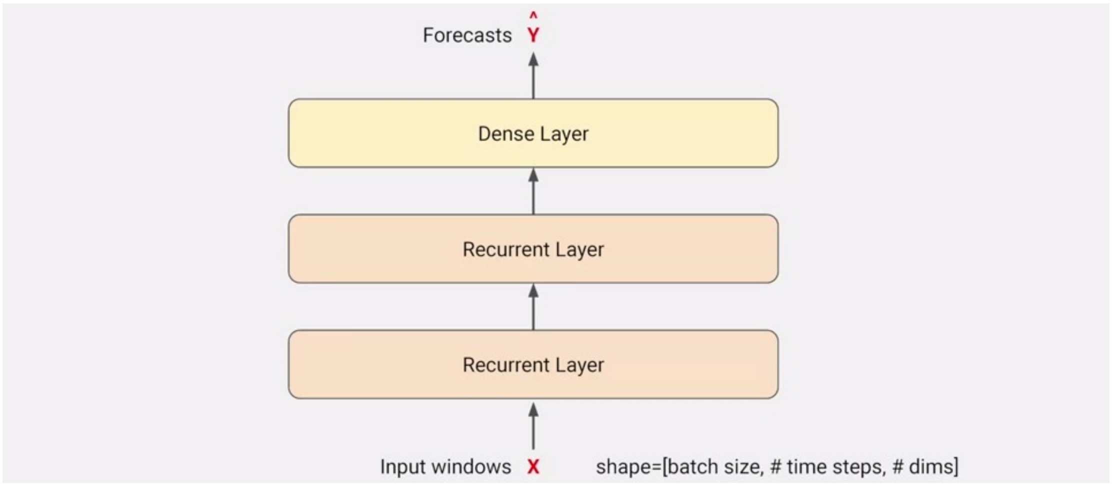
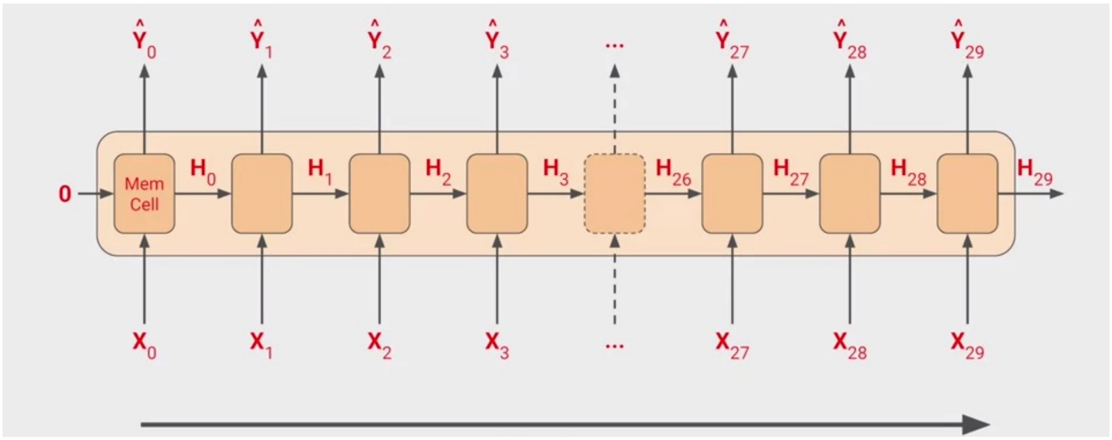
1. Overview

In the last couple of weeks, you've looked at creating neural networks to forecast time-series data. You started with some simple analytical techniques, which you then extend it to using Machine Learning to do a simple regression. From there you use the DNN that you tweaked a bit to get an even better model.



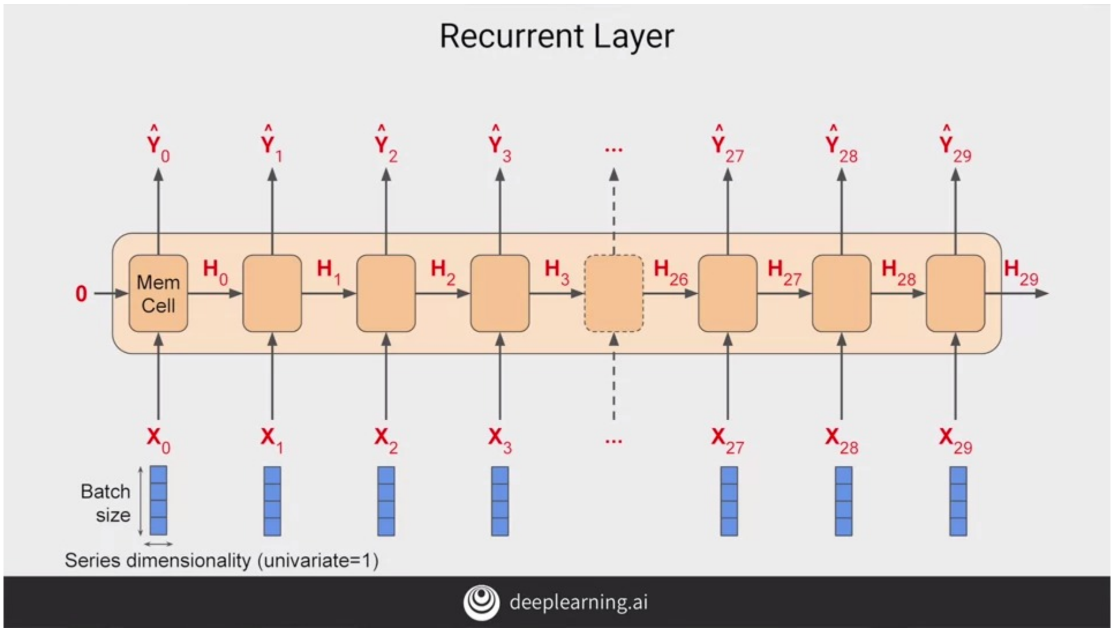
This week, we're going to look at RNNs for the task of prediction. A Recurrent Neural Network, or RNN is a neural network that contains recurrent layers. These are designed to sequentially processes sequence of inputs. RNNs are pretty flexible, able to process all kinds of sequences. As you saw in the previous course, they could've been used for predicting text. Here we'll use them to process the time series.
This example, will build an RNN that contains two recurrent layers and a final dense layer, which will serve as the output. With an RNN, you can feed it in batches of sequences, and it will output a batch of forecasts, just like we did last week. 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. But before going further, let's dig into the RNN layers to see how they work.



