```
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)
```

It's a convo D where we'll try to learn 32 filters. It's a one dimensional convolution. So we'll take a five number window and multiply out the values in that window by the filter values, in much the same way as image convolutions are done.

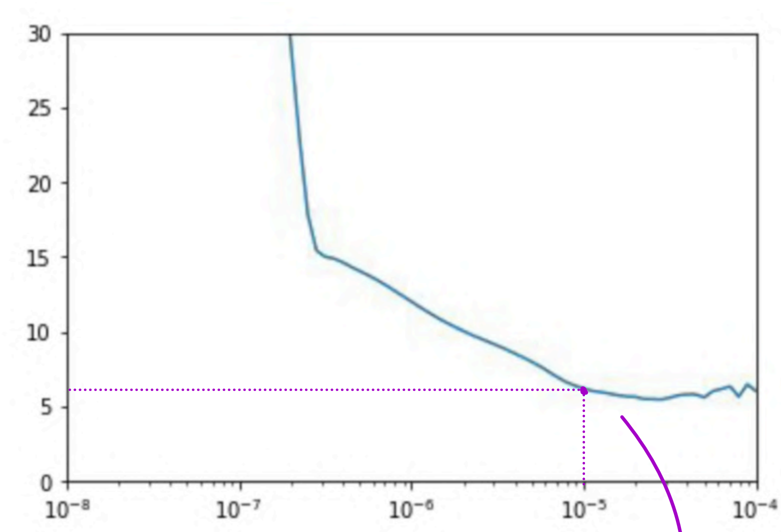
```
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv1D(filters=32, kernel_size=5,
                      strides=1, padding="causal",
                      activation="relu",
                      input_shape=[None, 1]),
  tf.keras.layers.LSTM(32, return_sequences=True),
  tf.keras.layers.LSTM(32,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

One important note is that while we got rid of the Lambda layer that reshaped the input for us to work with the LSTM's. So we're actually specifying an input shape on the curve 1D here.

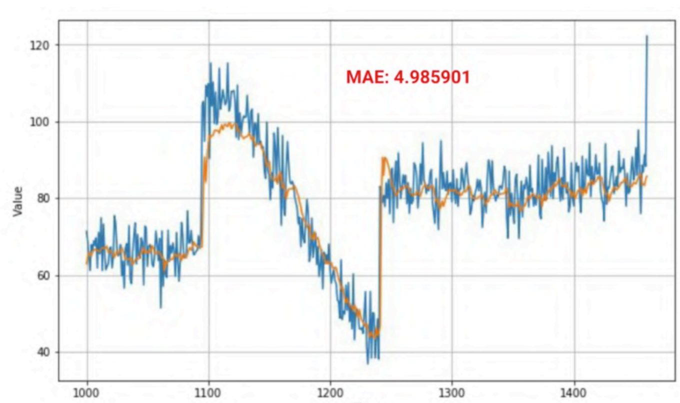
```
def windowed_dataset(series, window_size, batch_size, shuffle_buffer):
    series = tf.expand_dims(series, axis=-1)

    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)
```

This requires us to update the windowed_datasetet helper function that we've been working with all along. We'll simply use tf.expand dims in the helper function to expand the dimensions of the series before we process it.

The code will attempt lots of different learning rates changing them epoch by epoch and plotting the results.




```
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)
```

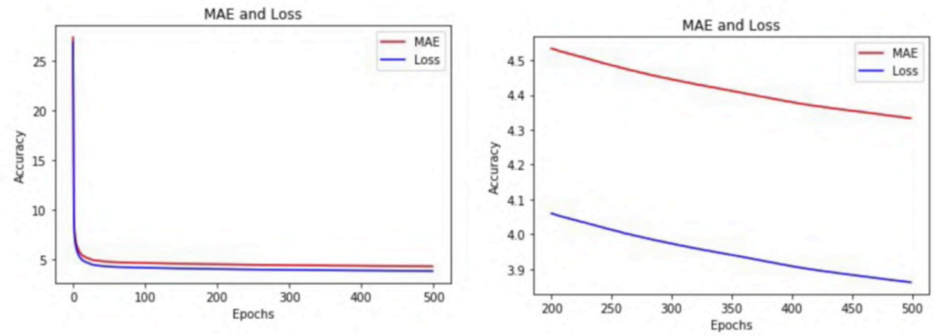


When we train for 500 epochs we'll get this curve. It's a huge improvement over earlier. The peak has lost its plateau but it's still not quite right, it's not getting high enough relative to the data.

Now of course noise is a factor and we can see crazy fluctuations in the peak caused by the noise, but I think our model could possibly do a bit better than this.

Our MAE is below five, but I would bet that outside of that first peak is probably a lot lower than that.

One solution might be to train a little bit longer. Even though our MAE loss curves look flat at 500 epochs, we can see when we zoom in that they're slowly diminishing. The network is still learning albeit slowly.

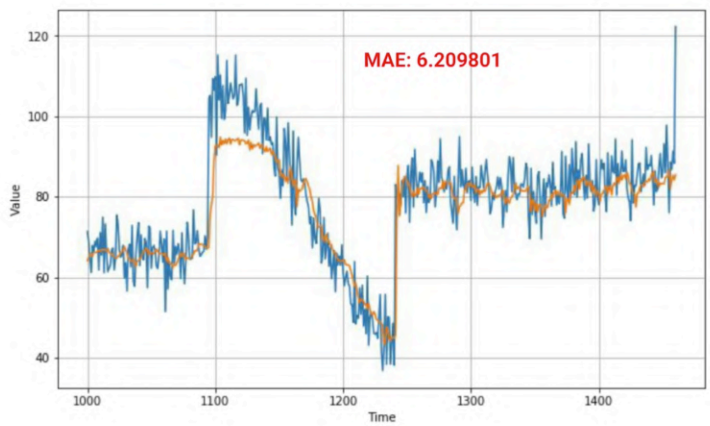



```
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv1D(filters=32, kernel_size=5,
                      strides=1, padding="causal",
                      activation="relu",
                      input_shape=[None, 1]),
  tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32, return_sequences=True)),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

