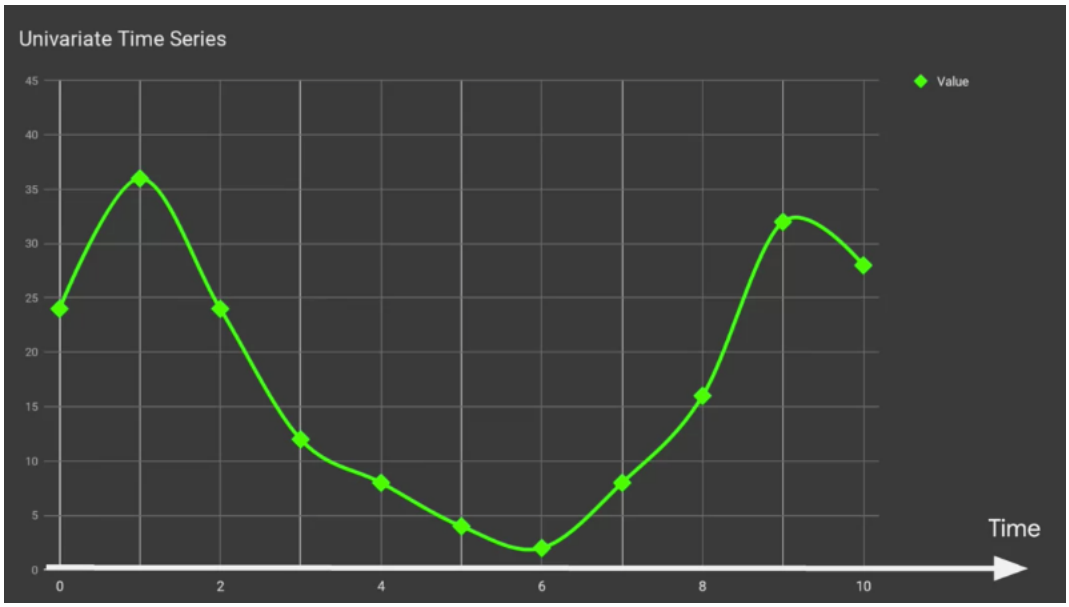
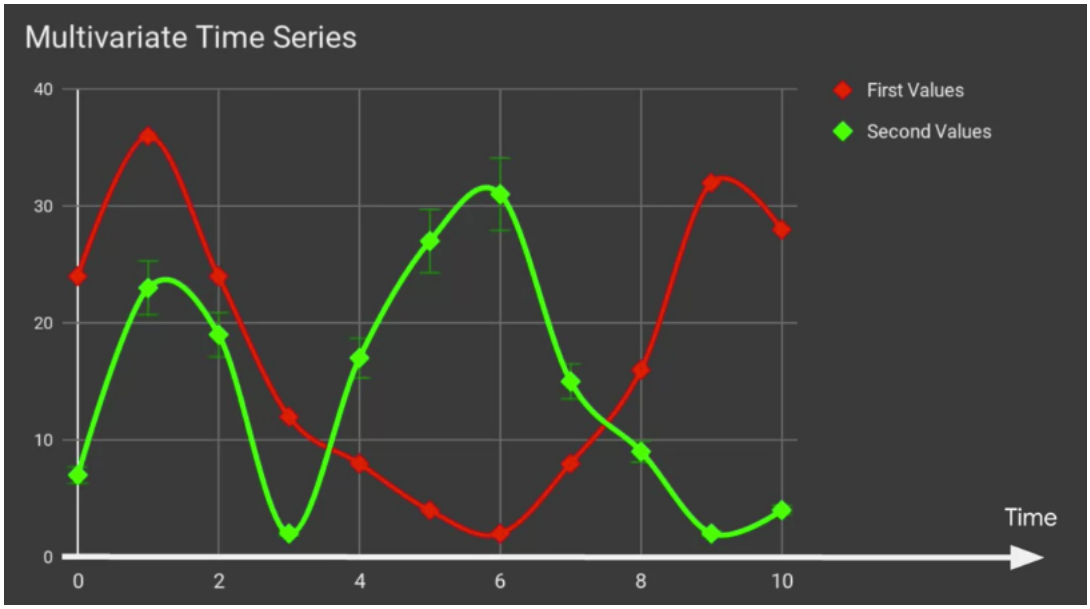
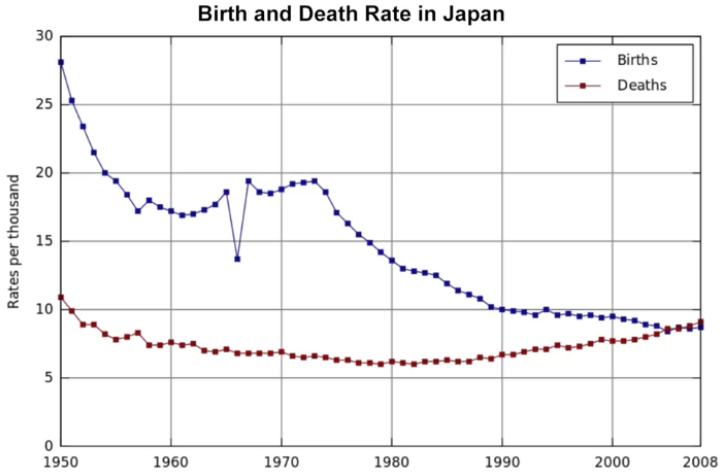
Sequences, Timeseries and Prediction

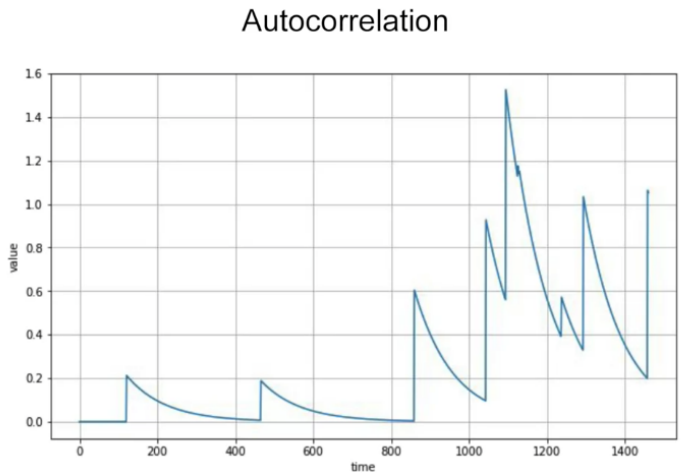
Week-1 (Sequences and Prediction)