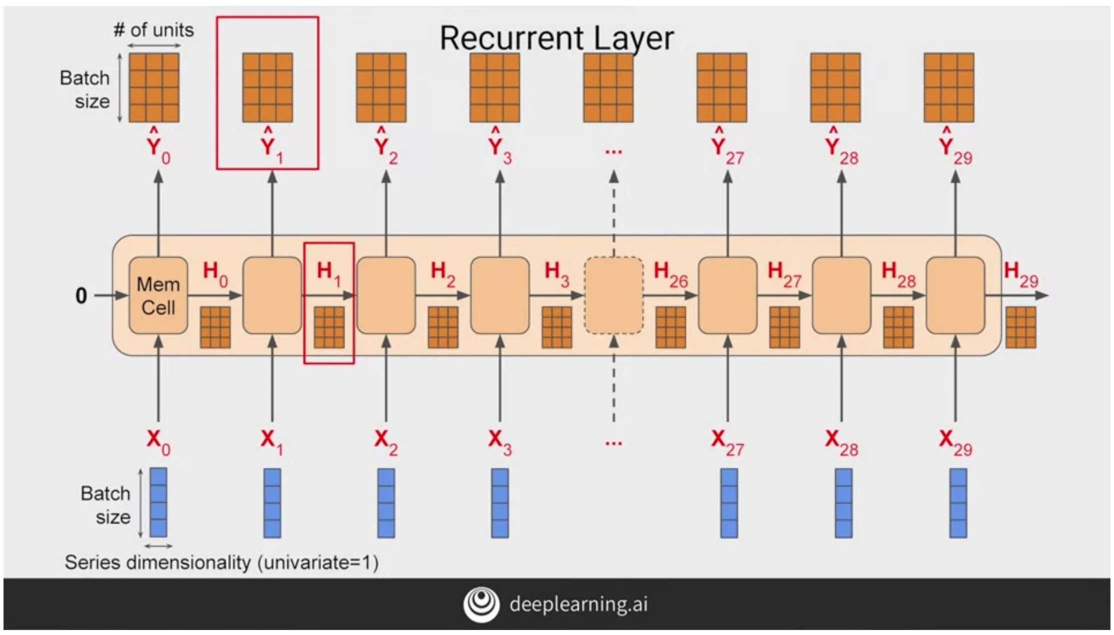
What it looks like there's lots of cells, there's actually only one, and it's used repeatedly to

compute the outputs. In this diagram, it looks like there's lots of them, but I'm just using the same one being reused multiple times by the layer. At each time step, the memory cell takes the input value for that step. So for example, it is zero at time zero, and zero state input. It then calculates the output for that step, in this case Y0, and a state vector H0 that's fed into the next step. H0 is fed into the cell with X1 to produce Y1 and H1, which is then fed into the cell at the next step with X2 to produce Y2 and H2. These steps will continue until we reach the end of our input dimension, which in this case has 30 values.
Now, this is what gives this type of architecture the name a recurrent neural network, because the values recur due to the output of the cell, a one-step being fed back into itself at the next time step. As we saw in the NLP course, this is really helpful in determining states. The location of a word in a sentence can determine it semantics. Similarly, for numeric series, things such as closer numbers in the series might have a greater impact than those further away from our target value.

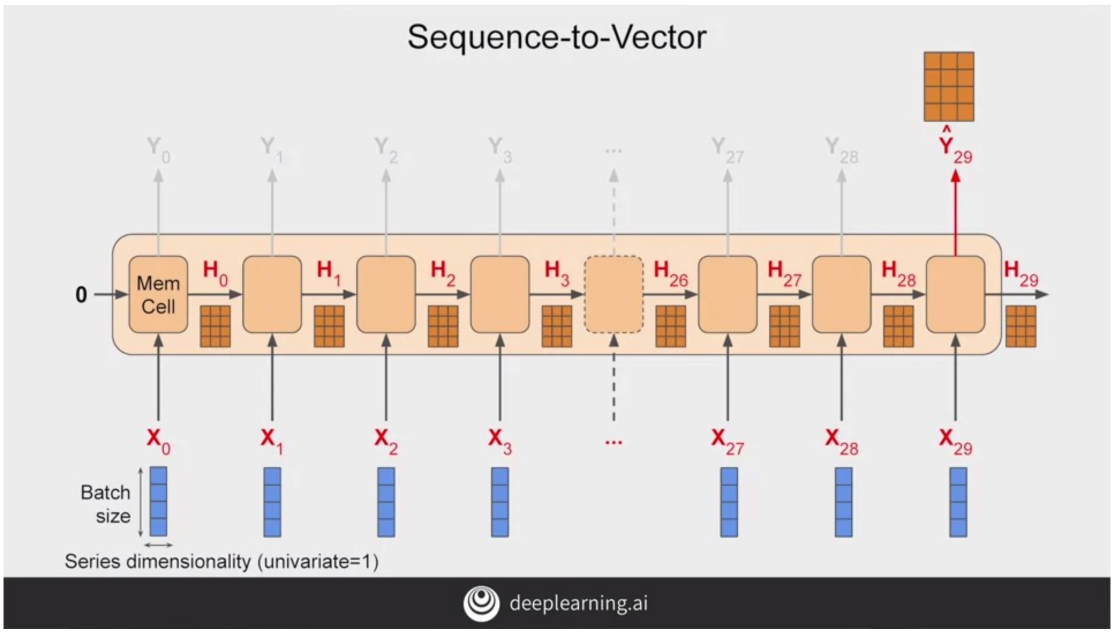
2. Shape of the inputs to the RNN



Okay. We've mentioned the shape of the data and the batches that the data is split up into. It's important to take a look at that, and let's dig into that next. 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 this. The cell will also take the input of the state matrix from the previous step. But of course in this case, in the first step, this will be zero. For subsequent ones, it'll be the output from the memory cell. But other than the state vector, the cell of course will output a Y value, which we can see here. 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.

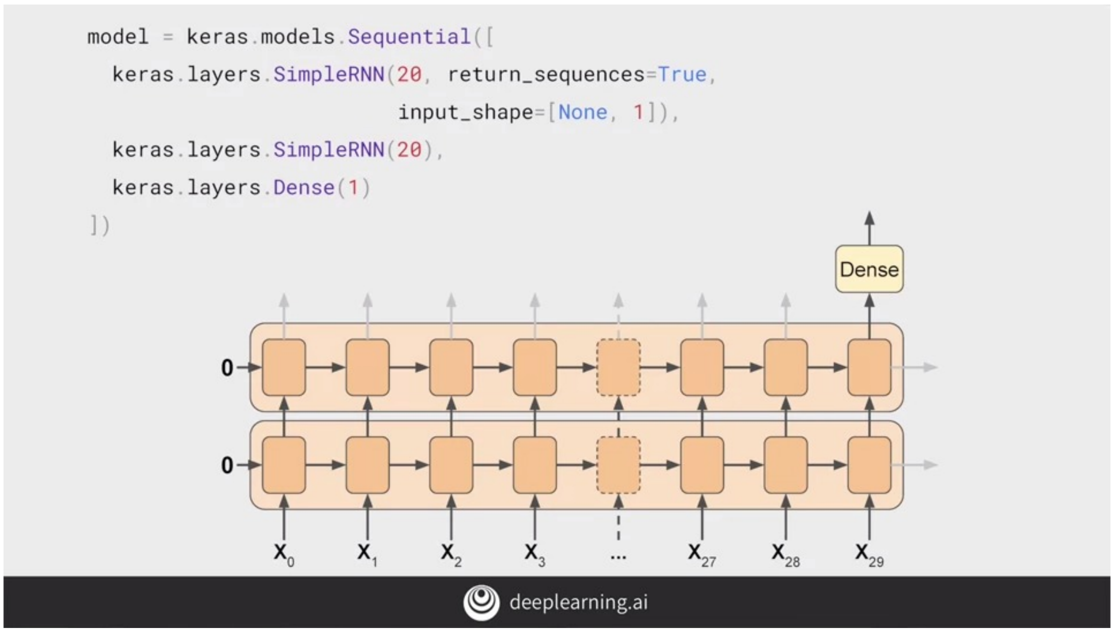

Recurrent Layer
# of units
Batch size
$\hat{Y}_0$ $\hat{Y}_1$ $\hat{Y}_2$ $\hat{Y}_3$ ... $\hat{Y}_{27}$ $\hat{Y}_{28}$ $\hat{Y}_{29}$
0
Mem Cell
H_0 H_1 H_2 H_3 H_{26} H_{27} H_{28} H_{29}
X_0 X_1 X_2 X_3 ... X_{27} X_{28} X_{29}
Batch size
Series dimensionality (univariate=1)
deeplearning.ai

So the full output of the layer is three dimensional, in this case, 4 by 30 by 3. With four being the batch size, three being the number of units, and 30 being the number of overall steps. In a simple RNN, the state output H is just a copy of the output matrix Y. So for example, H_0 is a copy of Y_0, H_1 is a copy of Y_1, and so on. So at each timestamp, the memory cell gets both the current input and also the previous output.