One method would be to make your LASTMs bidirectional like this

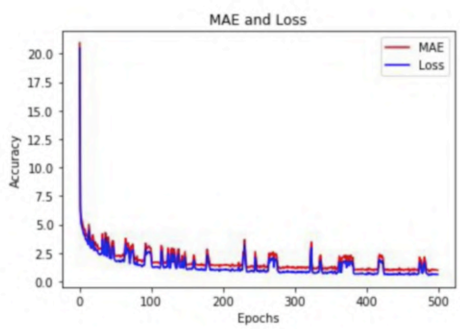
```
Epoch 495/500
31/31 [==============================] - 2s 58ms/step - loss: 0.6491 - mae: 1.0314
Epoch 496/500
31/31 [==============================] - 2s 60ms/step - loss: 0.6155 - mae: 0.9857
Epoch 497/500
31/31 [==============================] - 2s 59ms/step - loss: 0.6425 - mae: 1.0207
Epoch 498/500
31/31 [==============================] - 2s 59ms/step - loss: 0.6330 - mae: 1.0046
Epoch 499/500
31/31 [==============================] - 2s 59ms/step - loss: 0.6155 - mae: 0.9877
Epoch 500/500
31/31 [==============================] - 2s 59ms/step - loss: 0.6111 - mae: 0.9806
```

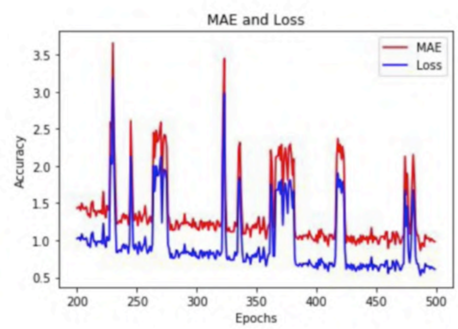
When training, this looks really good giving very low loss in MAE values sometimes even less than one



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 overfittingng.

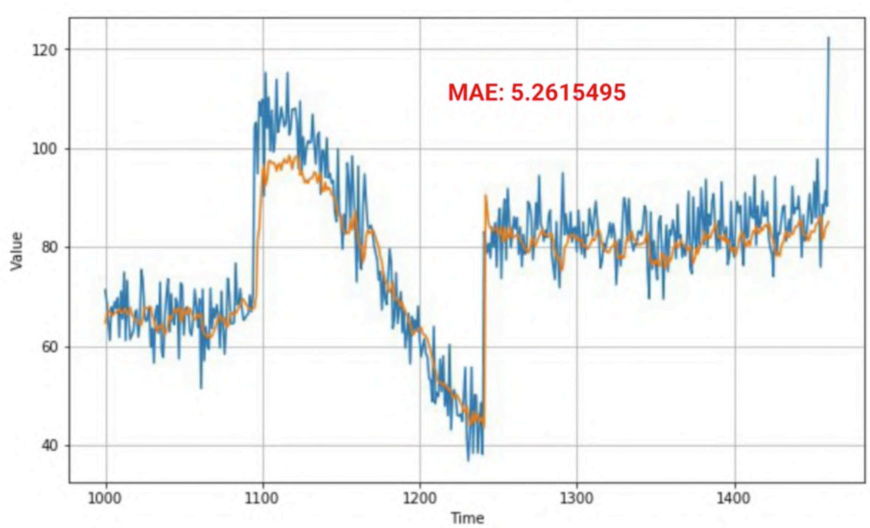
Some of the problems are clearly visualize when we plot the loss against the MAE, there's a lot of noise and instability in there. One common cause for small spikes like that is a small batch size introducing further random noise.

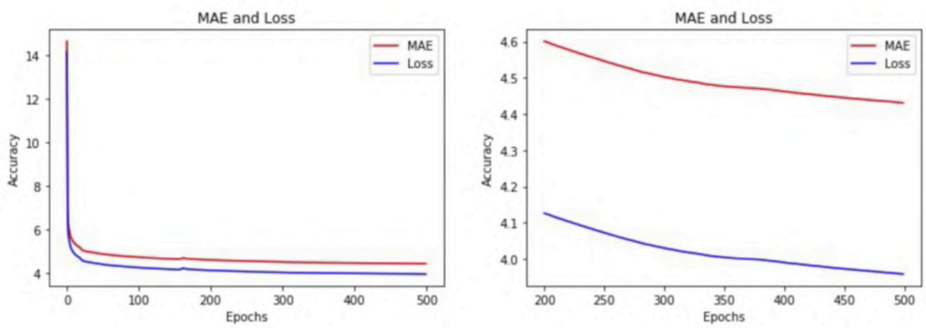




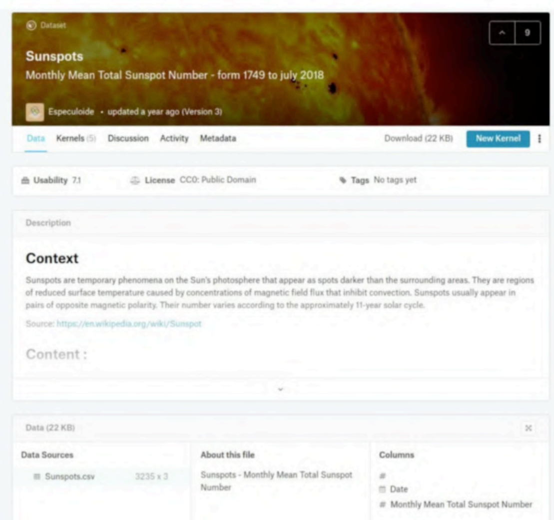
One hint was to explore the batch size and to make sure it's appropriate for my data. So in this case it's worth experimenting with different batch sizes. See optimization notes in DNN specialization.

— Batch Size: 16





https://www.kaggle.com/robervalt/sunspots



Sunspots.csv

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

```
!wget --no-check-certificate \
    https://storage.googleapis.com/laurencemoroney-blog.appspot.com/Sunspots.csv \
    -O /tmp/sunspots.csv
```

This line, next reader, is called Before we loop through the rows and the reader, and it's simply reads the first line and we end up throwing it away. That's because the column titles are in the first line of the file as you can see here.

```
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

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

```
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

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

```python
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]))
```

