
1314/1314 [==============================] - 3s 2ms/step - loss: 2035.3624 -
mean_absolute_error: 32.7289
```



2.3 2.2. CNN with multiple convolutions

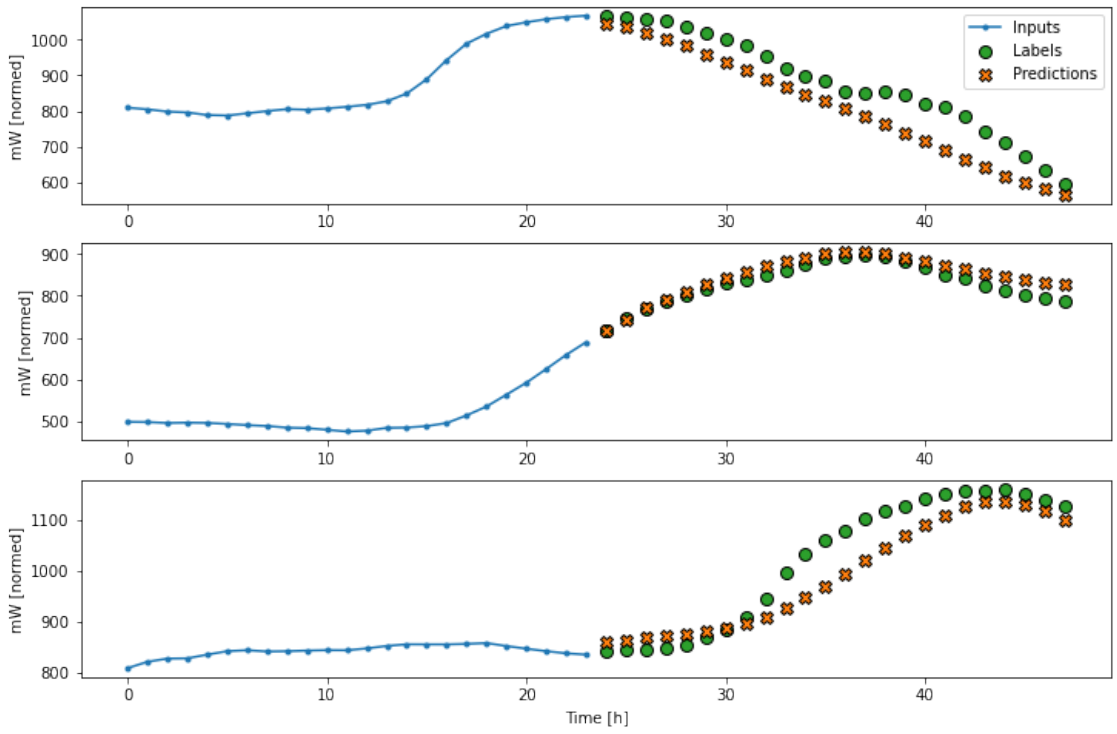
A convolutional model makes predictions based on a fixed-width history, which may lead to better performance than the dense model since it can see how things are changing over time.

To capture the patterns in the data, I use three stacked convolutional layers with MaxPool layers inbetween, followed by two Dense layers for prediction.

[62]:
```
CONV_WIDTH = 24
num_features = 1

multi_conv_model = tf.keras.Sequential([
        tf.keras.layers.Lambda(lambda x: x[:, -CONV_WIDTH:, :]),
        tf.keras.layers.Conv1D(256, activation='relu', kernel_size=(3)),
        tf.keras.layers.MaxPooling1D(pool_size=2),
        tf.keras.layers.Conv1D(filters=256, kernel_size=(3), activation='relu'),
        tf.keras.layers.MaxPooling1D(pool_size=2),
        tf.keras.layers.Conv1D(filters=256, kernel_size=(3), activation='relu'),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(256, activation='relu'),
        tf.keras.layers.Dense(100, activation='relu'),
```

```
        tf.keras.layers.Dense(output_steps*num_features,
                              kernel_initializer=tf.initializers.zeros()),
        tf.keras.layers.Reshape([output_steps, num_features])
])

multi_conv_model = compile_model(multi_conv_model, lr=0.001)
history = fit_model(multi_conv_model, multi_window)

IPython.display.clear_output()

multi_val_performance['Conv'] = multi_conv_model.evaluate(multi_window.val)
multi_performance['Conv'] = multi_conv_model.evaluate(multi_window.test,␣
 ↪verbose=0)
multi_window.plot(multi_conv_model)
```

