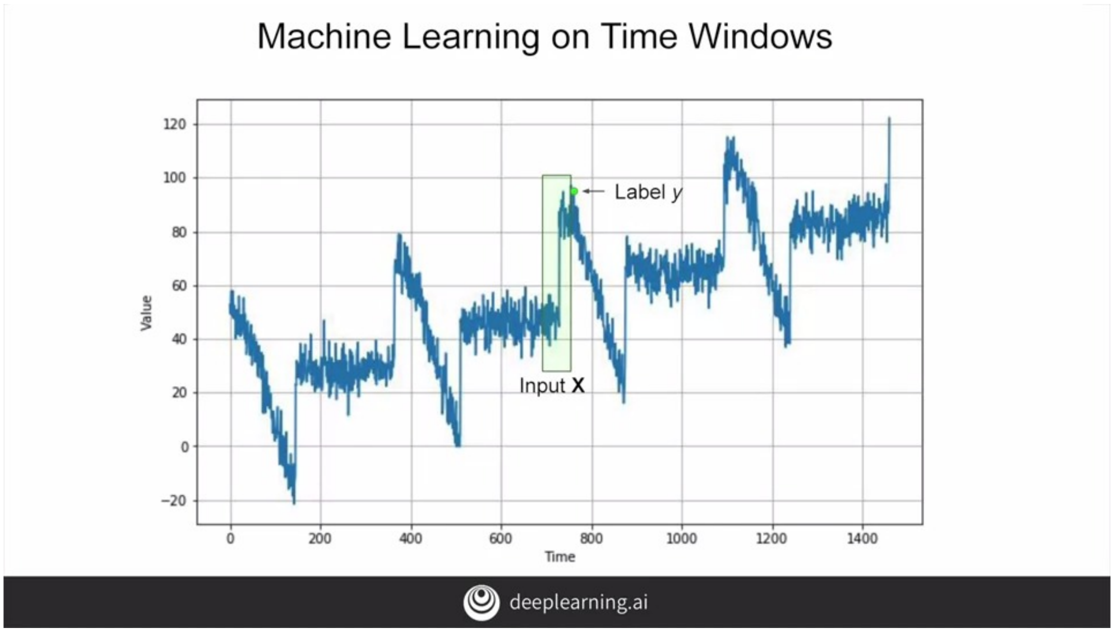
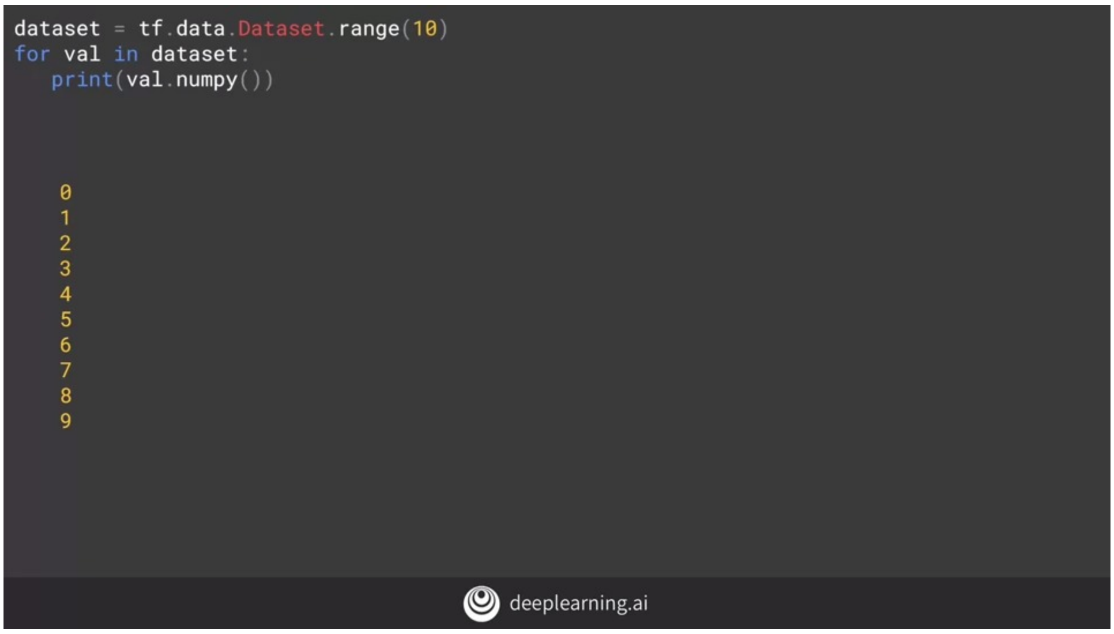
1. Preparing features and labels

First of all, as with any other ML problem, we have to divide our data into **features and labels**. In this case our **feature is effectively a number of values in the series**, with our **label being the next value**. We'll call that number of values that will treat as our feature, the **window size**, where we're taking a window of the data and training an ML model to predict the next value.



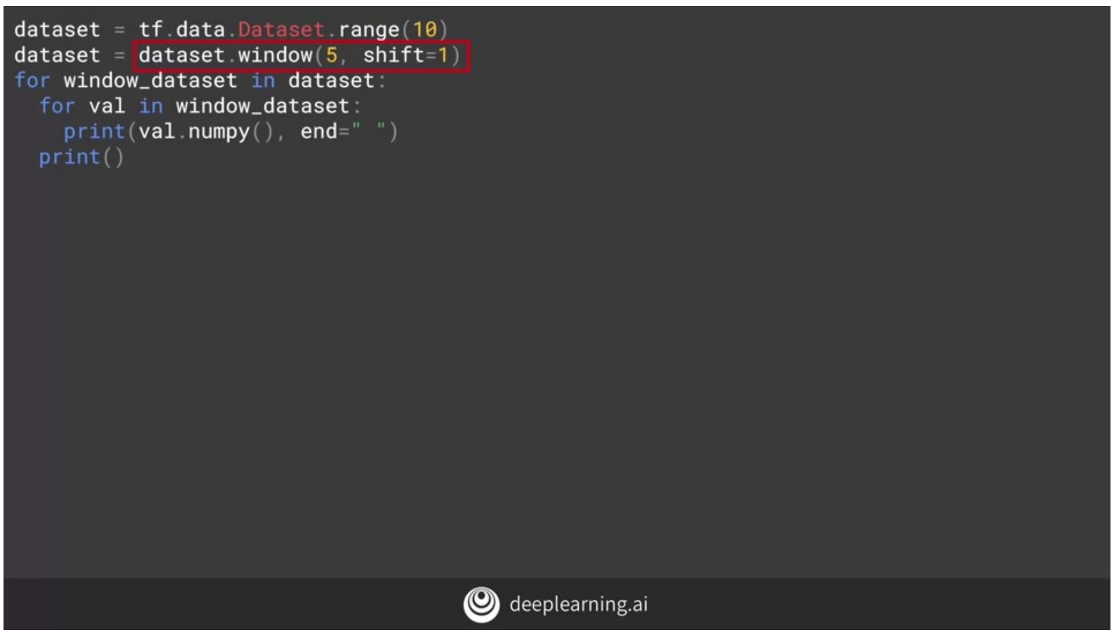
So for example, if we take our time series data, say, 30 days at a time, we'll use 30 values as the feature and the next value is the label. Then over time, we'll train a neural network to match the 30 features to the single label.



```
dataset = tf.data.Dataset.range(10)
for val in dataset:
   print(val.numpy())
```