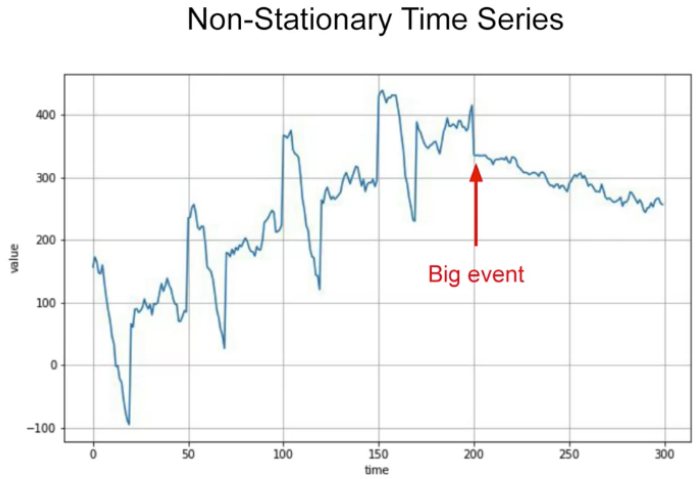




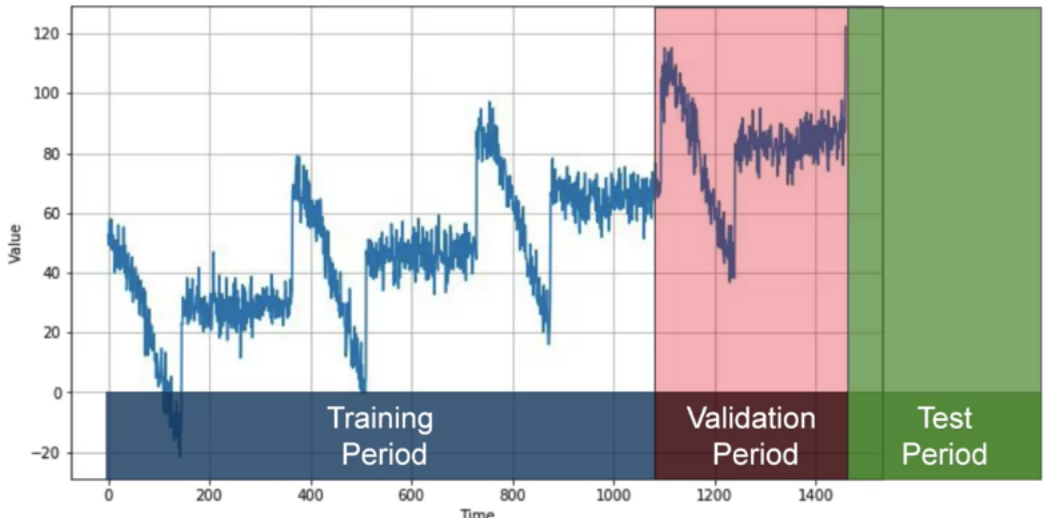
Example –


Birth and Death Rate in Japan
Births
Deaths
Rates per thousand
30
25
20
15
10
5
0
1950
1960
1970
1980
1990
2000
2008

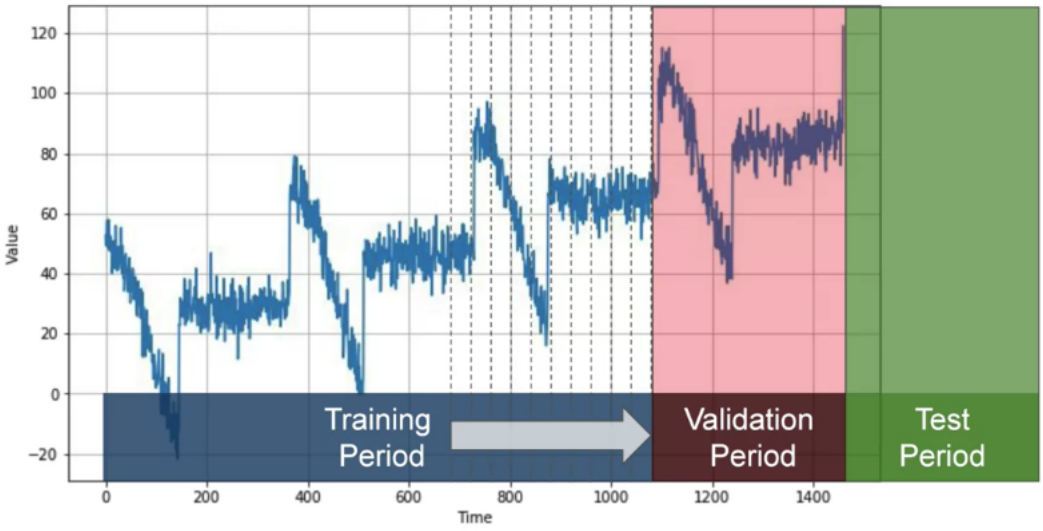

Autocorrelation
value
time
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000
1200
1400


Non-Stationary Time Series
value
time
Big event
400
300
200
100
0
−100
0
50
100
150
200
250
300