```
1314/1314 [==============================] - 4s 3ms/step - loss: 1370.6779 -
mean_absolute_error: 25.9363
```



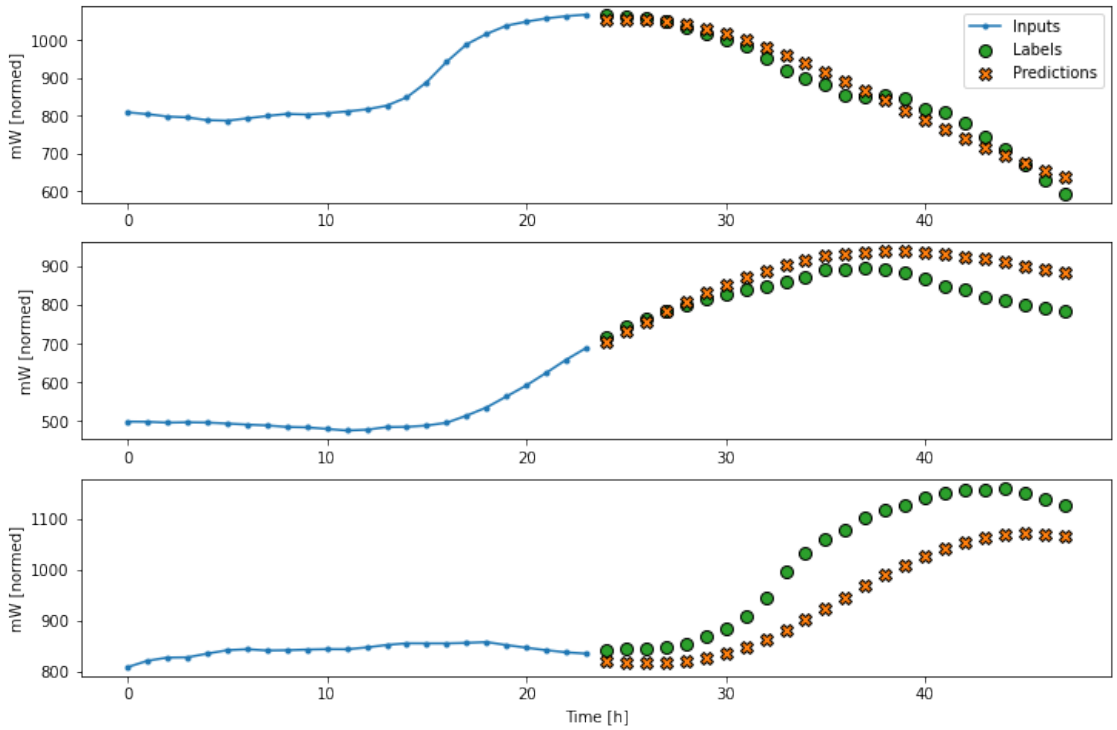
2.4 2.3. LSTM

A recurrent model can learn to use a long history of inputs, if it is relevant to predictions. It accumulates the state of the input time sequence before making a full prediction for the output sequence.

Here I use a layer with 512 LSTM units followed by a Dense layer.

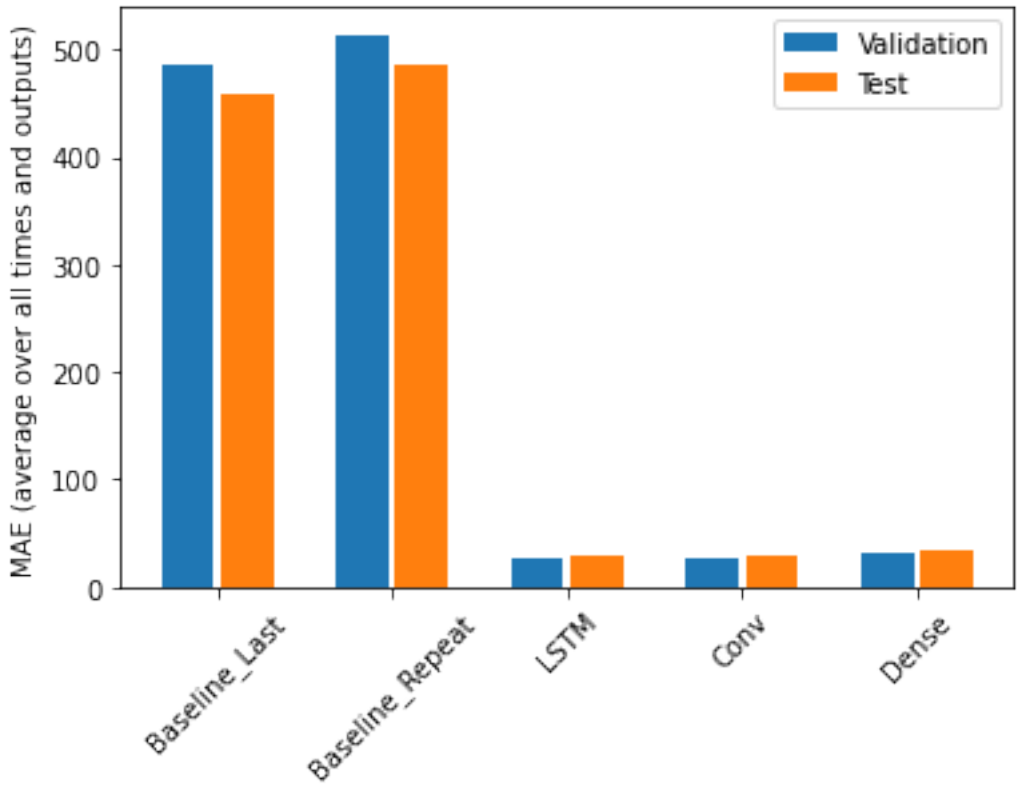
[63]:
```
multi_lstm_model = tf.keras.Sequential([
    tf.keras.layers.LSTM(512, return_sequences=False),
    tf.keras.layers.Dense(100),
    tf.keras.layers.Dense(output_steps*num_features,
                          kernel_initializer=tf.initializers.zeros()),
    tf.keras.layers.Reshape([output_steps, num_features])
])

multi_lstm_model = compile_model(multi_lstm_model, lr=0.001)
history = fit_model(multi_lstm_model, multi_window)