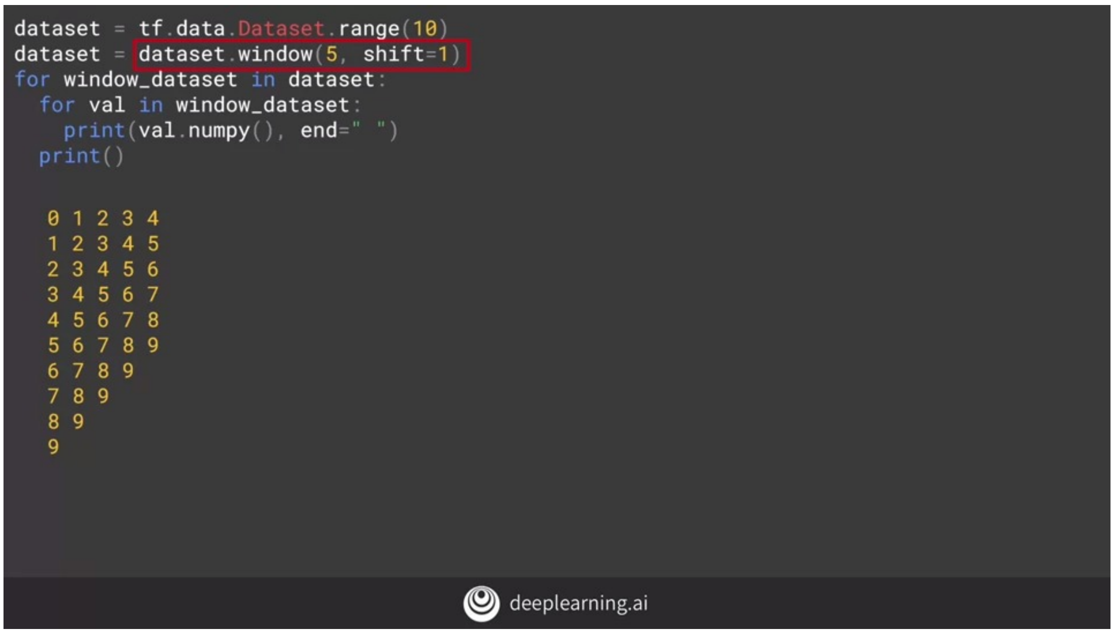
```
0
1
2
3
4
5
6
7
8
9
```

So let's, for example, use the tf.data.Dataset class to create some data for us, we'll make a range of 10 values. When we print them we'll see a series of data from 0 to 9.



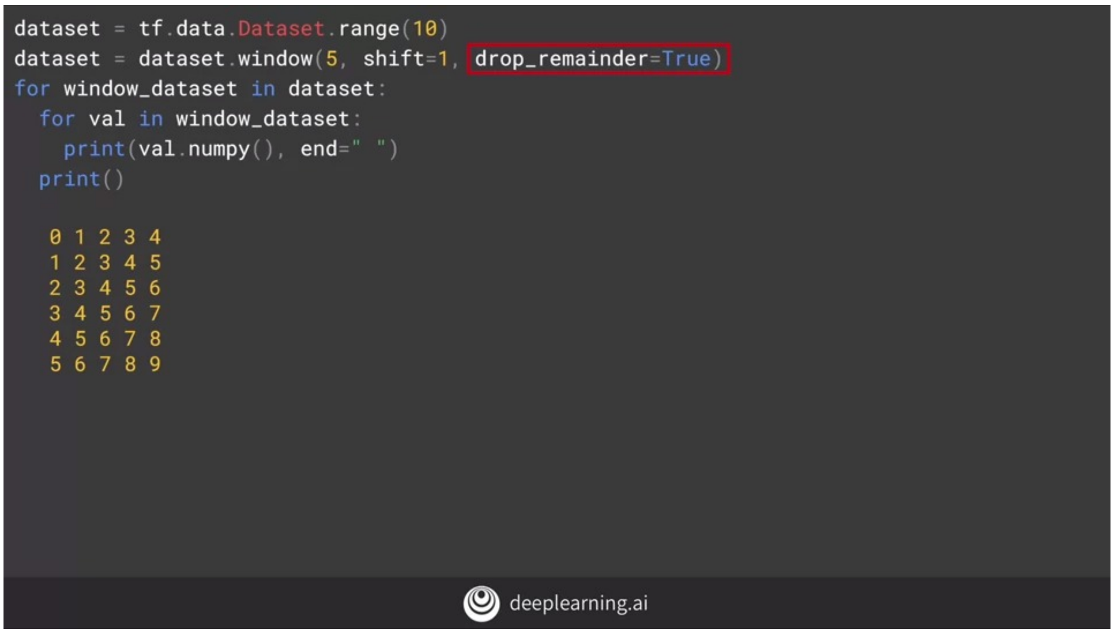
```
dataset = tf.data.Dataset.range(10)
dataset = dataset.window(5, shift=1)
for window_dataset in dataset:
  for val in window_dataset:
    print(val.numpy(), end=" ")
  print()
```

So now let's make it a little bit more interesting. We'll use the dataset.window to expand our data set using windowing. **Its parameters are the size of the window and how much we want to shift by each time.**