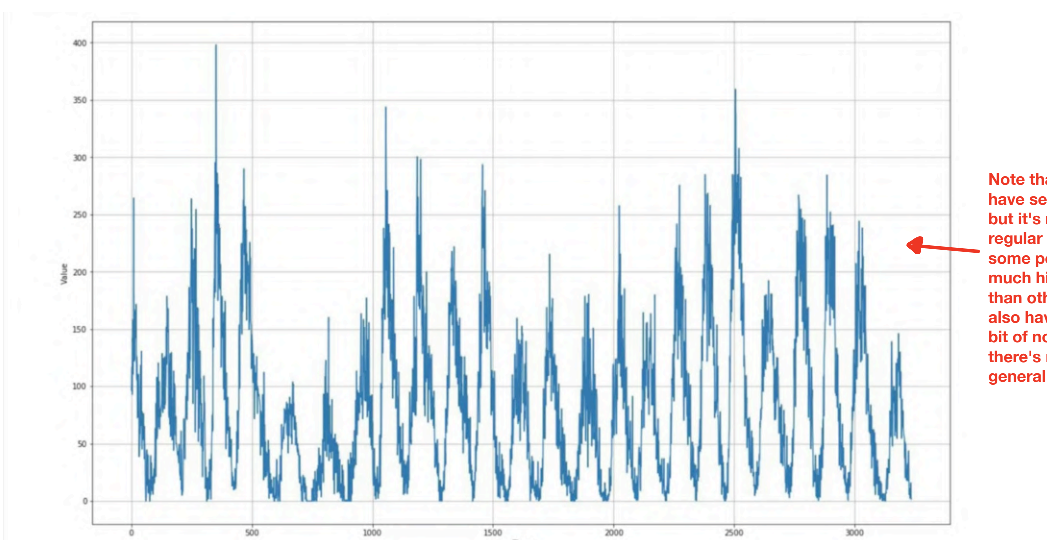
Sunspots.csv

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

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

As much of the code we'll be using to process these deals with NumPy arrays, we may as well now convert a list to NumPy arrays.

It's more efficient to do it this way, build-up your data in a throwaway list and then convert it to NumPy than I would have been to start with NumPy arrays, because every time you append an item to a NumPy, there's a lot of memory management going on to clone the list, maybe a lot of data that can get slow

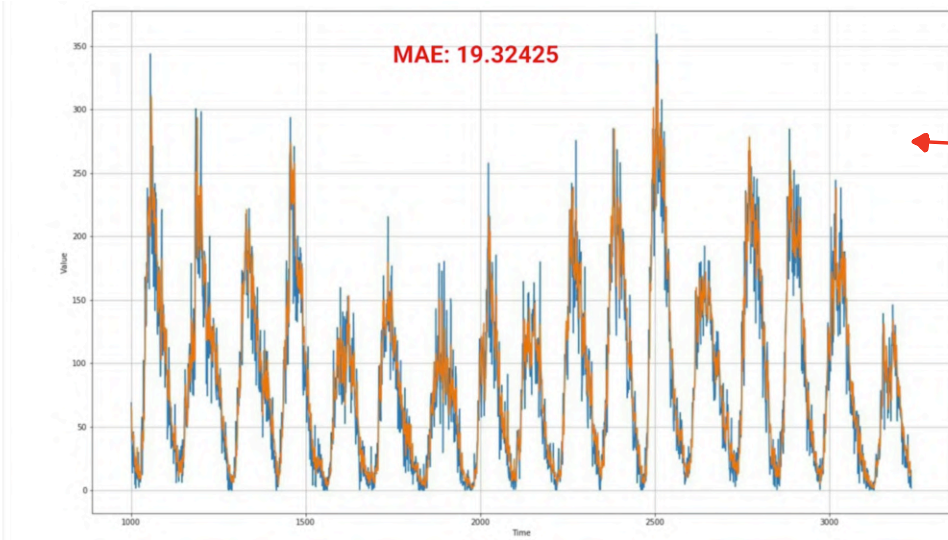


Note that we have seasonality, but it's not very regular with some peaks and much higher than others. We also have quite a bit of noise, but there's no general trend.

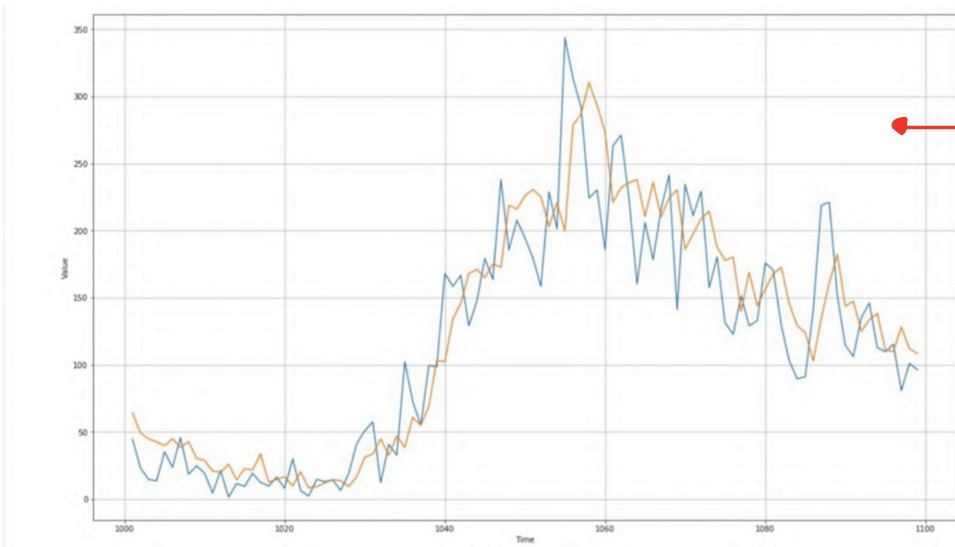
```python
split_time = 1000
time_train = time[:split_time]
x_train = series[:split_time]
time_valid = time[split_time:]
x_valid = series[split_time:]

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

```python
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
```

```python
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)
```



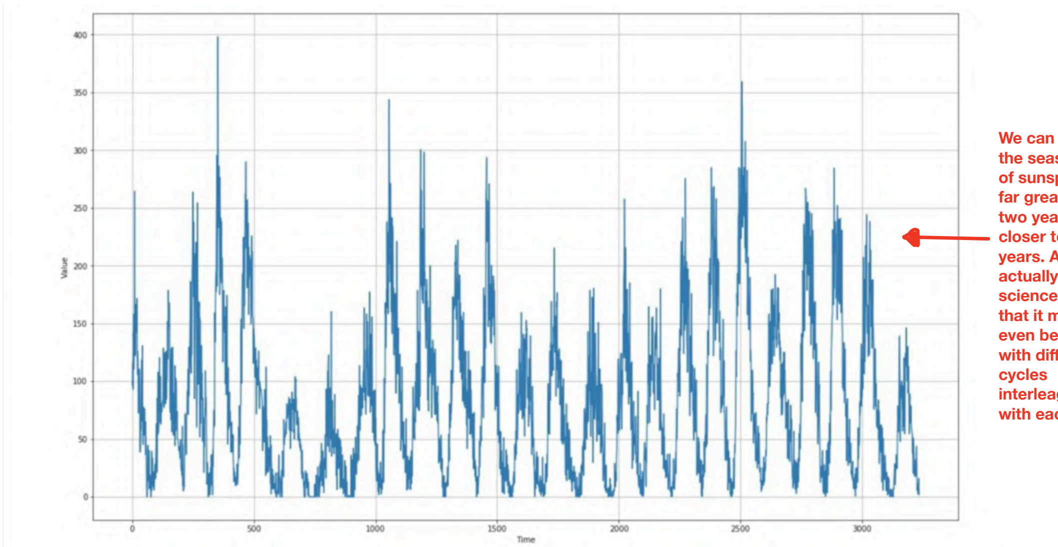
With a chart like this, which at least to the eyeball looks really good, but it has a very large MAE so something must be wrong.



Indeed, if we zoom into the results we can see in a little bit more detail about how the forecast behaves in the original data.