IPython.display.clear_output()

multi_val_performance['LSTM'] = multi_lstm_model.evaluate(multi_window.val)
multi_performance['LSTM'] = multi_lstm_model.evaluate(multi_window.test,␣
 ↪verbose=0)
multi_window.plot(multi_lstm_model)
```

```
1314/1314 [==============================] - 37s 28ms/step - loss: 1343.2423 -
mean_absolute_error: 26.4239
```



2.5 Performance comparison

[76]:
```
x = np.arange(len(multi_performance))
width = 0.3

metric_name = 'mean_absolute_error'
metric_index = multi_conv_model.metrics_names.index('mean_absolute_error')
val_mae = [v[metric_index] for v in multi_val_performance.values()]
test_mae = [v[metric_index] for v in multi_performance.values()]

plt.bar(x - 0.17, val_mae, width, label='Validation')
plt.bar(x + 0.17, test_mae, width, label='Test')
plt.xticks(ticks=x, labels=multi_performance.keys(),
           rotation=45)
plt.ylabel(f'MAE (average over all times and outputs)')
_ = plt.legend()
```



3 3. Grid search for best hyperparameters on the best model type

[ ]:
```
import re
```

[109]:
```
gridsearch_performance = {}
```

[110]:
```
def train_test_model(window, params, epochs=10):
    LR = params["LR"]
    DENSE_UNITS = params["DENSE_UNITS"]


    model = tf.keras.Sequential([
        tf.keras.layers.LSTM(512, return_sequences=False),
        tf.keras.layers.Dense(DENSE_UNITS),
        tf.keras.layers.Dense(output_steps*num_features,
                              kernel_initializer=tf.initializers.zeros()),
        tf.keras.layers.Reshape([output_steps, num_features])
    ])

    model.compile(loss=tf.losses.MeanSquaredError(),
                optimizer=tf.optimizers.Adam(learning_rate=LR),
                metrics=[tf.metrics.MeanAbsoluteError()])

    model.fit(window.train, epochs=epochs,
                          validation_data=window.val)


    _, mae = model.evaluate(multi_window.test, verbose=0)

    return mae
```

[111]:
```
LEARNING_RATES = [1e-3, 1e-4, 1e-5, 1e-6, 1e-10]
DENSE_UNITS = [100, 256]
```

[ ]:
```
session_num = 0

for dense_units in DENSE_UNITS:
    for lr in LEARNING_RATES:
        params = {
                "LR": lr,
                "DENSE_UNITS": dense_units
        }
        # redo this
        run_name = "run-%d" % session_num
        print('--- Starting trial: %s' % run_name)
        session_name = f"lr_{lr}_dense_{dense_units}"
        print(session_name)
```

```
        mae = train_test_model(multi_window, params, epochs=10)
        gridsearch_performance[session_name] = mae

        session_num += 1
```

[112]:
```
session_num = 0

for dense_units in DENSE_UNITS:
    for lr in LEARNING_RATES:
        params = {
                "LR": lr,
                "DENSE_UNITS": dense_units
        }
        # redo this
        run_name = "run-%d" % session_num
        print('--- Starting trial: %s' % run_name)
        session_name = f"lr_{lr}_dense_{dense_units}"
        print(session_name)
        mae = train_test_model(multi_window, params, epochs=10)
        gridsearch_performance[session_name] = mae

        session_num += 1
```

```
--- Starting trial: run-0
lr_0.001_dense_100
Epoch 1/10
4600/4600 [==============================] - 358s 78ms/step - loss: 19131.5117 -
mean_absolute_error: 89.6897 - val_loss: 3089.5229 - val_mean_absolute_error:
41.5166
Epoch 2/10
4600/4600 [==============================] - 359s 78ms/step - loss: 3212.7122 -
mean_absolute_error: 42.0756 - val_loss: 2481.0532 - val_mean_absolute_error:
37.3386
Epoch 3/10
4600/4600 [==============================] - 360s 78ms/step - loss: 2660.8936 -
mean_absolute_error: 37.8335 - val_loss: 1850.1487 - val_mean_absolute_error:
31.5851
Epoch 4/10
4600/4600 [==============================] - 362s 79ms/step - loss: 2485.9609 -
mean_absolute_error: 36.2973 - val_loss: 1892.2766 - val_mean_absolute_error:
31.8261
Epoch 5/10
4600/4600 [==============================] - 372s 81ms/step - loss: 2368.4270 -
mean_absolute_error: 35.3407 - val_loss: 2231.7292 - val_mean_absolute_error:
34.9839
Epoch 6/10
4600/4600 [==============================] - 361s 78ms/step - loss: 2303.9155 -
```

