```
%matplotlib inline

import datetime
import itertools
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import tensorflow as tf

# How to enable Colab GPUs https://colab.research.google.com/notebooks/gpu.ipynb
# Select the Runtime > "Change runtime type" menu to enable a GPU accelerator,
# and then re-execute this cell.
if 'google.colab' in str(get_ipython()):
    device_name = tf.test.gpu_device_name()
    if device_name != '/device:GPU:0':
        raise SystemError('GPU device not found')
    print('Found GPU at: {}'.format(device_name))

import tensorflow.keras as keras
from keras.models import Sequential
from keras.layers import Input, Dense, Dropout, Activation, Conv1D, \
                         BatchNormalization, GlobalAveragePooling1D, Flatten, \
                         Reshape, LSTM
from keras.optimizers import Adam, Adadelta
from keras.callbacks import EarlyStopping, ReduceLROnPlateau

# Reduces variance in results but won't eliminate it :-(
%env PYTHONHASHSEED=0
import random
random.seed(42)
np.random.seed(42)
tf.random.set_seed(42)
```

```
Found GPU at: /device:GPU:0
env: PYTHONHASHSEED=0
```

Keras Neural Networks for Weather Time Series Nowcasts

Building neural networks with keras for time series analysis of Cambridge UK weather data, using a streamlined version of data preparation from Tensorflow time series forecasting tutorial.

Import Data

Data has been cleaned but may still have issues. See the cleaning section in the Cambridge Temperature Model repository for details.

The `y` variable is temperature * 10. I'm primarily interested in short term temperature forecasts (less than 2 hours). Observations occur every 30 mins.

```
if 'google.colab' in str(get_ipython()):
    data_loc = "https://github.com/makeyourownmaker/CambridgeTemperatureModel/blob/master/data/CamUKWeather.csv?raw=true"
else:
    data_loc = "../data/CamUKWeather.csv"
df = pd.read_csv(data_loc, parse_dates = True)

df['ds'] = pd.to_datetime(df['ds'])

print("Shape:")
print(df.shape)
print("\nInfo:")
print(df.info())
print("\nSummary stats:")
display(df.describe())
print("\nRaw data:")
display(df)
print("\n")
```

```
def plot_examples(data, x_var):
    """Plot 9 sets of observations in 3 * 3 matrix ..."""

    assert len(data) == 9

    cols = [col for col in data[0].columns if col != x_var]

    fig, axs = plt.subplots(3, 3, figsize = (15, 10))
```

```
    axs = axs.ravel()  # apl for the win :-)

    for i in range(9):
      for col in cols:
        axs[i].plot(data[i][x_var], data[i][col])
        axs[i].xaxis.set_tick_params(rotation = 20, labelsize = 10)

    fig.legend(cols, loc = 'upper center',  ncol = len(cols))

    return None


cols = ['ds', 'y', 'humidity', 'dew.point', 'pressure',
        'wind.speed.mean', 'wind.bearing.mean', 'wind.speed.max']
plots  = 9
window = 24
starts = [random.randint(0, np.floor(df.shape[0] / window)) for _ in range(plots)]
p_data = [df.loc[starts[i] * window:starts[i] * window + window, cols]
          for i in range(plots)]
plot_examples(p_data, 'ds')
```

```
Shape:
(192885, 11)

Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 192885 entries, 0 to 192884
Data columns (total 11 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   ds                 192885 non-null  datetime64[ns]
 1   year               192885 non-null  int64
 2   doy                192885 non-null  int64
 3   time               192885 non-null  object
 4   y                  192885 non-null  int64
 5   humidity           192885 non-null  int64
 6   dew.point          192885 non-null  int64
 7   pressure           192885 non-null  int64
 8   wind.speed.mean    192885 non-null  int64
 9   wind.bearing.mean  192885 non-null  int64
 10  wind.speed.max     192885 non-null  int64
dtypes: datetime64[ns](1), int64(9), object(1)
memory usage: 16.2+ MB
None

Summary stats:
```

	year	doy	y	humidity	dew.point	pressure	wind.speed.mean	wind.bearing.mean
count	192885.000000	192885.000000	192885.000000	192885.000000	192885.000000	192885.000000	192885.000000	192885.000000
mean	2013.895803	186.882298	101.096819	79.239951	62.135174	1014.404153	44.588148	196.223423
std	3.283992	106.486420	64.465602	16.908724	51.016879	11.823922	40.025546	82.458390
min	2008.000000	1.000000	-138.000000	25.000000	-143.000000	963.000000	0.000000	0.000000
25%	2011.000000	94.000000	52.000000	69.000000	25.000000	1008.000000	12.000000	135.000000
50%	2014.000000	191.000000	100.000000	83.000000	64.000000	1016.000000	35.000000	225.000000
75%	2017.000000	280.000000	145.000000	92.000000	100.000000	1023.000000	67.000000	270.000000
max	2020.000000	366.000000	361.000000	100.000000	216.000000	1048.000000	291.000000	315.000000

```
Raw data:
```

	ds	year	doy	time	y	humidity	dew.point	pressure	wind.speed.mean	wind.bearing.mean	wind.speed.max
0	2008-08-01 08:30:00	2008	214	09:30:00	186	69	128	1010	123	180	280
1	2008-08-01 09:00:00	2008	214	10:00:00	191	70	135	1010	137	180	260
2	2008-08-01 09:30:00	2008	214	10:30:00	195	68	134	1010	133	180	260
	2008-08-										

Data Processing and Feature Engineering

The data must be reformatted before model building.

The following steps are necessary:

- Wind direction and speed transformation
- Impute missing data where possible
- Time conversion
- Split data
- Normalise data
- Window data

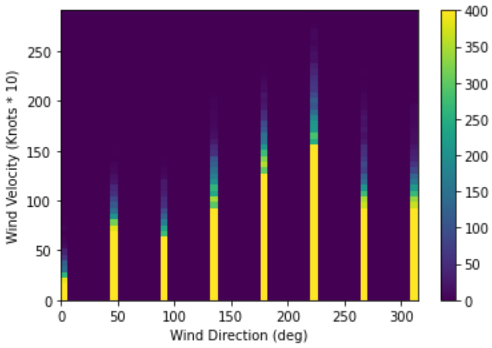
Wind direction and speed transformation

The `wind.bearing.mean` column gives wind direction in degrees but is categorised at 45 degree increments, i.e. 0, 45, 90, 135, 180, 225, 270, 315. Wind direction shouldn't matter if the wind is not blowing.

The distribution of wind direction and speed looks like this:

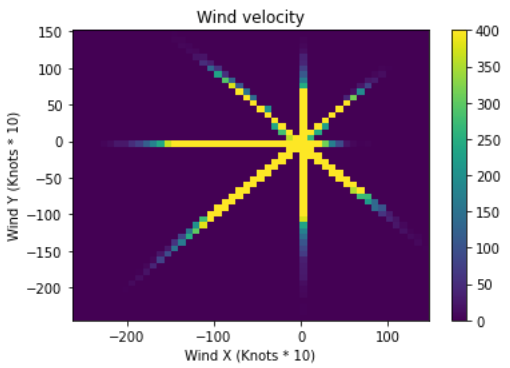
02:00:00

```
plt.hist2d(df['wind.bearing.mean'], df['wind.speed.mean'], bins = (50, 50), vmax = 400)
plt.colorbar()
plt.xlabel('Wind Direction (deg)')
plt.ylabel('Wind Velocity (Knots * 10)');
```



Convert wind direction and speed to *x* and *y* vectors, so the model can more easily interpret them.

```
wv = df['wind.speed.mean']
max_wv = df['wind.speed.max']

# Convert to radians
wd_rad = df['wind.bearing.mean'] * np.pi / 180

# Calculate the wind x and y components
df['wind.x'] = wv * np.cos(wd_rad)
df['wind.y'] = wv * np.sin(wd_rad)

# Calculate the max wind x and y components
df['max.wind.x'] = max_wv * np.cos(wd_rad)
df['max.wind.y'] = max_wv * np.sin(wd_rad)

df_orig = df

plt.hist2d(df['wind.x'], df['wind.y'], bins = (50, 50), vmax = 400)
plt.colorbar()
plt.xlabel('Wind X (Knots * 10)')
plt.ylabel('Wind Y (Knots * 10)')
plt.title('Wind velocity');
```



Better, but not ideal. Data augmentation with the [mixup] method is carried out at batch preparation time below.

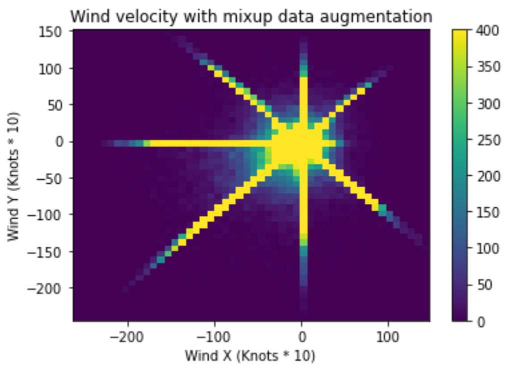
From the paper: "mixup trains a neural network on convex combinations of pairs of examples and their labels".

Further details on how I apply mixup to time series are included in the Window data section below.

Here is an illustration of the improvement in wind velocity with mixup augmentation.

```
def mixup(data, alpha = 1.0, factor = 1):
    batch_size = len(data) - 1

    data['epoch'] = data.index.astype(np.int64) // 10**9

    # random sample lambda value from beta distribution
    l   = np.random.beta(alpha, alpha, batch_size * factor)
    X_l = l.reshape(batch_size * factor, 1)

    # Get a pair of inputs and outputs
    y1  = data['y'].shift(-1).dropna()
    y1_ = pd.concat([y1] * factor)

    y2  = data['y'][0:batch_size]
    y2_ = pd.concat([y2] * factor)

    X1  = data.drop('y', 1).shift(-1).dropna()
```

```
    X1_ = pd.concat([X1] * factor)

    X2  = data.drop('y', 1)
    X2  = X2[0:batch_size]
    X2_ = pd.concat([X2] * factor)

    # Perform mixup
    X = X1_ * X_l + X2_ * (1 - X_l)
    y = y1_ * l   + y2_ * (1 - l)

    df = pd.DataFrame(y).join(X)
    df = data.append(df).sort_values('epoch', ascending = True)
    df = df.drop('epoch', 1)

    df = df.drop_duplicates(keep = False)

    return df


df_mix = mixup(df.loc[:, ['y','wind.x','wind.y']], factor = 2)
plt.hist2d(df_mix['wind.x'], df_mix['wind.y'], bins = (50, 50), vmax = 400)
plt.colorbar()
plt.xlabel('Wind X (Knots * 10)')
plt.ylabel('Wind Y (Knots * 10)')
plt.title('Wind velocity with mixup data augmentation');
```



Mixup improves the categorical legacy of the wind velocity data. Unfortunately, if outliers are present their influence will be reinforced.

Missing value interpolation

Missing data is much less of a problem for [prophet models](#) which handle it seamlessly.

Currently there are around 8,000 missing observations in approximately 600 sections or "gaps". The gaps range in length from 30 mins to 45 days.

Gaps have length across variables, i.e. if there is a gap of length 6 (3 hours) then all variables are missing for 6 consecutive observations.

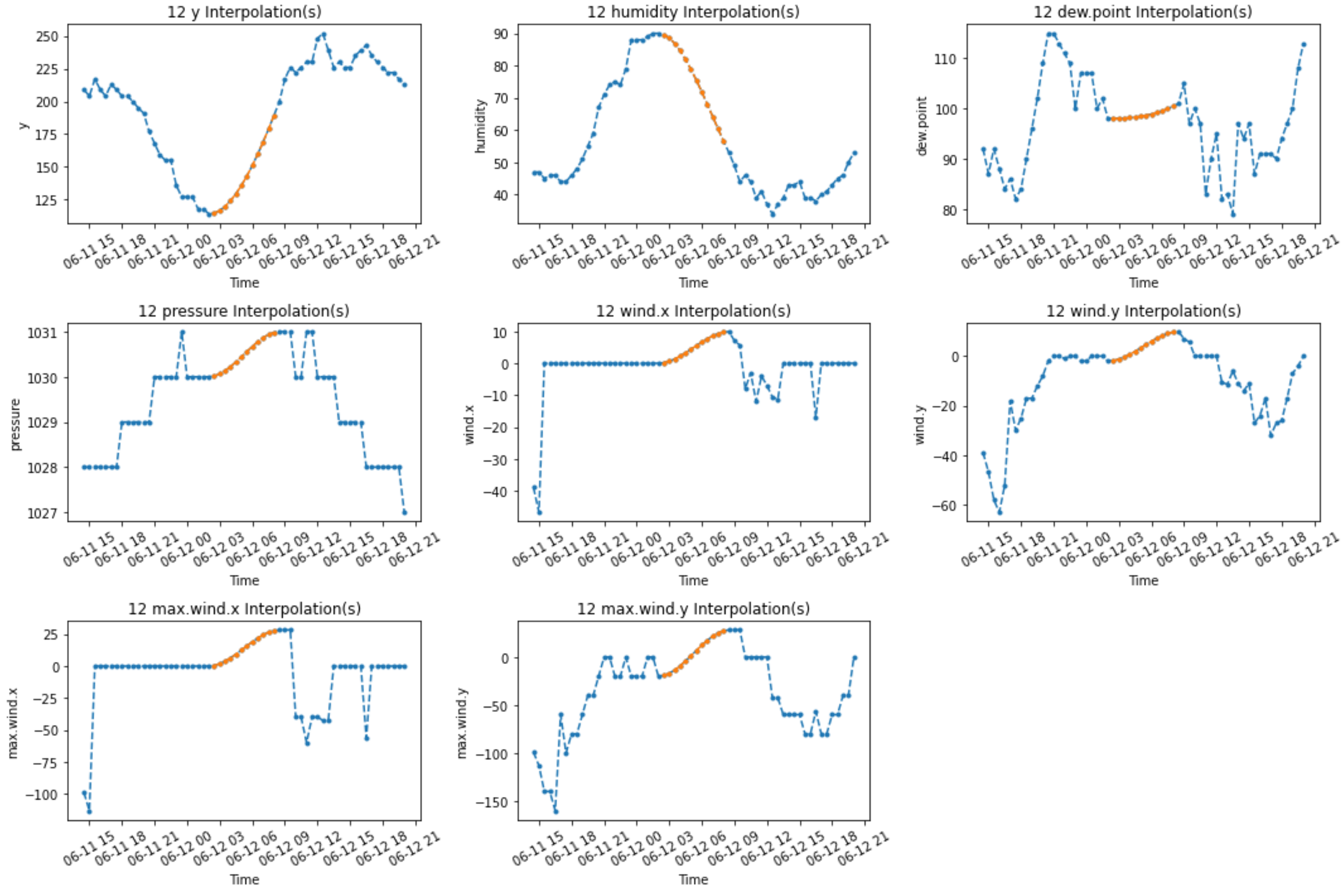
Missing observations can be [imputed](#).
Here I use a variation on [cubic interpolation](#). Vanilla cubic interpolation overshoots the data which introduces outlier values. I limit the gap length to 12 observations (6 hours).

