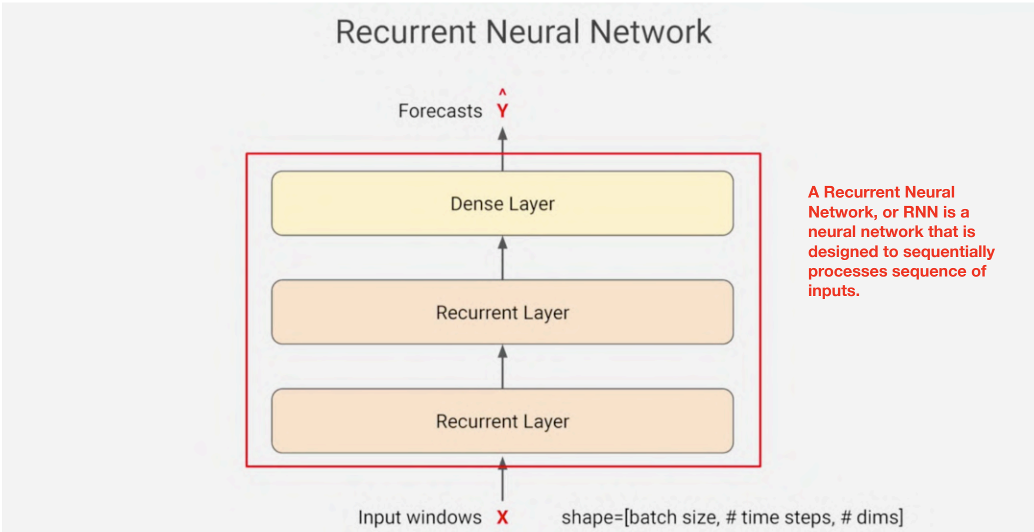
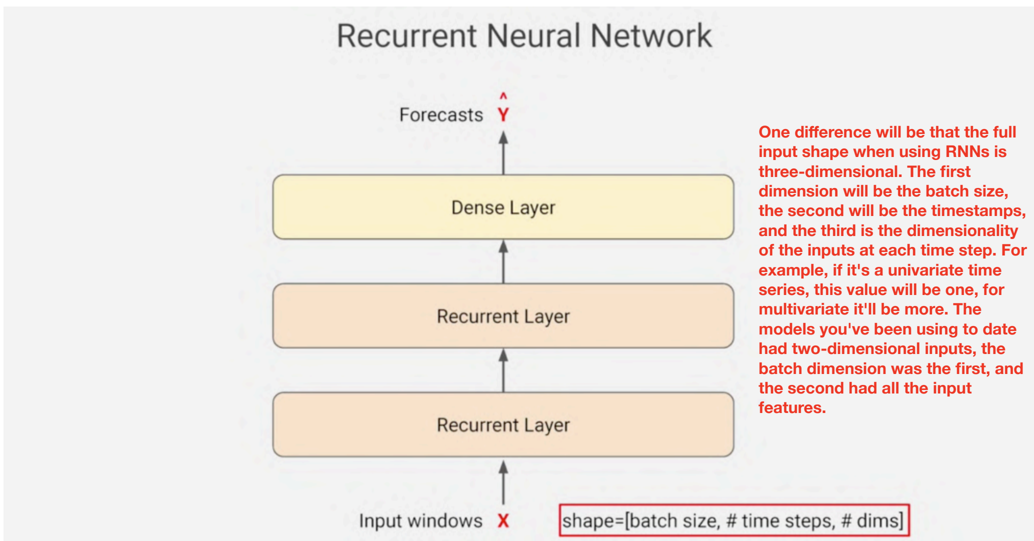
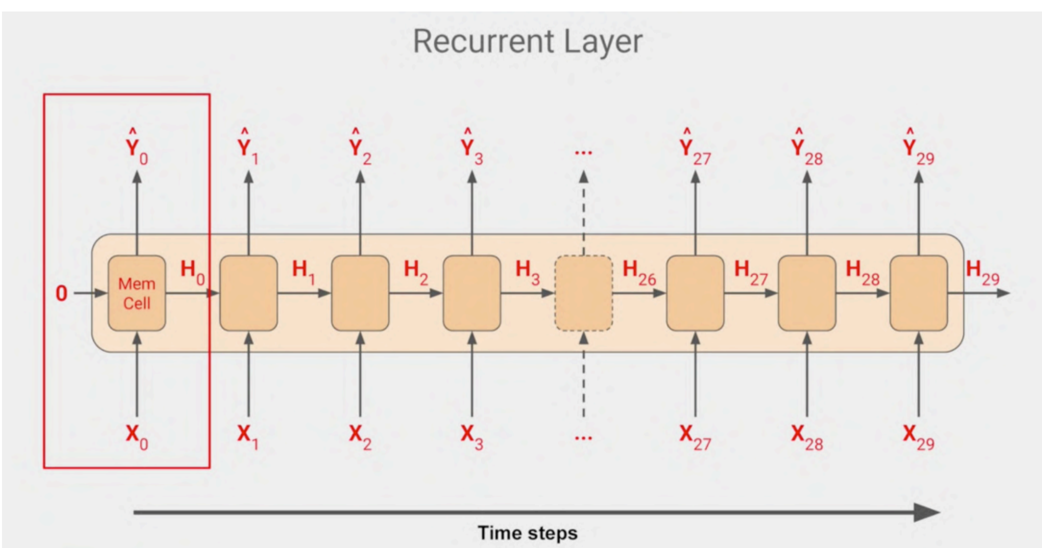