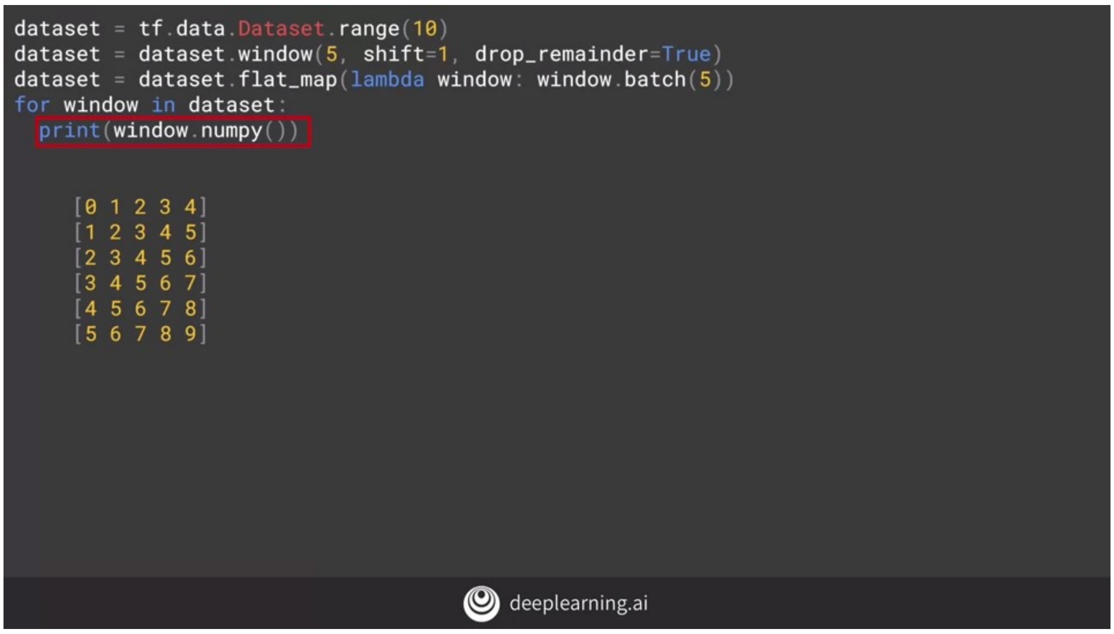
```
dataset = tf.data.Dataset.range(10)
dataset = dataset.window(5, shift=1)
for window_dataset in dataset:
  for val in window_dataset:
    print(val.numpy(), end=" ")
  print()

   0 1 2 3 4
   1 2 3 4 5
   2 3 4 5 6
   3 4 5 6 7
   4 5 6 7 8
   5 6 7 8 9
   6 7 8 9
   7 8 9
   8 9
   9
```

So if we set a window size of 5 with a shift of 1 when we print it we'll see something like this, 01234, which just stops there because it's five values, then we see 12345 etc, etc,. Once we get towards the end of the data set we'll have less values because they just don't exist. So we'll get 6789, and then 789, etc, etc,.



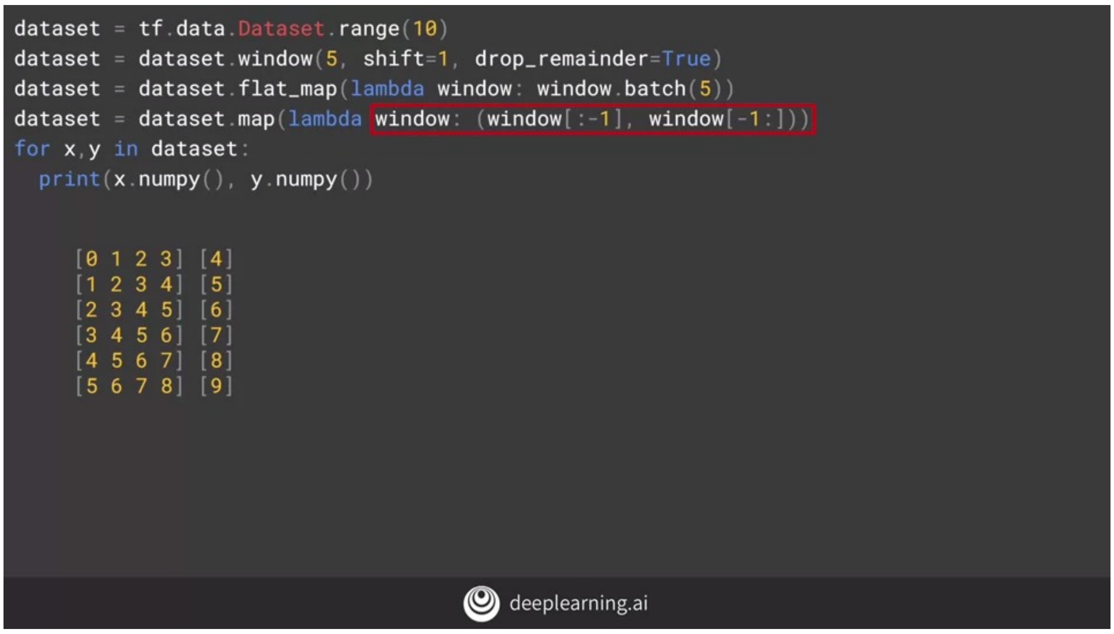
```
dataset = tf.data.Dataset.range(10)
dataset = dataset.window(5, shift=1, drop_remainder=True)
for window_dataset in dataset:
  for val in window_dataset:
    print(val.numpy(), end=" ")
  print()

   0 1 2 3 4
   1 2 3 4 5
   2 3 4 5 6
   3 4 5 6 7
   4 5 6 7 8
   5 6 7 8 9
```

So let's edit our window a little bit, so that we have regularly sized data. We can do that with an additional parameter on the window called drop_remainder. And if we set this to true, it will truncate the data by dropping all of the remainders. Namely, this means it will only give us windows of five items. So when we print it, it will now look like this, starting at 01234 and ending at 56789.



```
dataset = tf.data.Dataset.range(10)
dataset = dataset.window(5, shift=1, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(5))
for window in dataset:
  print(window.numpy())

     [0 1 2 3 4]
     [1 2 3 4 5]
     [2 3 4 5 6]
     [3 4 5 6 7]
     [4 5 6 7 8]
     [5 6 7 8 9]
```