```
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
```

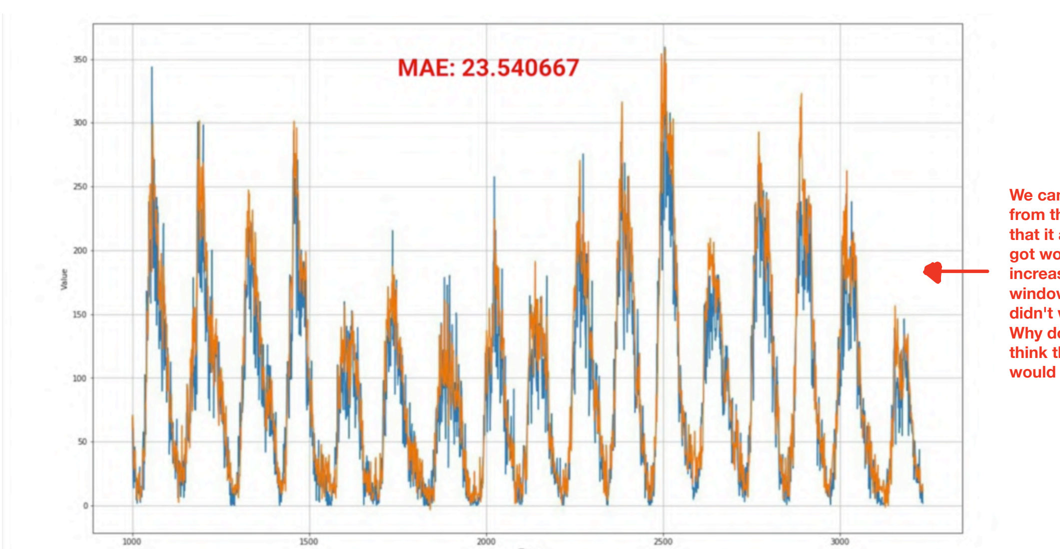
Our clue to the problem could be our window size. Remember earlier we said it's a 20 so our training window sizes are 20 time slices worth of data. And given that each time slice is a month in real time our window is a little under two years.



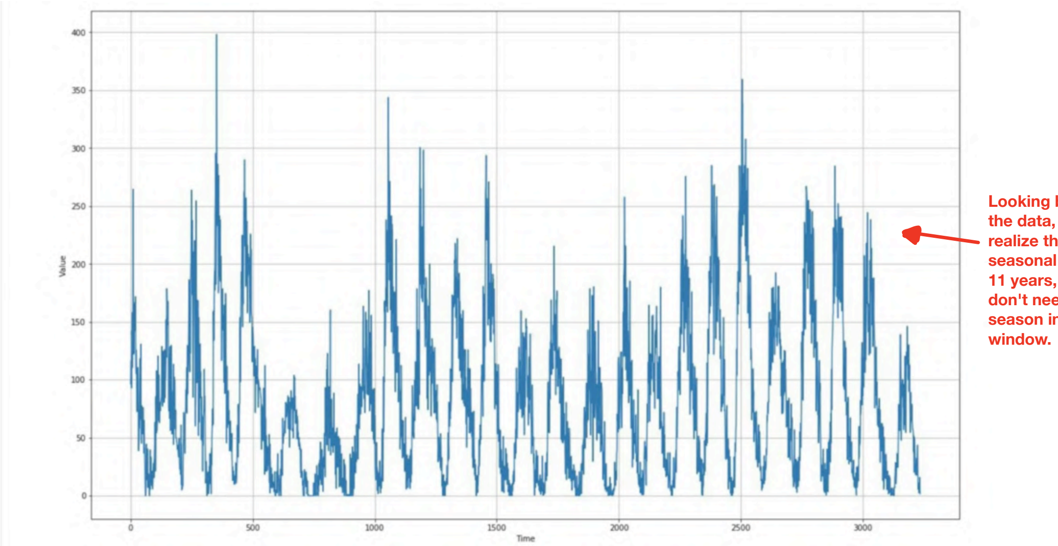
We can see that the seasonality of sunspots is far greater than two years. It's closer to 11 years. And actually some science tells us that it might even be 22 years with different cycles interleaguing with each other.

What would happen if we retrain with a window size of 132, which is 11 years worth of data as our window size.

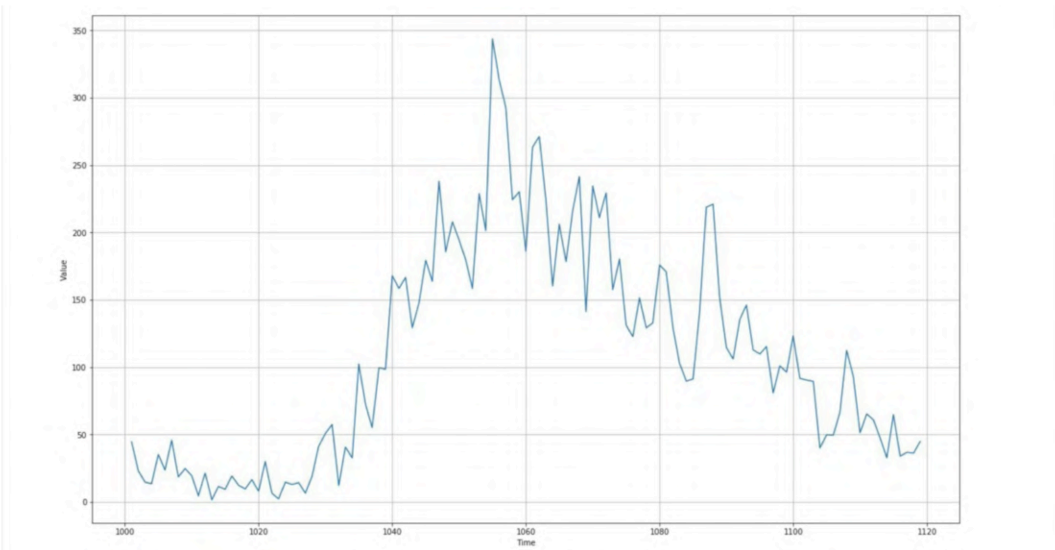
```
split_time = 1000
time_train = time[:split_time]
x_train = series[:split_time]
time_valid = time[split_time:]
x_valid = series[split_time:]