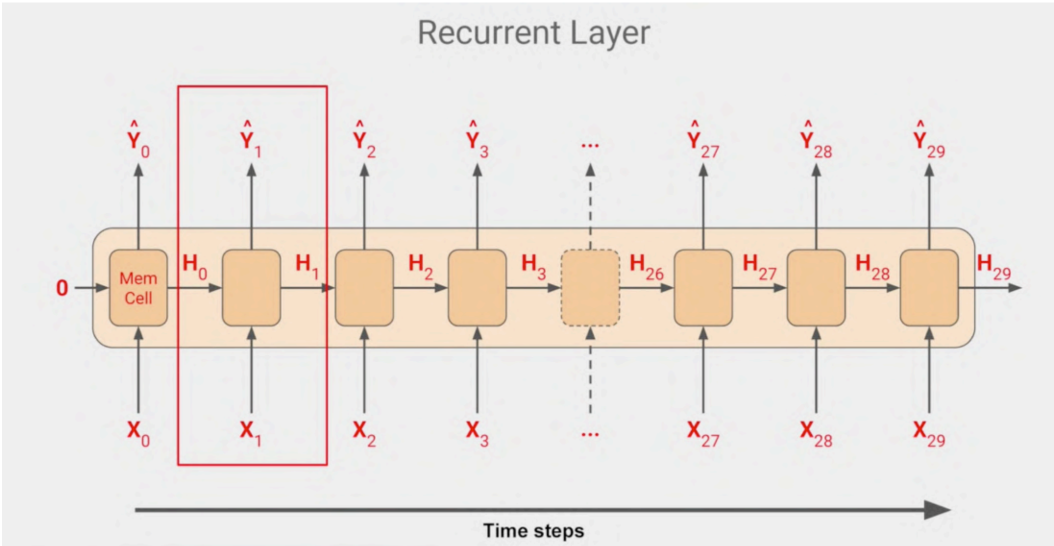
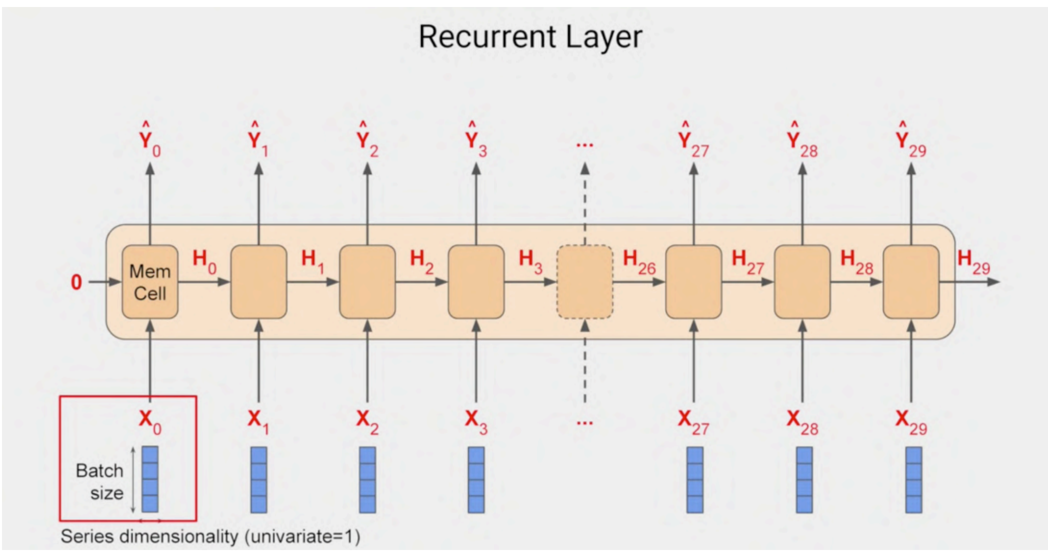
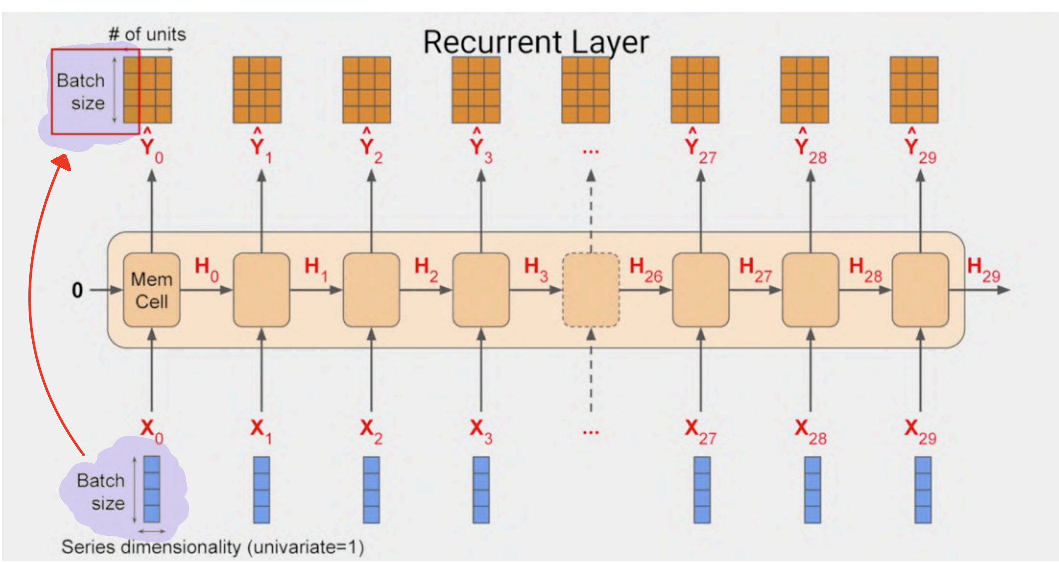