Sequence-to-Vector
Y_0 Y_1 Y_2 Y_3 ... Y_{27} 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)
deeplearning.ai

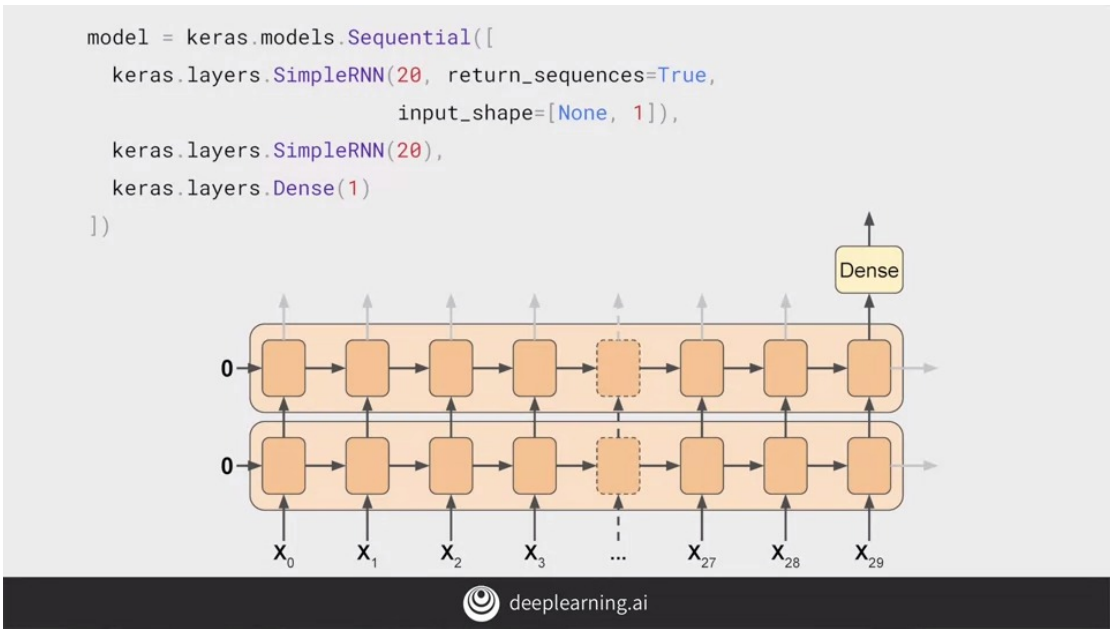
Now, in some cases, you might want to input a sequence, but you don't want to output on and you just want to get a single vector for each instance in the batch. This is typically called a sequence to vector RNN. But in reality, all you do is ignore all of the outputs, except the last one. When using 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.