```
mean_absolute_error: 34.8001 - val_loss: 1823.6439 - val_mean_absolute_error:
30.1970
Epoch 7/10
4600/4600 [==============================] - 365s 79ms/step - loss: 2234.1296 -
mean_absolute_error: 34.2031 - val_loss: 1779.5570 - val_mean_absolute_error:
30.6729
Epoch 8/10
4600/4600 [==============================] - 371s 81ms/step - loss: 2188.8245 -
mean_absolute_error: 33.8596 - val_loss: 1811.9919 - val_mean_absolute_error:
31.2683
Epoch 9/10
4600/4600 [==============================] - 369s 80ms/step - loss: 2169.8521 -
mean_absolute_error: 33.6655 - val_loss: 1628.1057 - val_mean_absolute_error:
29.6680
Epoch 10/10
4600/4600 [==============================] - 362s 79ms/step - loss: 2160.7808 -
mean_absolute_error: 33.5988 - val_loss: 1837.4335 - val_mean_absolute_error:
31.0171
--- Starting trial: run-1
lr_0.0001_dense_100
Epoch 1/10
4600/4600 [==============================] - 363s 78ms/step - loss: 73526.1953 -
mean_absolute_error: 175.0955 - val_loss: 9850.3711 - val_mean_absolute_error:
75.8944
Epoch 2/10
4600/4600 [==============================] - 360s 78ms/step - loss: 9756.8789 -
mean_absolute_error: 75.6077 - val_loss: 5582.2480 - val_mean_absolute_error:
57.4708
Epoch 3/10
4600/4600 [==============================] - 359s 78ms/step - loss: 3366.4006 -
mean_absolute_error: 44.1223 - val_loss: 2265.6086 - val_mean_absolute_error:
36.3675
Epoch 4/10
4600/4600 [==============================] - 359s 78ms/step - loss: 2591.9231 -
mean_absolute_error: 38.5309 - val_loss: 1894.1429 - val_mean_absolute_error:
32.5033
Epoch 5/10
4600/4600 [==============================] - 359s 78ms/step - loss: 1952.0659 -
mean_absolute_error: 32.1231 - val_loss: 1441.7744 - val_mean_absolute_error:
27.5311
Epoch 6/10
4600/4600 [==============================] - 359s 78ms/step - loss: 1687.9613 -
mean_absolute_error: 29.3403 - val_loss: 1280.3523 - val_mean_absolute_error:
25.3656
Epoch 7/10
4600/4600 [==============================] - 358s 78ms/step - loss: 1600.8020 -
mean_absolute_error: 28.4068 - val_loss: 1217.9825 - val_mean_absolute_error:
24.6259
```

```
Epoch 8/10
4600/4600 [==============================] - 358s 78ms/step - loss: 1548.3757 -
mean_absolute_error: 27.8493 - val_loss: 1260.0380 - val_mean_absolute_error:
24.9166
Epoch 9/10
4600/4600 [==============================] - 362s 79ms/step - loss: 1511.5880 -
mean_absolute_error: 27.4387 - val_loss: 1159.6246 - val_mean_absolute_error:
23.9967
Epoch 10/10
4600/4600 [==============================] - 382s 83ms/step - loss: 1466.2303 -
mean_absolute_error: 26.9262 - val_loss: 1138.4950 - val_mean_absolute_error:
23.7822
--- Starting trial: run-2
lr_1e-05_dense_100
Epoch 1/10
4600/4600 [==============================] - 382s 83ms/step - loss: 459551.9688
- mean_absolute_error: 643.0610 - val_loss: 268656.0312 -
val_mean_absolute_error: 483.3046
Epoch 2/10
4600/4600 [==============================] - 373s 81ms/step - loss: 181584.0000
- mean_absolute_error: 368.5462 - val_loss: 55714.7695 -
val_mean_absolute_error: 202.0530
Epoch 3/10
4600/4600 [==============================] - 373s 81ms/step - loss: 43086.2695 -
mean_absolute_error: 169.7485 - val_loss: 20756.2930 - val_mean_absolute_error:
115.2513
Epoch 4/10
4600/4600 [==============================] - 372s 81ms/step - loss: 14573.6797 -
mean_absolute_error: 93.9727 - val_loss: 9949.5635 - val_mean_absolute_error:
77.3852
Epoch 5/10
4600/4600 [==============================] - 372s 81ms/step - loss: 10478.2383 -
mean_absolute_error: 78.8341 - val_loss: 9641.9238 - val_mean_absolute_error:
76.0137
Epoch 6/10
4600/4600 [==============================] - 372s 81ms/step - loss: 10234.4287 -
mean_absolute_error: 77.6108 - val_loss: 9626.0352 - val_mean_absolute_error:
75.0809
Epoch 7/10
4600/4600 [==============================] - 372s 81ms/step - loss: 10137.7881 -
mean_absolute_error: 77.1027 - val_loss: 9535.7500 - val_mean_absolute_error:
74.5720
Epoch 8/10
4600/4600 [==============================] - 372s 81ms/step - loss: 10058.5879 -
mean_absolute_error: 76.7063 - val_loss: 9428.0830 - val_mean_absolute_error:
73.9212
Epoch 9/10
4600/4600 [==============================] - 374s 81ms/step - loss: 9974.7021 -
```