Recurrent Neural Network
Forecasts Ŷ
Dense Layer
Recurrent Layer
Recurrent Layer
Input windows X
shape=[batch size, # time steps, # dims]
A Recurrent Neural Network, or RNN is a neural network that is designed to sequentially processes sequence of inputs.

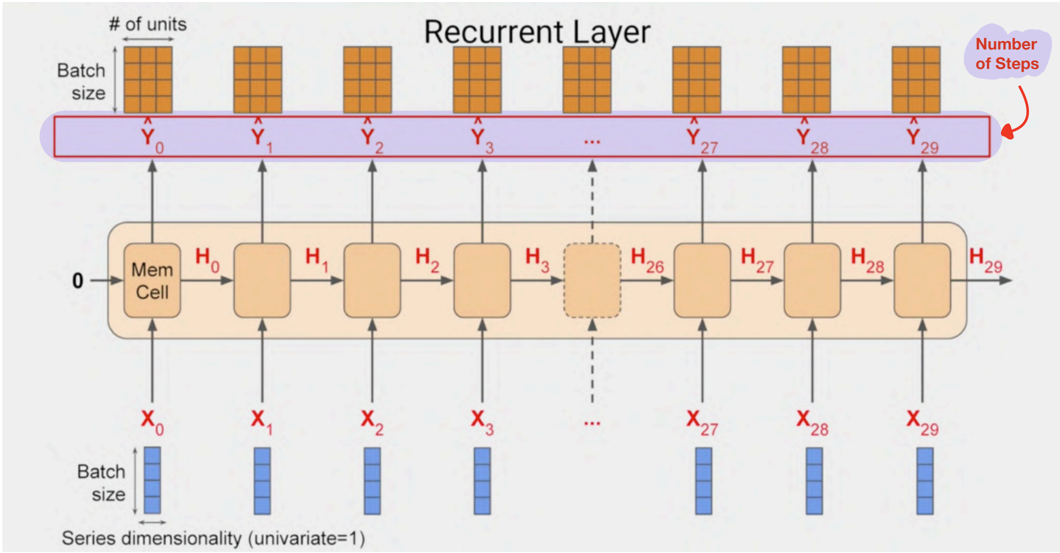

Recurrent Neural Network
Forecasts Ŷ
Dense Layer
Recurrent Layer
Recurrent Layer
Input windows X
shape=[batch size, # time steps, # dims]
One difference will be that the full input shape when using RNNs is three-dimensional. The first dimension will be the batch size, the second will be the timestamps, 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. The models you've been using to date had two-dimensional inputs, the batch dimension was the first, and the second had all the input features.


Recurrent Layer
$\hat{Y}_0$ $\hat{Y}_1$ $\hat{Y}_2$ $\hat{Y}_3$ ... $\hat{Y}_{27}$ $\hat{Y}_{28}$ $\hat{Y}_{29}$
0
Mem Cell
H_0 H_1 H_2 H_3 H_{26} H_{27} H_{28} H_{29}
X_0 X_1 X_2 X_3 ... X_{27} X_{28} X_{29}
Time steps

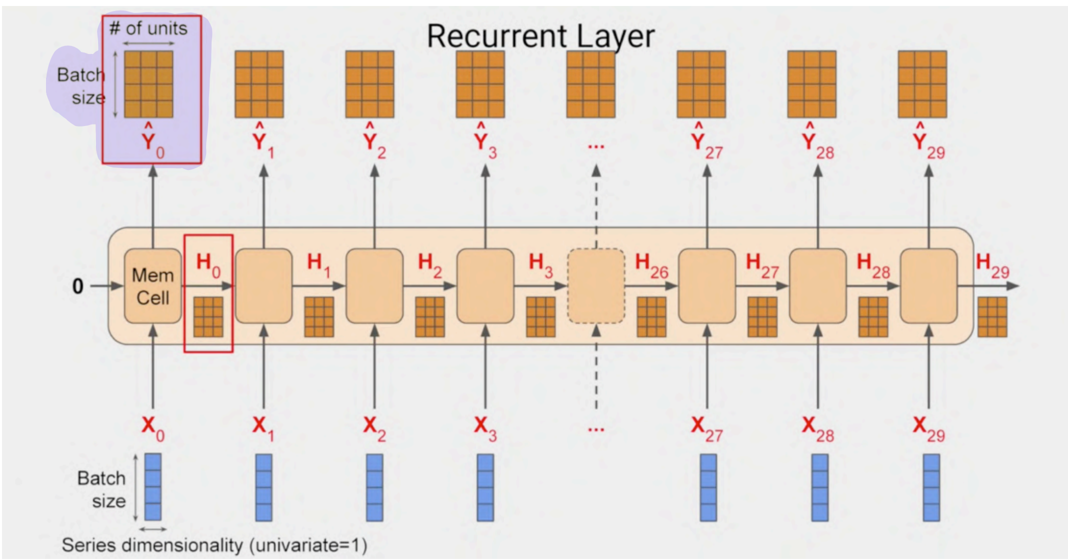

Recurrent Layer
$\hat{Y}_0$ $\hat{Y}_1$ $\hat{Y}_2$ $\hat{Y}_3$ ... $\hat{Y}_{27}$ $\hat{Y}_{28}$ $\hat{Y}_{29}$
0
Mem Cell
H_0 H_1 H_2 H_3 H_{26} H_{27} H_{28} H_{29}
X_0 X_1 X_2 X_3 ... X_{27} X_{28} X_{29}
Time steps


Recurrent Layer
$\hat{Y}_0$ $\hat{Y}_1$ $\hat{Y}_2$ $\hat{Y}_3$ ... $\hat{Y}_{27}$ $\hat{Y}_{28}$ $\hat{Y}_{29}$
0
Mem Cell
H_0 H_1 H_2 H_3 H_{26} H_{27} H_{28} H_{29}
X_0 X_1 X_2 X_3 ... X_{27} X_{28} X_{29}
Batch size
Series dimensionality (univariate=1)