Longer gaps will be accounted for at the train, test, validation split stage.

```
del_cols = ['doy', 'wind.bearing.mean', 'wind.speed.mean', 'wind.speed.max']
df_ts = df_orig.set_index('ds', drop = False)
df_ts.drop(del_cols, axis = 1, inplace = True)

# Add NaN values for missing observations
df_ts_na = df_ts.asfreq('30min')

# Set NaN year, time values using index
df_ts_na.ds   = df_ts_na.index
df_ts_na.year = df_ts_na.index.year
df_ts_na.time = df_ts_na.index.time

# Count number of consecutive missing values
#   There are more elegant ways to do this but they don't cope well with NaNs
#   As far as I can tell, neither numpy nor pandas have native run length encoding functions
len_holes = pd.Series([len(list(g)) for k, g in itertools.groupby(df_ts_na.y.isnull()) if k])
len_holes_long = pd.Series(list(itertools.repeat(l, l)) for l in len_holes)
len_holes_flat = pd.Series(list(itertools.chain(*len_holes_long)))

df_ts_na['missing_len'] = -100
df_ts_na['missing_len'] = df_ts_na['missing_len'].astype('Int64')
```

```python
df_ts_na.loc[df_ts_na.y.isnull(), 'missing_len'] = len_holes_flat.to_numpy()

# Mark 24 observations before and after each group of NaNs - for plotting
df_ts_na['around_nan'] = -100
df_ts_na['around_nan'] = df_ts_na['around_nan'].astype('Int64')

for i in range(-24, 25):
    df_ts_na.loc[df_ts_na.y.isna().shift(i).fillna(False), 'around_nan'] = i

df_ts_na.loc[df_ts_na.y.isna(), 'around_nan'] = 0


# Interpolate - method = 'spline' very slow :-(
#               cubic and quadratic overshoot the data and introduce outliers
limit  = 12
method = 'pchip'
for v in ['y', 'humidity', 'dew.point', 'pressure', 'wind.x', 'wind.y', 'max.wind.x', 'max.wind.y']:
    df_ts_na[v] = df_ts_na[v].interpolate(method = method, limit = limit)


# Extract missing observations and surrounding values into dict of lists for checking & plotting
# slow :-(
j = miss_len = 0
inner_list = []
miss_plus  = {}
for index, row in df_ts_na.iterrows():
    if (row['around_nan'] > -25) | (row['missing_len'] > 0):
        inner_list.append(index)
        if row['missing_len'] > 0:
            miss_len = row['missing_len']
        j  = 1
    else:
        if j == 1:
            miss_plus.setdefault(miss_len, []).append(inner_list)
            inner_list = []
        j = miss_len = 0

# print("keys: ",  len(miss_plus.keys()))
# print("sum:  ",  sum(miss_plus.keys()))
# print("keys:  ", sorted(miss_plus.keys()))
# print(len(miss_plus[29]))
# print(len(miss_plus[29][0]))
# print(miss_plus[29][0])
# df_ts_na.loc[miss_plus[29][0]]


def plot_interpolations(data):
    """Plot 8 labeled interpolation examples in 3 x 3 subplots"""

    fig, axs = plt.subplots(3, 3, figsize = (15, 10), tight_layout = True)
    axs = axs.ravel()
    i = 0

    for v in ['y', 'humidity', 'dew.point', 'pressure', 'wind.x', 'wind.y', 'max.wind.x', 'max.wind.y']:
        marks = data.loc[data.missing_len > 0, v]
        title = str(len(marks)) + ' ' + v + ' Interpolation(s)'

        axs[i].plot(data[v], marker = '.', linestyle = '--')
        axs[i].plot(marks, marker = '.', linestyle = '--')
        axs[i].xaxis.set_tick_params(rotation = 30, labelsize = 10)
        axs[i].set_title(title)
        axs[i].set_ylabel(v)
        axs[i].set_xlabel('Time')
        i += 1

    axs[i].set_visible(False)

    return None


plot_interpolations(df_ts_na.loc[miss_plus[12][1]])


# Remove gaps longer than 12
df_ts_na = df_ts_na.dropna()
drop_cols = ['missing_len', 'around_nan']
df_ts_na.drop(drop_cols, axis = 1, inplace = True)
df = df_ts_na
```

```
/usr/local/lib/python3.7/dist-packages/scipy/interpolate/_cubic.py:293: RuntimeWarning: invalid value encountered in add
  whmean = (w1/mk[:-1] + w2/mk[1:]) / (w1 + w2)
```



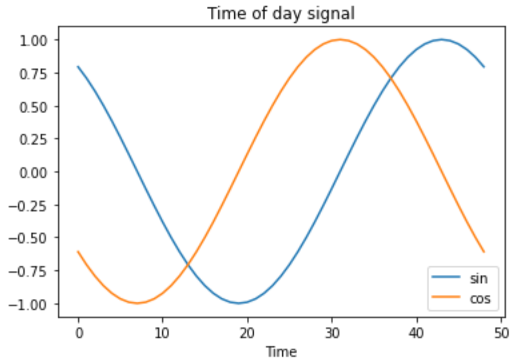
There is a 23 % reduction in missing values.

Alternative interpolation methods (piecewise methods in particular) may give more natural results for the wind and pressure variables.

Time conversion

Convert `ds` timestamps to "time of day" and "time of year" variables using `sin` and `cos`.

```
# Convert to secs
date_time   = pd.to_datetime(df['ds'], format = '%Y.%m.%d %H:%M:%S')
timestamp_s = date_time.map(datetime.datetime.timestamp)

day  = 24 * 60 * 60
year = (365.2425) * day

df['day.sin']  = np.sin(timestamp_s * (2 * np.pi / day))
df['day.cos']  = np.cos(timestamp_s * (2 * np.pi / day))
df['year.sin'] = np.sin(timestamp_s * (2 * np.pi / year))
df['year.cos'] = np.cos(timestamp_s * (2 * np.pi / year))

plt.plot(np.array(df['day.sin'])[:49])
plt.plot(np.array(df['day.cos'])[:49])
plt.xlabel('Time')
plt.legend(['sin', 'cos'], loc = 'lower right')
plt.title('Time of day signal');
```



Split data

Use data from 2018 for validation and 2019 for testing. These are entirely arbitrary choices. This results in an approximate 82%, 9%, 9% split for the training, validation, and test sets.

```
keep_cols = ['y', 'humidity', 'dew.point', 'pressure', 'wind.x', 'wind.y',
             'day.sin', 'day.cos', 'year.sin', 'year.cos']
del_cols = ['ds', 'time', 'max.wind.x', 'max.wind.y']
df.drop(del_cols, axis = 1, inplace = True)

train_df = df.loc[(df['year'] != 2018) & (df['year'] != 2019)]
valid_df = df.loc[df['year'] == 2018]
test_df  = df.loc[df['year'] == 2019]

train_df = train_df.drop('year', axis = 1)  # inplace = True gives SettingWithCopyWarning
valid_df = valid_df.drop('year', axis = 1)  # ...
test_df  = test_df.drop('year',  axis = 1)
df       = df.drop('year',       axis = 1)

print("df.drop shape: ", df.shape)
print("train shape:   ", train_df.shape)
print("valid shape:   ", valid_df.shape)
print("test shape:    ", test_df.shape)
```

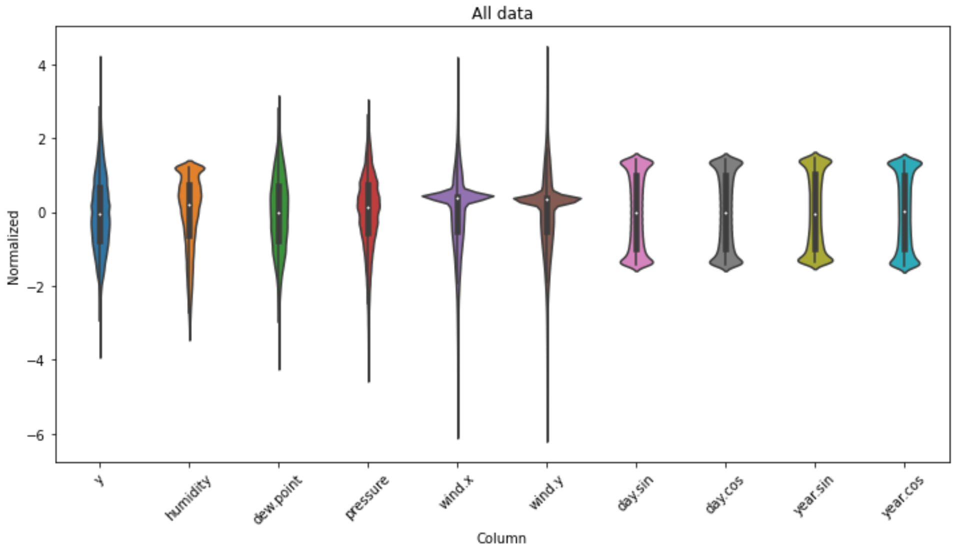
```
df.drop shape:  (194736, 10)
train shape:    (160059, 10)
valid shape:    (17236, 10)
test shape:     (17441, 10)
```

Normalise data

Features should be scaled before neural network training. Arguably, scaling should be done using moving averages to avoid accessing future values.
Instead, simple standard score normalisation will be used.

Plot violin plot to see distribution of features.

```
train_mean = train_df.mean()
train_std  = train_df.std()

train_df = (train_df - train_mean) / train_std
valid_df = (valid_df - train_mean) / train_std
test_df  = (test_df  - train_mean) / train_std

df_std = (df - train_mean) / train_std
df_std = df_std.melt(var_name = 'Column', value_name = 'Normalized')

plt.figure(figsize=(12, 6))
ax = sns.violinplot(x = 'Column', y = 'Normalized', data = df_std)
ax.set_xticklabels(df.keys(), rotation = 45)
ax.set_title('All data');
```



There may still be some outliers present but there are no glaring problems.

Window data

Models are trained using sliding windows of samples from the data.

Window parameters to consider for the [tf.keras.preprocessing.timeseries_dataset_from_array]() function:

- sequence_length:
  - Length of the output sequences (in number of timesteps), or number of **lag** observations to use
- sequence_stride:
  - Period between successive output sequences. For stride s, output samples start at index data[i], data[i + s], data[i + 2 * s] etc
  - s can include an **offset** and/or 1 or more **steps ahead** to forecast
- batch_size:
  - Number of samples in each batch
- shuffle:
  - Shuffle output samples, or use chronological order

Initial values used:

- sequence_length (aka lags): 24 (corresponds to 12 hours)
- steps ahead (what to forecast): 1 and separately 4 (corresponds to 30 mins and separately 30 mins, 60 mins, 90 mins, 120 mins)
- offset (space between lags and steps ahead): 0
- batch_size: 32
- shuffle: True for training data

The `make_dataset` function below generates [tensorflow datasets]() for:

- Lags, steps-ahead, offset, batch size and shuffle
- Optionally multiple y columns (Not extensively tested)

Stride is used to specify offset + steps-ahead. Offset will be 0 throughout this notebook.

TODO Insert figure illustrating lags, offsets and steps-ahead.

`shuffle = True` is used with train data. `shuffle = False` is used with validation and test data so the residuals can be checked for heteroscadicity.

Throughout this notebook I use a shorthand notation to describe lags and strides. For example:

- 24l_1s is 24 lags 1 step ahead
- 24l_4s is 24 lags 4 steps ahead

Mixup data augmentation

Data augmentation with [mixup: Beyond Empirical Risk Minimization]() by Zhang *et al* is used to help counter the categorical legacy from the wind bearing variable. Simple 'input mixup' is used as opposed to the batch-based mixup Zhang *et al* focus on. Input mixup has the advantage that it can be used with non-neural network methods. Mixup is performed for train and validation data separately. With current settings these datasets are approximately 3 times larger but this can be varied. Three times more training data is manageable on Colab. Test and validation data is left unmodified.

I apply mixup between consecutive observations in the time series instead of the usual random observations. This is a fairly conservative starting point. I'd be surprised if applying mixup between consecutive days of measurements didn't give better results. Applying mixup between inputs with equal temperature values will not improve performance and will increase run time.