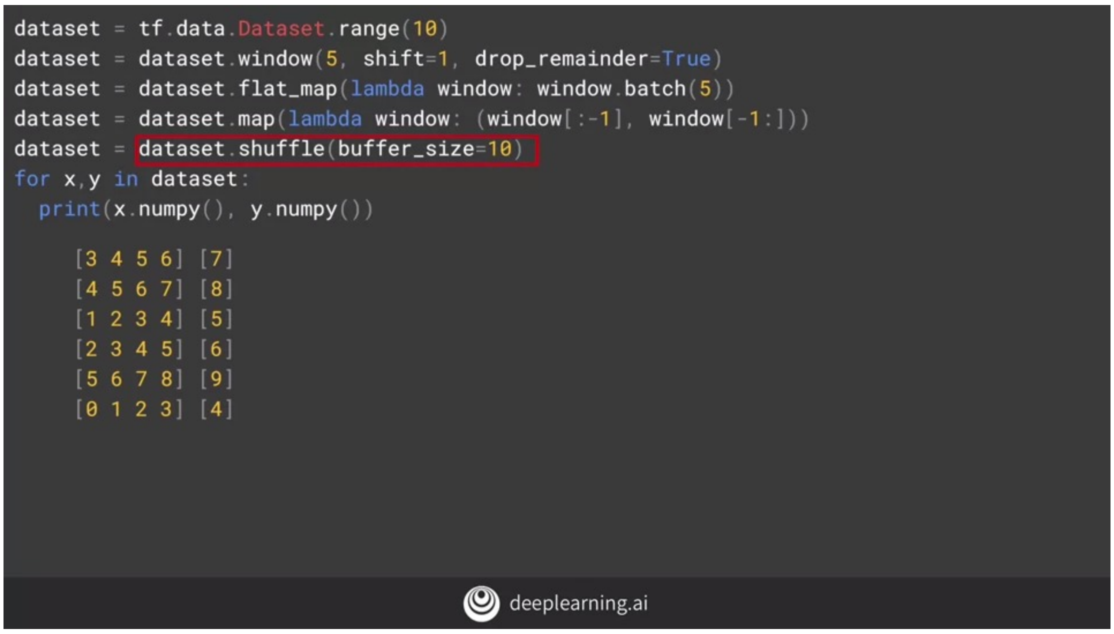
Great, now let's put these into numpy lists so that we can start using them with machine learning. Good news is, is that this is super easy, we just call the .numpy method on each item in the data set, and when we print we now see that we have a numpy list.



```
dataset = tf.data.Dataset.range(10)
dataset = dataset.window(5, shift=1, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(5))
dataset = dataset.map(lambda window: (window[:-1], window[-1:]))
for x,y in dataset:
  print(x.numpy(), y.numpy())


    [0 1 2 3] [4]
    [1 2 3 4] [5]
    [2 3 4 5] [6]
    [3 4 5 6] [7]
    [4 5 6 7] [8]
    [5 6 7 8] [9]
```

deeplearning.ai

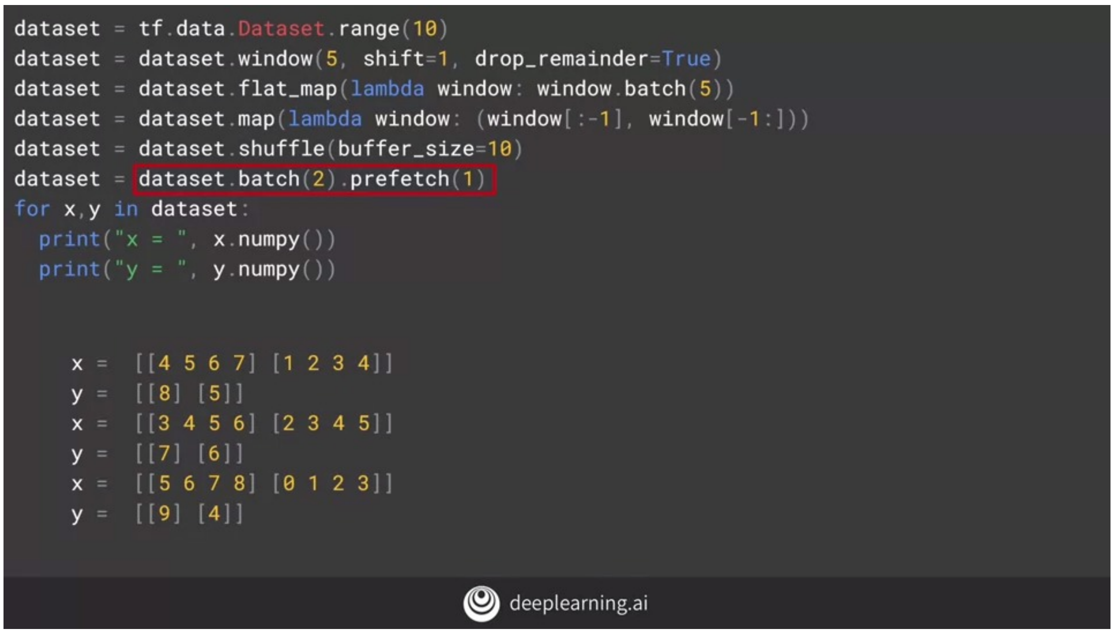
Okay, next up is to split the data into features and labels. For each item in the list it kind of makes sense to have all of the values but the last one to be the feature, and then the last one can be the label. And this can be achieved with mapping, like this, where we split into everything but the last one with :-1, and then just the last one itself with -1:. Which gives us this output when we print, which now looks like a nice set of features and labels.



```
dataset = tf.data.Dataset.range(10)
dataset = dataset.window(5, shift=1, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(5))
dataset = dataset.map(lambda window: (window[:-1], window[-1:]))
dataset = dataset.shuffle(buffer_size=10)
for x,y in dataset:
  print(x.numpy(), y.numpy())

    [3 4 5 6] [7]
    [4 5 6 7] [8]
    [1 2 3 4] [5]
    [2 3 4 5] [6]
    [5 6 7 8] [9]
    [0 1 2 3] [4]
```

deeplearning.ai

Typically, you would shuffle their data before training. And this is possible using the shuffle method. We call it with the buffer size of ten, because that's the amount of data items that we have. And when we print the results, we'll see our features and label sets have been shuffled.



```
dataset = tf.data.Dataset.range(10)
dataset = dataset.window(5, shift=1, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(5))
dataset = dataset.map(lambda window: (window[:-1], window[-1:]))
dataset = dataset.shuffle(buffer_size=10)
dataset = dataset.batch(2).prefetch(1)
for x,y in dataset:
  print("x = ", x.numpy())
  print("y = ", y.numpy())


    x =  [[4 5 6 7] [1 2 3 4]]
    y =  [[8] [5]]
    x =  [[3 4 5 6] [2 3 4 5]]
    y =  [[7] [6]]
    x =  [[5 6 7 8] [0 1 2 3]]
    y =  [[9] [4]]
```