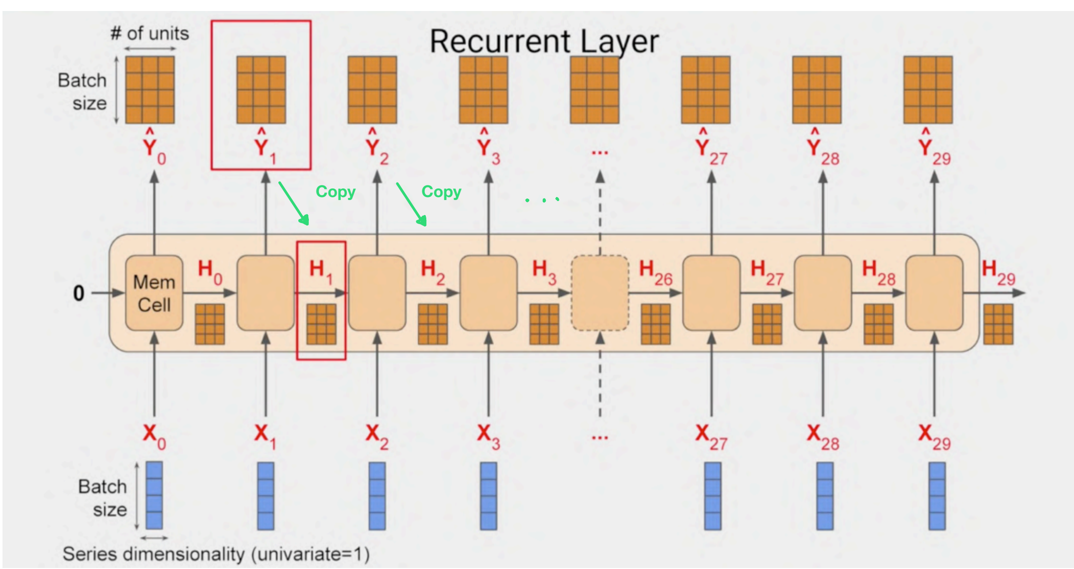

# of units
Batch size
Recurrent Layer
$\hat{Y}_0$ $\hat{Y}_1$ $\hat{Y}_2$ $\hat{Y}_3$ ... $\hat{Y}_{27}$ $\hat{Y}_{28}$ $\hat{Y}_{29}$
0
Mem Cell
H_0 H_1 H_2 H_3 H_{26} H_{27} H_{28} H_{29}
X_0 X_1 X_2 X_3 ... X_{27} X_{28} X_{29}
Batch size
Series dimensionality (univariate=1)



Recurrent Layer
# of units
Batch size
Number of Steps
$\hat{Y}_0$ $\hat{Y}_1$ $\hat{Y}_2$ $\hat{Y}_3$ ... $\hat{Y}_{27}$ $\hat{Y}_{28}$ $\hat{Y}_{29}$
0
Mem Cell
H_0 H_1 H_2 H_3 H_{26} H_{27} H_{28} H_{29}
X_0 X_1 X_2 X_3 ... X_{27} X_{28} X_{29}
Batch size
Series dimensionality (univariate=1)



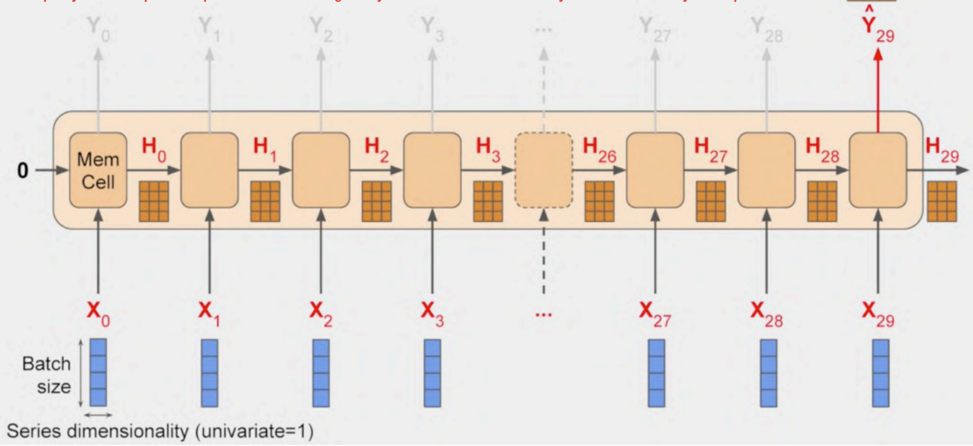
Recurrent Layer
# of units
Batch size
$\hat{Y}_0$ $\hat{Y}_1$ $\hat{Y}_2$ $\hat{Y}_3$ ... $\hat{Y}_{27}$ $\hat{Y}_{28}$ $\hat{Y}_{29}$
0
Mem Cell
H_0 H_1 H_2 H_3 H_{26} H_{27} H_{28} H_{29}
X_0 X_1 X_2 X_3 ... X_{27} X_{28} X_{29}
Batch size
Series dimensionality (univariate=1)



Recurrent Layer
# of units
Batch size
$\hat{Y}_0$ $\hat{Y}_1$ $\hat{Y}_2$ $\hat{Y}_3$ ... $\hat{Y}_{27}$ $\hat{Y}_{28}$ $\hat{Y}_{29}$
Copy
Copy
0
Mem Cell
H_0 H_1 H_2 H_3 H_{26} H_{27} H_{28} H_{29}
X_0 X_1 X_2 X_3 ... X_{27} X_{28} X_{29}
Batch size
Series dimensionality (univariate=1)

Sequence-to-Vector

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.