```
mean_absolute_error: 76.3098 - val_loss: 9354.6768 - val_mean_absolute_error:
73.3385
Epoch 10/10
4600/4600 [==============================] - 372s 81ms/step - loss: 9842.1562 -
mean_absolute_error: 75.7381 - val_loss: 9148.7402 - val_mean_absolute_error:
72.4303
--- Starting trial: run-3
lr_1e-06_dense_100
Epoch 1/10
4600/4600 [==============================] - 367s 79ms/step - loss: 554339.1875
- mean_absolute_error: 715.4788 - val_loss: 488722.4062 -
val_mean_absolute_error: 673.5330
Epoch 2/10
4600/4600 [==============================] - 369s 80ms/step - loss: 551725.5625
- mean_absolute_error: 713.6349 - val_loss: 484847.3438 -
val_mean_absolute_error: 670.6270
Epoch 3/10
4600/4600 [==============================] - 363s 79ms/step - loss: 545757.3750
- mean_absolute_error: 709.3885 - val_loss: 477114.8438 -
val_mean_absolute_error: 664.7855
Epoch 4/10
4600/4600 [==============================] - 363s 79ms/step - loss: 535509.3750
- mean_absolute_error: 702.0519 - val_loss: 465215.9062 -
val_mean_absolute_error: 655.7371
Epoch 5/10
4600/4600 [==============================] - 361s 78ms/step - loss: 521234.2812
- mean_absolute_error: 691.7720 - val_loss: 450346.0312 -
val_mean_absolute_error: 644.2763
Epoch 6/10
4600/4600 [==============================] - 366s 80ms/step - loss: 503985.3750
- mean_absolute_error: 679.1889 - val_loss: 432997.9375 -
val_mean_absolute_error: 630.7110
Epoch 7/10
4600/4600 [==============================] - 360s 78ms/step - loss: 484671.5000
- mean_absolute_error: 664.8770 - val_loss: 414384.0938 -
val_mean_absolute_error: 615.8127
Epoch 8/10
4600/4600 [==============================] - 360s 78ms/step - loss: 463857.5000
- mean_absolute_error: 649.0642 - val_loss: 394048.1250 -
val_mean_absolute_error: 599.0823
Epoch 9/10
4600/4600 [==============================] - 358s 78ms/step - loss: 441243.7188
- mean_absolute_error: 631.4187 - val_loss: 372244.1250 -
val_mean_absolute_error: 580.6212
Epoch 10/10
4600/4600 [==============================] - 358s 78ms/step - loss: 417145.4688
- mean_absolute_error: 612.0532 - val_loss: 349152.0938 -
val_mean_absolute_error: 560.3894
```

```
--- Starting trial: run-4
lr_1e-10_dense_100
Epoch 1/10
4600/4600 [==============================] - 366s 79ms/step - loss: 554847.8125
- mean_absolute_error: 715.8343 - val_loss: 490032.0625 -
val_mean_absolute_error: 674.5076
Epoch 2/10
4600/4600 [==============================] - 367s 80ms/step - loss: 554848.2500
- mean_absolute_error: 715.8339 - val_loss: 490031.7500 -
val_mean_absolute_error: 674.5076
Epoch 3/10
4600/4600 [==============================] - 375s 81ms/step - loss: 554847.0625
- mean_absolute_error: 715.8331 - val_loss: 490031.3750 -
val_mean_absolute_error: 674.5068
Epoch 4/10
4600/4600 [==============================] - 372s 81ms/step - loss: 554848.5000
- mean_absolute_error: 715.8350 - val_loss: 490031.7500 -
val_mean_absolute_error: 674.5067
Epoch 5/10
4600/4600 [==============================] - 366s 80ms/step - loss: 554847.1250
- mean_absolute_error: 715.8341 - val_loss: 490031.7500 -
val_mean_absolute_error: 674.5065
Epoch 6/10
4600/4600 [==============================] - 365s 79ms/step - loss: 554848.3750
- mean_absolute_error: 715.8340 - val_loss: 490031.5625 -
val_mean_absolute_error: 674.5066
Epoch 7/10
4600/4600 [==============================] - 365s 79ms/step - loss: 554847.1250
- mean_absolute_error: 715.8339 - val_loss: 490032.1875 -
val_mean_absolute_error: 674.5074
Epoch 8/10
4600/4600 [==============================] - 372s 81ms/step - loss: 554848.3750
- mean_absolute_error: 715.8342 - val_loss: 490031.6250 -
val_mean_absolute_error: 674.5070
Epoch 9/10
4600/4600 [==============================] - 371s 81ms/step - loss: 554847.2500
- mean_absolute_error: 715.8334 - val_loss: 490030.7812 -
val_mean_absolute_error: 674.5071
Epoch 10/10
4600/4600 [==============================] - 367s 80ms/step - loss: 554847.3125
- mean_absolute_error: 715.8337 - val_loss: 490031.8125 -
val_mean_absolute_error: 674.5069
--- Starting trial: run-5
lr_0.001_dense_256
Epoch 1/10
4600/4600 [==============================] - 366s 79ms/step - loss: 14362.4316 -
mean_absolute_error: 74.9435 - val_loss: 2758.9214 - val_mean_absolute_error:
39.1152
```