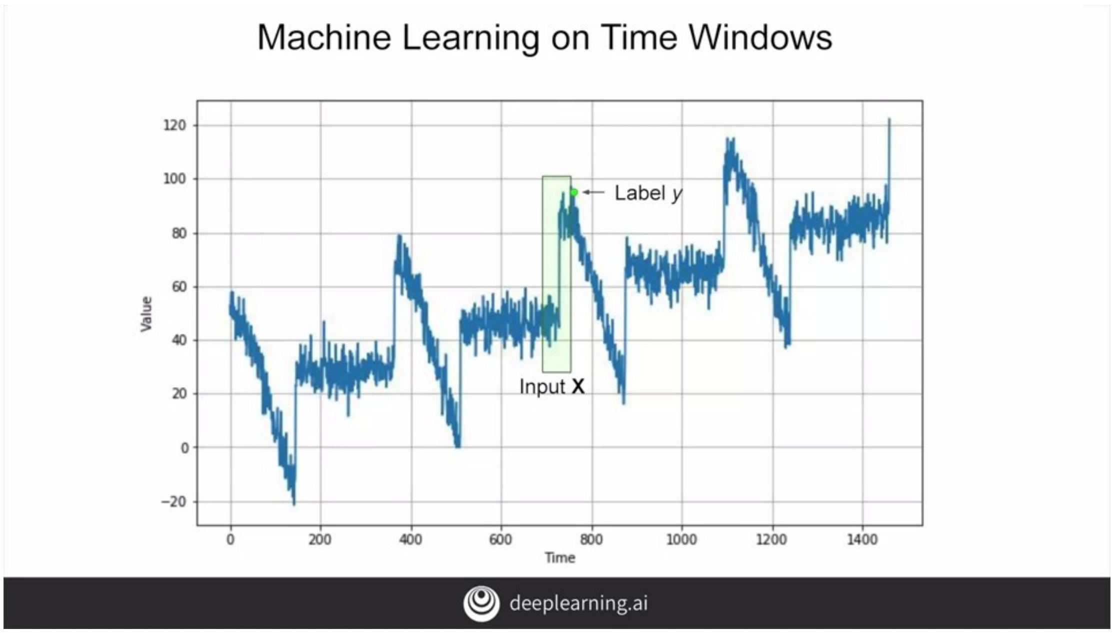
Finally, we can look at batching the data, and this is done with the batch method. It'll take a size parameter, and in this case it's 2. So what we'll do is we'll batch the data into sets of two, and if we print them out, we'll see this. We now have three batches of two data items each. And if you look at the first set, you'll see the corresponding x and y. So when x is four, five, six and seven, our y is eight, or when x is zero, one, two, three, you'll see our y is four.
Okay, now that you've seen the tools that let us create a series of x and y's, or features and labels, you have everything you need to work on a data set in order to get predictions from it. We'll take a look at a screen cast of this code next, before moving on to creating our first neural networks to run predictions on this data.

2. Preparing features and labels

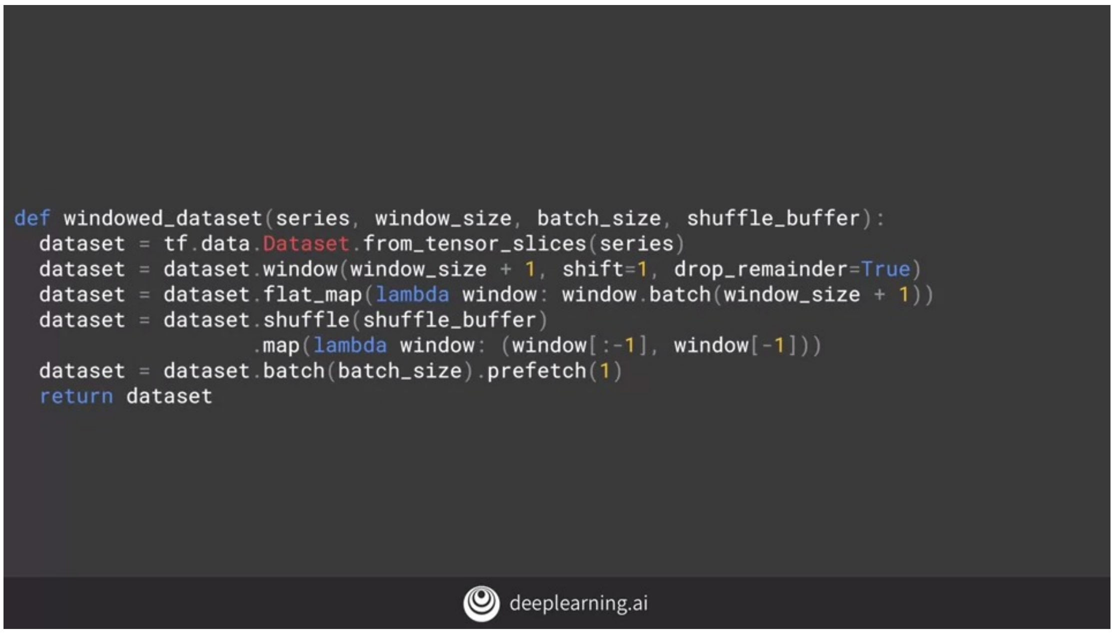
S+P Week 2 Lesson 1

First, we'll create a simple dataset, and it's just a simple range containing 10 elements from zero to nine. We'll print each one out on its own line as you can see. Next, we'll window the data into chunks of five items, shifting by one each time. We'll see that this gives us the output of the first five items, and then the second five items, and then the third five items, etc. At the end of the dataset, when there isn't enough data to give us five items, you'll see shorter lines. To just get chunks of five records, we'll set drop_reminder to true. When we run it, we'll see that our data looks like this. We've got even sets that are the same size. TensorFlow likes its data to be in numpy format. So we can convert it easily by calling the dot numpy method and when we print it, we can see it's now listed in square brackets. Next up is to split into x's and y's or features and labels. We'll take the last column as the label, and we'll split using a lambda. We'll split the data into column minus one, which is all of the columns except the last one, and minus one column which is the last one only. Now we can see that we have a set of four items and a single item. Remember that the minus one column denotes the last value in the list, and column minus one denotes everything about the last value. As such, we can see zero, one, two, three and one, two, three, four before the split just for example. Next of course, is to shuffle the data. This is achieved with the shuffle method. This helps us to rearrange the data so as not to accidentally introduce a sequence bias. Multiple runs will show the data in different arrangements because it gets shuffled randomly. Finally, comes batching. By setting a batch size of two, our data gets batched into two x's and two y's at a time. For example, as we saw earlier, if x is zero, one, two, three, we can see that the corresponding y is four or if x is five, six, seven, eight, then our y is nine. So that's the workbook with the code that splits a data series into windows. Try it out for yourself, and once you're familiar with what it does, proceed to the next video. There you'll move to the seasonal dataset that you've been using two dates, and with this windowing technique, you'll see how to set up x's and y's that can be fed into a neural network to see how it performs with predicting values.