window_size = 132
batch_size = 32
shuffle_buffer_size = 1000
```



We can see from the MAE that it actually got worse so increasing the window size didn't work. Why do you think that would be?



Looking back to the data, we can realize that it is seasonal to about 11 years, but we don't need a full season in our window.




```
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 = 30
batch_size = 32
shuffle_buffer_size = 1000
```

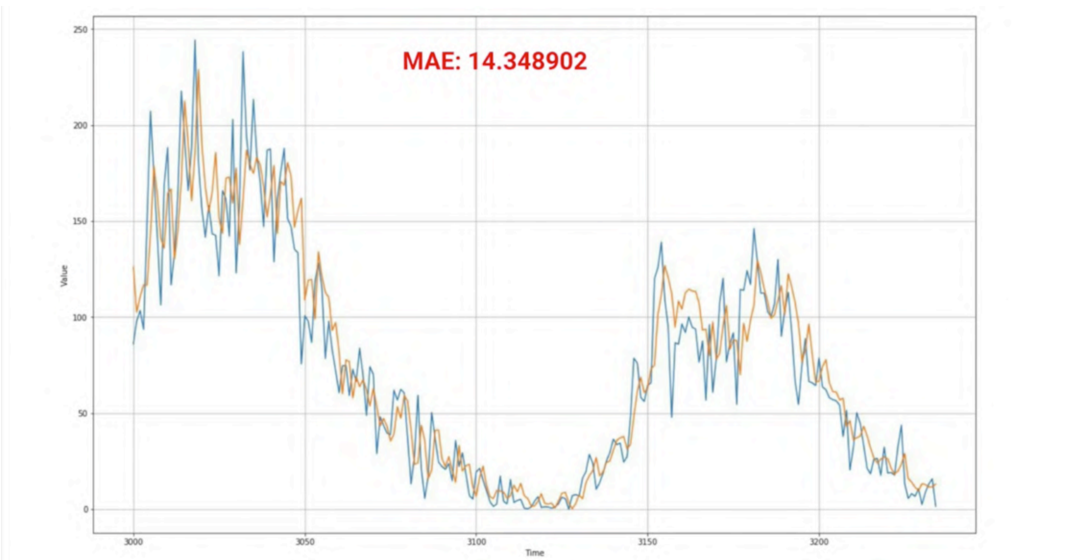
```
split_time = 1000
time_train = time[:split_time]
x_train = series[:split_time]
time_valid = time[split_time:]
x_valid = series[split_time:]

window_size = 30
batch_size = 32
shuffle_buffer_size = 1000
```

So if we look back at this code, we can change our window size to 30. But then look at the split time, the data set has around 3,500 items of data, but we're splitting it into training and validation.

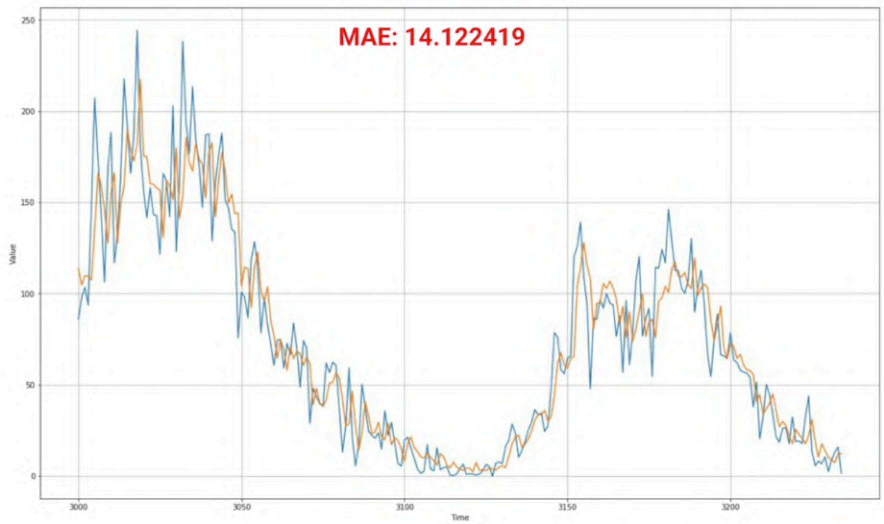
Now 1,000, which means only 1,000 for training and 2,500 for validation. That's a really bad split. There's not enough training data. So let's make it 3,500 instead.

```
split_time = 3000
time_train = time[:split_time]
x_train = series[:split_time]
time_valid = time[split_time:]
x_valid = series[split_time:]

window_size = 30
batch_size = 32
shuffle_buffer_size = 1000
```

```
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)
```




```
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)
```




```
model.predict(series[3205:3235][np.newaxis])