Fixed Partitioning



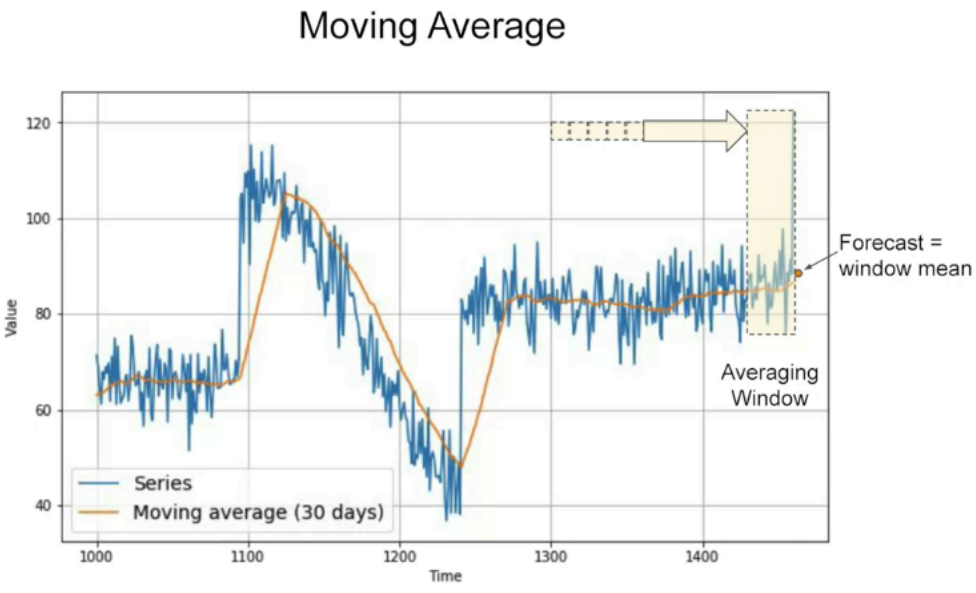
Roll-Forward Partitioning



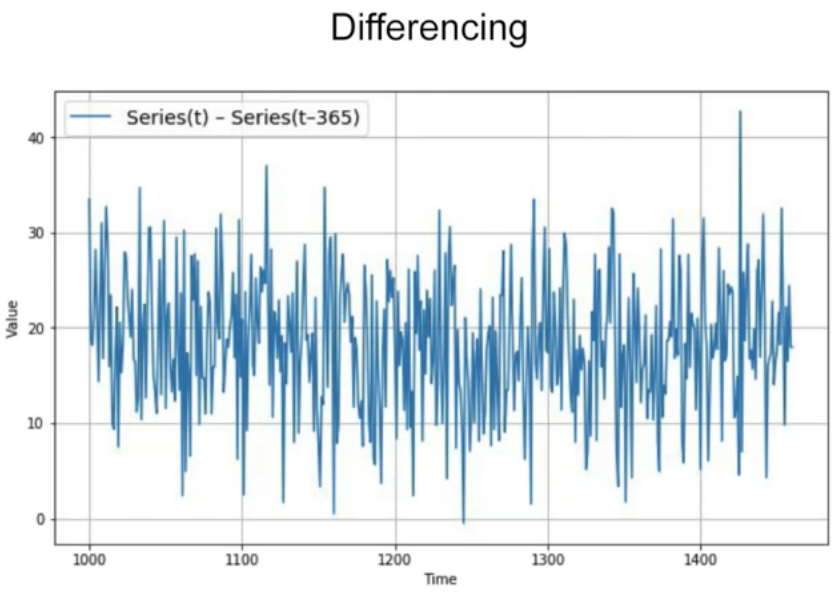
Metrics

```
errors = forecasts - actual

mse = np.square(errors).mean()

rmse = np.sqrt(mse)

mae = np.abs(errors).mean()

mape = np.abs(errors / x_valid).mean()
```

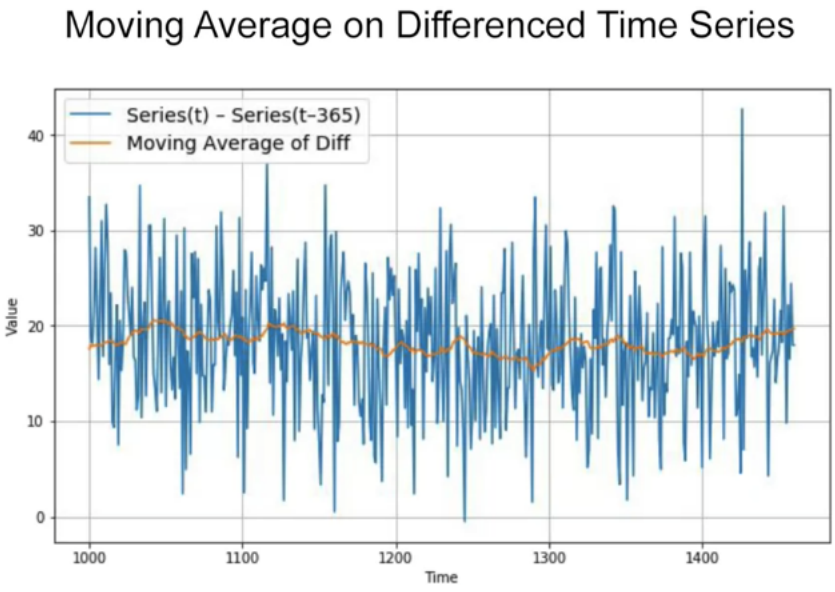
A common and very simple forecasting method is to calculate a moving average. The idea here is that the yellow line is a plot of the average of the blue values over a fixed period called an averaging window, for example, 30 days. Now this nicely eliminates a lot of the noise and it gives us a curve roughly emulating the original series, but it does not anticipate trend or seasonality. Depending on the current time i.e. the period after which you want to forecast for the future, it can actually end up being worse than a naive forecast. In this case, for example, I got a mean absolute error of about 7.14.



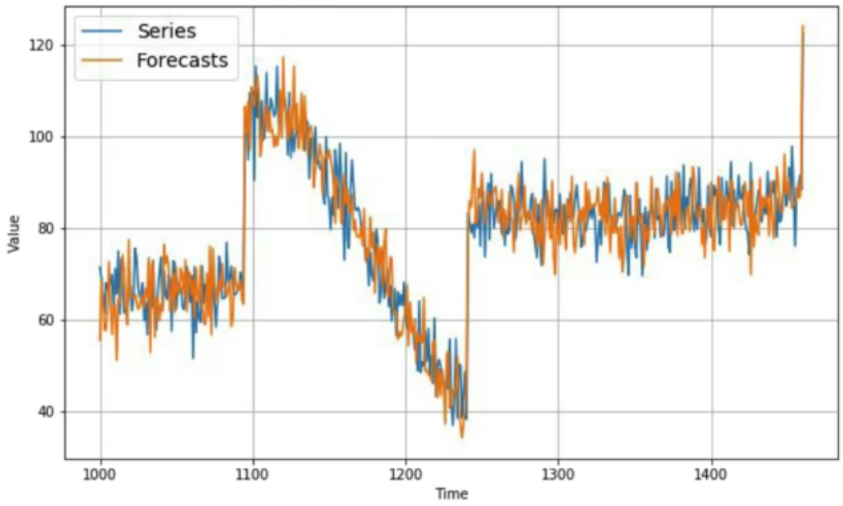
One method to avoid this is to remove the trend and seasonality from the time series with a technique called differencing. So instead of studying the time series itself, we study the difference between the value at time T and the value at an earlier period.



Depending on the time of your data, that period might be a year, a day, a month or whatever. Let's look at a year earlier. So for this data, at time T minus 365, we'll get this difference time series which has no trend and no seasonality. We can then use a moving average to forecast this time series which gives us these forecasts. But these are just forecasts for the difference time series, not the original time series. To get the final forecasts for the original time series, we just need to add back the value at time T minus 365, and we'll get these forecasts. They look much better, don't they? If we measure the mean absolute error on the validation period, we get about 5.8. So it's slightly better than naive forecasting but not tremendously better.