3. Feeding windowed dataset into neural network



In the last videos you saw how to prepare time series data for machine learning by creating a window dataset where the previous n values could be seen as the input features are x. And the current value with any time stamp is the output label or the y. It would then look a little bit like this. With a number of input values on x, typically called a window on the data. You saw in the previous videos how to use the tools in tensor flow datasets to create these windows. This week you'll adapt that code to feed a neural network and then train it on the data.

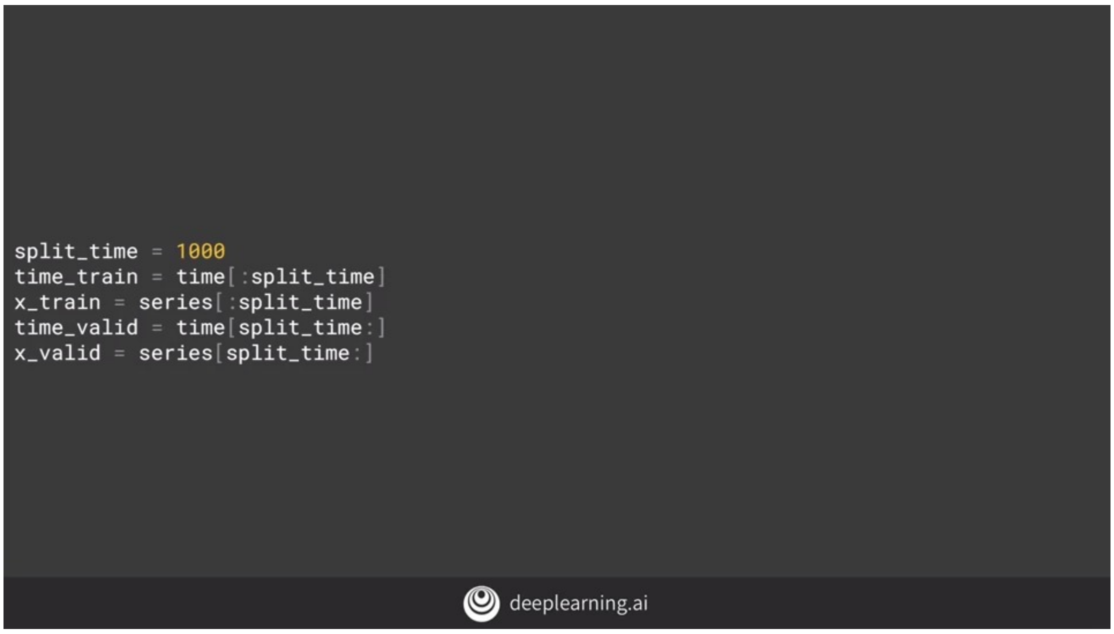


```
def windowed_dataset(series, window_size, batch_size, shuffle_buffer):
  dataset = tf.data.Dataset.from_tensor_slices(series)
  dataset = dataset.window(window_size + 1, shift=1, drop_remainder=True)
  dataset = dataset.flat_map(lambda window: window.batch(window_size + 1))
  dataset = dataset.shuffle(shuffle_buffer)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

So let's start with this function that will call a windows dataset. It will take in a data series along with the parameters for the size of the window that we want. The size of the batches to use when training, and the size of the shuffle buffer, which determines how the data will be shuffled.
The first step will be to create a dataset from the series using a tf.data dataset. And we'll pass the series to it using its from_tensor_slices method.
We will then use the window method of the dataset based on our window_size to slice the data up into the appropriate windows. Each one being shifted by one time set. We'll keep them all the same size by setting drop remainder to true.
We then flatten the data out to make it easier to work with. And it will be flattened into chunks in the size of our window_size + 1.
Once it's flattened, it's easy to shuffle it. You call a shuffle and you pass it the shuffle buffer. Using a shuffle buffer speeds things up a bit. So for example, if you have 100,000 items in your dataset, but you set the buffer to a thousand. It will just fill the buffer with the first thousand elements, pick one of them at random. And then it will replace that with the 1,000 and first element before randomly picking again, and so on. This way with super large datasets, the random element choosing can choose from a smaller number which effectively speeds things up.
The shuffled dataset is then split into the xs, which is all of the elements except the last, and the y which is the last element.
It's then batched into the selected batch size and returned.

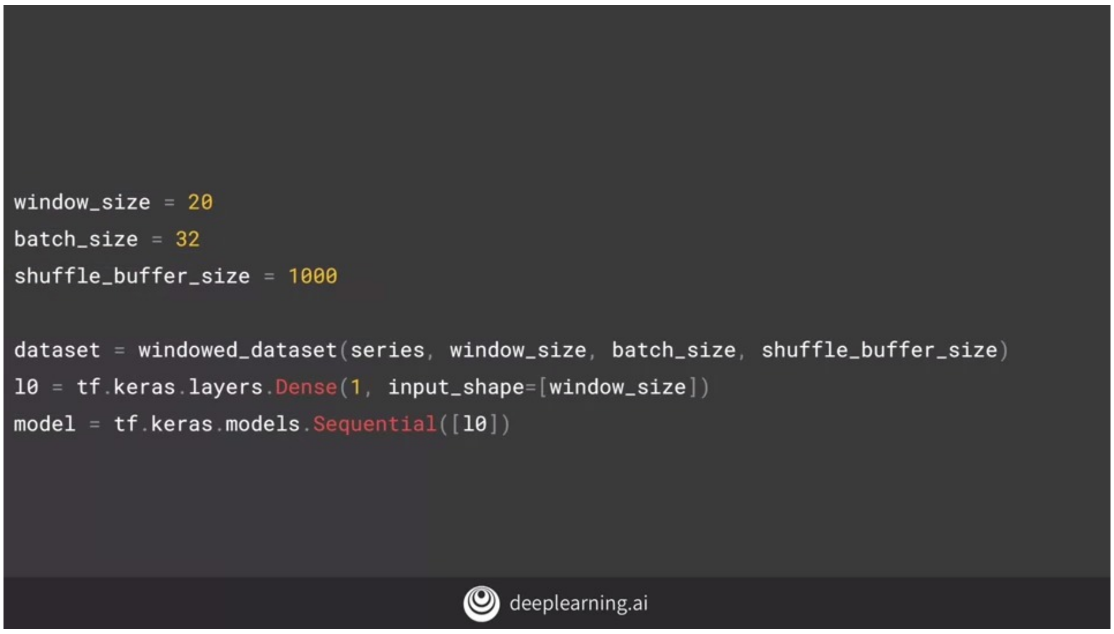
4. Single layer neural network

Now that we have a window datasets, we can start training neural networks with it. Let's start with a super simple one that's effectively a linear regression. We'll measure its accuracy, and then we'll work from there to improve that.



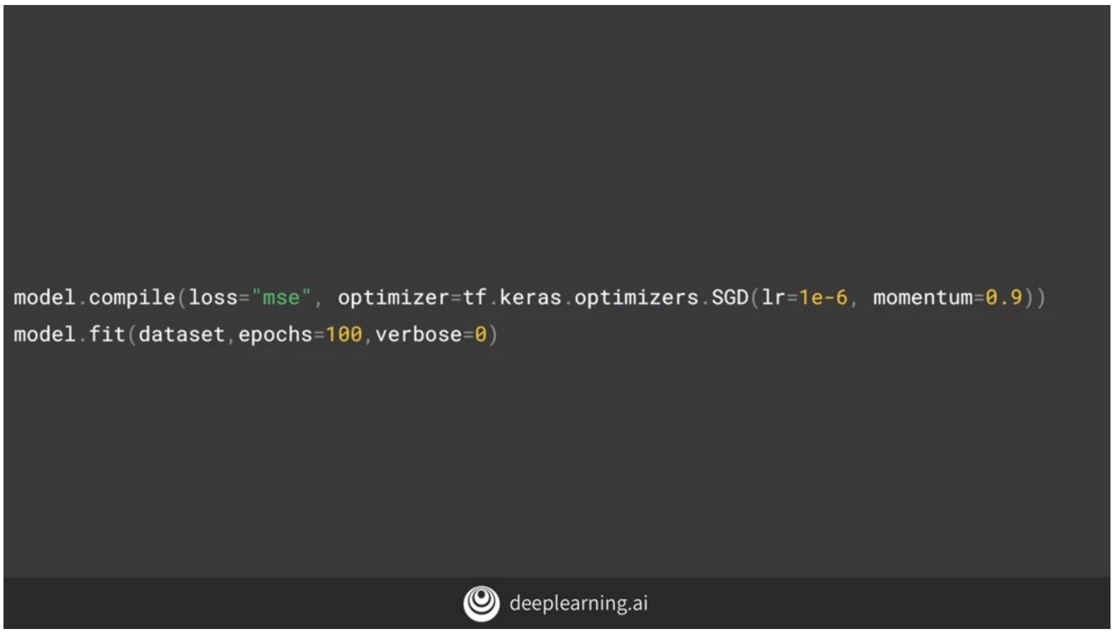
```
split_time = 1000
time_train = time[:split_time]
x_train = series[:split_time]
time_valid = time[split_time:]
x_valid = series[split_time:]
```

Before we can do a training, we have to split our dataset into training and validation sets. Here's the code to do that at time step 1000. We can see that the training data is the subset of the series called x train up to the split time.



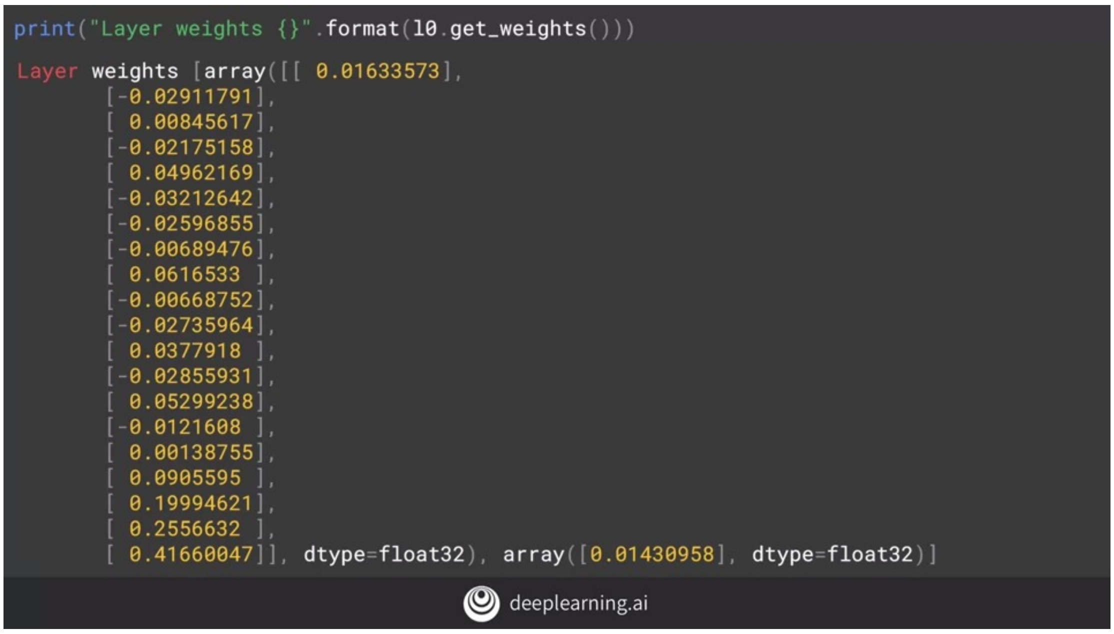
```
window_size = 20
batch_size = 32
shuffle_buffer_size = 1000