      7.0773993
```

https://www.sws.bom.gov.au/Solar/1/6

(last updated 01 Jun 2019 09:42 UT)

OBSERVED MONTHLY SUNSPOT NUMBERS

2001	142.6	121.5	165.8	161.7	142.1	202.9	123.0	161.5	238.2	194.1	176.6	213.4
2002	184.6	170.2	147.1	186.9	187.5	128.8	161.0	175.6	187.9	151.2	147.2	135.3
2003	133.5	75.7	100.7	97.9	86.8	118.7	128.3	115.4	78.5	97.8	82.9	72.2
2004	60.6	74.6	74.8	59.2	72.8	66.5	83.8	69.7	48.8	74.2	70.1	28.9
2005	48.1	43.5	39.6	38.7	61.9	56.8	62.4	60.5	37.2	13.2	27.5	59.3
2006	20.9	5.7	17.3	50.3	37.2	24.5	22.2	20.8	23.7	14.9	35.7	22.3
2007	29.3	18.4	7.2	5.4	19.5	21.3	15.1	9.8	4.0	1.5	2.8	17.3
2008	4.1	2.9	15.5	3.6	4.6	5.2	0.6	0.3	1.2	4.2	6.6	1.0
2009	1.3	1.2	0.6	1.2	2.9	6.3	5.5	0.0	7.1	7.7	6.9	16.3
2010	19.5	28.5	24.0	10.4	13.9	18.8	25.2	29.6	36.4	33.6	34.4	24.5
2011	27.3	48.3	78.6	76.1	58.2	56.1	64.5	65.8	120.1	125.7	139.1	109.3
2012	94.4	47.8	86.6	85.9	96.5	92.0	100.1	94.8	93.7	76.5	87.6	56.8
2013	96.1	60.9	78.3	107.3	120.2	76.7	86.2	91.8	54.5	114.4	113.9	124.2
2014	117.0	146.1	128.7	112.5	112.5	102.9	100.2	106.9	130.0	90.0	103.6	112.9
2015	93.0	66.7	54.5	75.3	88.8	66.5	65.8	64.4	78.6	63.6	62.2	58.0
2016	57.0	56.4	54.1	37.9	51.5	20.5	32.4	50.2	44.6	33.4	21.4	18.5
2017	26.1	26.4	17.7	32.3	18.9	19.2	17.8	32.6	43.7	13.2	5.7	8.2
2018	6.8	10.7	2.5	8.9	13.1	15.6	1.6	$\boxed{8.7}$	3.3	4.9	4.9	3.1
2019	7.8	0.8	9.5	9.1	10.1							

```
split_time = 3000
window_size = 60

model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(20, 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-7, momentum=0.9))
```

Doing accuracy based on a single prediction like this is also a recipe for disappointment, and you're much better off evaluating mean accuracy over a number of readings.

```
window_size = 60
batch_size = 64
train_set = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)

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(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])
```

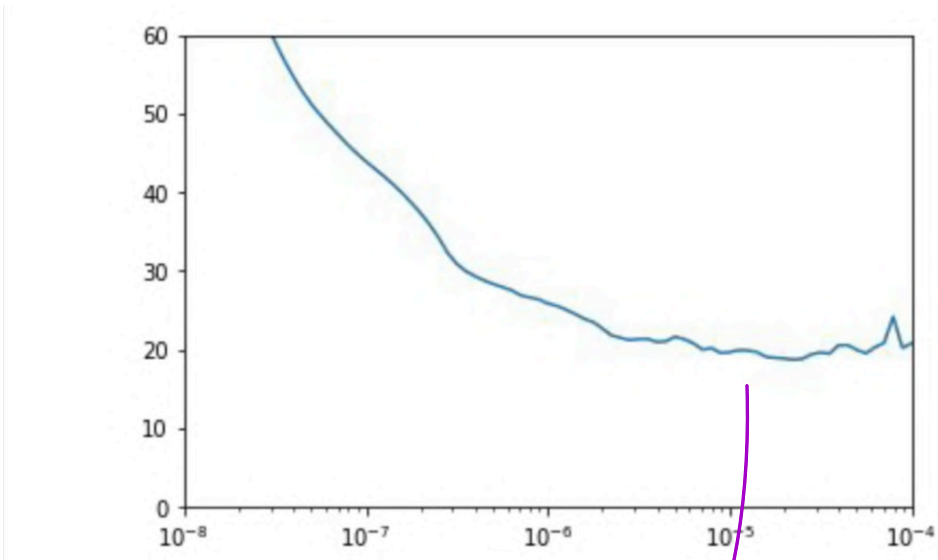

```
window_size = 60
batch_size = 64
train_set = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)

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(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])
```

```
window_size = 60
batch_size = 64
train_set = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)

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(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])
```

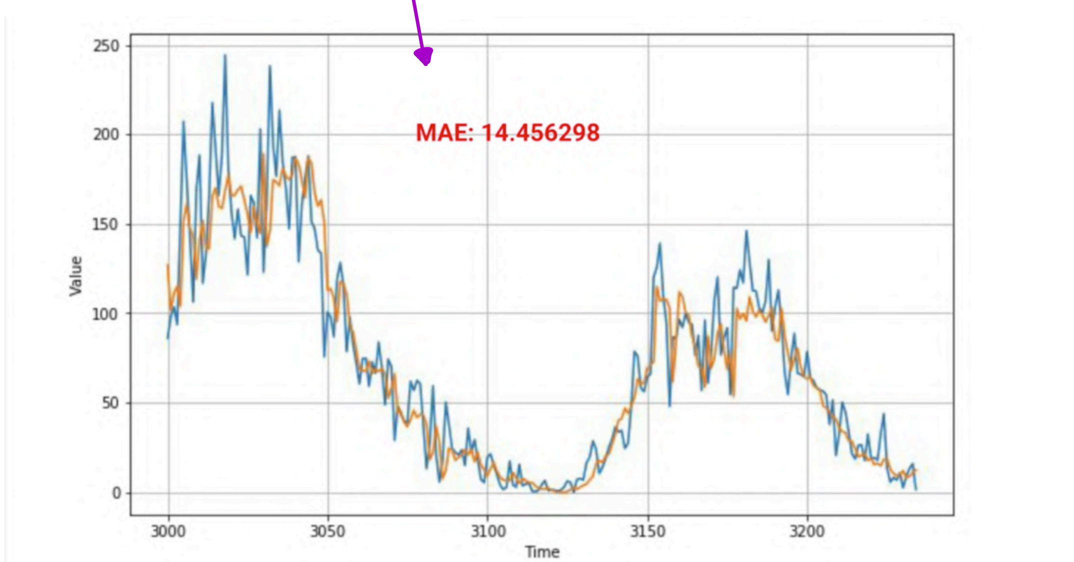
```
window_size = 60
batch_size = 64
train_set = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)

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(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])
```

```
window_size = 60
batch_size = 64
train_set = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)