3. Outputting a sequence



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

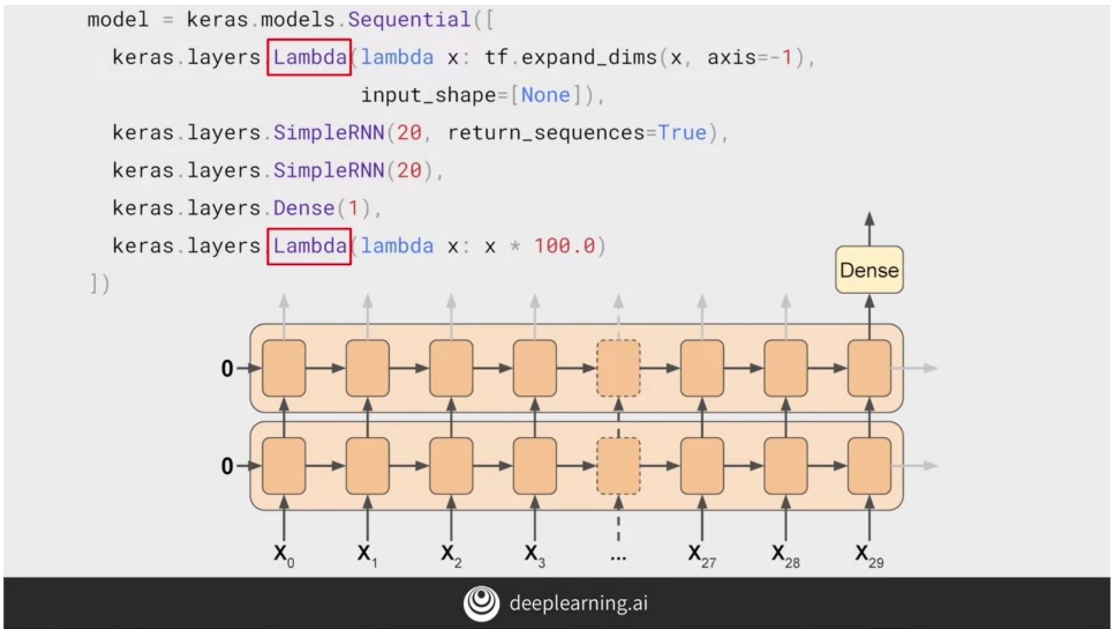
So consider this RNN, these has two recovered layers, and the first has return_sequences=True set up. It will output a sequence which is fed to the next layer. The next layer does not have return_sequence that's set to True, so it will only output to the final step. But notice the input_shape, it's set 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**.
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.SimpleRNN(20, return_sequences=True,
                         input_shape=[None, 1]),
  keras.layers.SimpleRNN(20),
  keras.layers.Dense(1)
])
```

So let's now return to a two-layer RNN that has the second one not return sequences. This will give us an output to a single dense.

4. Lambda layers



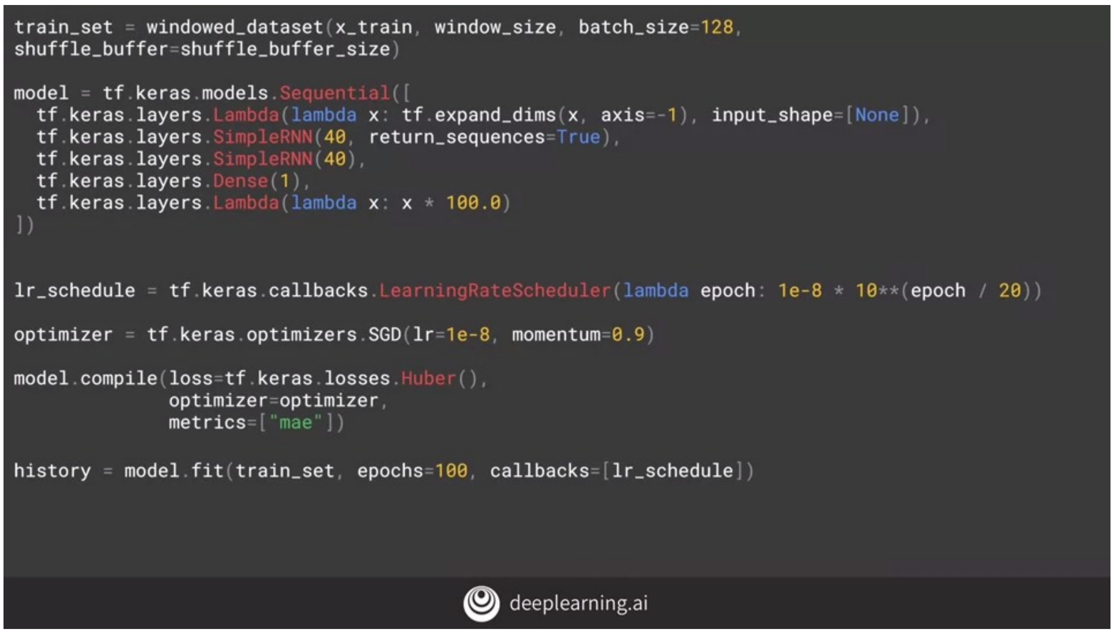
```
model = keras.models.Sequential([
  keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1),
                      input_shape=[None]),
  keras.layers.SimpleRNN(20, return_sequences=True),
  keras.layers.SimpleRNN(20),
  keras.layers.Dense(1),
  keras.layers.Lambda(lambda x: x * 100.0)
])
```

But I'd like to add a couple of new layers to this, layers that use the Lambda type. 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. 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. 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. So let's now take a look at what it takes to build out the full RNN so we can start doing some predictions with it. You'll see that in the next video.

5. Adjusting the learning rate dynamically

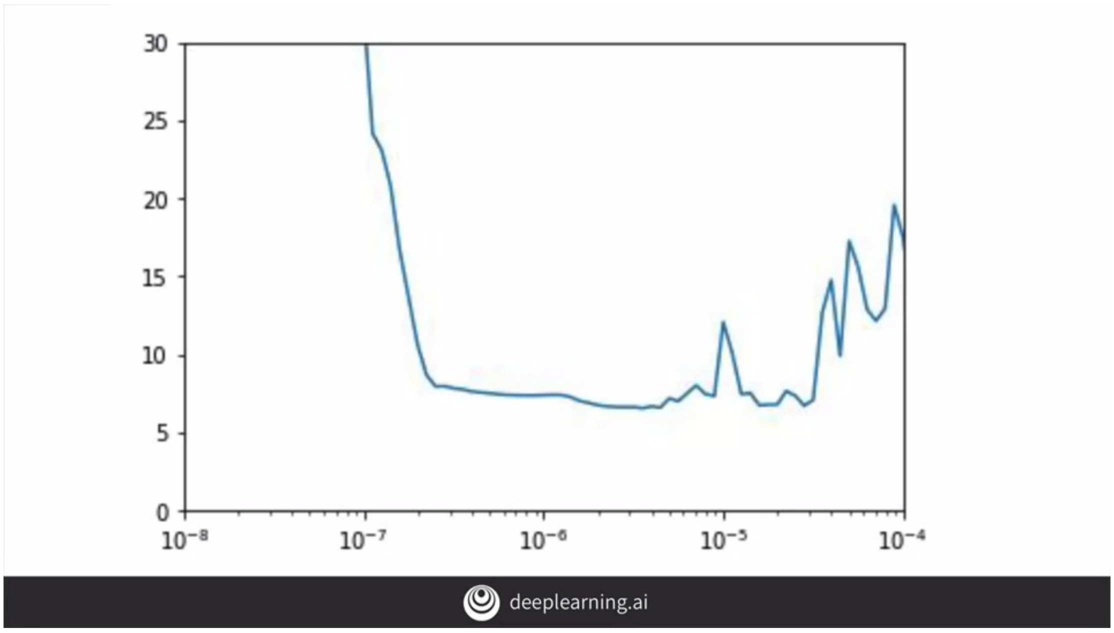
In the previous video, you got a look at RNNs and how they can be used for sequence to vector to sequence to sequence prediction. Let's now take a look at coding them for the problem at hand and seeing if we can get good predictions in our time series using them. One thing you'll see in the rest of the lessons going forward is that I'd like to write a little bit of code to optimize the neural network for the learning rate of the optimizer. Can be pretty quick to train and we can from there save a lot of time in our hyper-parameter tuning.



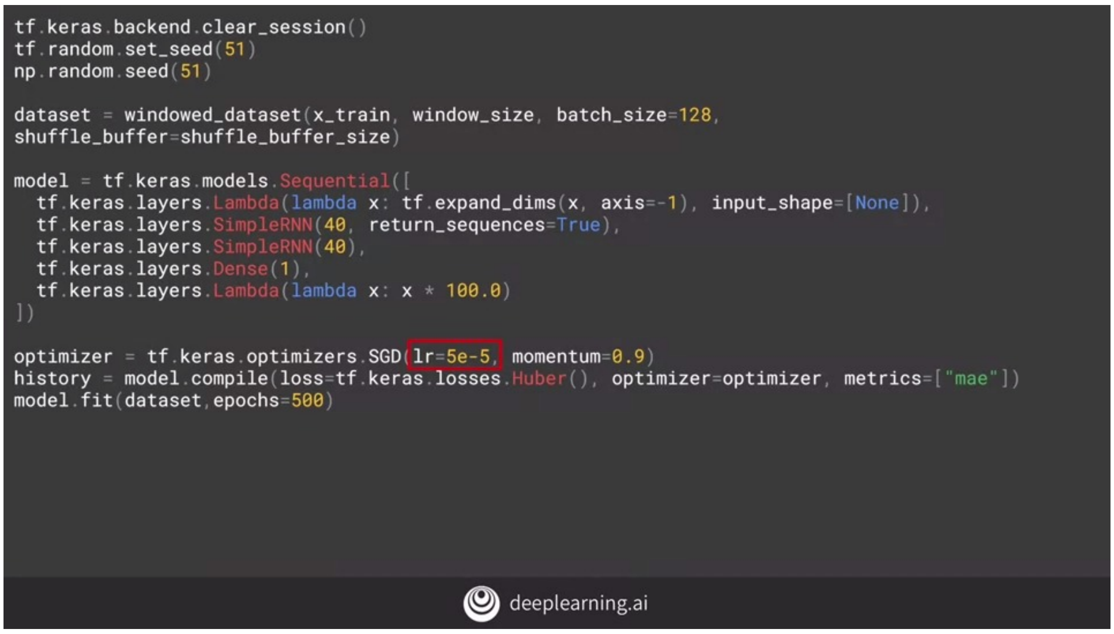
```
train_set = windowed_dataset(x_train, window_size, batch_size=128,
shuffle_buffer=shuffle_buffer_size)