TODO Insert couple of examples of mixup - use `plot_examples()`

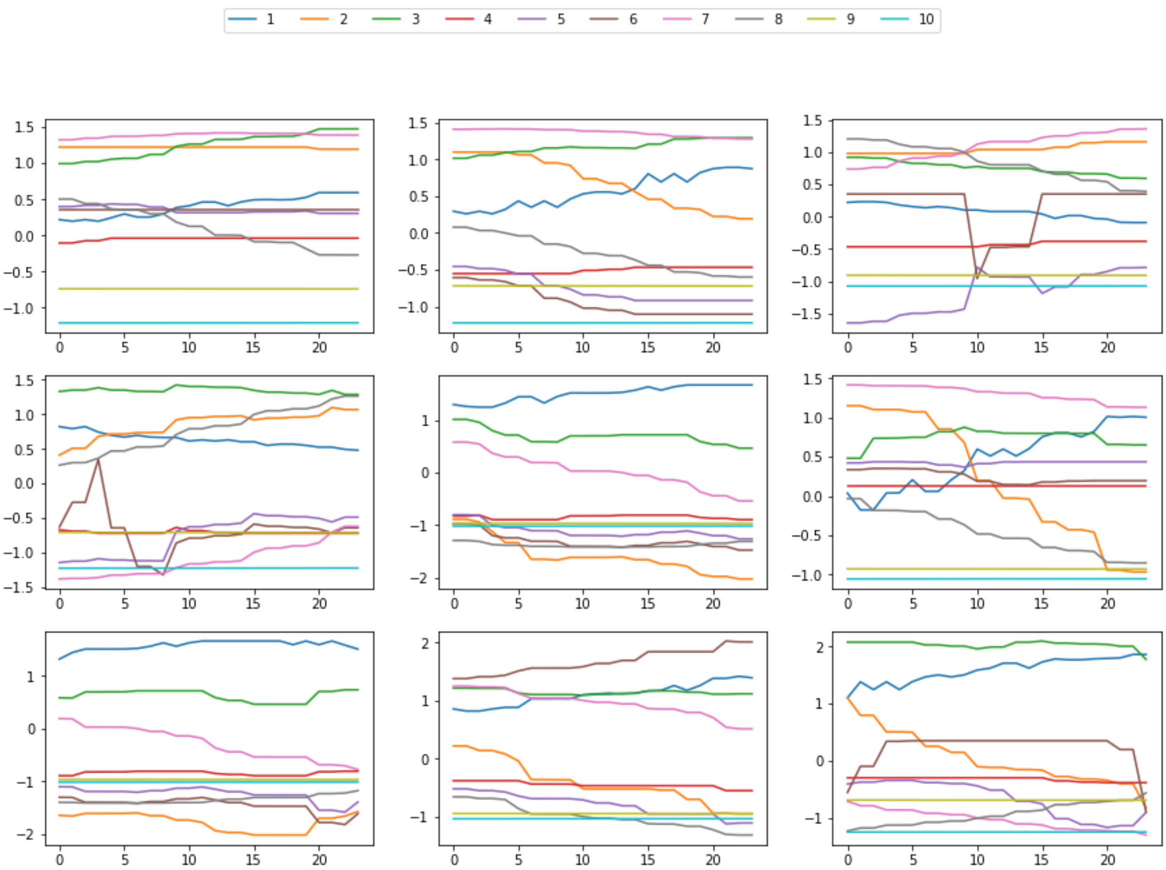
I don't show it in this notebook, but adding this data augmentation makes a big difference to loss values for all three model architectures. For example, here are comparable results for MLP, 24 largs, 1 step ahead, 20 epochs.

Augmentation	Train rmse	Train mae	Valid rmse	Valid mae
No augmentation	0.0058	0.053	0.0054	0.052
Input mixup	0.0016	0.025	0.0015	0.025

See this commit for results from other architectures without input mixup.

```
def make_dataset(data, y_cols, lags = 1, steps_ahead = 1, stride = 1, bs = 32, shuffle = False, mix = 2):
    assert stride >= steps_ahead

    total_window_size = lags + stride

    # Add NaN values for missing observations
    data = data.asfreq('30min')

    # Split data into subsets (blocks) on NaNs - SLOW - https://stackoverflow.com/a/21404655/100129
    blocks = np.split(data, np.where(np.isnan(data.y))[0])
    # Removing NaN entries
    blocks = [bl[~np.isnan(bl.y)] for bl in blocks if not isinstance(bl, np.ndarray)]
    # Removing empty DataFrames
    blocks = [bl for bl in blocks if not bl.empty]

    i = 0
    for block in blocks:
        i += 1
        if mix != 0:
            block_mix = mixup(block, factor = mix)
        else:
            block_mix = block
        block_np  = np.array(block_mix, dtype = np.float32)

        ds = tf.keras.preprocessing.timeseries_dataset_from_array(
              data     = block_np,
              targets = None,
              sequence_length = total_window_size,
              sequence_stride = 1,
              shuffle    = shuffle,
              batch_size = bs)

        col_indices = {name: i for i, name in enumerate(data.columns)}
        X_slice = slice(0, lags)
        y_start = total_window_size - steps_ahead
        y_slice = slice(y_start, None)


        def split_window(features):
            X = features[:, X_slice, :]
            y = features[:, y_slice, :]

            X = tf.stack(
                [X[:, :, col_indices[name]] for name in data.columns],
                axis = -1)
            y = tf.stack(
                [y[:, :, col_indices[name]] for name in y_cols],
                axis = -1)

            # Slicing doesn't preserve static shape information, so set the shapes manually.
            # This way the `tf.data.Datasets` are easier to inspect.
            X.set_shape([None, lags,        None])
            y.set_shape([None, steps_ahead, None])

            return X, y


        ds = ds.map(split_window)

        if i == 1:
            combined_dataset = ds
        else:
            combined_dataset = combined_dataset.concatenate(ds)

    return combined_dataset
```

```
def make_datasets(train, valid, test,
                  y_cols = 'y', lags = 1, steps_ahead = 1,
                  stride = 1, bs = 32, shuffle = False):
    ds_train = make_dataset(train, y_cols,
                            lags = lags, steps_ahead = steps_ahead,
                            stride = stride, shuffle = shuffle)
    ds_valid = make_dataset(valid, y_cols,
                            lags = lags, steps_ahead = steps_ahead,
                            stride = stride, shuffle = False)
    ds_test  = make_dataset(test,  y_cols,
                            lags = lags, steps_ahead = steps_ahead,
                            stride = stride, shuffle = False, mix = 0)

    return ds_train, ds_valid, ds_test


def dataset_sanity_checks(data, name):
    print(name, "batches: ", data.cardinality().numpy())
    for batch in data.take(1):
        print("\tX (batch_size, time, features): ", batch[0].shape)
        print("\ty (batch_size, time, features): ", batch[1].shape)
        print("\tX[0][0]: ", batch[0][0])
        print("\ty[0][0]: ", batch[1][0])


# Single step-ahead
ds = {}
bs = 32
shuffle = True
ds['train_24l_1s'], ds['valid_24l_1s'], ds['test_24l_1s'] = make_datasets(train_df,
                                                                         valid_df,
                                                                         test_df,
                                                                         lags = 24,
                                                                         shuffle = shuffle,
                                                                         bs = bs)
# dataset_sanity_checks(ds_train_4l_1s, '4l 1s train');


# 4 steps-ahead
steps = stride = 4
ds['train_24l_4s'], ds['valid_24l_4s'], ds['test_24l_4s'] = make_datasets(train_df,
                                                                         valid_df,
                                                                         test_df,
                                                                         lags = 24,
                                                                         steps_ahead = steps,
                                                                         stride = stride,
                                                                         shuffle = shuffle,
                                                                         bs = bs)

# lags = 4
# display(train_df.head(lags + steps))
# dataset_sanity_checks(ds_train_4l_4s, '4l 4s train');


# Plot 9 examples from ds['train_24l_1s']
def plot_dataset_examples(dataset):
    fig, axs = plt.subplots(3, 3, figsize = (15, 10))
    axs = axs.ravel()

    for batch in dataset.take(1):
        for i in range(9):
          x = batch[0][i].numpy()
          axs[i].plot(x)

    fig.legend(range(1, 11), loc = 'upper center',  ncol = 10);


plot_dataset_examples(ds['train_24l_1s'])
```



Model Building

Model architectures considered:

- MLP
- FCN
- ResNet
- LSTM

The architectures considered were mostly inspired by those used by Wang *et al* in Time Series Classification from Scratch with Deep Neural Networks: A Strong Baseline. They did not consider LSTMs. Initial hyperparameter settings came from Deep learning for time series classification: a review.

I'm primarily interested in "now-casting" or forecasting in the next 1 or 2 hours. The following model outputs are investigated:

- Single step ahead - 30 mins
- Multi-step ahead - 30, 60, 90 and 120 mins

The training and validation code are stored in the `compile_fit_validate` function below.

Multi-layer perceptron

It is useful to check the performance of the multi-layer perceptron (MLP) before using more sophisticated models. The MLP is described in the `build_mlp_model` function below. It deviates from the Wang *et al*/Fawaz *et al* model. Specifically, I use a `Flatten` layer for the first layer to train on multiple input lags, reduce the number of layers from 3 to 2 and reduce the number of neurons in each layer from 500 to 64.

First, check single step-ahead predictions.

```
def compile_fit_validate(model, train, valid, optimizer, epochs = 5, verbose = 2):
    # Reduces variance in results but won't eliminate it :-(
    random.seed(42)
    np.random.seed(42)
    tf.random.set_seed(42)

    if optimizer.lower() == 'adadelta':
        opt = Adadelta(lr = 1.0)
    else:
        opt = Adam(lr = 0.001)

    es = EarlyStopping(monitor = 'val_loss', mode = 'min', verbose = 1, patience = 10)
    lr = ReduceLROnPlateau(monitor = 'val_loss', factor = 0.2, patience = 5, min_lr = 0.0001)
```

```python
    model.compile(optimizer = opt, loss = 'mse', metrics = ['mae', 'mape'])
    h = model.fit(train, validation_data = valid,
                  epochs = epochs, verbose = verbose, callbacks = [es, lr])

    return h


def plot_history(h, name, epochs = 10):
    fig, axs = plt.subplots(1, 2, figsize = (9, 6), tight_layout = True)
    axs = axs.ravel()

    axs[0].plot(h.history['loss'])
    axs[0].plot(h.history['val_loss'])
    axs[0].set_title(name + ' loss')
    axs[0].set_xticklabels(range(1, epochs + 1))
    axs[0].set_xticks(range(0, epochs))
    axs[0].set_ylabel('loss')
    axs[0].set_xlabel('epoch')
    axs[0].legend(['train', 'valid'], loc = 'upper right')

    axs[1].plot(h.history['mape'])
    axs[1].plot(h.history['val_mape'])
    axs[1].set_title(name + ' mape')
    axs[1].set_xticks(range(0, epochs))
    axs[1].set_xticklabels(range(1, epochs + 1))
    axs[1].set_title(name + ' mape')
    axs[1].set_ylabel('mape')
    axs[1].set_xlabel('epoch')
    axs[1].legend(['train', 'valid'], loc = 'upper right')
    plt.show()

    return None


def print_min_loss(h, name):
    argmin_loss     = np.argmin(np.array(h.history['loss']))
    argmin_val_loss = np.argmin(np.array(h.history['val_loss']))
    min_loss        = h.history['loss'][argmin_loss]
    min_val_loss    = h.history['val_loss'][argmin_val_loss]
    mape            = h.history['mape'][argmin_loss]
    val_mape        = h.history['val_mape'][argmin_val_loss]
    mae             = h.history['mae'][argmin_loss]
    val_mae         = h.history['val_mae'][argmin_val_loss]

    txt = "{0:s} {1:s} min loss: {2:f}\tmae: {3:f}\tmape: {4:f}\tepoch: {5:d}"
    print(txt.format(name, "train", min_loss,     mae,     mape,     argmin_loss + 1))
    print(txt.format(name, "valid", min_val_loss, val_mae, val_mape, argmin_val_loss + 1))
    print()

    return None


def get_io_shapes(data):
    for batch in data.take(1):
        in_shape  = batch[0][0].shape
        out_shape = batch[1][0].shape

    return in_shape, out_shape


def build_mlp_model(name, data, neurons = 64):
    in_shape, out_shape = get_io_shapes(data)
    out_steps = out_shape[0]

    mlp = Sequential(name = name)

    mlp.add(Input(shape = in_shape))
    mlp.add(Flatten())  # Shape: (time, features) => (time*features)
    # mlp.add(Dropout(0.1))

    mlp.add(Dense(neurons, activation = 'relu'))
    # mlp.add(Dropout(0.1))

    mlp.add(Dense(neurons, activation = 'relu'))
    # mlp.add(Dropout(0.1))

    mlp.add(Dense(out_steps))
    mlp.add(Reshape([1, -1]))

    return mlp
```

```
def run_model(model, train, valid, optimizer = 'adam', epochs = 5):
    in_shape, out_shape = get_io_shapes(train)
    model_id = model.name + ' model - ' + str(in_shape[0]) + \
               ' lags ' + str(out_shape[0]) + ' steps-ahead -'

    model.summary()
    h = compile_fit_validate(model, train, valid, optimizer, epochs)
    plot_history(h, model_id, epochs)
    print_min_loss(h, model_id)

    return h


h = {}  # history
name = 'MLP'
models = {}
models['mlp_24l_1s'] = build_mlp_model(name, ds['train_24l_1s'])
models['mlp_24l_4s'] = build_mlp_model(name, ds['train_24l_4s'])
```

Learning rate finder

Leslie Smith was one of the first people to work on finding optimal learning rates for deep learning networks in [Cyclical Learning Rates for Training Neural Networks](). Jeremy Howard from [fast.ai]() popularised the learning rate finder used here.

Before building any models, I use a modified version of [Pavel Surmenok's Keras learning rate finder]() to get reasonably close to the optimal learning rate. It's a single small class which I add support for tensorflow datasets to, customise the graphics and add a simple summary function to.