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

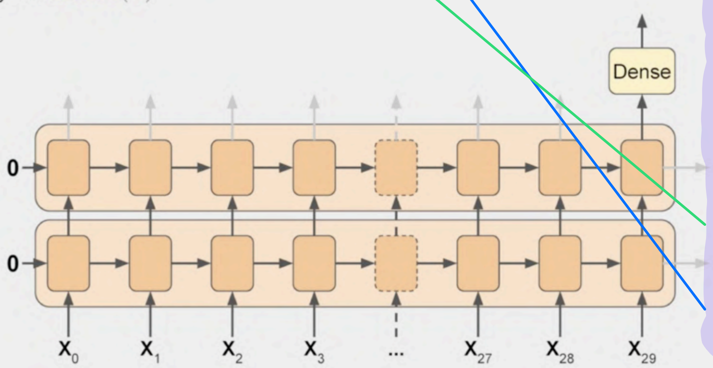
Consider this RNN, these has two recovered layers, and the first has return_sequences=True set up. It will output a sequence which is fed to the next layer.

The next layer does not have return_sequence that's set to True, so it will only output to the final step. But 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.

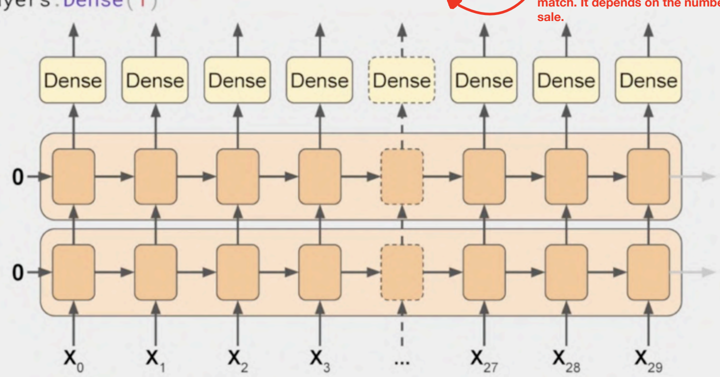
The last dimension is just one because we're using a unit vary of time series.




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

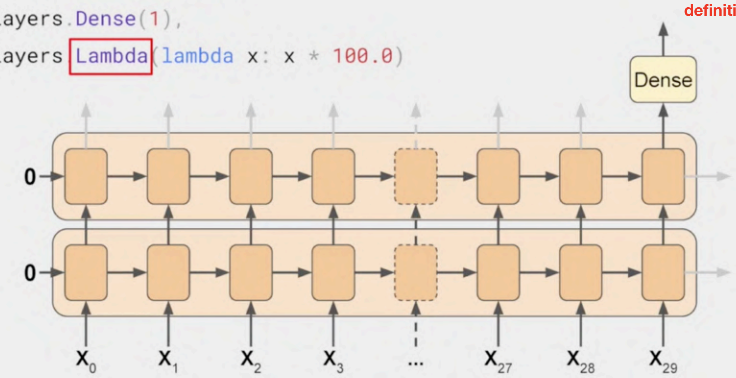
If we set return_sequences to true and all recurrent layers, then they will all output sequences and the dense layer will get a sequence as its inputs. Keras handles this by using the same dense layer independently at each time stamp.

It might look like multiple ones here but it's the same one that's being reused at each time step. This gives us what is called a sequence to sequence RNN. It's fed a batch of sequences and it returns a batch of sequences of the same length. The dimensionality may not always match. It depends on the number of units in the memory sale.



```
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)
])
```

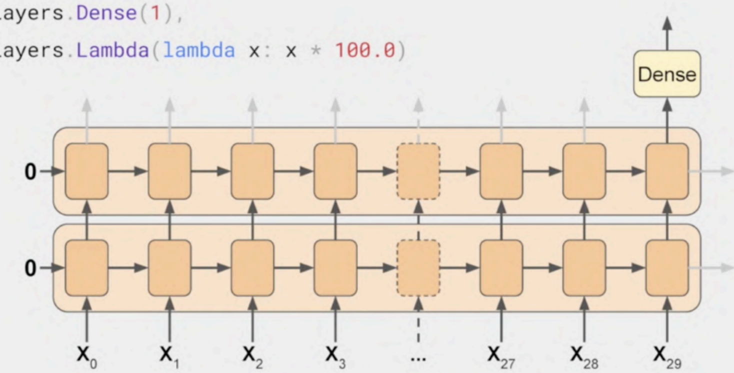
This type of layer is one that allows us to perform arbitrary operations to effectively expand the functionality of TensorFlow's kares, and we can do this within the model definition itself.




```
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)
])
```

So the first Lambda layer will be used to help us with our dimensionality. 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.

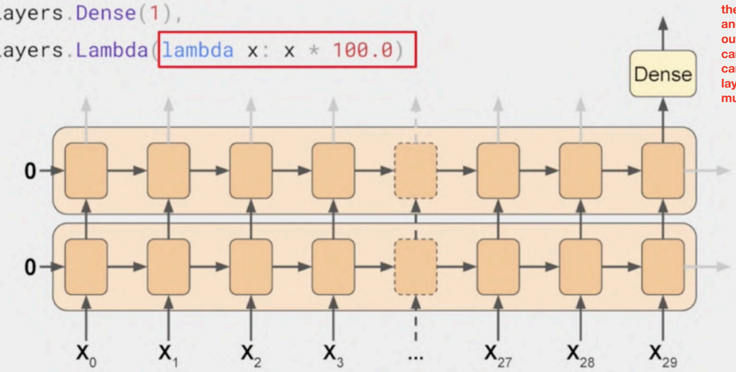
Using the Lambda, we just expand the array by one dimension. By setting input shape to none, we're saying that the model can take sequences of any length.




```
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)
])
```

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.