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])
```

So here's the code for training the RNN with two layers each with 40 cells. 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. You can see that setup here while training.
I've also introduced **a new loss function to use called Huber** which you can see here. 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.

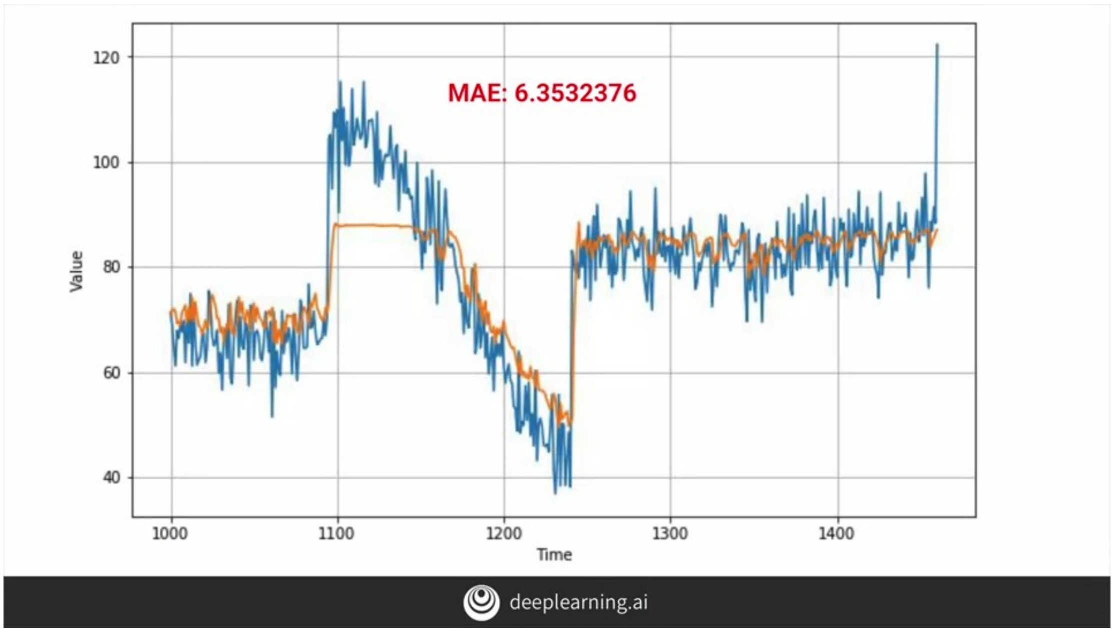


If I run this for 100 epochs and measure the loss at each epoch, I will see that my optimum learning rate for stochastic gradient descent is between about 10 to the minus 5 and 10 to the minus 6. So I'm going to set it's 5 times 10 to the minus 5.

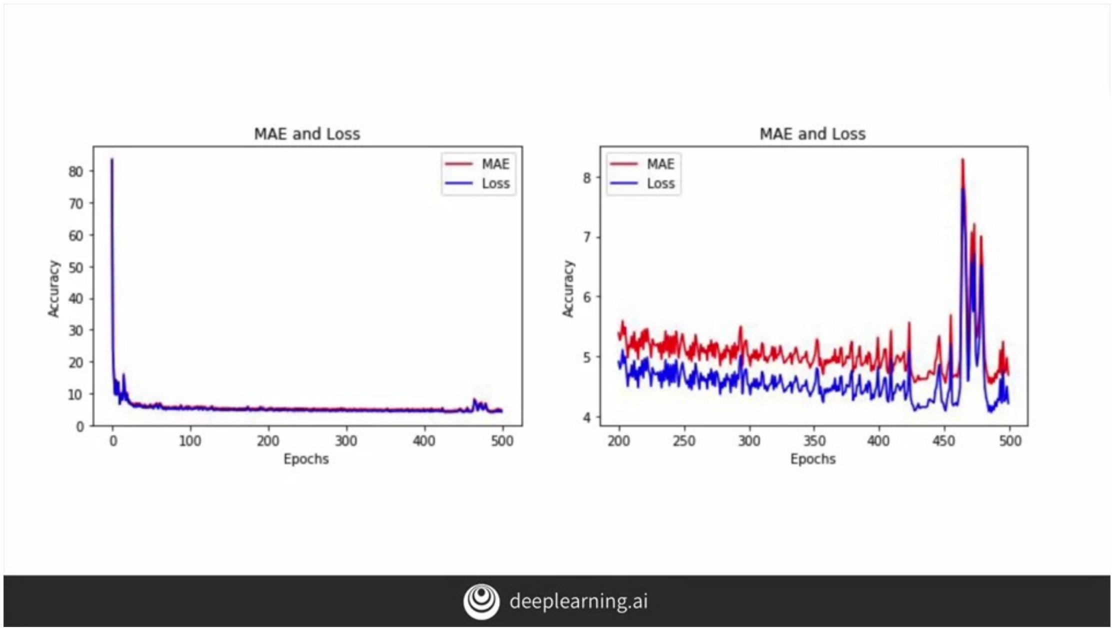


```
tf.keras.backend.clear_session()
tf.random.set_seed(51)
np.random.seed(51)