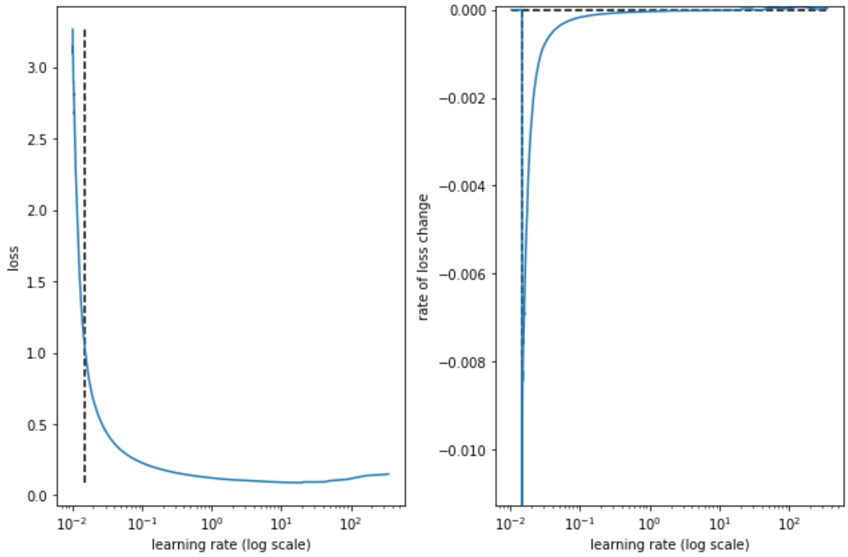
```
from keras.callbacks import LambdaCallback
import keras.backend as K
import math


class LRFinder:
    """
    Plots the change of the loss function of a Keras model when the learning rate is exponentially increasing.
    See for details:
    https://towardsdatascience.com/estimating-optimal-learning-rate-for-a-deep-neural-network-ce32f2556ce0
    """

    def __init__(self, model):
        self.model = model
        self.losses = []
        self.lrs = []
        self.best_loss = 1e9

    def on_batch_end(self, batch, logs):
        # Log the learning rate
        lr = K.get_value(self.model.optimizer.lr)
        self.lrs.append(lr)

        # Log the loss
        loss = logs['loss']
        self.losses.append(loss)

        # Check whether the loss got too large or NaN
        if batch > 5 and (math.isnan(loss) or loss > self.best_loss * 4):
            self.model.stop_training = True
            return

        if loss < self.best_loss:
            self.best_loss = loss

        # Increase the learning rate for the next batch
        lr *= self.lr_mult
        K.set_value(self.model.optimizer.lr, lr)

    def find_ds(self, train_ds, start_lr, end_lr, batch_size=64, epochs=1, **kw_fit):
        # If x_train contains data for multiple inputs, use length of the first input.
        # Assumption: the first element in the list is single input; NOT a list of inputs.
        # N = x_train[0].shape[0] if isinstance(x_train, list) else x_train.shape[0]
        N = train_ds.cardinality().numpy()

        # Compute number of batches and LR multiplier
        num_batches = epochs * N / batch_size
        self.lr_mult = (float(end_lr) / float(start_lr)) ** (float(1) / float(num_batches))
        #print(self.lr_mult)
        # Save weights into a file
```

```python
        initial_weights = self.model.get_weights()

        # Remember the original learning rate
        original_lr = K.get_value(self.model.optimizer.lr)

        # Set the initial learning rate
        K.set_value(self.model.optimizer.lr, start_lr)

        callback = LambdaCallback(on_batch_end=lambda batch, logs: self.on_batch_end(batch, logs))

        self.model.fit(train_ds,
                       batch_size=batch_size, epochs=epochs,
                       callbacks=[callback],
                       **kw_fit)

        # Restore the weights to the state before model fitting
        self.model.set_weights(initial_weights)

        # Restore the original learning rate
        K.set_value(self.model.optimizer.lr, original_lr)

    def plot_loss(self, axs, sma, n_skip_beginning=10, n_skip_end=5, x_scale='log'):
        """
        Plots the loss.
        Parameters:
            n_skip_beginning - number of batches to skip on the left.
            n_skip_end - number of batches to skip on the right.
        """
        lrs = self.lrs[n_skip_beginning:-n_skip_end]
        losses = self.losses[n_skip_beginning:-n_skip_end]
        best_lr = self.get_best_lr(sma=sma, n_skip_beginning=10, n_skip_end=5)

        axs[0].set_ylabel("loss")
        axs[0].set_xlabel("learning rate (log scale)")
        axs[0].plot(lrs, losses)
        axs[0].vlines(best_lr, np.min(losses), np.max(losses), linestyles='dashed')
        axs[0].set_xscale(x_scale)

    def plot_loss_change(self, axs, sma=1, n_skip_beginning=10, n_skip_end=5, y_lim=None):
        """
        Plots rate of change of the loss function.
        Parameters:
            axs - subplot axes
            sma - number of batches for simple moving average to smooth out the curve.
            n_skip_beginning - number of batches to skip on the left.
            n_skip_end - number of batches to skip on the right.
            y_lim - limits for the y axis.
        """
        derivatives = self.get_derivatives(sma)[n_skip_beginning:-n_skip_end]
        lrs = self.lrs[n_skip_beginning:-n_skip_end]
        best_lr = self.get_best_lr(sma=sma, n_skip_beginning=n_skip_beginning, n_skip_end=n_skip_end)
        y_min, y_max = np.min(derivatives), np.max(derivatives)
        x_min, x_max = np.min(lrs), np.max(lrs)

        axs[1].set_ylabel("rate of loss change")
        axs[1].set_xlabel("learning rate (log scale)")
        axs[1].plot(lrs, derivatives)
        axs[1].vlines(best_lr, y_min, y_max, linestyles='dashed')
        axs[1].hlines(0, x_min, x_max, linestyles='dashed')
        axs[1].set_xscale('log')
        if y_lim == None:
            axs[1].set_ylim([y_min, y_max])
        else:
            axs[1].set_ylim(y_lim)

    def get_derivatives(self, sma):
        assert sma >= 1
        derivatives = [0] * sma
        for i in range(sma, len(self.lrs)):
            derivatives.append((self.losses[i] - self.losses[i - sma]) / sma)
        return derivatives

    def get_best_lr(self, sma, n_skip_beginning=10, n_skip_end=5):
        derivatives = self.get_derivatives(sma)
        best_der_idx = np.argmin(derivatives[n_skip_beginning:-n_skip_end])
        return self.lrs[n_skip_beginning:-n_skip_end][best_der_idx]

    def summarise_lr(self, train_ds, start_lr, end_lr, batch_size=64, epochs=1, sma=1, n_skip_beginning=200, **kw_fit):
        self.find_ds(train_ds, start_lr, end_lr, batch_size, epochs)

        fig, axs = plt.subplots(1, 2, figsize = (9, 6), tight_layout = True)
        axs = axs.ravel()
```

```
        self.plot_loss(axs, sma)
        self.plot_loss_change(axs, sma=sma, n_skip_beginning=n_skip_beginning, n_skip_end=5)
        plt.show()

        best_lr = self.get_best_lr(sma=sma, n_skip_beginning=n_skip_beginning, n_skip_end=5)
        print("best lr:", best_lr, "\n")
```

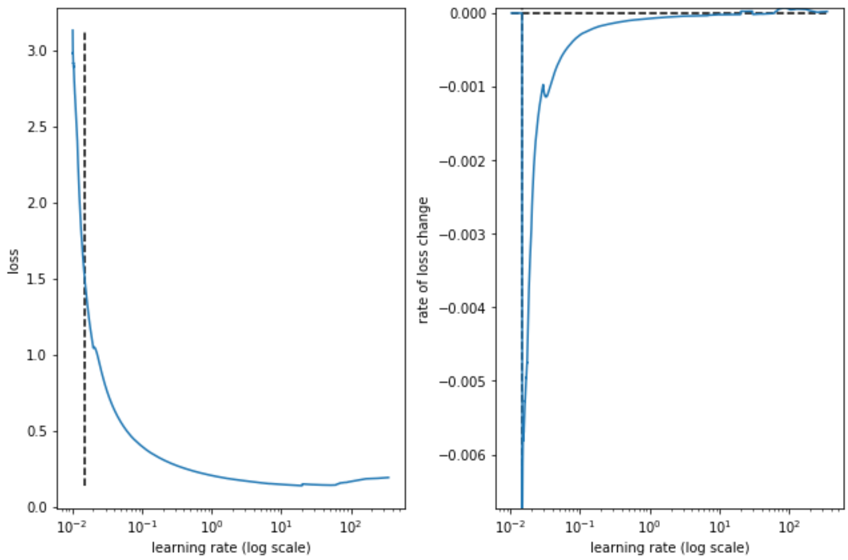
```
lrf = {}
model = models['mlp_24l_1s']
model.compile(optimizer = 'adadelta', loss = 'mse', metrics = ['mae', 'mape'])
lrf_mlp_24l_1s = LRFinder(model)
lrf_mlp_24l_1s.summarise_lr(ds['train_24l_1s'], 0.01, 1, 32, 5, 250, 25)
lrf['mlp_24l_1s'] = lrf_mlp_24l_1s

model = models['mlp_24l_4s']
model.compile(optimizer = 'adadelta', loss = 'mse', metrics = ['mae', 'mape'])
lrf_mlp_24l_4s = LRFinder(model)
lrf_mlp_24l_4s.summarise_lr(ds['train_24l_4s'], 0.01, 1, 32, 5, 250, 25)
lrf['mlp_24l_4s'] = lrf_mlp_24l_4s
```

```
Epoch 1/5
18904/18904 [==============================] - 32s 2ms/step - loss: 13.8065 - mae: 0.2728 - mape: 118.5735
```




```
best lr: 0.014789497
```

```
Epoch 1/5
18869/18869 [==============================] - 33s 2ms/step - loss: 21.2164 - mae: 0.3185 - mape: 152.5291
```




```
best lr: 0.014892996
```

The learning rate finder has a surprisingly low run time; possibly because the loss quickly becomes infinite at high learning rates.

The learning rate finder has not been very useful with any of these architectures and this data (see below for results from the other architectures). The models currently converge to the minimum loss value within 20 epochs with default learning rates. So, I default back to the accepted learning rate of 1.0 for adadelta and 0.001 for adam.

The smoothing value `sma`, is relatively high for the MLPs. It's possible to get lower rate of loss change values by using a lower `start_lr` but the rate of loss change has high variance in these regions. For MLPs learning rates in the region 0.01 to 1.0 give acceptible rates of loss change.

I leave the learning rate finder code in this notebook for possible future personal reference. It may also prove useful with other architectures and/or addition of exogenous regressors from for example the Global Forecast System model.

First, check single step-ahead model.

```
h['mlp_24l_1s'] = run_model(models['mlp_24l_1s'], ds['train_24l_1s'], ds['valid_24l_1s'], optimizer = 'adadelta', epochs = 2(
```

```
Model: "MLP"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 240)               0
_________________________________________________________________
dense_6 (Dense)              (None, 64)                15424
_________________________________________________________________
dense_7 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 65
_________________________________________________________________
reshape_2 (Reshape)          (None, 1, 1)              0
=================================================================
Total params: 19,649
Trainable params: 19,649
Non-trainable params: 0
_________________________________________________________________
Epoch 1/20
18904/18904 - 70s - loss: 0.0060 - mae: 0.0527 - mape: 26.4303 - val_loss: 0.0044 - val_mae: 0.0504 - val_mape: 21.3368
Epoch 2/20
18904/18904 - 70s - loss: 0.0029 - mae: 0.0378 - mape: 19.7297 - val_loss: 0.0023 - val_mae: 0.0339 - val_mape: 16.8933
Epoch 3/20
18904/18904 - 71s - loss: 0.0025 - mae: 0.0350 - mape: 18.3935 - val_loss: 0.0022 - val_mae: 0.0338 - val_mape: 16.4769
Epoch 4/20
18904/18904 - 69s - loss: 0.0024 - mae: 0.0337 - mape: 17.7326 - val_loss: 0.0026 - val_mae: 0.0365 - val_mape: 14.7955
Epoch 5/20
18904/18904 - 70s - loss: 0.0022 - mae: 0.0323 - mape: 17.4271 - val_loss: 0.0029 - val_mae: 0.0400 - val_mape: 15.5878
Epoch 6/20
18904/18904 - 71s - loss: 0.0021 - mae: 0.0317 - mape: 16.8221 - val_loss: 0.0028 - val_mae: 0.0401 - val_mape: 15.8053
Epoch 7/20
18904/18904 - 71s - loss: 0.0021 - mae: 0.0311 - mape: 16.7360 - val_loss: 0.0020 - val_mae: 0.0314 - val_mape: 13.9622
Epoch 8/20
18904/18904 - 69s - loss: 0.0021 - mae: 0.0310 - mape: 16.8856 - val_loss: 0.0018 - val_mae: 0.0292 - val_mape: 13.2568
Epoch 9/20
18904/18904 - 69s - loss: 0.0021 - mae: 0.0308 - mape: 16.4686 - val_loss: 0.0019 - val_mae: 0.0311 - val_mape: 13.9141
Epoch 10/20
18904/18904 - 71s - loss: 0.0020 - mae: 0.0304 - mape: 16.3769 - val_loss: 0.0019 - val_mae: 0.0304 - val_mape: 13.7852
Epoch 11/20
18904/18904 - 71s - loss: 0.0020 - mae: 0.0302 - mape: 16.4885 - val_loss: 0.0019 - val_mae: 0.0305 - val_mape: 13.2742
Epoch 12/20
18904/18904 - 69s - loss: 0.0020 - mae: 0.0301 - mape: 16.3227 - val_loss: 0.0018 - val_mae: 0.0300 - val_mape: 13.2368
Epoch 13/20
18904/18904 - 71s - loss: 0.0020 - mae: 0.0301 - mape: 16.1918 - val_loss: 0.0022 - val_mae: 0.0342 - val_mape: 14.5272
Epoch 14/20
18904/18904 - 71s - loss: 0.0016 - mae: 0.0258 - mape: 15.1422 - val_loss: 0.0015 - val_mae: 0.0259 - val_mape: 12.7123
Epoch 15/20
18904/18904 - 69s - loss: 0.0016 - mae: 0.0257 - mape: 15.1254 - val_loss: 0.0016 - val_mae: 0.0271 - val_mape: 12.8926
Epoch 16/20
18904/18904 - 70s - loss: 0.0016 - mae: 0.0257 - mape: 15.1013 - val_loss: 0.0015 - val_mae: 0.0260 - val_mape: 12.6072
Epoch 17/20
18904/18904 - 71s - loss: 0.0016 - mae: 0.0257 - mape: 15.1451 - val_loss: 0.0015 - val_mae: 0.0252 - val_mape: 12.5537
Epoch 18/20
18904/18904 - 69s - loss: 0.0016 - mae: 0.0257 - mape: 15.0979 - val_loss: 0.0016 - val_mae: 0.0265 - val_mape: 12.5819
Epoch 19/20
18904/18904 - 70s - loss: 0.0016 - mae: 0.0257 - mape: 15.0406 - val_loss: 0.0015 - val_mae: 0.0252 - val_mape: 12.5789
Epoch 20/20
18904/18904 - 71s - loss: 0.0016 - mae: 0.0252 - mape: 14.8436 - val_loss: 0.0015 - val_mae: 0.0250 - val_mape: 12.4887
```