```
Epoch 2/10
4600/4600 [==============================] - 364s 79ms/step - loss: 2881.4441 -
mean_absolute_error: 39.7223 - val_loss: 1903.7422 - val_mean_absolute_error:
31.6111
Epoch 3/10
4600/4600 [==============================] - 363s 79ms/step - loss: 2431.3064 -
mean_absolute_error: 35.8628 - val_loss: 1976.2578 - val_mean_absolute_error:
32.2704
Epoch 4/10
4600/4600 [==============================] - 363s 79ms/step - loss: 2278.8938 -
mean_absolute_error: 34.5528 - val_loss: 1689.0612 - val_mean_absolute_error:
30.4862
Epoch 5/10
4600/4600 [==============================] - 364s 79ms/step - loss: 2179.6448 -
mean_absolute_error: 33.6874 - val_loss: 1909.5768 - val_mean_absolute_error:
32.6402
Epoch 6/10
4600/4600 [==============================] - 362s 79ms/step - loss: 2140.9250 -
mean_absolute_error: 33.3458 - val_loss: 1986.8784 - val_mean_absolute_error:
32.7423
Epoch 7/10
4600/4600 [==============================] - 362s 79ms/step - loss: 2081.5474 -
mean_absolute_error: 32.8179 - val_loss: 1834.2406 - val_mean_absolute_error:
31.3298
Epoch 8/10
4600/4600 [==============================] - 363s 79ms/step - loss: 2053.2729 -
mean_absolute_error: 32.5504 - val_loss: 1641.9606 - val_mean_absolute_error:
29.7448
Epoch 9/10
4600/4600 [==============================] - 362s 79ms/step - loss: 2024.0991 -
mean_absolute_error: 32.2533 - val_loss: 1610.0160 - val_mean_absolute_error:
29.4201
Epoch 10/10
4600/4600 [==============================] - 363s 79ms/step - loss: 1997.2643 -
mean_absolute_error: 31.9735 - val_loss: 2113.5085 - val_mean_absolute_error:
35.9833
--- Starting trial: run-6
lr_0.0001_dense_256
Epoch 1/10
4600/4600 [==============================] - 373s 81ms/step - loss: 50194.3242 -
mean_absolute_error: 140.8630 - val_loss: 9758.5732 - val_mean_absolute_error:
74.8172
Epoch 2/10
4600/4600 [==============================] - 371s 81ms/step - loss: 5705.4604 -
mean_absolute_error: 56.1820 - val_loss: 2445.1062 - val_mean_absolute_error:
37.9988
Epoch 3/10
4600/4600 [==============================] - 372s 81ms/step - loss: 2654.3721 -
```

```
mean_absolute_error: 38.8669 - val_loss: 1956.0156 - val_mean_absolute_error:
32.1523
Epoch 4/10
4600/4600 [==============================] - 372s 81ms/step - loss: 1891.1486 -
mean_absolute_error: 31.3898 - val_loss: 1425.9447 - val_mean_absolute_error:
27.2484
Epoch 5/10
4600/4600 [==============================] - 372s 81ms/step - loss: 1719.2272 -
mean_absolute_error: 29.5412 - val_loss: 1270.8936 - val_mean_absolute_error:
25.4619
Epoch 6/10
4600/4600 [==============================] - 372s 81ms/step - loss: 1636.9434 -
mean_absolute_error: 28.5910 - val_loss: 1342.2131 - val_mean_absolute_error:
26.2881
Epoch 7/10
4600/4600 [==============================] - 372s 81ms/step - loss: 1558.3190 -
mean_absolute_error: 27.7036 - val_loss: 1294.7478 - val_mean_absolute_error:
24.9840
Epoch 8/10
4600/4600 [==============================] - 372s 81ms/step - loss: 1508.2736 -
mean_absolute_error: 27.1535 - val_loss: 1175.8636 - val_mean_absolute_error:
23.7566
Epoch 9/10
4600/4600 [==============================] - 372s 81ms/step - loss: 1461.8118 -
mean_absolute_error: 26.6545 - val_loss: 1250.4105 - val_mean_absolute_error:
24.5192
Epoch 10/10
4600/4600 [==============================] - 372s 81ms/step - loss: 1416.4886 -
mean_absolute_error: 26.2063 - val_loss: 1255.1174 - val_mean_absolute_error:
24.7473
--- Starting trial: run-7
lr_1e-05_dense_256
Epoch 1/10
4600/4600 [==============================] - 368s 80ms/step - loss: 352770.9688
- mean_absolute_error: 540.3782 - val_loss: 89012.2891 -
val_mean_absolute_error: 250.6539
Epoch 2/10
4600/4600 [==============================] - 365s 79ms/step - loss: 49479.6836 -
mean_absolute_error: 179.5621 - val_loss: 23489.9785 - val_mean_absolute_error:
121.2537
Epoch 3/10
4600/4600 [==============================] - 365s 79ms/step - loss: 23295.6133 -
mean_absolute_error: 121.6837 - val_loss: 17673.1484 - val_mean_absolute_error:
105.4964
Epoch 4/10
4600/4600 [==============================] - 365s 79ms/step - loss: 12151.3418 -
mean_absolute_error: 86.3155 - val_loss: 9785.3838 - val_mean_absolute_error:
75.8107
```