dataset = windowed_dataset(series, window_size, batch_size, shuffle_buffer_size)
l0 = tf.keras.layers.Dense(1, input_shape=[window_size])
model = tf.keras.models.Sequential([l0])
```

Here's the code to do a simple linear regression. Let's look at it line by line. We'll start by setting up all the constants that we want to pass to the window dataset function. These include the window size on the data, the batch size that we want for training, and the size of the shuffled buffer as we've just discussed. Then we'll create our dataset. We'll do this by taking our series, and in the notebook that you'll go through later, you'll create the same synthetic series as you did in week one. You'll pass it your series along what your desired window size, batch size, and shuffled buffer size, and it will give you back a formatted datasets that you could use for training. I'm then going to create a single dense layer with its input shape being the window size. For linear regression, that's all you need. I'm using this approach. By passing the layer to a variable called L0, because later I'm want to print out its learned weights, and it's a lot easier for me to do that if I have a variable to refer to the layer for that. Then I simply define my model as a sequential containing the sole layer just like this.



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

Now I'll compile and fit my model with this code. I'll use the mean squared error loss function by setting loss to MSE, and my optimizer will use Stochastic Gradient Descent. I'd use this methodology instead of the raw string, so I can set parameters on it to initialize it such as the learning rate or LR and the momentum. Experiment with different values here to see if you can get your model to converge more quickly or more accurately. Next you can fit your model by just passing it the dataset, which has already been preformatted with the x and y values. I'm going to run for a 100 epochs here. Ignoring the epoch but epoch output by setting verbose to zero.

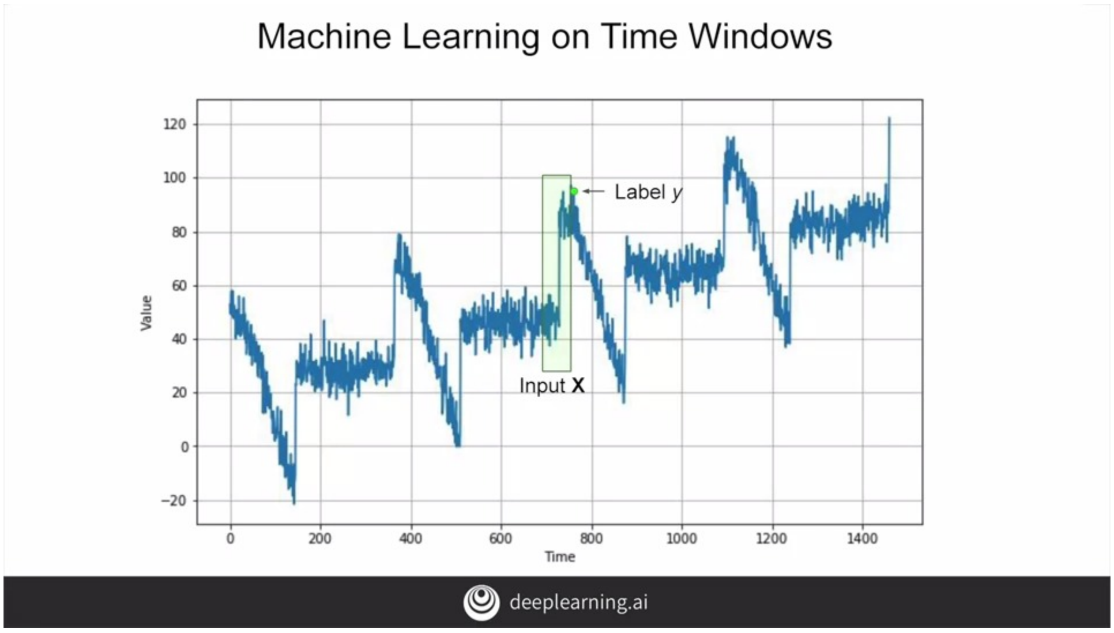