Second, check multiple time-steps.


```
h['mlp_24l_4s'] = run_model(models['mlp_24l_4s'], ds['train_24l_4s'], ds['valid_24l_4s'], optimizer = 'adadelta', epochs = 2
```

```
Model: "MLP"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_3 (Flatten)          (None, 240)               0
_________________________________________________________________
dense_9 (Dense)              (None, 64)                15424
_________________________________________________________________
dense_10 (Dense)             (None, 64)                4160
_________________________________________________________________
dense_11 (Dense)             (None, 4)                 260
_________________________________________________________________
reshape_3 (Reshape)          (None, 1, 4)              0
=================================================================
Total params: 19,844
Trainable params: 19,844
Non-trainable params: 0
_________________________________________________________________
Epoch 1/20
18869/18869 - 70s - loss: 0.0079 - mae: 0.0616 - mape: 48.5360 - val_loss: 0.0047 - val_mae: 0.0513 - val_mape: 26.9024
Epoch 2/20
18869/18869 - 70s - loss: 0.0047 - mae: 0.0489 - mape: 39.8717 - val_loss: 0.0043 - val_mae: 0.0489 - val_mape: 28.2949
Epoch 3/20
18869/18869 - 72s - loss: 0.0042 - mae: 0.0461 - mape: 36.7081 - val_loss: 0.0038 - val_mae: 0.0442 - val_mape: 26.6455
Epoch 4/20
18869/18869 - 72s - loss: 0.0040 - mae: 0.0446 - mape: 38.5765 - val_loss: 0.0046 - val_mae: 0.0512 - val_mape: 26.2144
Epoch 5/20
18869/18869 - 70s - loss: 0.0038 - mae: 0.0438 - mape: 38.0278 - val_loss: 0.0034 - val_mae: 0.0411 - val_mape: 26.2134
Epoch 6/20
18869/18869 - 70s - loss: 0.0037 - mae: 0.0428 - mape: 39.3668 - val_loss: 0.0032 - val_mae: 0.0390 - val_mape: 23.8590
Epoch 7/20
18869/18869 - 70s - loss: 0.0037 - mae: 0.0425 - mape: 36.4837 - val_loss: 0.0033 - val_mae: 0.0401 - val_mape: 22.7384
Epoch 8/20
18869/18869 - 71s - loss: 0.0036 - mae: 0.0419 - mape: 36.1011 - val_loss: 0.0036 - val_mae: 0.0427 - val_mape: 23.8585
Epoch 9/20
18869/18869 - 71s - loss: 0.0035 - mae: 0.0416 - mape: 36.2540 - val_loss: 0.0034 - val_mae: 0.0412 - val_mape: 22.7704
Epoch 10/20
18869/18869 - 72s - loss: 0.0035 - mae: 0.0415 - mape: 34.8899 - val_loss: 0.0040 - val_mae: 0.0440 - val_mape: 24.1475
Epoch 11/20
18869/18869 - 72s - loss: 0.0035 - mae: 0.0413 - mape: 33.5526 - val_loss: 0.0036 - val_mae: 0.0440 - val_mape: 25.2787
Epoch 12/20
18869/18869 - 71s - loss: 0.0031 - mae: 0.0378 - mape: 33.7398 - val_loss: 0.0030 - val_mae: 0.0375 - val_mape: 22.4531
Epoch 13/20
18869/18869 - 72s - loss: 0.0031 - mae: 0.0377 - mape: 33.9282 - val_loss: 0.0031 - val_mae: 0.0384 - val_mape: 22.5544
Epoch 14/20
18869/18869 - 70s - loss: 0.0031 - mae: 0.0377 - mape: 34.7801 - val_loss: 0.0031 - val_mae: 0.0384 - val_mape: 22.0597
Epoch 15/20
18869/18869 - 71s - loss: 0.0031 - mae: 0.0377 - mape: 34.2808 - val_loss: 0.0031 - val_mae: 0.0386 - val_mape: 22.2223
Epoch 16/20
18869/18869 - 70s - loss: 0.0030 - mae: 0.0376 - mape: 34.3899 - val_loss: 0.0031 - val_mae: 0.0388 - val_mape: 21.9613
Epoch 17/20
18869/18869 - 70s - loss: 0.0030 - mae: 0.0376 - mape: 34.5809 - val_loss: 0.0030 - val_mae: 0.0375 - val_mape: 22.0642
Epoch 18/20
18869/18869 - 71s - loss: 0.0030 - mae: 0.0373 - mape: 33.4497 - val_loss: 0.0029 - val_mae: 0.0371 - val_mape: 21.8037
Epoch 19/20
18869/18869 - 70s - loss: 0.0030 - mae: 0.0373 - mape: 33.6596 - val_loss: 0.0029 - val_mae: 0.0369 - val_mape: 21.8714
Epoch 20/20
18869/18869 - 70s - loss: 0.0030 - mae: 0.0373 - mape: 33.5927 - val_loss: 0.0029 - val_mae: 0.0370 - val_mape: 21.8664
```

Fully convolutional network

See Time Series Classification from Scratch with Deep Neural Networks: A Strong Baseline for a detailed description of the Fully Convolutional Network (FCN) architecture. The FCN was first described in Time-series modeling with undecimated fully convolutional neural networks.

The FCN architecture is a variant of the Convolutional Neural Network (CNN). A Convolutional Neural Network (CNN) usually contains fully-connected layers or a MLP at the end of the network. The FCN does not include these final layers, so it is learning convolutional filters everywhere.

TODO Include figure comparing FCNs and CNNs

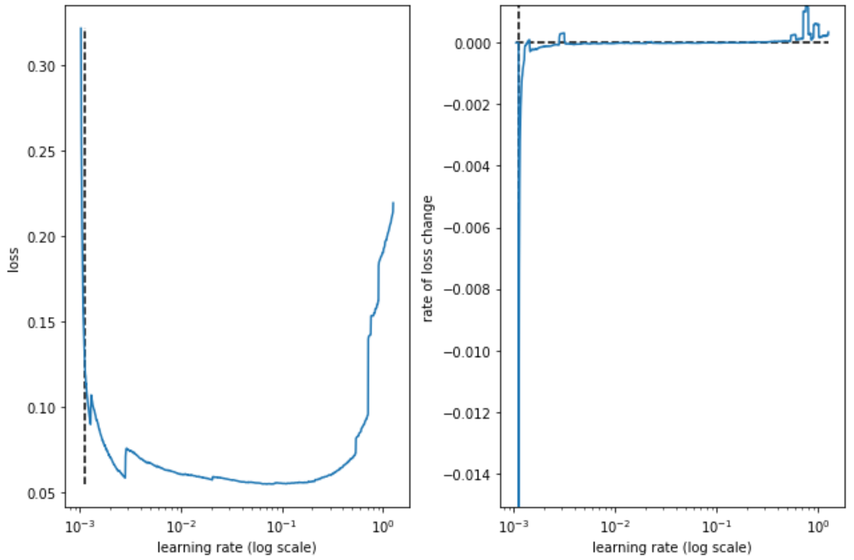
The Keras Conv1D layer is used for temporal convolution.

Next, run the learning rate finder for FCNs.

```
def build_fcn_model(name, data, n_feature_maps = 64):
    in_shape, out_shape = get_io_shapes(data)
    out_steps = out_shape[0]

    fcn = Sequential(name = name)
    fcn.add(Input(shape = in_shape))

    fcn.add(Conv1D(filters = n_feature_maps, kernel_size = 8, padding = 'same'))
    fcn.add(BatchNormalization())
    fcn.add(Activation(activation = 'relu'))

    fcn.add(Conv1D(filters = n_feature_maps, kernel_size = 5, padding = 'same'))
```

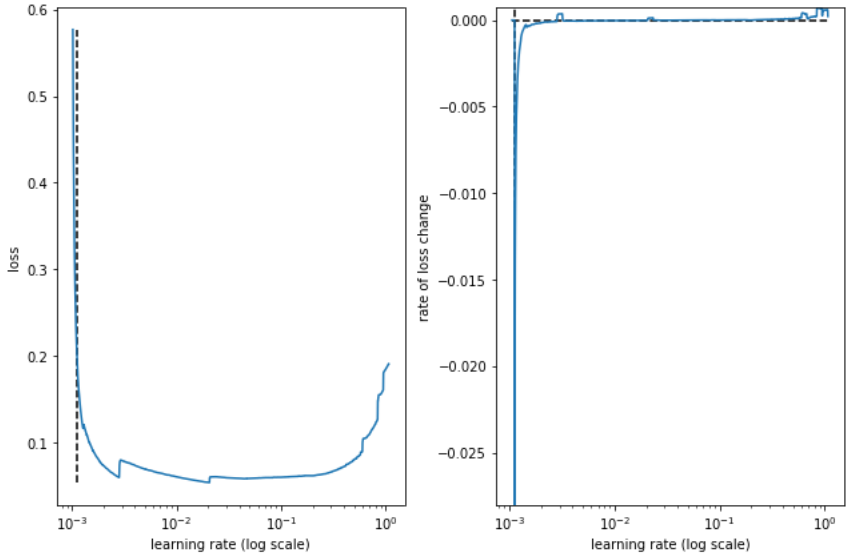
```
    fcn.add(BatchNormalization())
    fcn.add(Activation(activation = 'relu'))

    fcn.add(Conv1D(filters = n_feature_maps, kernel_size = 3, padding = 'same'))
    fcn.add(BatchNormalization())
    fcn.add(Activation(activation = 'relu'))

    fcn.add(GlobalAveragePooling1D())
    fcn.add(Dense(out_steps))

    return fcn


name = 'FCN'
models['fcn_24l_1s'] = build_fcn_model(name, ds['train_24l_1s'])
models['fcn_24l_4s'] = build_fcn_model(name, ds['train_24l_4s'])


model = models['fcn_24l_1s']
model.compile(loss = 'mse', metrics = ['mae', 'mape'])
lrf_fcn_24l_1s = LRFinder(model)
lrf_fcn_24l_1s.summarise_lr(ds['train_24l_1s'], 0.001, 1, 32, 5, 50, 25)
lrf['fcn_24l_1s'] = lrf_fcn_24l_1s

model = models['fcn_24l_4s']
model.compile(loss = 'mse', metrics = ['mae', 'mape'])
lrf_fcn_24l_4s = LRFinder(model)
lrf_fcn_24l_4s.summarise_lr(ds['train_24l_4s'], 0.001, 1, 32, 5, 50, 25)
lrf['fcn_24l_4s'] = lrf_fcn_24l_4s
```

```
Epoch 1/5
18904/18904 [==============================] - 25s 1ms/step - loss: 0.1970 - mae: 0.2458 - mape: 114.9170
```




```
best lr: 0.0011266746
```

```
Epoch 1/5
18869/18869 [==============================] - 25s 1ms/step - loss: 0.2256 - mae: 0.2510 - mape: 124.7014
```



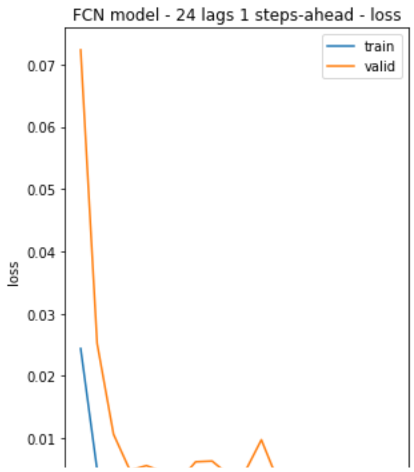

```
best lr: 0.0011242866
```

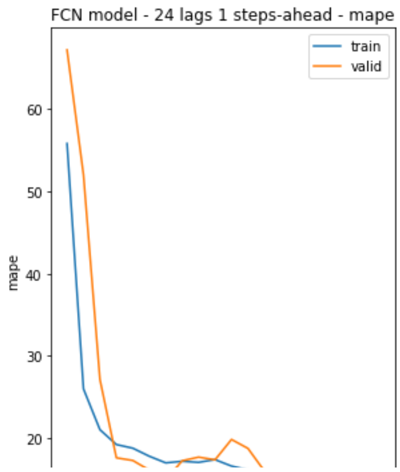
Best learning rates from the learning rate finder are close to the accepted adam learning rate of 0.001. So, I default back to that value for FCNs.