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)
```

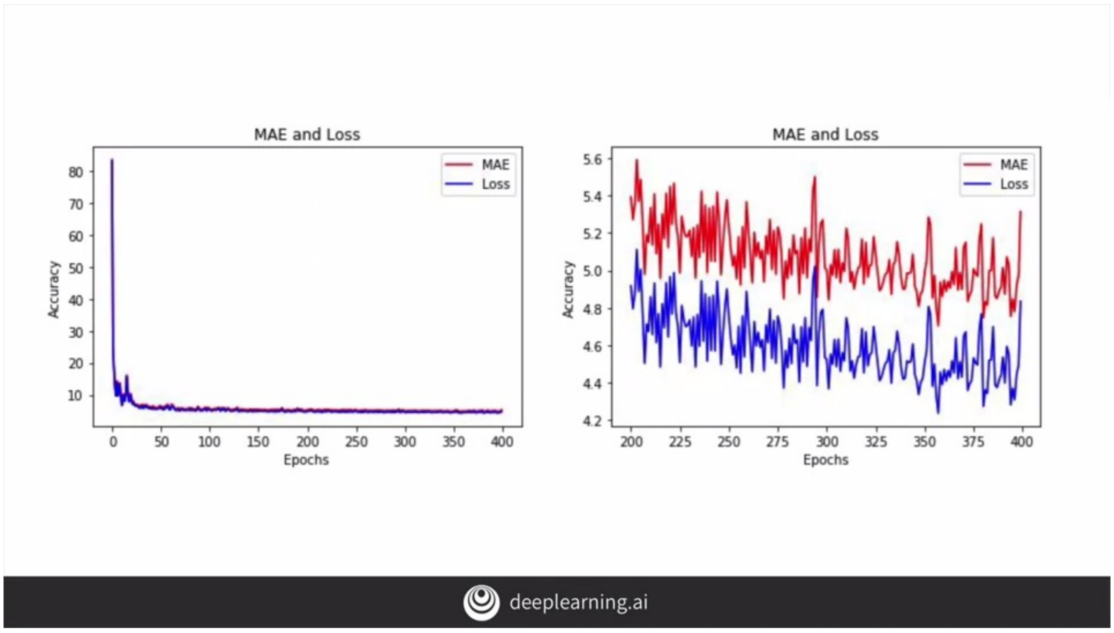
So now, I'll set my models compiled with that learning rate and the stochastic gradient descent optimizer.



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.



So 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. When I do that, I get these results. That's pretty much the same with the MAE only a tiny little bit higher, but we've saved 100 epochs worth of training to get it. So it's worth it.



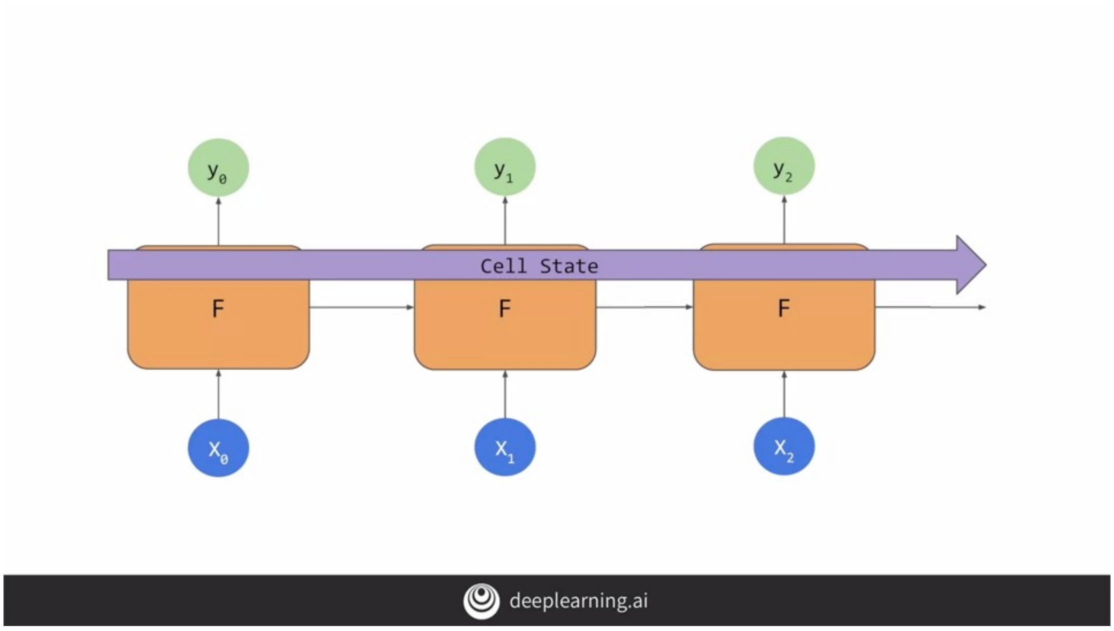
A quick look at the training MAE and loss gives us this results. So we've done quite well, and that was just using a simple RNN. Let's see how we can improve this with LSTMs and you'll see that in the next video.
Huber loss: https://en.wikipedia.org/wiki/Huber_loss

6. RNN

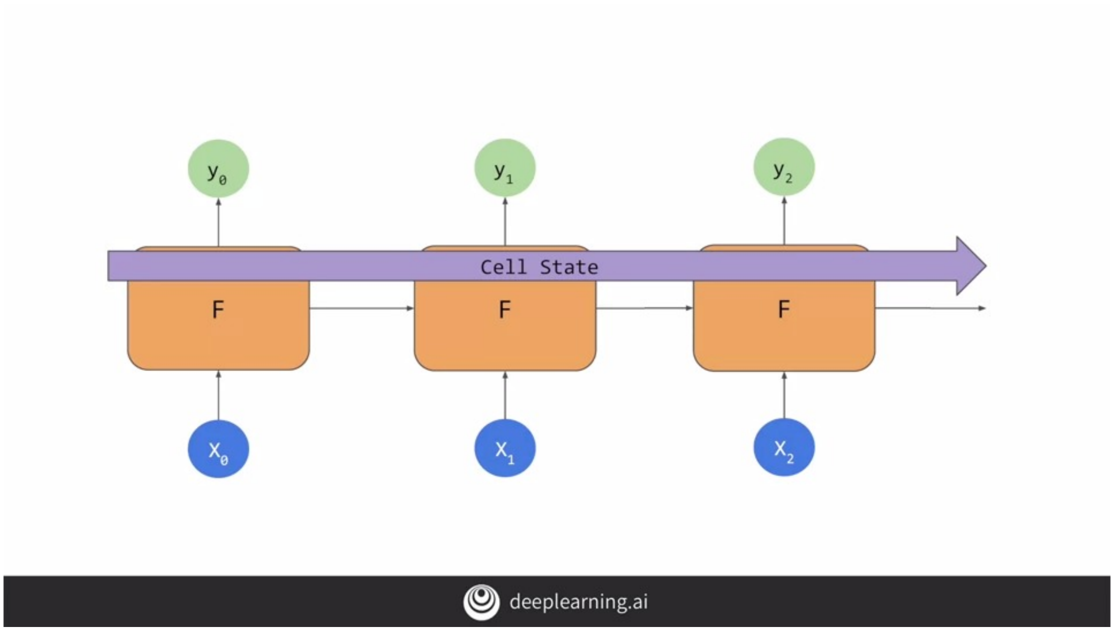
In the last video, we saw how to use RNNs for prediction of sequences, and we learned a little about how to construct a neural network with them. In this video, you'll work through a notebook with all of the code, and when you're done watching, you can try out the notebook for yourself. As usual, let's first ensure that we're running 2.0, and then we'll run the code to set up the data and create the dataset. Then we'll run the neural network code but adjusting the learning rate per epoch in order to find the best one for the full training. We'll plot the results to try to find the optimal learning rate. You can see here, it's between 10_minus 6 and 10_minus 5. So I pick a value halfway say, 5 times 10_minus 5. Here's the code to train the neural network. I've set the optimal learning rates and I've picked 400 epochs for which to train. I'll now train the neural network for the 400 epochs. Once it's trained, I can use it to forecast for the validation range and plot the results. On all my plot, I can see that my prediction isn't too bad other than this plateau, which is definitely going to impact my MA in a bad way. But despite that my MA is