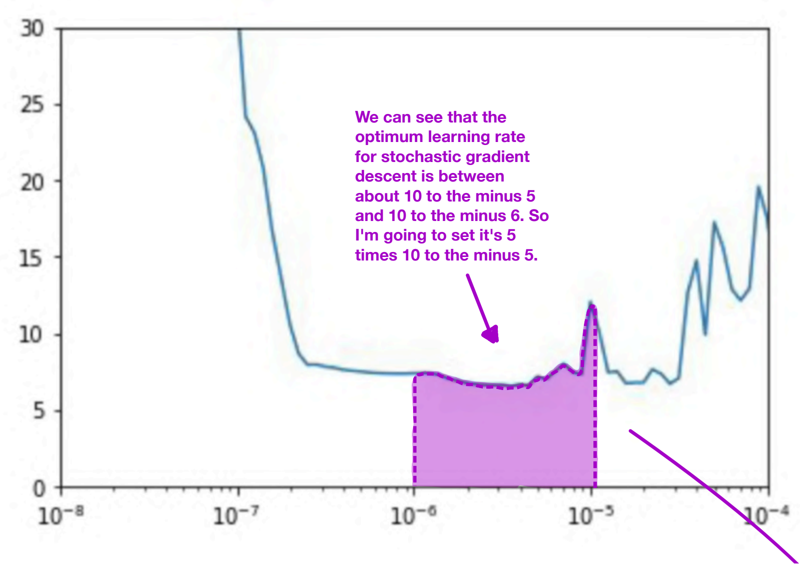
```python
train_set = windowed_dataset(x_train, window_size, batch_size=128,
shuffle_buffer=shuffle_buffer_size)

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])
```

To tune the learning rate, we'll set up a callback, which you can see here. Every epoch this just changes the learning rate a little so that it steps all the way from 1 times 10 to the minus 8 to 1 times 10 to the minus 6.

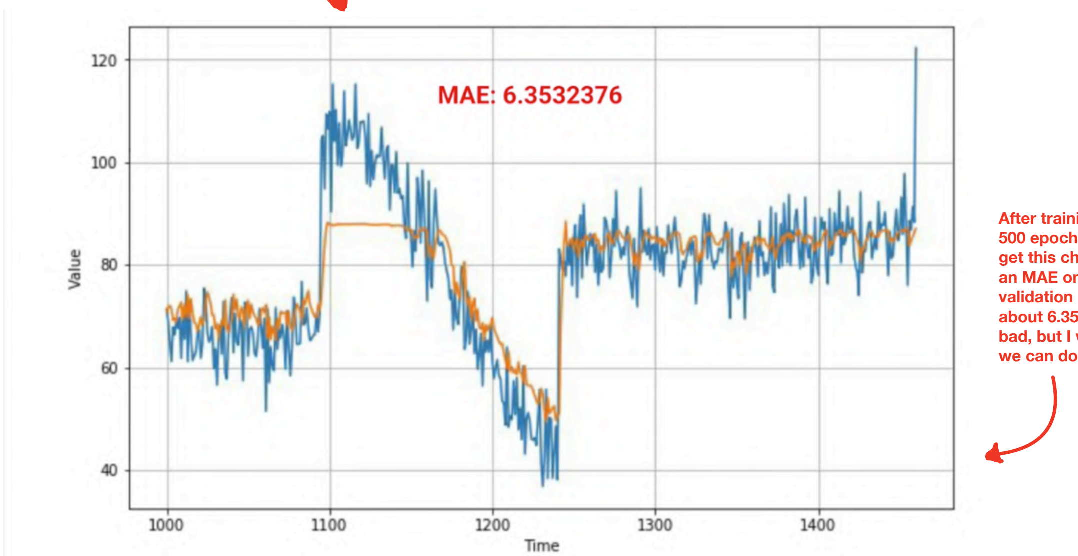

```python
train_set = windowed_dataset(x_train, window_size, batch_size=128,
shuffle_buffer=shuffle_buffer_size)

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.

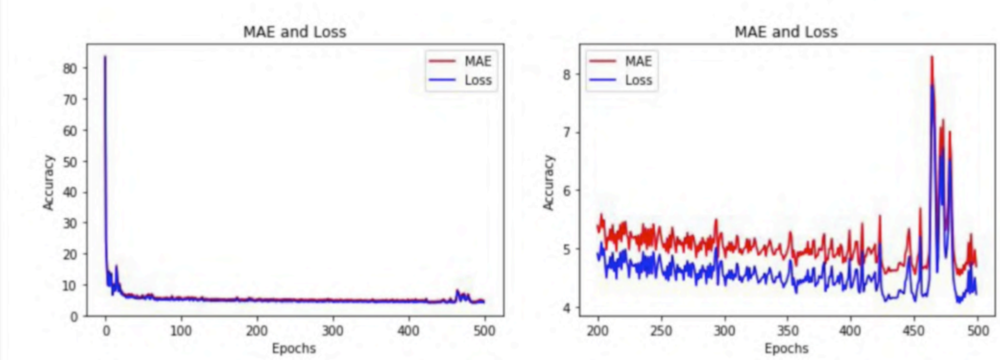
https://en.wikipedia.org/wiki/Huber_loss



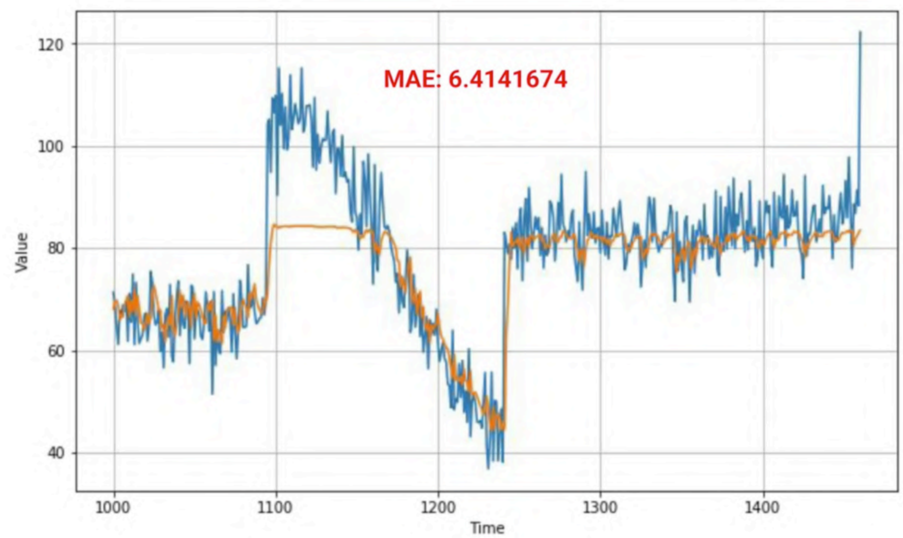
```
tf.keras.backend.clear_session()
tf.random.set_seed(51)
np.random.seed(51)