```
Epoch 5/10
4600/4600 [==============================] - 364s 79ms/step - loss: 10314.1357 -
mean_absolute_error: 77.9537 - val_loss: 9903.5801 - val_mean_absolute_error:
75.4447
Epoch 6/10
4600/4600 [==============================] - 365s 79ms/step - loss: 10101.2979 -
mean_absolute_error: 76.9353 - val_loss: 9532.2422 - val_mean_absolute_error:
74.2050
Epoch 7/10
4600/4600 [==============================] - 365s 79ms/step - loss: 9740.0391 -
mean_absolute_error: 75.4382 - val_loss: 8906.9033 - val_mean_absolute_error:
71.0731
Epoch 8/10
4600/4600 [==============================] - 365s 79ms/step - loss: 8489.6279 -
mean_absolute_error: 70.4696 - val_loss: 6628.5747 - val_mean_absolute_error:
62.2910
Epoch 9/10
4600/4600 [==============================] - 364s 79ms/step - loss: 5516.5552 -
mean_absolute_error: 57.3941 - val_loss: 3633.2017 - val_mean_absolute_error:
46.0884
Epoch 10/10
4600/4600 [==============================] - 364s 79ms/step - loss: 3539.9880 -
mean_absolute_error: 45.9430 - val_loss: 2839.1328 - val_mean_absolute_error:
39.9057
--- Starting trial: run-8
lr_1e-06_dense_256
Epoch 1/10
4600/4600 [==============================] - 368s 80ms/step - loss: 553685.1250
- mean_absolute_error: 715.0184 - val_loss: 486970.2812 -
val_mean_absolute_error: 672.2272
Epoch 2/10
4600/4600 [==============================] - 366s 80ms/step - loss: 547506.0625
- mean_absolute_error: 710.6507 - val_loss: 477866.8438 -
val_mean_absolute_error: 665.3662
Epoch 3/10
4600/4600 [==============================] - 368s 80ms/step - loss: 532953.5625
- mean_absolute_error: 700.1695 - val_loss: 457987.5938 -
val_mean_absolute_error: 650.1165
Epoch 4/10
4600/4600 [==============================] - 371s 81ms/step - loss: 507088.6250
- mean_absolute_error: 681.2767 - val_loss: 429405.9062 -
val_mean_absolute_error: 627.6577
Epoch 5/10
4600/4600 [==============================] - 367s 80ms/step - loss: 473016.6562
- mean_absolute_error: 655.6525 - val_loss: 393865.9688 -
val_mean_absolute_error: 598.6326
Epoch 6/10
4600/4600 [==============================] - 377s 82ms/step - loss: 432567.9375
```