only about 6.41, so it's not too bad. If I plot the MA and loss for training, I'll see this. Showing that while it looks quite flat after about epoch 50, it was actually still reducing slowly, and when we zoom in, we can see that. We saw instability after about epoch 400 earlier, so I just stopped training at that point.
S+P Week 3 Lesson 2 - RNN

7. LSTM

In the previous videos, you experimented with using RNNs to predict values in your sequence. The results were good but they did need a bit of improvement as you were hitting strange plateaus in your predictions. You experimented with using different hyperparameters and you saw some improvement, but perhaps a better approach would be to use LSTMs instead of RNNs to see the impact.

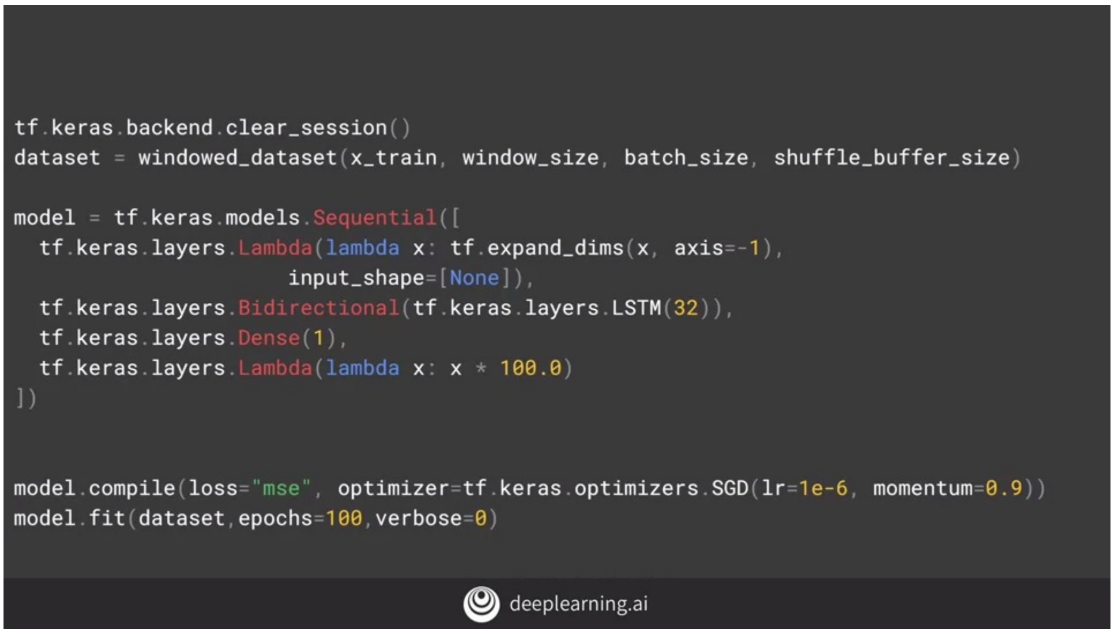


If you remember when you looked at RNNs, they looked a little bit like this. They had cells that took patches as inputs or X, and they calculated a Y output as well as the state vector, that fed into the cell along with the next X which then resulted in the Y, and the state vector and so on. The impact of this is that while state is a factor in subsequent calculations, its impacts can diminish greatly over timestamps. LSTMs are the cell state to this that keep a state throughout the life of the training so that the state is passed from cell to cell, timestamp to timestamp, and it can be better maintained. This means that the data from earlier in the window can have a greater impact on the overall projection than in the case of RNNs.



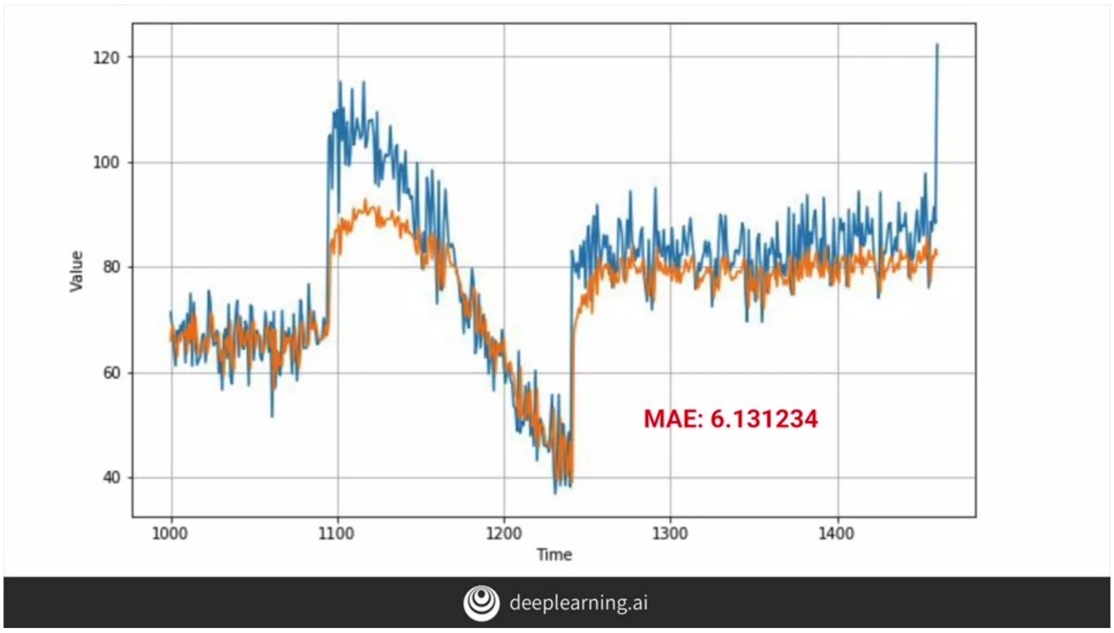
The state can also be bidirectional so that state moves forwards and backwards. In the case of texts, this was really powerful. Within the prediction of numeric sequences, it may or may not be, and it'll be interesting to experiment with. I'm not going to go into a lot of detail here but hundreds videos around LSTM are terrific. From there, you can really understand how they work under the hood.

8. Coding LSTMs

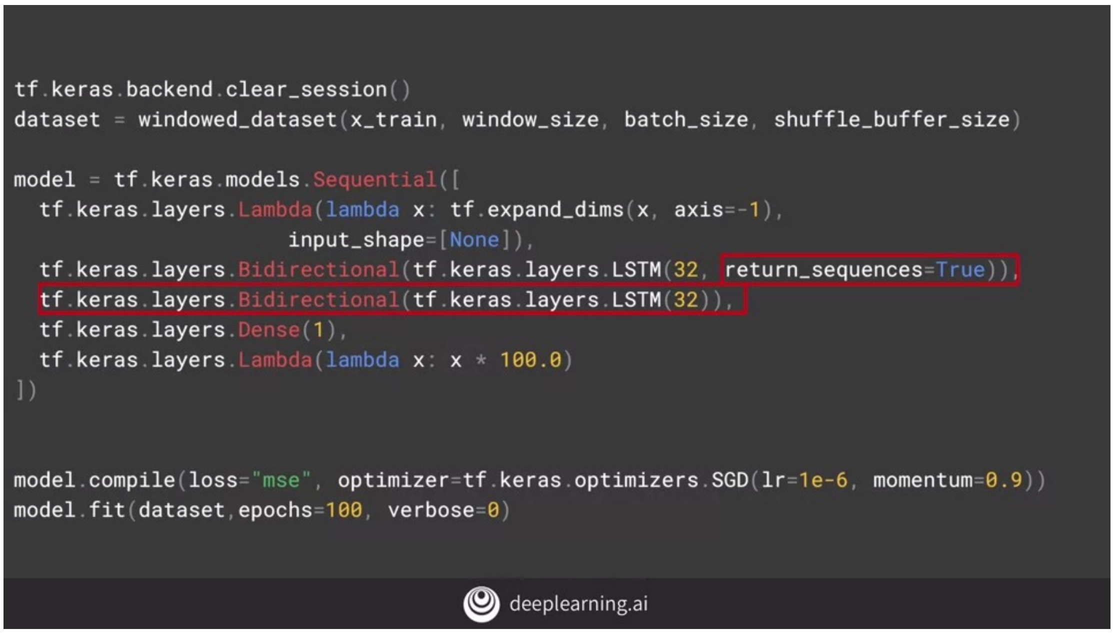