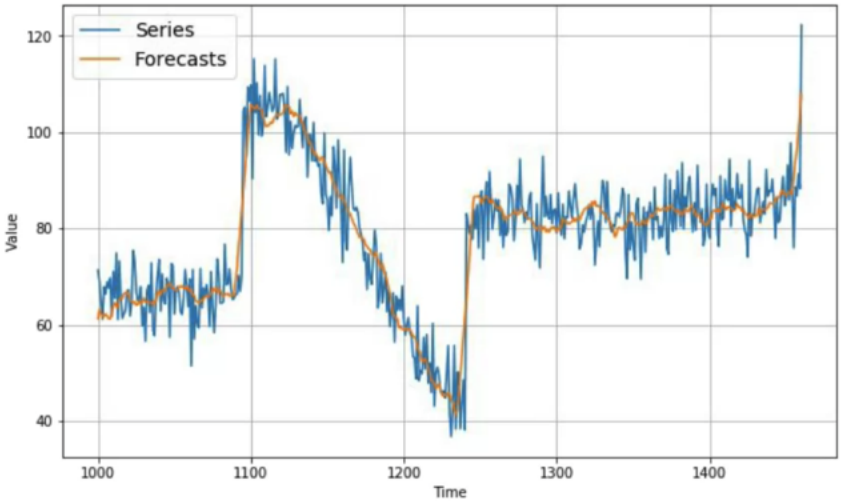


Restoring the Trend and Seasonality



Forecasts = moving average of differenced series + series(t – 365)

Smoothing Both Past and Present Values



Forecasts = trailing moving average of differenced series + centered moving average of past series (t – 365)

Week-2 (DNN for timeseries)

Feeding windowed dataset into neural network

```
def windowed_dataset(series, window_size, batch_size, shuffle_buffer):
  dataset = tf.data.Dataset.from_tensor_slices(series)
  dataset = dataset.window(window_size + 1, shift=1, drop_remainder=True)
  dataset = dataset.flat_map(lambda window: window.batch(window_size + 1))
  dataset = dataset.shuffle(shuffle_buffer)
                   .map(lambda window: (window[:-1], window[-1]))
  dataset = dataset.batch(batch_size).prefetch(1)
  return dataset
```

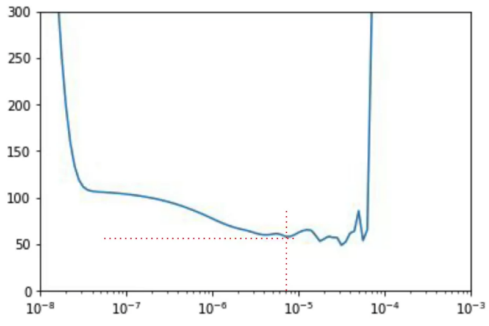
Deep neural network training, tuning and prediction

```
dataset = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)

model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(10, input_shape=[window_size], activation="relu"),
    tf.keras.layers.Dense(10, activation="relu"),
    tf.keras.layers.Dense(1)
])

model.compile(loss="mse", optimizer=tf.keras.optimizers.SGD(lr=1e-6, momentum=0.9))
model.fit(dataset,epochs=100,verbose=0)
```

Getting the optimized Learning rate sing callbacks

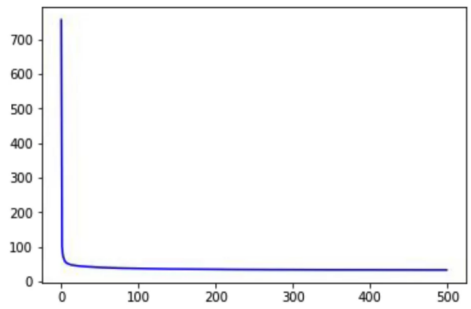
```
dataset = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)

model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(10, input_shape=[window_size], activation="relu"),
    tf.keras.layers.Dense(10, activation="relu"),
    tf.keras.layers.Dense(1)
])

lr_schedule = tf.keras.callbacks.LearningRateScheduler(
    lambda epoch: 1e-8 * 10**(epoch / 20))

optimizer = tf.keras.optimizers.SGD(lr=1e-8, momentum=0.9)

model.compile(loss="mse", optimizer=optimizer)

history = model.fit(dataset, epochs=100, callbacks=[lr_schedule])
```

```
lrs = 1e-8 * (10 ** (np.arange(100) / 20))
plt.semilogx(lrs, history.history["loss"])
plt.axis([1e-8, 1e-3, 0, 300])
```

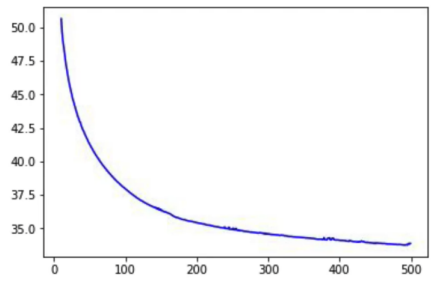



```
window_size = 30
dataset = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)

model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(10, activation="relu", input_shape=[window_size]),
  tf.keras.layers.Dense(10, activation="relu"),
  tf.keras.layers.Dense(1)
])

optimizer = tf.keras.optimizers.SGD(lr=7e-6, momentum=0.9)
model.compile(loss="mse", optimizer=optimizer)
history = model.fit(dataset, epochs=500)
```

```
loss = history.history['loss']
epochs = range(len(acc))
plt.plot(epochs, loss, 'b', label='Training Loss')
plt.show()
```