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(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])
```

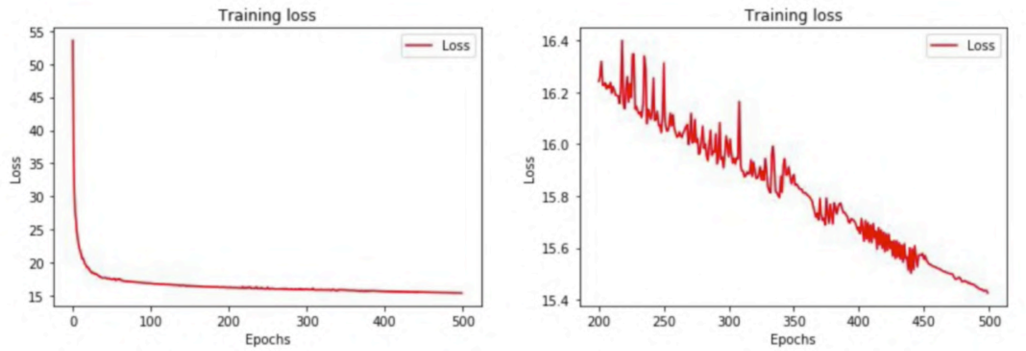
As our numbers are in the 1-400 range, there is a Lambda layer that multiplies out our X by 400.



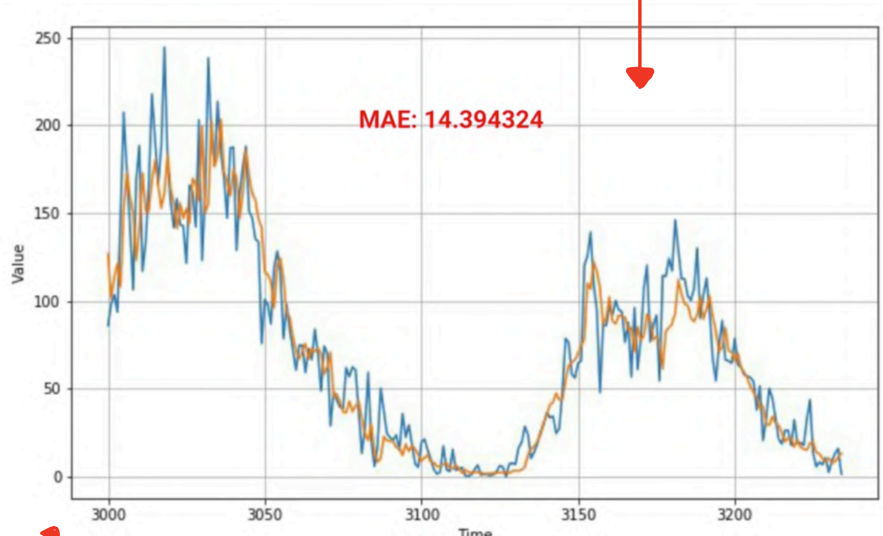

```
train_set = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv1D(filters=32, kernel_size=5,
                      strides=1, padding="causal",
                      activation="relu",
                      input_shape=[None,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


optimizer = tf.keras.optimizers.SGD(lr=1e-5, momentum=0.9)
model.compile(loss=tf.keras.losses.Huber(),
              optimizer=optimizer,
              metrics=["mae"])
history = model.fit(train_set, epochs=500)
```



When I look at my loss function during training, I can see that there's a lot of noise which tells me that I can certainly optimize it a bit

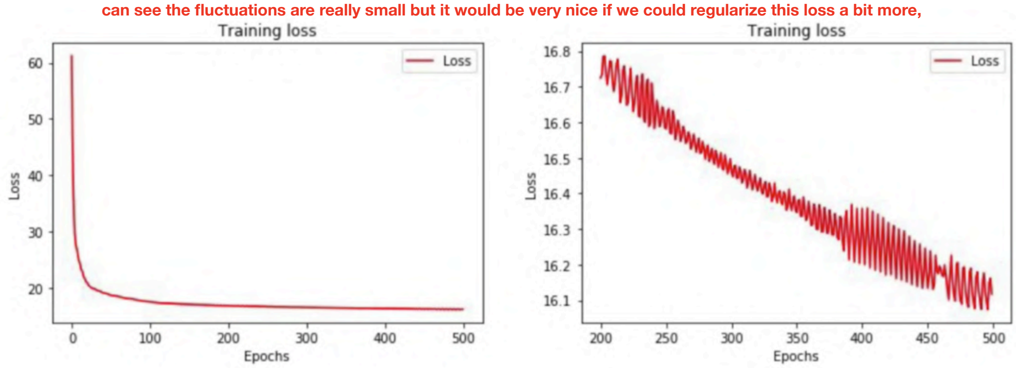



```
train_set = windowed_dataset(x_train, window_size, batch_size=256, shuffle_buffer_size)
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv1D(filters=32, kernel_size=5,
                      strides=1, padding="causal",
                      activation="relu",
                      input_shape=[None, 1]),
  tf.keras.layers.LSTM(32, return_sequences=True),
  tf.keras.layers.LSTM(32, return_sequences=True),
  tf.keras.layers.Dense(30, activation="relu"),
  tf.keras.layers.Dense(10, activation="relu"),
  tf.keras.layers.Dense(1),
  tf.keras.layers.Dense(1),
  tf.keras.layers.Lambda(lambda x: x * 400)
])


optimizer = tf.keras.optimizers.SGD(lr=1e-5, momentum=0.9)
model.compile(loss=tf.keras.losses.Huber(),
              optimizer=optimizer,
              metrics=["mae"])
history = model.fit(train_set,epochs=500)
```



After 500 epochs, my predictions have improved a little which is a step in the right direction.

But look at my training noise. Particularly towards the end of the training is really noisy but it's a very regular looking wave. This suggests that my larger batch size was good, but maybe a little off. It's not catastrophic because as you can see the fluctuations are really small but it would be very nice if we could regularize this loss a bit more,




```
train_set = windowed_dataset(x_train, window_size=60, batch_size=250, shuffle_buffer_size)
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv1D(filters=60, kernel_size=5,
                      strides=1, padding="causal",
                      activation="relu",
                      input_shape=[None, 1]),
  tf.keras.layers.LSTM(60, return_sequences=True),
  tf.keras.layers.LSTM(60, 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)
])