```
tf.keras.backend.clear_session()
dataset = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)

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)),
  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)
```

So let's now take a look at some code for LSTMs and how they can work with the data that we've been playing with all week. So here's the update to our code to use LSTMs. Let's unpack it and take a look at the interesting parts. 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. 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. The output neuron will give us our prediction value. I'm also using a learning rate of one times 10 _ minus six and that might be worth experimenting with too.

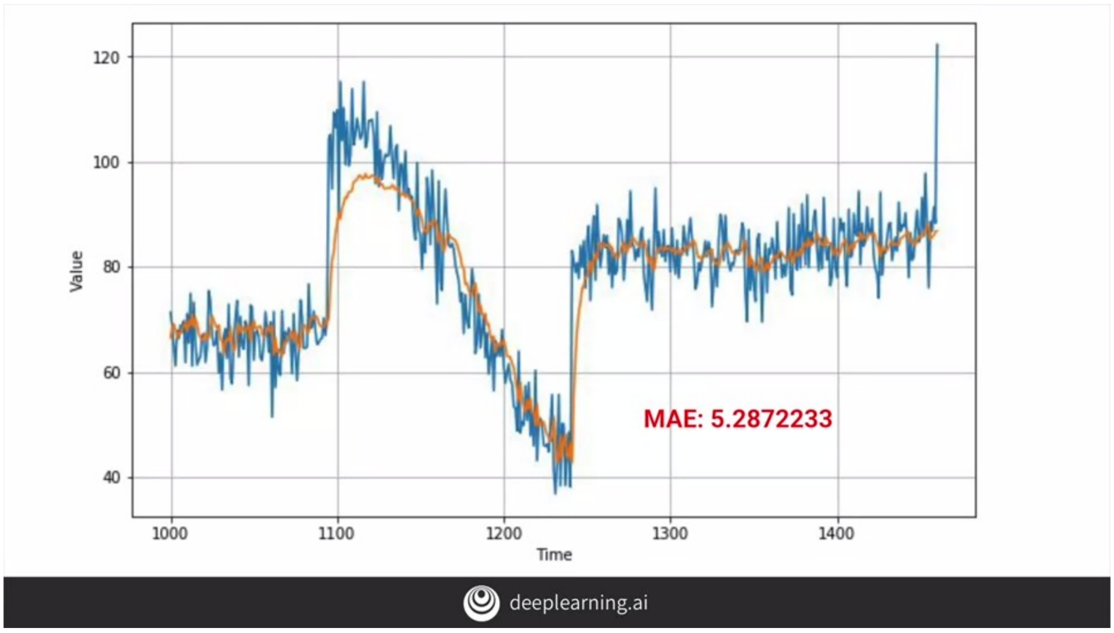


So here's the results of running this LSTM on the synthetic data that we've been using throughout the course. The plateau under the big spike is still there and are MAE is in the low sixes. It's not bad, it's not great, but it's not bad. The predictions look like there might be a little bit on the low side too.

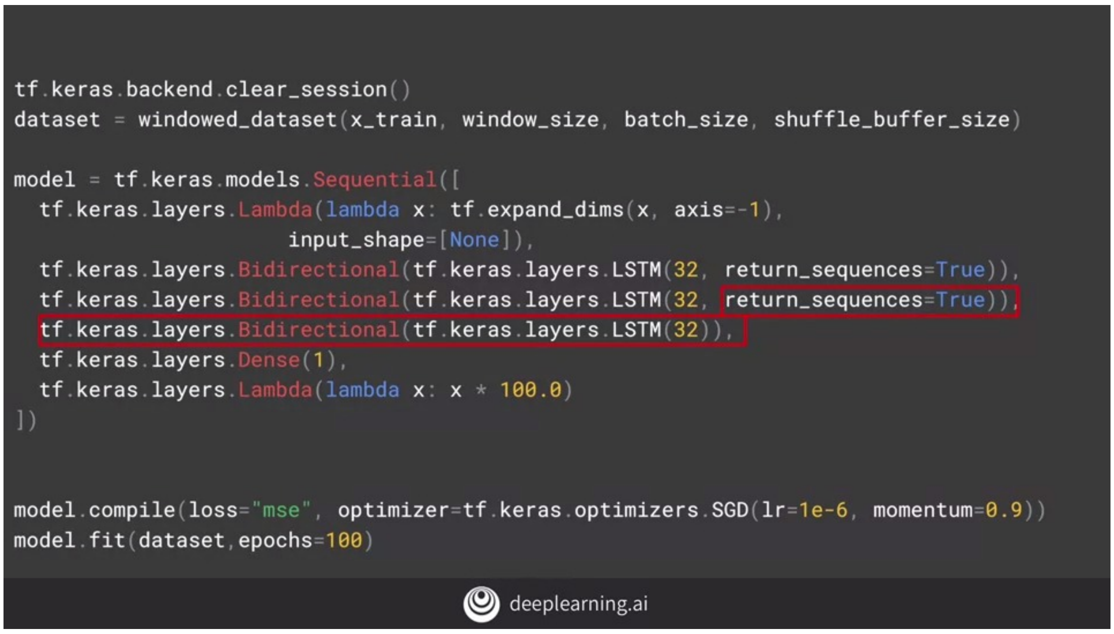


```
tf.keras.backend.clear_session()
dataset = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)

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)
```

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.

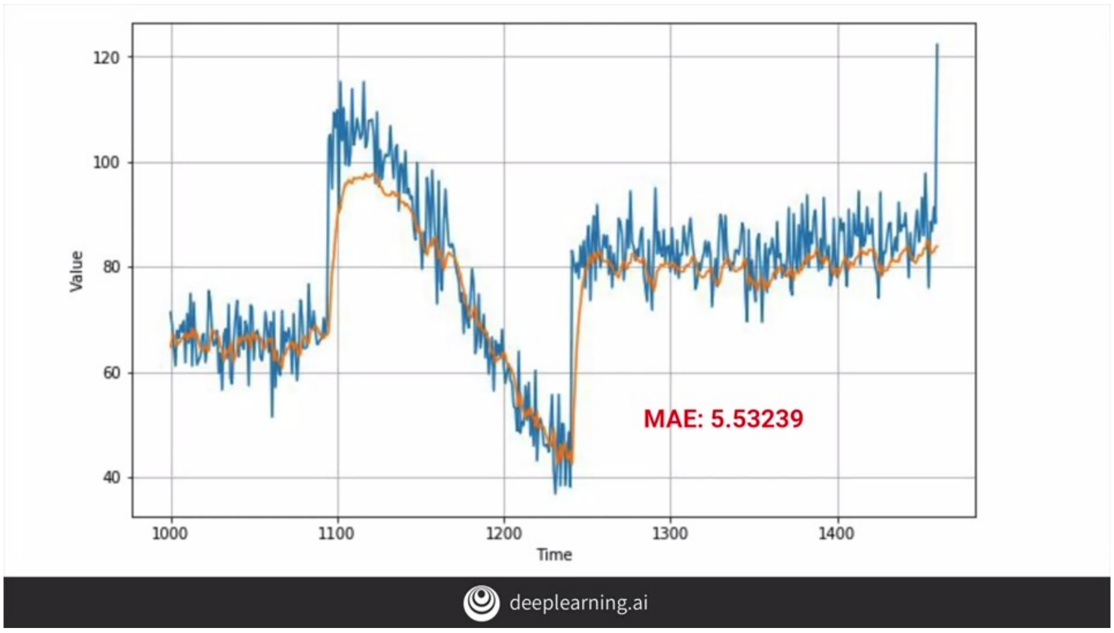


We train on this and now we will see the following results. Here's the chart. Now it's tracking much better and closer to the original data. Maybe not keeping up with the sharp increase but at least it's tracking close. It also gives us a mean average error that's a lot better and it's showing that we're heading in the right direction.



```
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. So that's it for looking at LSTMs and predicting our sequences. In the next video you'll take a look at using convolutions to see what the impact will be on using them for predicting time-series content.
S+P Week 3 Lesson 4 - LSTM