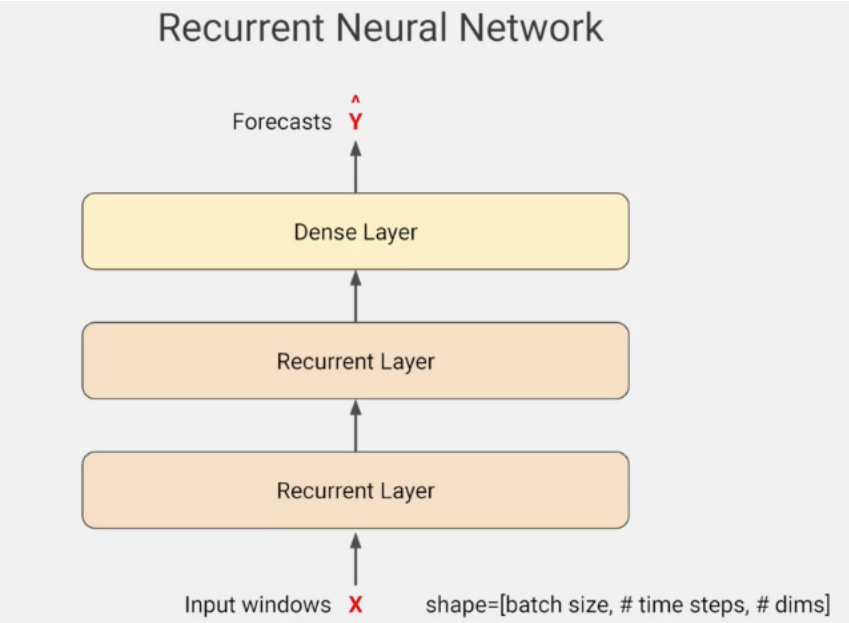


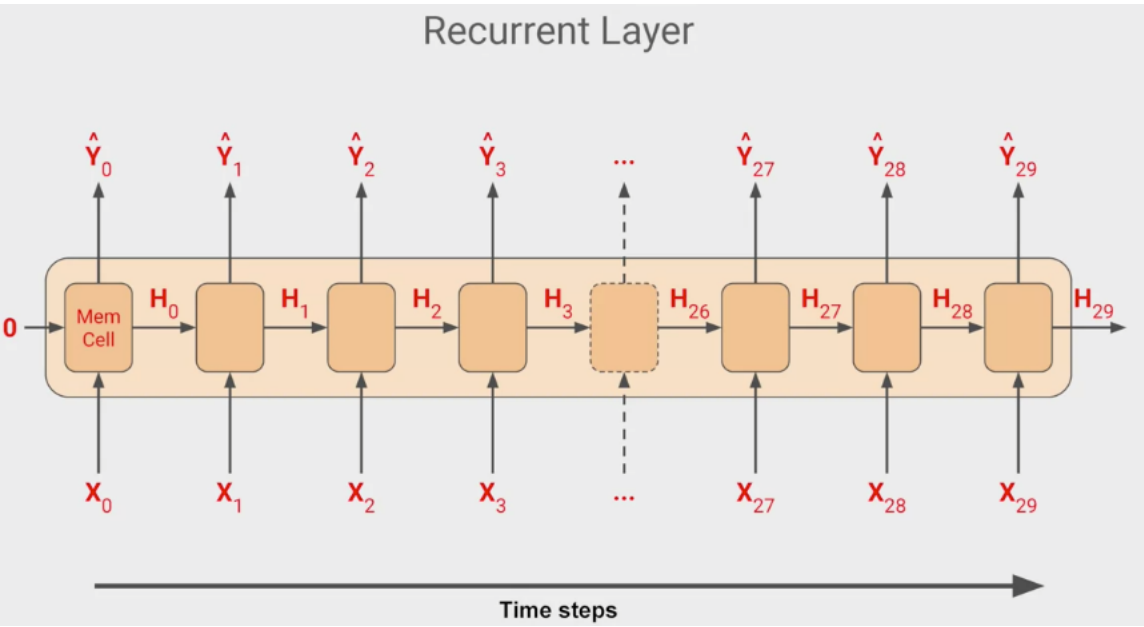

```
# Plot all but the first 10
loss = history.history['loss']
epochs = range(10, len(acc))
plot_loss = loss[10:]
print(plot_loss)
plt.plot(epochs, plot_loss, 'b', label='Training Loss')
plt.show()
```



Week-3 (RNN for timeseries)

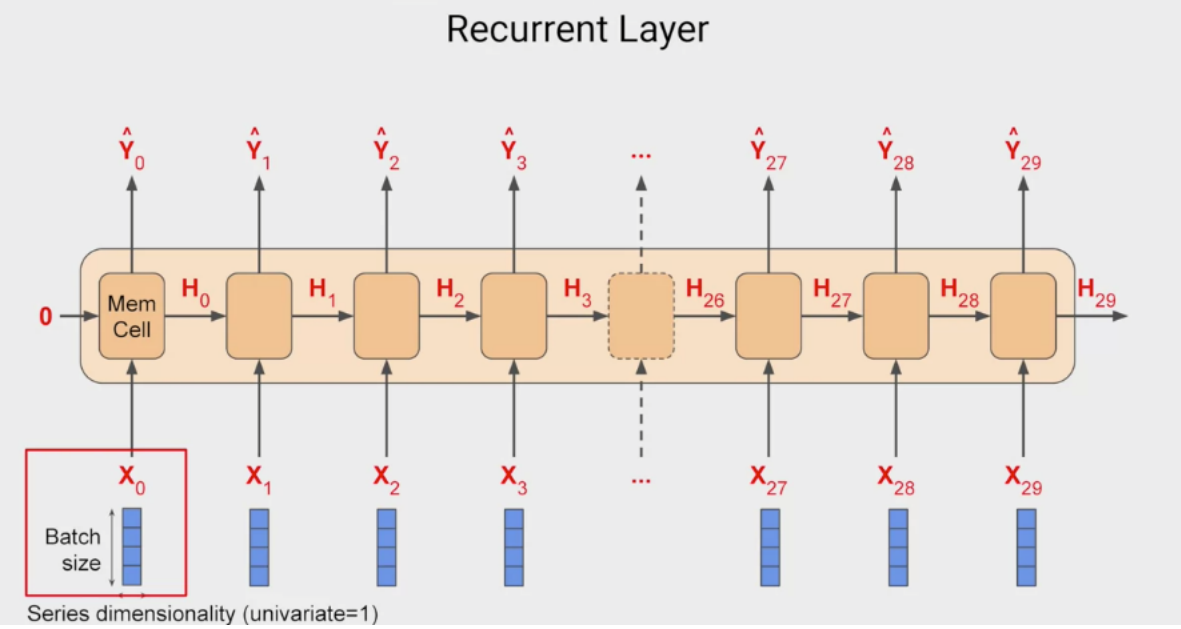
The full input shape when using RNNs is three-dimensional. The first dimension will be the batch size, the second will be the timestamps, and the third is the dimensionality of the inputs at each time step. For example, if it's a univariate time series, this value will be one, for multivariate it'll be more.