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)
```

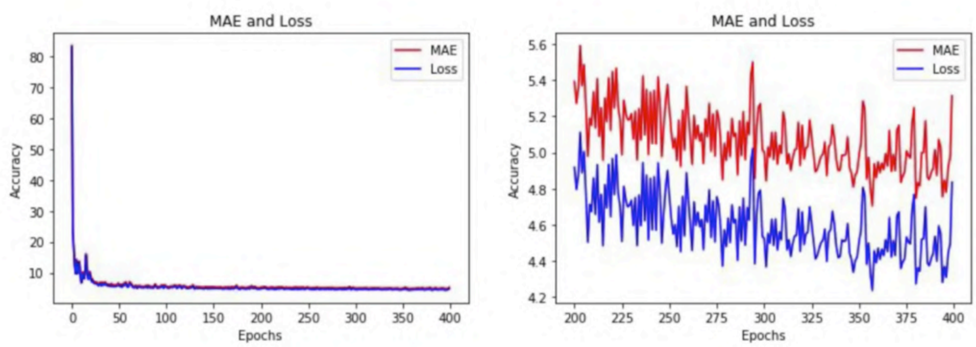


After training for 500 epochs, I will get this chart, with an MAE on the validation set of about 6.35. It's not bad, but I wonder if we can do better.

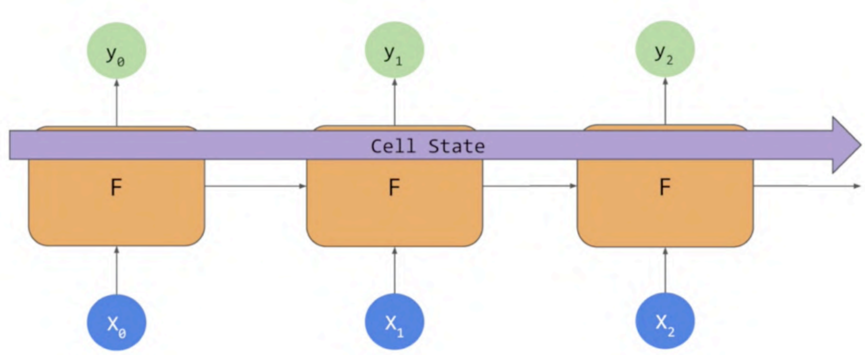


Here's the loss and the MAE during training with the chart on the right is zoomed into the last few epochs. As you can see, the trend was genuinely downward until a little after 400 epochs, when it started getting unstable. Given this, it's probably worth only training for about 400 epochs.

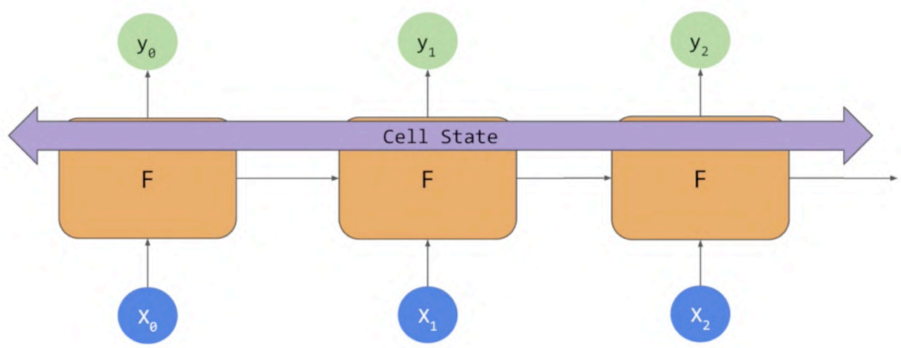

MAE: 6.4141674
Value
Time
120
100
80
60
40
1000
1100
1200
1300
1400


MAE and Loss
MAE
Loss
Accuracy
Epochs
MAE and Loss
MAE
Loss
Accuracy
Epochs

RNNs


Cell State
F
F
F

Bidirectional RNN



First of all is the tf.keras.backend.clear_session, and this clears any internal variables. That makes it easy for us to experiment without models impacting later versions of themselves.