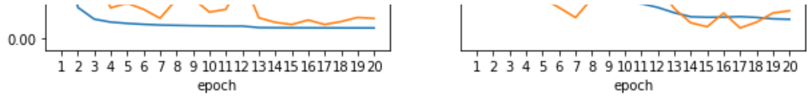
First, check single step-ahead predictions.

```
h['fcn_24l_1s'] = run_model(models['fcn_24l_1s'], ds['train_24l_1s'], ds['valid_24l_1s'], epochs = 20)
```

```
Model: "FCN"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_34 (Conv1D)           (None, 24, 64)            5184
_________________________________________________________________
batch_normalization_36 (Batc (None, 24, 64)            256
_________________________________________________________________
activation_30 (Activation)   (None, 24, 64)            0
_________________________________________________________________
conv1d_35 (Conv1D)           (None, 24, 64)            20544
_________________________________________________________________
batch_normalization_37 (Batc (None, 24, 64)            256
_________________________________________________________________
activation_31 (Activation)   (None, 24, 64)            0
_________________________________________________________________
conv1d_36 (Conv1D)           (None, 24, 64)            12352
_________________________________________________________________
batch_normalization_38 (Batc (None, 24, 64)            256
_________________________________________________________________
activation_32 (Activation)   (None, 24, 64)            0
_________________________________________________________________
global_average_pooling1d_6 ( (None, 64)                0
_________________________________________________________________
dense_18 (Dense)             (None, 1)                 65
=================================================================
Total params: 38,913
Trainable params: 38,529
Non-trainable params: 384
_________________________________________________________________
Epoch 1/20
18904/18904 - 109s - loss: 0.0244 - mae: 0.1049 - mape: 55.8223 - val_loss: 0.0724 - val_mae: 0.1911 - val_mape: 67.2142
Epoch 2/20
18904/18904 - 107s - loss: 0.0049 - mae: 0.0497 - mape: 25.9807 - val_loss: 0.0253 - val_mae: 0.1245 - val_mape: 52.0181
Epoch 3/20
18904/18904 - 109s - loss: 0.0030 - mae: 0.0393 - mape: 21.0184 - val_loss: 0.0107 - val_mae: 0.0812 - val_mape: 27.1264
Epoch 4/20
18904/18904 - 109s - loss: 0.0026 - mae: 0.0357 - mape: 19.2125 - val_loss: 0.0049 - val_mae: 0.0552 - val_mape: 17.5977
Epoch 5/20
18904/18904 - 109s - loss: 0.0024 - mae: 0.0339 - mape: 18.7675 - val_loss: 0.0056 - val_mae: 0.0568 - val_mape: 17.2739
Epoch 6/20
18904/18904 - 109s - loss: 0.0022 - mae: 0.0326 - mape: 17.7924 - val_loss: 0.0046 - val_mae: 0.0528 - val_mape: 16.1029
Epoch 7/20
18904/18904 - 109s - loss: 0.0021 - mae: 0.0315 - mape: 16.9849 - val_loss: 0.0032 - val_mae: 0.0427 - val_mape: 14.8911
Epoch 8/20
18904/18904 - 109s - loss: 0.0020 - mae: 0.0310 - mape: 17.1765 - val_loss: 0.0062 - val_mae: 0.0588 - val_mape: 17.2404
Epoch 9/20
18904/18904 - 109s - loss: 0.0020 - mae: 0.0305 - mape: 17.0491 - val_loss: 0.0063 - val_mae: 0.0585 - val_mape: 17.6799
Epoch 10/20
18904/18904 - 109s - loss: 0.0020 - mae: 0.0301 - mape: 17.3789 - val_loss: 0.0042 - val_mae: 0.0458 - val_mape: 17.3576
Epoch 11/20
18904/18904 - 109s - loss: 0.0019 - mae: 0.0297 - mape: 16.5931 - val_loss: 0.0046 - val_mae: 0.0546 - val_mape: 19.8289
Epoch 12/20
18904/18904 - 109s - loss: 0.0019 - mae: 0.0296 - mape: 16.1211 - val_loss: 0.0098 - val_mae: 0.0707 - val_mape: 18.7449
Epoch 13/20
18904/18904 - 109s - loss: 0.0017 - mae: 0.0268 - mape: 15.4780 - val_loss: 0.0033 - val_mae: 0.0426 - val_mape: 15.9895
Epoch 14/20
18904/18904 - 107s - loss: 0.0017 - mae: 0.0265 - mape: 14.9915 - val_loss: 0.0025 - val_mae: 0.0370 - val_mape: 14.2585
Epoch 15/20
18904/18904 - 108s - loss: 0.0017 - mae: 0.0264 - mape: 14.9296 - val_loss: 0.0022 - val_mae: 0.0338 - val_mape: 13.7792
Epoch 16/20
18904/18904 - 108s - loss: 0.0017 - mae: 0.0264 - mape: 14.9456 - val_loss: 0.0029 - val_mae: 0.0397 - val_mape: 15.4468
Epoch 17/20
18904/18904 - 107s - loss: 0.0016 - mae: 0.0263 - mape: 15.0056 - val_loss: 0.0022 - val_mae: 0.0338 - val_mape: 13.6309
Epoch 18/20
18904/18904 - 108s - loss: 0.0016 - mae: 0.0262 - mape: 14.9100 - val_loss: 0.0027 - val_mae: 0.0378 - val_mape: 14.3438
Epoch 19/20
18904/18904 - 108s - loss: 0.0016 - mae: 0.0262 - mape: 14.7391 - val_loss: 0.0033 - val_mae: 0.0426 - val_mape: 15.4516
Epoch 20/20
18904/18904 - 107s - loss: 0.0016 - mae: 0.0261 - mape: 14.6824 - val_loss: 0.0032 - val_mae: 0.0408 - val_mape: 15.7144
```





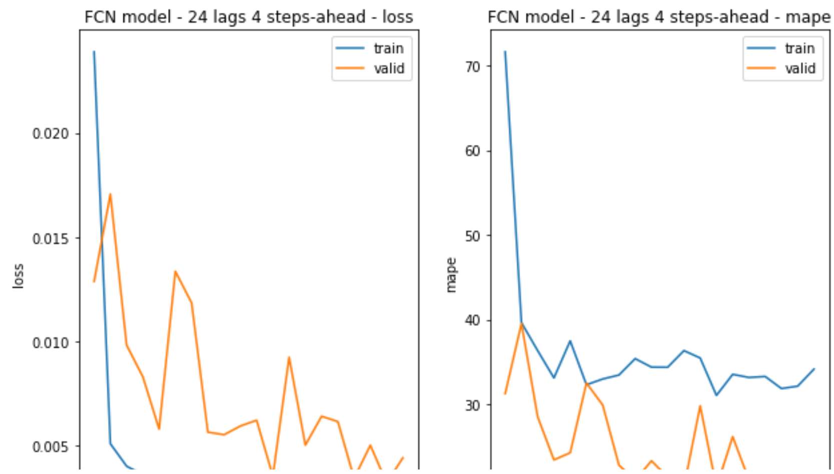


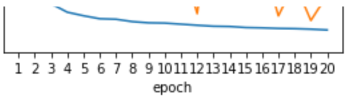

```
FCN model - 24 lags 1 steps-ahead - train min loss: 0.001631    mae: 0.026096   mape: 14.682421 epoch: 20
FCN model - 24 lags 1 steps-ahead - valid min loss: 0.002183    mae: 0.033800   mape: 13.779160 epoch: 15
```

Second, check multiple step-ahead predictions.

```
h['fcn_24l_4s'] = run_model(models['fcn_24l_4s'], ds['train_24l_4s'], ds['valid_24l_4s'], epochs = 20)
```

```
Model: "FCN"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_37 (Conv1D)           (None, 24, 64)            5184
_________________________________________________________________
batch_normalization_39 (Batc (None, 24, 64)            256
_________________________________________________________________
activation_33 (Activation)   (None, 24, 64)            0
_________________________________________________________________
conv1d_38 (Conv1D)           (None, 24, 64)            20544
_________________________________________________________________
batch_normalization_40 (Batc (None, 24, 64)            256
_________________________________________________________________
activation_34 (Activation)   (None, 24, 64)            0
_________________________________________________________________
conv1d_39 (Conv1D)           (None, 24, 64)            12352
_________________________________________________________________
batch_normalization_41 (Batc (None, 24, 64)            256
_________________________________________________________________
activation_35 (Activation)   (None, 24, 64)            0
_________________________________________________________________
global_average_pooling1d_7 ( (None, 64)                0
_________________________________________________________________
dense_19 (Dense)             (None, 4)                 260
=================================================================
Total params: 39,108
Trainable params: 38,724
Non-trainable params: 384
_________________________________________________________________
Epoch 1/20
18869/18869 - 109s - loss: 0.0239 - mae: 0.1021 - mape: 71.6267 - val_loss: 0.0129 - val_mae: 0.0876 - val_mape: 31.2475
Epoch 2/20
18869/18869 - 108s - loss: 0.0051 - mae: 0.0514 - mape: 39.6259 - val_loss: 0.0171 - val_mae: 0.1082 - val_mape: 39.5168
Epoch 3/20
18869/18869 - 108s - loss: 0.0040 - mae: 0.0452 - mape: 36.3123 - val_loss: 0.0098 - val_mae: 0.0788 - val_mape: 28.4839
Epoch 4/20
18869/18869 - 108s - loss: 0.0036 - mae: 0.0426 - mape: 33.0940 - val_loss: 0.0083 - val_mae: 0.0704 - val_mape: 23.4077
Epoch 5/20
18869/18869 - 108s - loss: 0.0034 - mae: 0.0414 - mape: 37.4728 - val_loss: 0.0058 - val_mae: 0.0586 - val_mape: 24.2517
Epoch 6/20
18869/18869 - 108s - loss: 0.0033 - mae: 0.0403 - mape: 32.2820 - val_loss: 0.0134 - val_mae: 0.0918 - val_mape: 32.4899
Epoch 7/20
18869/18869 - 108s - loss: 0.0033 - mae: 0.0400 - mape: 32.9658 - val_loss: 0.0118 - val_mae: 0.0901 - val_mape: 29.8904
Epoch 8/20
18869/18869 - 109s - loss: 0.0032 - mae: 0.0392 - mape: 33.4380 - val_loss: 0.0056 - val_mae: 0.0567 - val_mape: 22.7888
Epoch 9/20
18869/18869 - 108s - loss: 0.0031 - mae: 0.0388 - mape: 35.3697 - val_loss: 0.0055 - val_mae: 0.0578 - val_mape: 21.0027
Epoch 10/20
18869/18869 - 108s - loss: 0.0031 - mae: 0.0386 - mape: 34.3711 - val_loss: 0.0059 - val_mae: 0.0591 - val_mape: 23.2947
Epoch 11/20
18869/18869 - 108s - loss: 0.0030 - mae: 0.0382 - mape: 34.3594 - val_loss: 0.0062 - val_mae: 0.0610 - val_mape: 21.3616
Epoch 12/20
18869/18869 - 108s - loss: 0.0030 - mae: 0.0379 - mape: 36.3199 - val_loss: 0.0035 - val_mae: 0.0430 - val_mape: 20.1916
Epoch 13/20
18869/18869 - 108s - loss: 0.0029 - mae: 0.0375 - mape: 35.4463 - val_loss: 0.0092 - val_mae: 0.0798 - val_mape: 29.7993
Epoch 14/20
18869/18869 - 108s - loss: 0.0029 - mae: 0.0374 - mape: 31.0279 - val_loss: 0.0050 - val_mae: 0.0547 - val_mape: 20.1532
Epoch 15/20
18869/18869 - 108s - loss: 0.0029 - mae: 0.0370 - mape: 33.5142 - val_loss: 0.0064 - val_mae: 0.0633 - val_mape: 26.1554
Epoch 16/20
18869/18869 - 108s - loss: 0.0029 - mae: 0.0369 - mape: 33.1496 - val_loss: 0.0062 - val_mae: 0.0615 - val_mape: 21.0425
Epoch 17/20
18869/18869 - 108s - loss: 0.0028 - mae: 0.0367 - mape: 33.2664 - val_loss: 0.0034 - val_mae: 0.0431 - val_mape: 19.5610
Epoch 18/20
18869/18869 - 108s - loss: 0.0028 - mae: 0.0366 - mape: 31.8402 - val_loss: 0.0050 - val_mae: 0.0530 - val_mape: 20.5192
Epoch 19/20
18869/18869 - 108s - loss: 0.0028 - mae: 0.0364 - mape: 32.1231 - val_loss: 0.0032 - val_mae: 0.0406 - val_mape: 19.7312
Epoch 20/20
18869/18869 - 108s - loss: 0.0028 - mae: 0.0361 - mape: 34.1318 - val_loss: 0.0044 - val_mae: 0.0495 - val_mape: 20.6855
```






```
FCN model - 24 lags 4 steps-ahead - train min loss: 0.002759    mae: 0.036098    mape: 34.131786 epoch: 20
FCN model - 24 lags 4 steps-ahead - valid min loss: 0.003212    mae: 0.040593    mape: 19.731224 epoch: 19
```

Residual network

Residual networks, or ResNets, were originally proposed in Deep Residual Learning for Image Recognition.

Residual neural networks use "identity shortcut connections" to skip over some layers. Typical ResNet models are implemented with blocks of layers that contain nonlinearities (ReLU) and batch normalization. Skipping over layers may avoid the problem of vanishing gradients, by reusing activations from a previous layer until the adjacent layer learns its weights. This should allow training networks with more layers.

TODO Include basic ResNet diagram

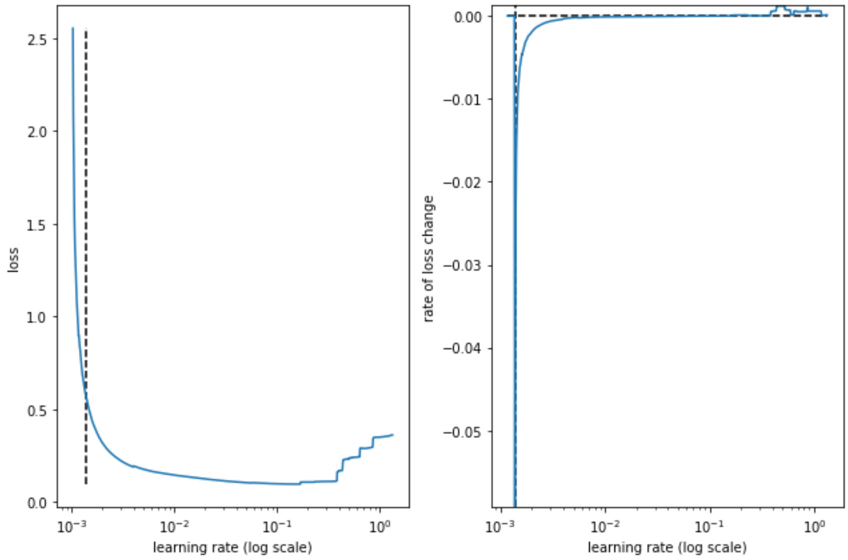
Again, the Keras Conv1D layer is used for temporal convolution.