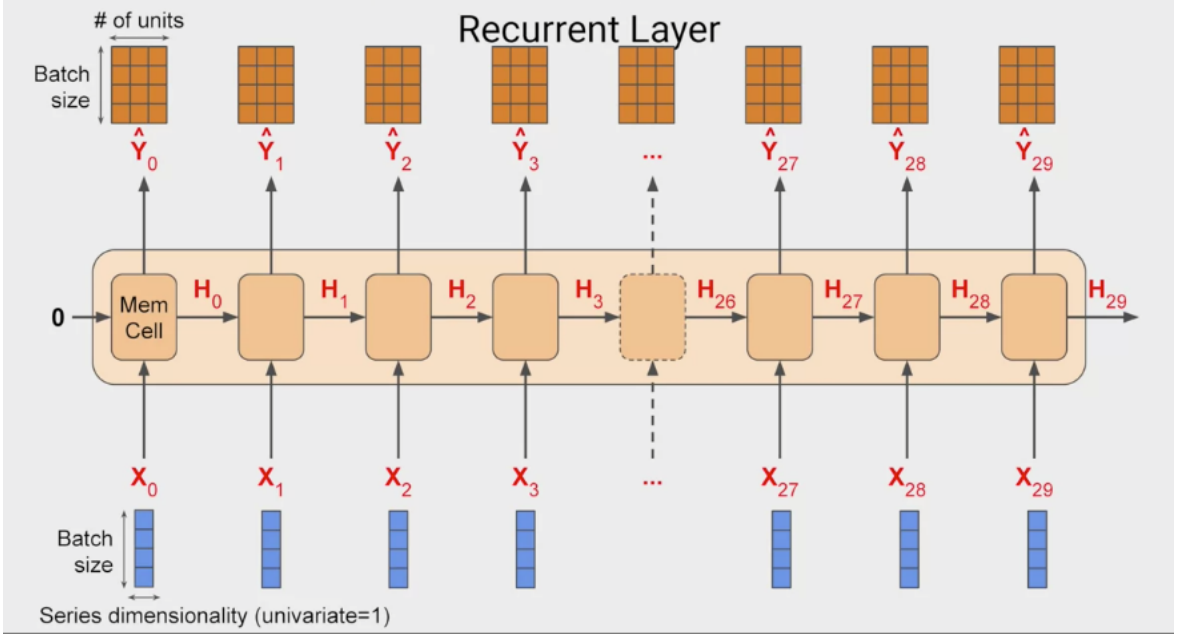


Shape of the inputs to the RNN

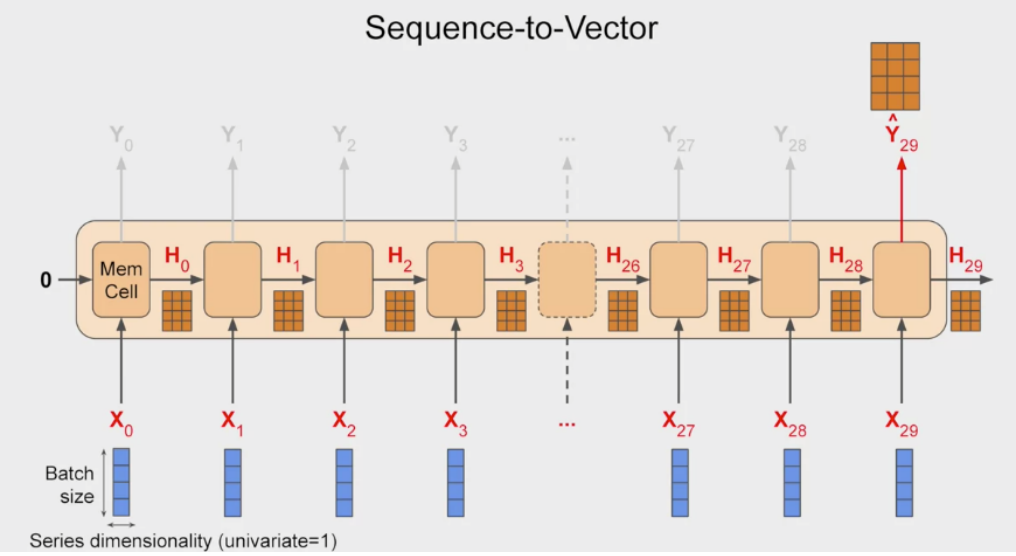
The inputs are three dimensional. So for example, if we have a window size of 30 timestamps and we're batching them in sizes of four, the shape will be 4 times 30 times 1, and each timestamp, the memory cell input will be a four by one matrix, like below.



If the memory cell is comprised of three neurons, then the output matrix will be four by three because the batch size coming in was four and the number of neurons is three. So the full output of the layer is three dimensional, in this case, 4 by 30 by 3.



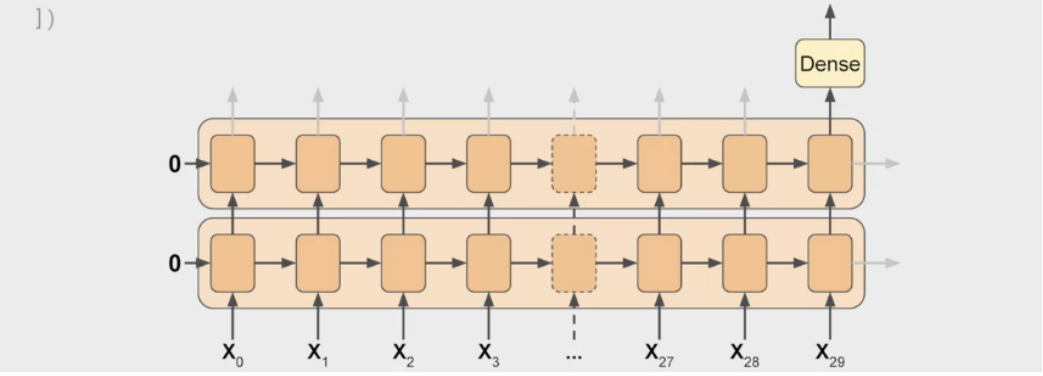
Now, in some cases, you might want to input a sequence, but you don't want to output on and you just want to get a single vector for each instance in the batch. This is typically called a sequence to vector RNN. But in reality, all you do is ignore all of the outputs, except the last one. When using Keras in TensorFlow, this is the default behavior. So if you want the recurrent layer to output a sequence, you have to specify returns sequences equals true when creating the layer. You'll need to do this when you stack one RNN layer on top of another.



Outputting a sequence

Notice the **input_shape,** it's set to None and 1. TensorFlow assumes that the first dimension is the batch size, and that it can have any size at all, so you don't need to define it. Then the next dimension is the number of timestamps, which we can set to none, which means that the RNN can handle sequences of any length. The last dimension is just one because we're using a unit vary of time series.

```
model = keras.models.Sequential([
  keras.layers.SimpleRNN(20, return_sequences=True,
                         input_shape=[None, 1]),
  keras.layers.SimpleRNN(20),
  keras.layers.Dense(1)
])
```