```
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)),
  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,verbose=0)
```

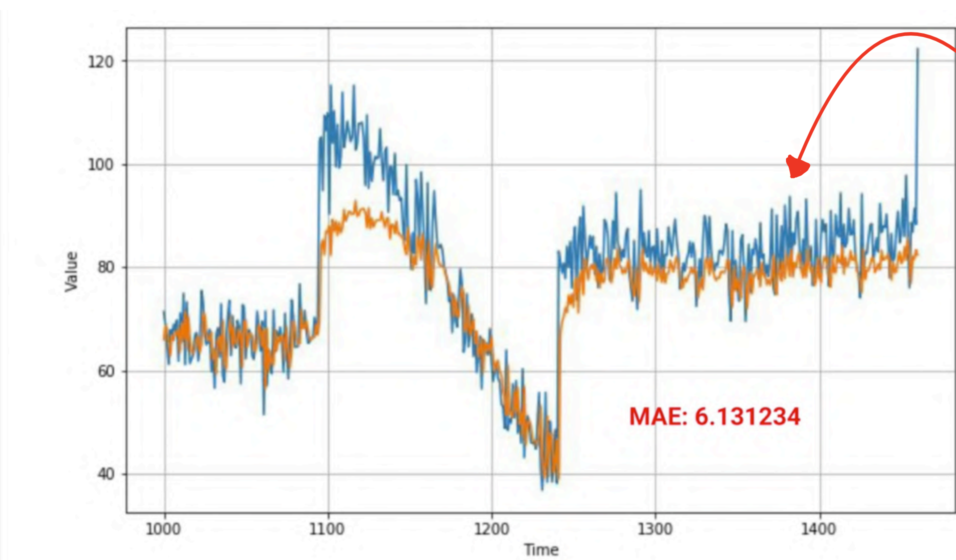
```
tf.keras.backend.clear_session()
dataset = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)

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)),
  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,verbose=0)
```

After the Lambda layer that expands the dimensions for us I've added a single LSTM layer with 32 cells. I've also made a bidirectional to see the impact of that on a prediction.

```
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)),
  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,verbose=0)
```

The output neuron will give us our prediction value.

```
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)),
  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,verbose=0)
```

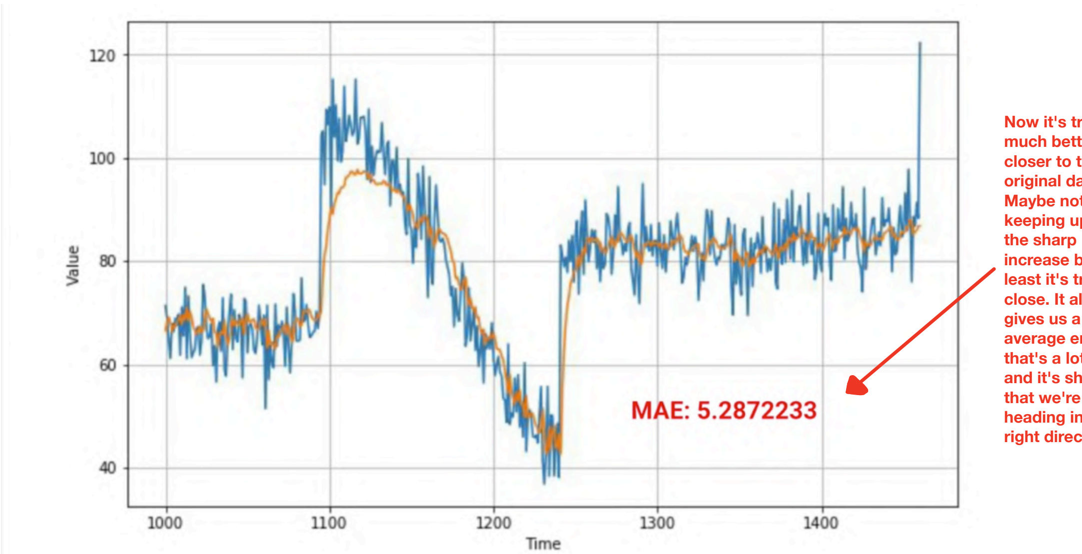


The plateau under the big spike is still there and are MAE is in the low sixes. It's not bad, it's not great, but it's not bad.

The predictions look like there might be a little bit on the low side too.

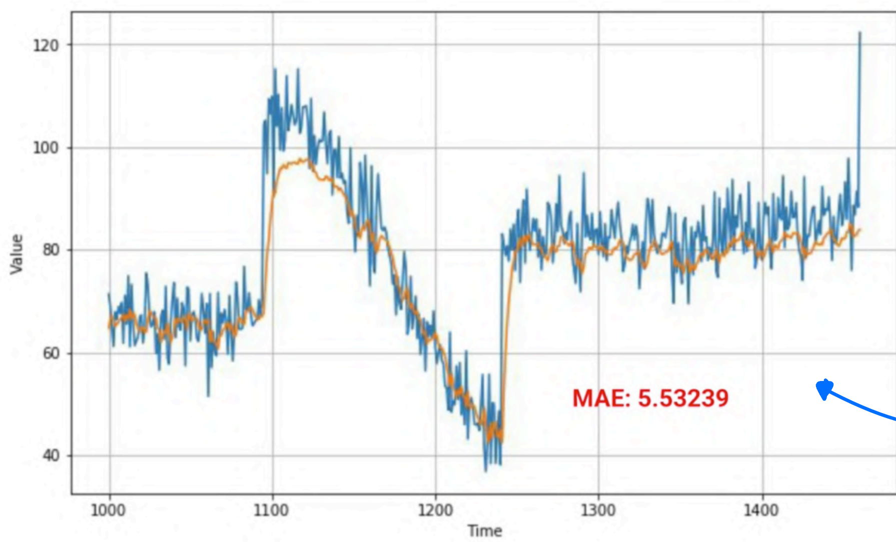
So let's edit our code to add another LSTM to see the impact. Now you can see the second layer and note that we had to set return sequences equal to true on the first one in order for this to work. We train on this and now we will see the following results

```python
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)),
  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, verbose=0)
```



Now it's tracking much better and closer to the original data. Maybe not keeping up with the sharp increase but at least it's tracking close. It also gives us a mean average error that's a lot better and it's showing that we're heading in the right direction.

```python
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)
```



If we edit our source to add a third LSTM layer like this, by adding the layer and having the second layer return sequences is true we can then train and run it, and we'll get the following output.

There's really not that much of a difference and are MAE has actually gone down.