Next, run the learning rate finder.

```python
def build_resnet_model(name, data, n_feature_maps = 64):
    in_shape, out_shape = get_io_shapes(data)
    out_steps = out_shape[0]

    input_layer = keras.layers.Input(in_shape)

    # BLOCK 1
    conv_x = keras.layers.Conv1D(filters = n_feature_maps, kernel_size = 8, padding = 'same')(input_layer)
    conv_x = keras.layers.BatchNormalization()(conv_x)
    conv_x = keras.layers.Activation('relu')(conv_x)

    conv_y = keras.layers.Conv1D(filters = n_feature_maps, kernel_size = 5, padding = 'same')(conv_x)
    conv_y = keras.layers.BatchNormalization()(conv_y)
    conv_y = keras.layers.Activation('relu')(conv_y)

    conv_z = keras.layers.Conv1D(filters = n_feature_maps, kernel_size = 3, padding='same')(conv_y)
    conv_z = keras.layers.BatchNormalization()(conv_z)

    # expand channels for the sum
    shortcut_y = keras.layers.Conv1D(filters = n_feature_maps, kernel_size = 1, padding = 'same')(input_layer)
    shortcut_y = keras.layers.BatchNormalization()(shortcut_y)

    output_block_1 = keras.layers.add([shortcut_y, conv_z])
    output_block_1 = keras.layers.Activation('relu')(output_block_1)


    # BLOCK 2
    conv_x = keras.layers.Conv1D(filters = n_feature_maps * 2, kernel_size = 8, padding = 'same')(output_block_1)
    conv_x = keras.layers.BatchNormalization()(conv_x)
    conv_x = keras.layers.Activation('relu')(conv_x)

    conv_y = keras.layers.Conv1D(filters = n_feature_maps * 2, kernel_size = 5, padding = 'same')(conv_x)
    conv_y = keras.layers.BatchNormalization()(conv_y)
    conv_y = keras.layers.Activation('relu')(conv_y)

    conv_z = keras.layers.Conv1D(filters = n_feature_maps * 2, kernel_size = 3, padding = 'same')(conv_y)
    conv_z = keras.layers.BatchNormalization()(conv_z)

    # expand channels for the sum
    shortcut_y = keras.layers.Conv1D(filters = n_feature_maps * 2, kernel_size = 1, padding = 'same')(output_block_1)
    shortcut_y = keras.layers.BatchNormalization()(shortcut_y)

    output_block_2 = keras.layers.add([shortcut_y, conv_z])
    output_block_2 = keras.layers.Activation('relu')(output_block_2)


    # BLOCK 3
    conv_x = keras.layers.Conv1D(filters = n_feature_maps * 2, kernel_size = 8, padding = 'same')(output_block_2)
    conv_x = keras.layers.BatchNormalization()(conv_x)
    conv_x = keras.layers.Activation('relu')(conv_x)

    conv_y = keras.layers.Conv1D(filters = n_feature_maps * 2, kernel_size = 5, padding = 'same')(conv_x)
    conv_y = keras.layers.BatchNormalization()(conv_y)
    conv_y = keras.layers.Activation('relu')(conv_y)

    conv_z = keras.layers.Conv1D(filters = n_feature_maps * 2, kernel_size = 3, padding = 'same')(conv_y)
    conv_z = keras.layers.BatchNormalization()(conv_z)
```

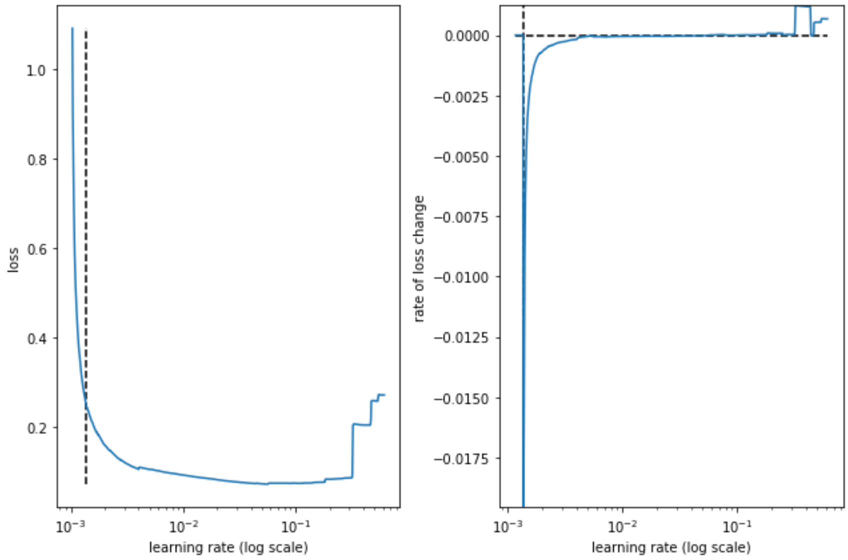
```
    # no need to expand channels because they are equal
    shortcut_y = keras.layers.BatchNormalization()(output_block_2)

    output_block_3 = keras.layers.add([shortcut_y, conv_z])
    output_block_3 = keras.layers.Activation('relu')(output_block_3)

    # FINAL
    gap_layer = keras.layers.GlobalAveragePooling1D()(output_block_3)
    output_layer = keras.layers.Dense(out_steps)(gap_layer)
    resnet = keras.models.Model(name = name, inputs = input_layer, outputs = output_layer)

    return resnet


name = 'ResNet'
models['resnet_24l_1s'] = build_resnet_model(name, ds['train_24l_1s'])
models['resnet_24l_4s'] = build_resnet_model(name, ds['train_24l_4s'])

model = models['resnet_24l_1s']
model.compile(loss = 'mse', metrics = ['mae', 'mape'])
lrf_resnet_24l_1s = LRFinder(model)
lrf_resnet_24l_1s.summarise_lr(ds['train_24l_1s'], 0.001, 10, 32, 5, 100, 50)
lrf['resnet_24l_1s'] = lrf_resnet_24l_1s

model = models['resnet_24l_4s']
model.compile(loss = 'mse', metrics = ['mae', 'mape'])
lrf_resnet_24l_4s = LRFinder(model)
lrf_resnet_24l_4s.summarise_lr(ds['train_24l_4s'], 0.001, 10, 32, 5, 100, 50)
lrf['resnet_24l_4s'] = lrf_resnet_24l_4s
```

```
Epoch 1/5
18904/18904 [==============================] - 75s 2ms/step - loss: 0.5768 - mae: 0.3080 - mape: 143.1178
```




```
best lr: 0.0013744514
```

```
Epoch 1/5
18869/18869 [==============================] - 40s 2ms/step - loss: 0.3239 - mae: 0.2483 - mape: 124.0002
```




```
best lr: 0.0013752629
```

Best learning rates from the learning rate finder are close to the accepted learning rate of 0.001. So, I default back to that value for resnets.

First, check single step-ahead predictions.

```
h['resnet_24l_1s'] = run_model(models['resnet_24l_1s'], ds['train_24l_1s'], ds['valid_24l_1s'], epochs = 20)
```

```
Model: "ResNet"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_7 (InputLayer)            [(None, 24, 10)]     0
__________________________________________________________________________________________________
conv1d_22 (Conv1D)              (None, 24, 64)       5184        input_7[0][0]
__________________________________________________________________________________________________
batch_normalization_24 (BatchNo (None, 24, 64)       256         conv1d_22[0][0]
__________________________________________________________________________________________________
activation_18 (Activation)      (None, 24, 64)       0           batch_normalization_24[0][0]
__________________________________________________________________________________________________
conv1d_23 (Conv1D)              (None, 24, 64)       20544       activation_18[0][0]
__________________________________________________________________________________________________
batch_normalization_25 (BatchNo (None, 24, 64)       256         conv1d_23[0][0]
__________________________________________________________________________________________________
activation_19 (Activation)      (None, 24, 64)       0           batch_normalization_25[0][0]
__________________________________________________________________________________________________
conv1d_25 (Conv1D)              (None, 24, 64)       704         input_7[0][0]
__________________________________________________________________________________________________
conv1d_24 (Conv1D)              (None, 24, 64)       12352       activation_19[0][0]
__________________________________________________________________________________________________
batch_normalization_27 (BatchNo (None, 24, 64)       256         conv1d_25[0][0]
__________________________________________________________________________________________________
batch_normalization_26 (BatchNo (None, 24, 64)       256         conv1d_24[0][0]
__________________________________________________________________________________________________
add_6 (Add)                     (None, 24, 64)       0           batch_normalization_27[0][0]
                                                                 batch_normalization_26[0][0]
__________________________________________________________________________________________________
activation_20 (Activation)      (None, 24, 64)       0           add_6[0][0]
__________________________________________________________________________________________________
conv1d_26 (Conv1D)              (None, 24, 128)      65664       activation_20[0][0]
__________________________________________________________________________________________________
batch_normalization_28 (BatchNo (None, 24, 128)      512         conv1d_26[0][0]
__________________________________________________________________________________________________
activation_21 (Activation)      (None, 24, 128)      0           batch_normalization_28[0][0]
__________________________________________________________________________________________________
conv1d_27 (Conv1D)              (None, 24, 128)      82048       activation_21[0][0]
__________________________________________________________________________________________________
batch_normalization_29 (BatchNo (None, 24, 128)      512         conv1d_27[0][0]
__________________________________________________________________________________________________
activation_22 (Activation)      (None, 24, 128)      0           batch_normalization_29[0][0]
__________________________________________________________________________________________________
conv1d_29 (Conv1D)              (None, 24, 128)      8320        activation_20[0][0]
__________________________________________________________________________________________________
conv1d_28 (Conv1D)              (None, 24, 128)      49280       activation_22[0][0]
__________________________________________________________________________________________________
batch_normalization_31 (BatchNo (None, 24, 128)      512         conv1d_29[0][0]
__________________________________________________________________________________________________
batch_normalization_30 (BatchNo (None, 24, 128)      512         conv1d_28[0][0]
__________________________________________________________________________________________________
add_7 (Add)                     (None, 24, 128)      0           batch_normalization_31[0][0]
                                                                 batch_normalization_30[0][0]
__________________________________________________________________________________________________
activation_23 (Activation)      (None, 24, 128)      0           add_7[0][0]
__________________________________________________________________________________________________
conv1d_30 (Conv1D)              (None, 24, 128)      131200      activation_23[0][0]
__________________________________________________________________________________________________
```