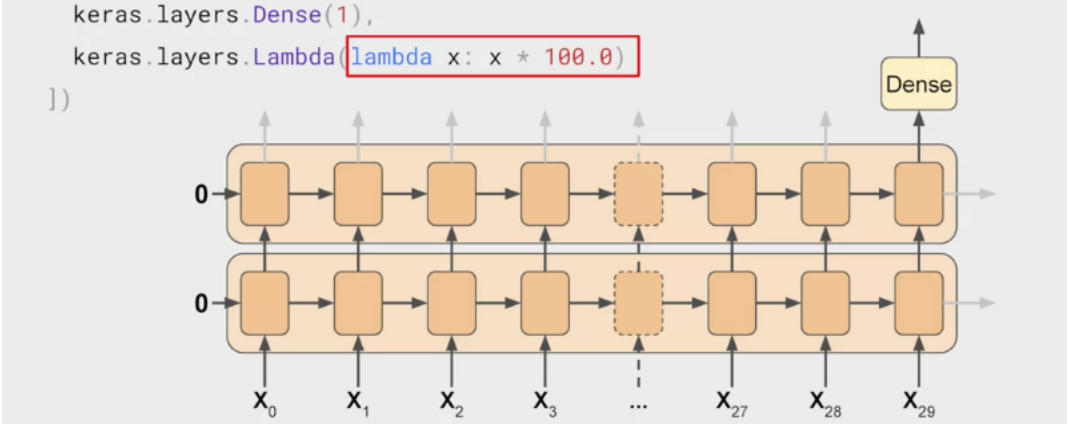


Lambda layers

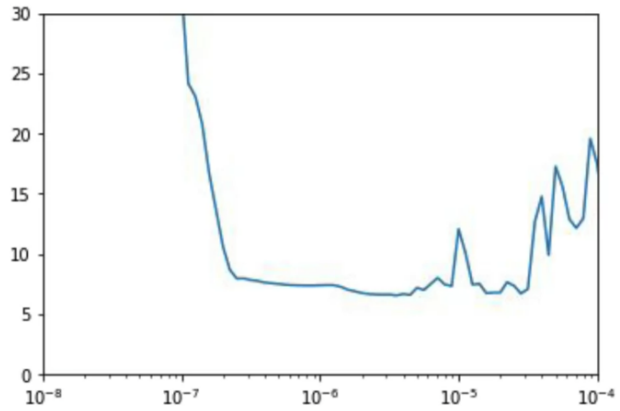
This type of layer is one that allows us to perform arbitrary operations to effectively expand the functionality of TensorFlow's Keras.

If you recall when we wrote the window dataset helper function, it returned two-dimensional batches of Windows on the data, with the first being the batch size and the second the number of timestamps. But an RNN expects three-dimensions; batch size, the number of timestamps, and the series dimensionality. With the Lambda layer, we can fix this without rewriting our Window dataset helper function. Using the Lambda, we just expand the array by one dimension. By setting input shape to none, we're saying that the model can take sequences of any length. Similarly, if we scale up the outputs by 100, we can help training. The default activation function in the RNN layers is tan H which is the hyperbolic tangent activation. This outputs values between negative one and one. Since the time series values are in that order usually in the 10s like 40s, 50s, 60s, and 70s, then scaling up the outputs to the same ballpark can help us with learning. We can do that in a Lambda layer too, we just simply multiply that by a 100.

```
model = keras.models.Sequential([
  keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1),
                      input_shape=[None]),
  keras.layers.SimpleRNN(20, return_sequences=True),
  keras.layers.SimpleRNN(20),
  keras.layers.Dense(1),
  keras.layers.Lambda(lambda x: x * 100.0)
])
```



```
train_set = windowed_dataset(x_train, window_size, batch_size=128,
shuffle_buffer=shuffle_buffer_size)

model = tf.keras.models.Sequential([
  tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1), input_shape=[None]),
  tf.keras.layers.SimpleRNN(40, return_sequences=True),
  tf.keras.layers.SimpleRNN(40),
  tf.keras.layers.Dense(1),
  tf.keras.layers.Lambda(lambda x: x * 100.0)
])

lr_schedule = tf.keras.callbacks.LearningRateScheduler(lambda epoch: 1e-8 * 10**(epoch / 20))

optimizer = tf.keras.optimizers.SGD(lr=1e-8, momentum=0.9)

model.compile(loss=tf.keras.losses.Huber(),
              optimizer=optimizer,
              metrics=["mae"])

history = model.fit(train_set, epochs=100, callbacks=[lr_schedule])
```



The Huber function is a loss function that's less sensitive to outliers and as this data can get a little bit noisy, it's worth giving it a shot. If I run this for 100 epochs and measure the loss at each epoch, I will see that my optimum learning rate for stochastic gradient descent is between about 10 to the minus 5 and 10 to the minus 6. So I'm going to set it's 5 times 10 to the minus 5.

```
tf.keras.backend.clear_session()
tf.random.set_seed(51)
np.random.seed(51)

dataset = windowed_dataset(x_train, window_size, batch_size=128,
shuffle_buffer=shuffle_buffer_size)

model = tf.keras.models.Sequential([
  tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1), input_shape=[None]),
  tf.keras.layers.SimpleRNN(40, return_sequences=True),
  tf.keras.layers.SimpleRNN(40),
  tf.keras.layers.Dense(1),
  tf.keras.layers.Lambda(lambda x: x * 100.0)
])

optimizer = tf.keras.optimizers.SGD(lr=5e-5, momentum=0.9)
history = model.compile(loss=tf.keras.losses.Huber(), optimizer=optimizer, metrics=["mae"])
model.fit(dataset,epochs=500)
```

Coding LSTMs

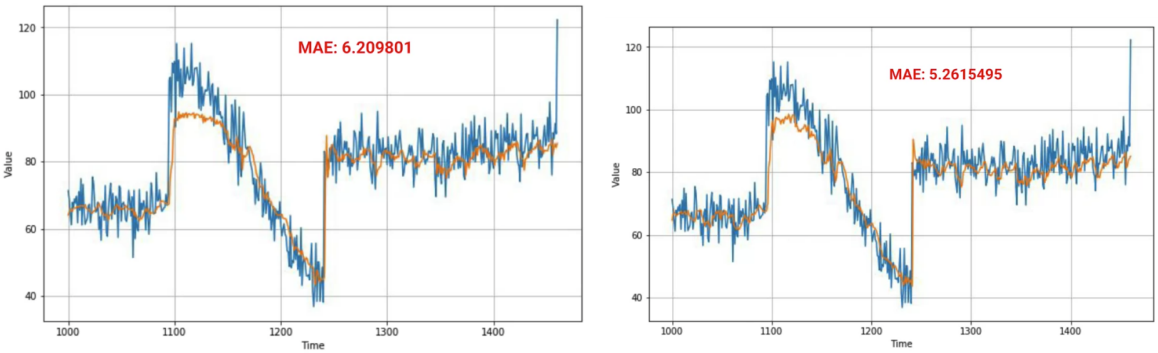
```
tf.keras.backend.clear_session()
dataset = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)

model = tf.keras.models.Sequential([
  tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1),
                      input_shape=[None]),
  tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32, return_sequences=True)),
  tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32, return_sequences=True)),
  tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
  tf.keras.layers.Dense(1),
  tf.keras.layers.Lambda(lambda x: x * 100.0)
])


model.compile(loss="mse", optimizer=tf.keras.optimizers.SGD(lr=1e-6, momentum=0.9))
model.fit(dataset,epochs=100)
```