optimizer = tf.keras.optimizers.SGD(lr=1e-5, momentum=0.9)
model.compile(loss=tf.keras.losses.Huber(),
              optimizer=optimizer,
              metrics=["mae"])
history = model.fit(train_set,epochs=500)
```

My training data has 3,000 data points in it. So why are things like my window size and batch size powers of two that aren't necessarily evenly divisible into 3,000?

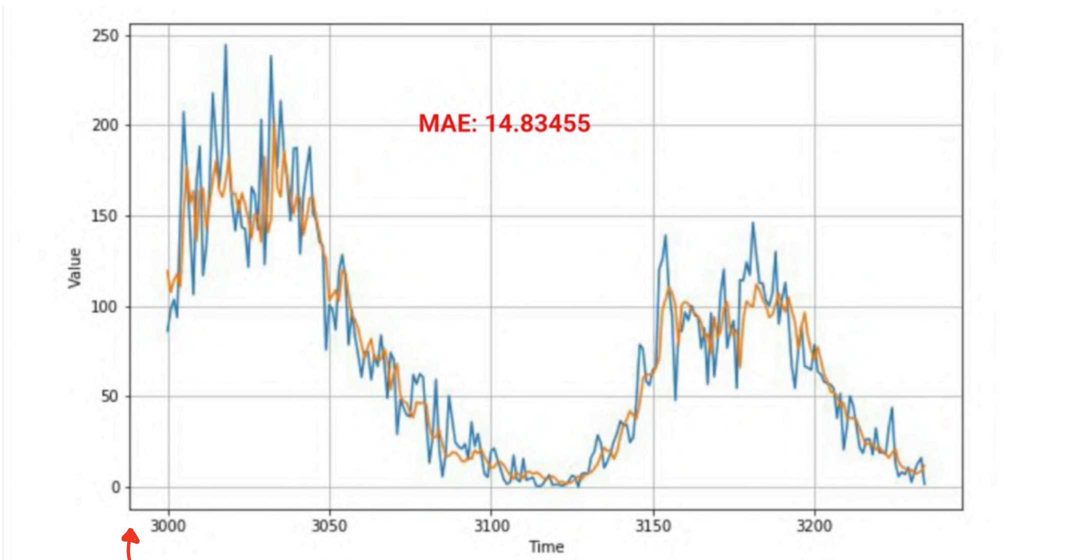
What would happen if I were to change my parameters to suit,

and not just the window and batch size, how about changing the filters too?

So what if I set that to 60, and the LSTMs to 60 instead of 32 or 64?

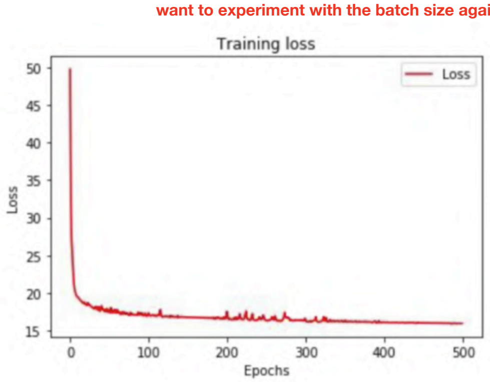
```
train_set = windowed_dataset(x_train, window_size=60, batch_size=250, shuffle_buffer_size)
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv1D(filters=60, kernel_size=5,
                      strides=1, padding="causal",
                      activation="relu",
                      input_shape=[None, 1]),
  tf.keras.layers.LSTM(60, return_sequences=True),
  tf.keras.layers.LSTM(60, 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)
])


optimizer = tf.keras.optimizers.SGD(lr=1e-5, momentum=0.9)
model.compile(loss=tf.keras.losses.Huber(),
              optimizer=optimizer,
              metrics=["mae"])
history = model.fit(train_set,epochs=500)
```

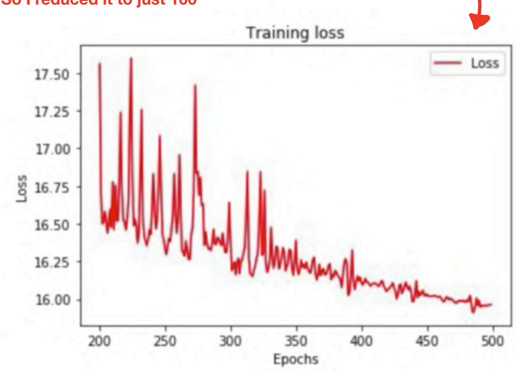
```
train_set = windowed_dataset(x_train, window_size=60, batch_size=250, shuffle_buffer_size)
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv1D(filters=60, kernel_size=5,
                      strides=1, padding="causal",
                      activation="relu",
                      input_shape=[None, 1]),
  tf.keras.layers.LSTM(60, return_sequences=True),
  tf.keras.layers.LSTM(60, 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)
])


optimizer = tf.keras.optimizers.SGD(lr=1e-5, momentum=0.9)
model.compile(loss=tf.keras.losses.Huber(),
              optimizer=optimizer,
              metrics=["mae"])
history = model.fit(train_set,epochs=500)
```

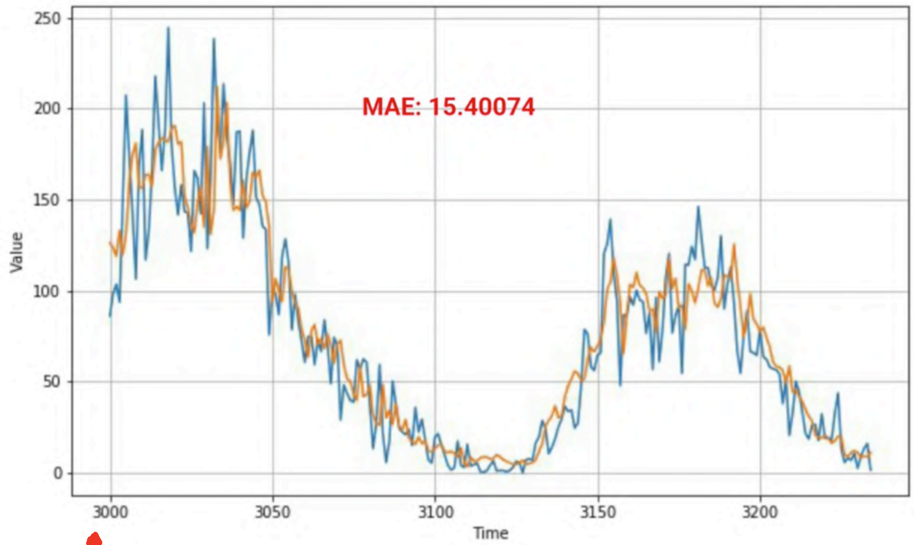


So after training this for 500 epochs, my scores improved again albeit slightly. So it shows we're heading in the right direction.

What's interesting is that the noise and the loss function actually increased the bits, and that made me want to experiment with the batch size again. So I reduced it to just 100







Now here my MAE has actually gone up a little.

The projections are doing much better in the higher peaks than earlier but the overall accuracy has gone down, and the loss has smoothed out except for a couple of large blips.