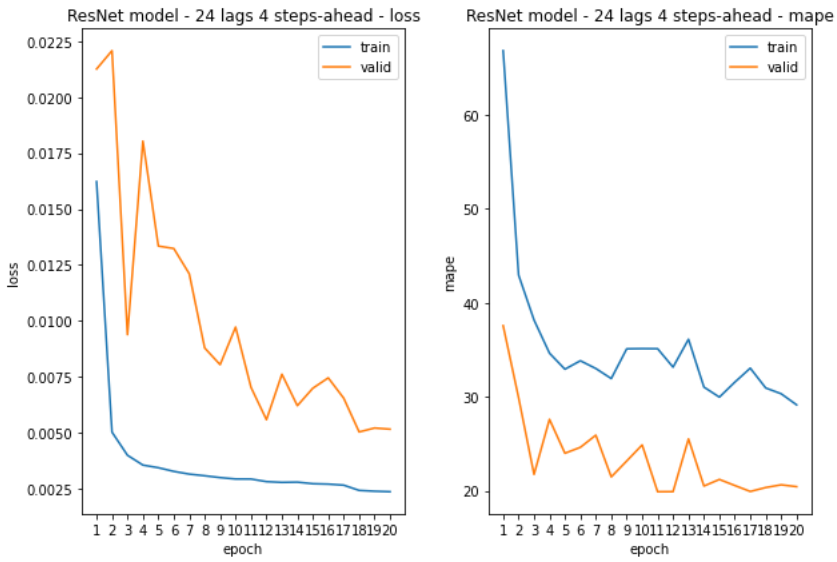
Second, check multiple step-ahead predictions.

```
activation_24 (Activation)      (None, 24, 128)      0           batch_normalization_32[0][0]
```

```
h['resnet_24l_4s'] = run_model(models['resnet_24l_4s'], ds['train_24l_4s'], ds['valid_24l_4s'], epochs = 20)
```

```
Model: "ResNet"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_8 (InputLayer)            [(None, 24, 10)]     0
__________________________________________________________________________________________________
conv1d_33 (Conv1D)              (None, 24, 64)       5184        input_8[0][0]
__________________________________________________________________________________________________
batch_normalization_36 (BatchNo (None, 24, 64)       256         conv1d_33[0][0]
__________________________________________________________________________________________________
activation_27 (Activation)      (None, 24, 64)       0           batch_normalization_36[0][0]
__________________________________________________________________________________________________
conv1d_34 (Conv1D)              (None, 24, 64)       20544       activation_27[0][0]
__________________________________________________________________________________________________
batch_normalization_37 (BatchNo (None, 24, 64)       256         conv1d_34[0][0]
__________________________________________________________________________________________________
activation_28 (Activation)      (None, 24, 64)       0           batch_normalization_37[0][0]
__________________________________________________________________________________________________
conv1d_36 (Conv1D)              (None, 24, 64)       704         input_8[0][0]
__________________________________________________________________________________________________
conv1d_35 (Conv1D)              (None, 24, 64)       12352       activation_28[0][0]
__________________________________________________________________________________________________
batch_normalization_39 (BatchNo (None, 24, 64)       256         conv1d_36[0][0]
__________________________________________________________________________________________________
batch_normalization_38 (BatchNo (None, 24, 64)       256         conv1d_35[0][0]
__________________________________________________________________________________________________
add_9 (Add)                     (None, 24, 64)       0           batch_normalization_39[0][0]
                                                                 batch_normalization_38[0][0]
__________________________________________________________________________________________________
activation_29 (Activation)      (None, 24, 64)       0           add_9[0][0]
__________________________________________________________________________________________________
conv1d_37 (Conv1D)              (None, 24, 128)      65664       activation_29[0][0]
__________________________________________________________________________________________________
batch_normalization_40 (BatchNo (None, 24, 128)      512         conv1d_37[0][0]
__________________________________________________________________________________________________
activation_30 (Activation)      (None, 24, 128)      0           batch_normalization_40[0][0]
__________________________________________________________________________________________________
conv1d_38 (Conv1D)              (None, 24, 128)      82048       activation_30[0][0]
__________________________________________________________________________________________________
batch_normalization_41 (BatchNo (None, 24, 128)      512         conv1d_38[0][0]
__________________________________________________________________________________________________
activation_31 (Activation)      (None, 24, 128)      0           batch_normalization_41[0][0]
__________________________________________________________________________________________________
conv1d_40 (Conv1D)              (None, 24, 128)      8320        activation_29[0][0]
__________________________________________________________________________________________________
conv1d_39 (Conv1D)              (None, 24, 128)      49280       activation_31[0][0]
__________________________________________________________________________________________________
batch_normalization_43 (BatchNo (None, 24, 128)      512         conv1d_40[0][0]
__________________________________________________________________________________________________
batch_normalization_42 (BatchNo (None, 24, 128)      512         conv1d_39[0][0]
__________________________________________________________________________________________________
add_10 (Add)                    (None, 24, 128)      0           batch_normalization_43[0][0]
                                                                 batch_normalization_42[0][0]
__________________________________________________________________________________________________
activation_32 (Activation)      (None, 24, 128)      0           add_10[0][0]
__________________________________________________________________________________________________
conv1d_41 (Conv1D)              (None, 24, 128)      131200      activation_32[0][0]
__________________________________________________________________________________________________
batch_normalization_44 (BatchNo (None, 24, 128)      512         conv1d_41[0][0]
__________________________________________________________________________________________________
activation_33 (Activation)      (None, 24, 128)      0           batch_normalization_44[0][0]
__________________________________________________________________________________________________
conv1d_42 (Conv1D)              (None, 24, 128)      82048       activation_33[0][0]
__________________________________________________________________________________________________
batch_normalization_45 (BatchNo (None, 24, 128)      512         conv1d_42[0][0]
__________________________________________________________________________________________________
activation_34 (Activation)      (None, 24, 128)      0           batch_normalization_45[0][0]
__________________________________________________________________________________________________
conv1d_43 (Conv1D)              (None, 24, 128)      49280       activation_34[0][0]
__________________________________________________________________________________________________
batch_normalization_47 (BatchNo (None, 24, 128)      512         activation_32[0][0]
__________________________________________________________________________________________________
batch_normalization_46 (BatchNo (None, 24, 128)      512         conv1d_43[0][0]
__________________________________________________________________________________________________
add_11 (Add)                    (None, 24, 128)      0           batch_normalization_47[0][0]
                                                                 batch_normalization_46[0][0]
__________________________________________________________________________________________________
activation_35 (Activation)      (None, 24, 128)      0           add_11[0][0]
__________________________________________________________________________________________________
global_average_pooling1d_3 (Glo (None, 128)          0           activation_35[0][0]
__________________________________________________________________________________________________
dense_15 (Dense)                (None, 4)            516         global_average_pooling1d_3[0][0]
==================================================================================================
Total params: 512,260
Trainable params: 509,700
Non-trainable params: 2,560
__________________________________________________________________________________________________
Epoch 1/20
18869/18869 - 257s - loss: 0.0162 - mae: 0.0850 - mape: 66.8201 - val_loss: 0.0213 - val_mae: 0.1162 - val_mape: 37.5702
Epoch 2/20
```

```
18869/18869 - 256s - loss: 0.0050 - mae: 0.0507 - mape: 42.9599 - val_loss: 0.0221 - val_mae: 0.1123 - val_mape: 29.9136
Epoch 3/20
18869/18869 - 258s - loss: 0.0040 - mae: 0.0449 - mape: 38.1732 - val_loss: 0.0094 - val_mae: 0.0741 - val_mape: 21.7524
Epoch 4/20
18869/18869 - 257s - loss: 0.0036 - mae: 0.0422 - mape: 34.6486 - val_loss: 0.0180 - val_mae: 0.1072 - val_mape: 27.6115
Epoch 5/20
18869/18869 - 257s - loss: 0.0034 - mae: 0.0412 - mape: 32.9408 - val_loss: 0.0133 - val_mae: 0.0916 - val_mape: 24.0021
Epoch 6/20
18869/18869 - 258s - loss: 0.0033 - mae: 0.0400 - mape: 33.8418 - val_loss: 0.0132 - val_mae: 0.0891 - val_mape: 24.6326
Epoch 7/20
18869/18869 - 255s - loss: 0.0031 - mae: 0.0392 - mape: 33.0154 - val_loss: 0.0121 - val_mae: 0.0817 - val_mape: 25.9249
Epoch 8/20
18869/18869 - 257s - loss: 0.0031 - mae: 0.0386 - mape: 31.9466 - val_loss: 0.0088 - val_mae: 0.0711 - val_mape: 21.4855
Epoch 9/20
18869/18869 - 258s - loss: 0.0030 - mae: 0.0380 - mape: 35.1108 - val_loss: 0.0080 - val_mae: 0.0680 - val_mape: 23.1770
Epoch 10/20
18869/18869 - 254s - loss: 0.0029 - mae: 0.0375 - mape: 35.1308 - val_loss: 0.0097 - val_mae: 0.0741 - val_mape: 24.8891
Epoch 11/20
18869/18869 - 256s - loss: 0.0029 - mae: 0.0374 - mape: 35.1222 - val_loss: 0.0070 - val_mae: 0.0632 - val_mape: 19.9127
Epoch 12/20
18869/18869 - 257s - loss: 0.0028 - mae: 0.0366 - mape: 33.1597 - val_loss: 0.0056 - val_mae: 0.0561 - val_mape: 19.9193
Epoch 13/20
18869/18869 - 256s - loss: 0.0028 - mae: 0.0364 - mape: 36.1271 - val_loss: 0.0076 - val_mae: 0.0669 - val_mape: 25.5291
Epoch 14/20
18869/18869 - 254s - loss: 0.0028 - mae: 0.0364 - mape: 31.0407 - val_loss: 0.0062 - val_mae: 0.0608 - val_mape: 20.5167
Epoch 15/20
18869/18869 - 255s - loss: 0.0027 - mae: 0.0359 - mape: 29.9716 - val_loss: 0.0070 - val_mae: 0.0643 - val_mape: 21.2172
Epoch 16/20
18869/18869 - 256s - loss: 0.0027 - mae: 0.0357 - mape: 31.5652 - val_loss: 0.0075 - val_mae: 0.0673 - val_mape: 20.5711
Epoch 17/20
18869/18869 - 255s - loss: 0.0027 - mae: 0.0353 - mape: 33.0526 - val_loss: 0.0065 - val_mae: 0.0627 - val_mape: 19.9382
Epoch 18/20
18869/18869 - 257s - loss: 0.0024 - mae: 0.0331 - mape: 30.9467 - val_loss: 0.0050 - val_mae: 0.0545 - val_mape: 20.3570
Epoch 19/20
18869/18869 - 257s - loss: 0.0024 - mae: 0.0328 - mape: 30.3295 - val_loss: 0.0052 - val_mae: 0.0557 - val_mape: 20.6455
Epoch 20/20
18869/18869 - 257s - loss: 0.0024 - mae: 0.0326 - mape: 29.1513 - val_loss: 0.0052 - val_mae: 0.0553 - val_mape: 20.4484
```




```
ResNet model - 24 lags 4 steps-ahead - train min loss: 0.002362 mae: 0.032584   mape: 29.151272 epoch: 20
ResNet model - 24 lags 4 steps-ahead - valid min loss: 0.005034 mae: 0.054451   mape: 20.356956 epoch: 18
```

LSTM network

Long Short Term Memory networks, or LSTMs, were originally proposed in [LONG SHORT TERM MEMORY](). They are recurrent neural networks which have feedback connections.

LSTMs can take entire sequences of data as input and keep track of arbitrary long-term dependencies. A LSTM unit is composed of a cell and three gates. The cell remembers values over arbitrary time intervals and the input, output and forget gates regulate the flow of information into and out of the cell.

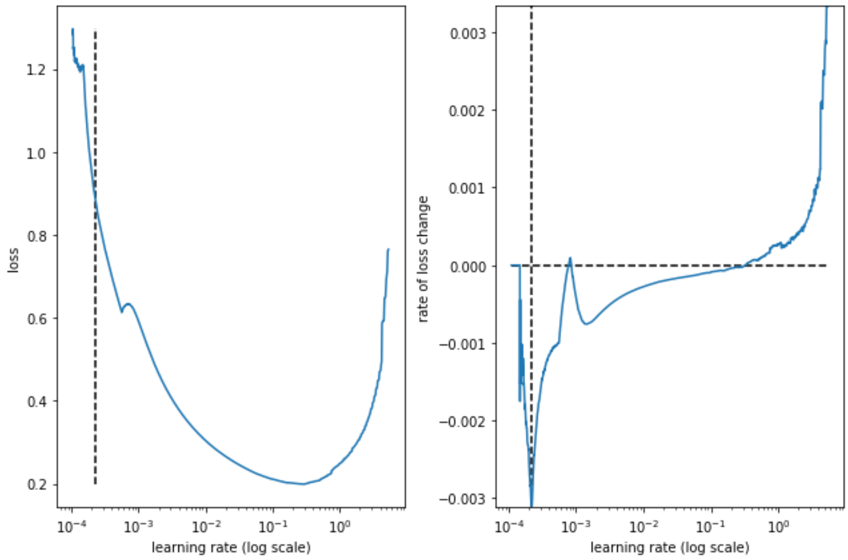
TODO Include basic LSTM diagram

Next, run the learning rate finder.

```
def build_lstm_model(name, data, n_feature_maps = 8):
    in_shape, out_shape = get_io_shapes(data)
    out_steps = out_shape[0]
```

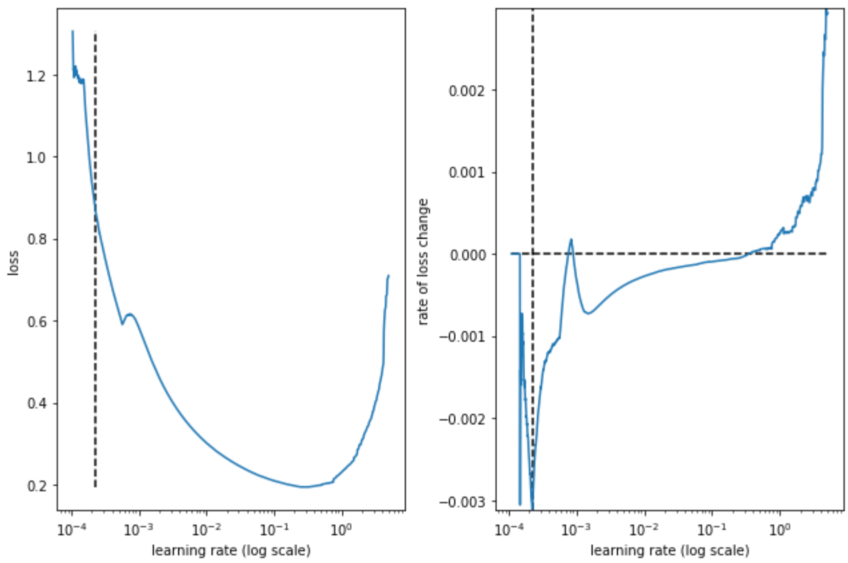
```
    lstm = Sequential(name = name)
    lstm.add(Input(shape = in_shape))

    # Shape [batch, time, features] => [batch, n_feature_maps]
    lstm.add(LSTM(n_feature_maps, return_sequences=False))

    # Shape => [batch, out_steps]
    lstm.add(Dense(out_steps,
                   kernel_initializer=tf.initializers.zeros()))

    return lstm


name = 'LSTM'
models['lstm_24l_1s'] = build_lstm_model(name, ds['train_24l_1s'])
models['lstm_24l_4s'] = build_lstm_model(name, ds['train_24l_4s'])

model = models['lstm_24l_1s']
model.compile(loss = 'mse', metrics = ['mae', 'mape'])
lrf_lstm_24l_1s = LRFinder(model)
lrf_lstm_24l_1s.summarise_lr(ds['train_24l_1s'], 0.0001, 10, 32, 5, 100, 25)
lrf['lstm_24l_1s'] = lrf_lstm_24l_1s

model = models['lstm_24l_4s']
model.compile(loss = 'mse', metrics = ['mae', 'mape'])
lrf_lstm_24l_4s = LRFinder(model)
lrf_lstm_24l_4s.summarise_lr(ds['train_24l_4s'], 0.0001, 10, 32, 5, 100, 25)
lrf['lstm_24l_4s'] = lrf_lstm_24l_4s
```

```
Epoch 1/5
18904/18904 [==============================] - 19s 943us/step - loss: 0.7534 - mae: 0.4657 - mape: 167.8633
```




```
best lr: 0.00022147449
```

```
Epoch 1/5
18869/18869 [==============================] - 20s 966us/step - loss: 0.7260 - mae: 0.4561 - mape: 175.8322
```




```
best lr: 0.00022354048
```