Week-4 (Real World Timeseries data)

Convolutions

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv1D(filters=32, kernel_size=5,
                         strides=1, padding="causal",
                         activation="relu",
                         input_shape=[None, 1]),
  tf.keras.layers.LSTM(32, return_sequences=True),
  tf.keras.layers.LSTM(32, return_sequences=True),
  tf.keras.layers.Dense(1),
  tf.keras.layers.Lambda(lambda x: x * 200)
])

optimizer = tf.keras.optimizers.SGD(lr=1e-5, momentum=0.9)

model.compile(loss=tf.keras.losses.Huber(),
              optimizer=optimizer,
              metrics=["mae"])

model.fit(dataset,epochs=500)
```

```
def windowed_dataset(series, window_size, batch_size, shuffle_buffer):
    series = tf.expand_dims(series, axis=-1)

    ds = tf.data.Dataset.from_tensor_slices(series)
    ds = ds.window(window_size + 1, shift=1, drop_remainder=True)
    ds = ds.flat_map(lambda w: w.batch(window_size + 1))
    ds = ds.shuffle(shuffle_buffer)
    ds = ds.map(lambda w: (w[:-1], w[1:]))

    return ds.batch(batch_size).prefetch(1)
```



When training, this looks really good giving very low loss in MAE values sometimes even less than one. But unfortunately it's overfittingng when we plot the predictions against the validation set, we don't see much improvement and in fact our MAE has gone down. So it's still a step in the right direction and consider an architecture like this one as you go forward, but perhaps you might need to tweak some of the parameters to avoid overfitting. Some of the problems are clearly visualize when we plot the loss against the MAE, there's a lot of noise and instability in there. One common cause for small spikes like that is a small batch size introducing further random noise. One hint was to explore the batch size and to make sure it's appropriate for my data. So in this case it's worth experimenting with different batch sizes. So for example experimented with different batch sizes both larger and smaller than the original 32, and when I tried 16 you can see the impact here on the validation set, and here on the training loss and MAE data.

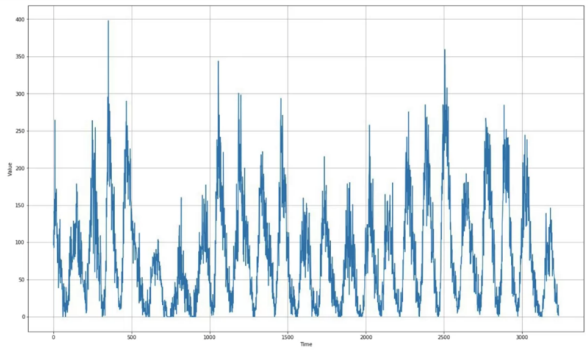
Real data - sunspots

```
import csv
time_step = []
sunspots = []

with open('/tmp/sunspots.csv') as csvfile:
  reader = csv.reader(csvfile, delimiter=',')
  next(reader)
  for row in reader:
    sunspots.append(float(row[2]))
    time_step.append(int(row[0]))
```

Sunspots.csv

1	,Date,Monthly Mean Total Sunspot Number
2	0,1749-01-31,96.7
3	1,1749-02-28,104.3
4	2,1749-03-31,116.7
5	3,1749-04-30,92.8
6	4,1749-05-31,141.7
7	5,1749-06-30,139.2
8	6,1749-07-31,158.0
9	7,1749-08-31,110.5
10	8,1749-09-30,126.5
11	9,1749-10-31,125.8
12	10,1749-11-30,264.3
13	11,1749-12-31,142.0
14	12,1750-01-31,122.2
15	13,1750-02-28,126.5
16	14,1750-03-31,148.7
17	15,1750-04-30,147.2
18	16,1750-05-31,150.0
19	17,1750-06-30,166.7

```
series = np.array(sunspots)
time = np.array(time_step)
```



```
split_time = 1000
time_train = time[:split_time]
x_train = series[:split_time]
time_valid = time[split_time:]
x_valid = series[split_time:]

window_size = 20
batch_size = 32
shuffle_buffer_size = 1000
```

```
def windowed_dataset(series, window_size, batch_size, shuffle_buffer):
  dataset = tf.data.Dataset.from_tensor_slices(series)
  dataset = dataset.window(window_size + 1, shift=1, drop_remainder=True)
  dataset = dataset.flat_map(lambda window: window.batch(window_size + 1))
  dataset = dataset.shuffle(shuffle_buffer)
                   .map(lambda window: (window[:-1], window[-1]))
  dataset = dataset.batch(batch_size).prefetch(1)
  return dataset
```

```
tf.keras.backend.clear_session()
tf.random.set_seed(51)
np.random.seed(51)
window_size = 64
batch_size = 256
train_set = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)
print(train_set)
print(x_train.shape)

model = tf.keras.models.Sequential([
  tf.keras.layers.Conv1D(filters=32, kernel_size=5,
                         strides=1, padding="causal",
                         activation="relu",
                         input_shape=[None, 1]),
  tf.keras.layers.LSTM(64, return_sequences=True),
  tf.keras.layers.LSTM(64, return_sequences=True),
  tf.keras.layers.Dense(30, activation="relu"),
  tf.keras.layers.Dense(10, activation="relu"),
  tf.keras.layers.Dense(1),
  tf.keras.layers.Lambda(lambda x: x * 400)
])

lr_schedule = tf.keras.callbacks.LearningRateScheduler(
    lambda epoch: 1e-8 * 10**(epoch / 20))
optimizer = tf.keras.optimizers.SGD(lr=1e-8, momentum=0.9)
model.compile(loss=tf.keras.losses.Huber(),
              optimizer=optimizer,
              metrics=["mae"])
history = model.fit(train_set, epochs=100, callbacks=[lr_schedule])
```

```
[9] def model_forecast(model, series, window_size):
        ds = tf.data.Dataset.from_tensor_slices(series)
        ds = ds.window(window_size, shift=1, drop_remainder=True)
        ds = ds.flat_map(lambda w: w.batch(window_size))
        ds = ds.batch(32).prefetch(1)
        forecast = model.predict(ds)
        return forecast
```

```
rnn_forecast = model_forecast(model, series[..., np.newaxis], window_size)
rnn_forecast = rnn_forecast[split_time - window_size:-1, -1, 0]
```