```
- mean_absolute_error: 624.0191 - val_loss: 354336.6562 -
val_mean_absolute_error: 564.6814
Epoch 7/10
4600/4600 [==============================] - 374s 81ms/step - loss: 388417.7500
- mean_absolute_error: 587.6516 - val_loss: 312296.6875 -
val_mean_absolute_error: 526.2828
Epoch 8/10
4600/4600 [==============================] - 375s 81ms/step - loss: 342435.2500
- mean_absolute_error: 547.2861 - val_loss: 269833.5000 -
val_mean_absolute_error: 484.3159
Epoch 9/10
4600/4600 [==============================] - 375s 81ms/step - loss: 295844.0938
- mean_absolute_error: 502.9242 - val_loss: 226683.2656 -
val_mean_absolute_error: 437.5651
Epoch 10/10
4600/4600 [==============================] - 367s 80ms/step - loss: 248974.0938
- mean_absolute_error: 454.0645 - val_loss: 184956.4844 -
val_mean_absolute_error: 387.1129
--- Starting trial: run-9
lr_1e-10_dense_256
Epoch 1/10
4600/4600 [==============================] - 368s 80ms/step - loss: 554848.3125
- mean_absolute_error: 715.8344 - val_loss: 490031.7188 -
val_mean_absolute_error: 674.5069
Epoch 2/10
4600/4600 [==============================] - 367s 80ms/step - loss: 554848.1250
- mean_absolute_error: 715.8354 - val_loss: 490031.4375 -
val_mean_absolute_error: 674.5070
Epoch 3/10
4600/4600 [==============================] - 368s 80ms/step - loss: 554847.7500
- mean_absolute_error: 715.8349 - val_loss: 490031.6562 -
val_mean_absolute_error: 674.5071
Epoch 4/10
4600/4600 [==============================] - 367s 80ms/step - loss: 554847.5000
- mean_absolute_error: 715.8342 - val_loss: 490031.6250 -
val_mean_absolute_error: 674.5074
Epoch 5/10
4600/4600 [==============================] - 367s 80ms/step - loss: 554847.1875
- mean_absolute_error: 715.8339 - val_loss: 490031.6250 -
val_mean_absolute_error: 674.5070
Epoch 6/10
4600/4600 [==============================] - 368s 80ms/step - loss: 554846.7500
- mean_absolute_error: 715.8336 - val_loss: 490031.6250 -
val_mean_absolute_error: 674.5070
Epoch 7/10
4600/4600 [==============================] - 368s 80ms/step - loss: 554848.6250
- mean_absolute_error: 715.8339 - val_loss: 490031.2812 -
val_mean_absolute_error: 674.5072
```

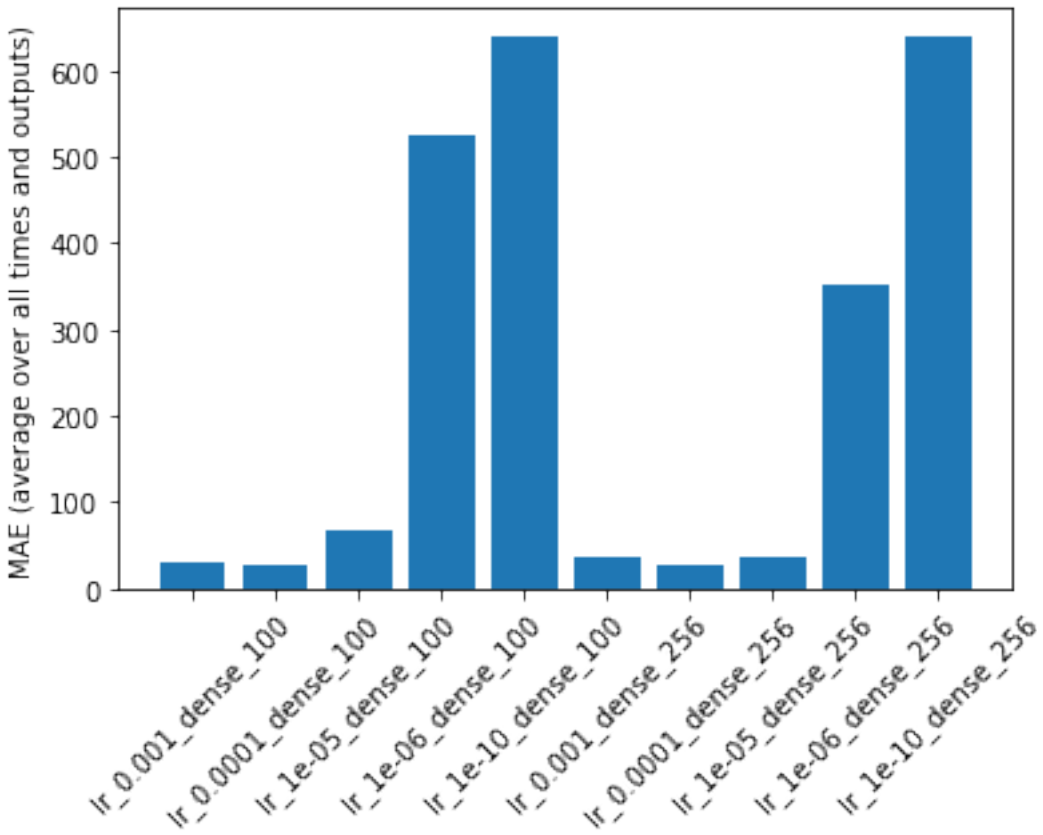
```
Epoch 8/10
4600/4600 [==============================] - 368s 80ms/step - loss: 554847.4375
- mean_absolute_error: 715.8339 - val_loss: 490031.2812 -
val_mean_absolute_error: 674.5068
Epoch 9/10
4600/4600 [==============================] - 369s 80ms/step - loss: 554848.6250
- mean_absolute_error: 715.8347 - val_loss: 490031.4688 -
val_mean_absolute_error: 674.5063
Epoch 10/10
4600/4600 [==============================] - 368s 80ms/step - loss: 554846.8750
- mean_absolute_error: 715.8336 - val_loss: 490031.3750 -
val_mean_absolute_error: 674.5066
```

[113]:
```
gridsearch_performance
```

[113]:
```
{'lr_0.001_dense_100': 30.289310455322266,
 'lr_0.0001_dense_100': 26.845367431640625,
 'lr_1e-05_dense_100': 67.11715698242188,
 'lr_1e-06_dense_100': 526.0383911132812,
 'lr_1e-10_dense_100': 640.2051391601562,
 'lr_0.001_dense_256': 35.28562545776367,
 'lr_0.0001_dense_256': 27.672861099243164,
 'lr_1e-05_dense_256': 36.73331832885742,
 'lr_1e-06_dense_256': 352.7760925292969,
 'lr_1e-10_dense_256': 640.2050170898438}
```

[114]:
```
plt.bar(*zip(*gridsearch_performance.items()))
plt.xticks(rotation=45)
plt.ylabel(f'MAE (average over all times and outputs)')
```

[114]:
```
Text(0, 0.5, 'MAE (average over all times and outputs)')
```



4 Train best combination of params for a longer amount of epochs to get the final model

[75]:
```
best_params = min(gridsearch_performance, key=gridsearch_performance.get)
best_lr = float(re.search("(?<=lr_)(.*?)(?=_)", best_params)[0])
best_dense_units = int(re.search("[^|dense_]*$", best_params)[0])
```

[78]:
```
best_model = tf.keras.Sequential([
    tf.keras.layers.LSTM(512, return_sequences=False),
    tf.keras.layers.Dense(best_dense_units),
    tf.keras.layers.Dense(output_steps*num_features,
                          kernel_initializer=tf.initializers.zeros()),
    tf.keras.layers.Reshape([output_steps, num_features])
])

best_model.compile(loss=tf.losses.MeanSquaredError(),
            optimizer=tf.optimizers.Adam(learning_rate=best_lr),
            metrics=[tf.metrics.MeanAbsoluteError()])
```

```
best_model.fit(multi_window.train, epochs=35,
                      validation_data=multi_window.val)

IPython.display.clear_output()
```

[79]:
```
best_model.evaluate(multi_window.test, verbose=0)
```

[79]:
```
[1675.600341796875, 26.683202743530273]
```

Saving the model in Tensorflow SavedModel format for later use in the prediction application.

[81]:
```
best_model.save('model')
```