```
print("Layer weights {}".format(l0.get_weights()))
Layer weights [array([[ 0.01633573],
       [-0.02911791],
       [ 0.00845617],
       [-0.02175158],
       [ 0.04962169],
       [-0.03212642],
       [-0.02596855],
       [-0.00689476],
       [ 0.0616533 ],
       [-0.00668752],
       [-0.02735964],
       [ 0.0377918 ],
       [-0.02855931],
       [ 0.05299238],
       [-0.0121608 ],
       [ 0.00138755],
       [ 0.0905595 ],
       [ 0.19994621],
       [ 0.2556632 ],
       [ 0.41660047]], dtype=float32), array([0.01430958], dtype=float32)]
```

Once it's done training, you can actually inspect the different weights with this code.

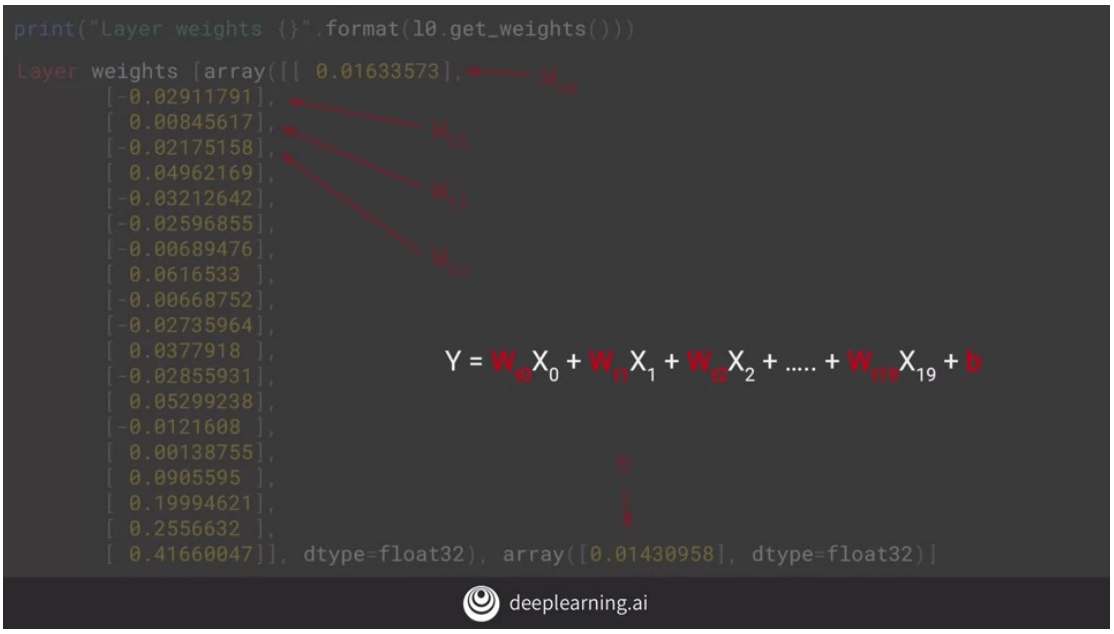
Remember earlier when we referred to the layer with a variable called L 0? Well, here's where that's useful. The output will look like this. If you inspect it closely, you will see that the first array has 20 values in it, and the secondary has only one value. This is because the network has learned a linear regression to fit the values as best as they can. So each of the values in the first array can be seen as the weights for the 20 values in x, and the value for the second array is the b value.

5. Machine learning on time windows



So if you think back to this diagram and you consider the input window to be 20 values wide, then let's call them x0, x1, x2, etc, all the way up to x19. But let's be clear. That's not the value on the horizontal axis which is commonly called the x-axis, it's the value of the time series at that point on the horizontal axis. So the value at time t0, which is 20 steps before the current value is called x0, and t1 is called x1, etc. Similarly, for the output, which we would then consider to be the value at the current time to be the y.

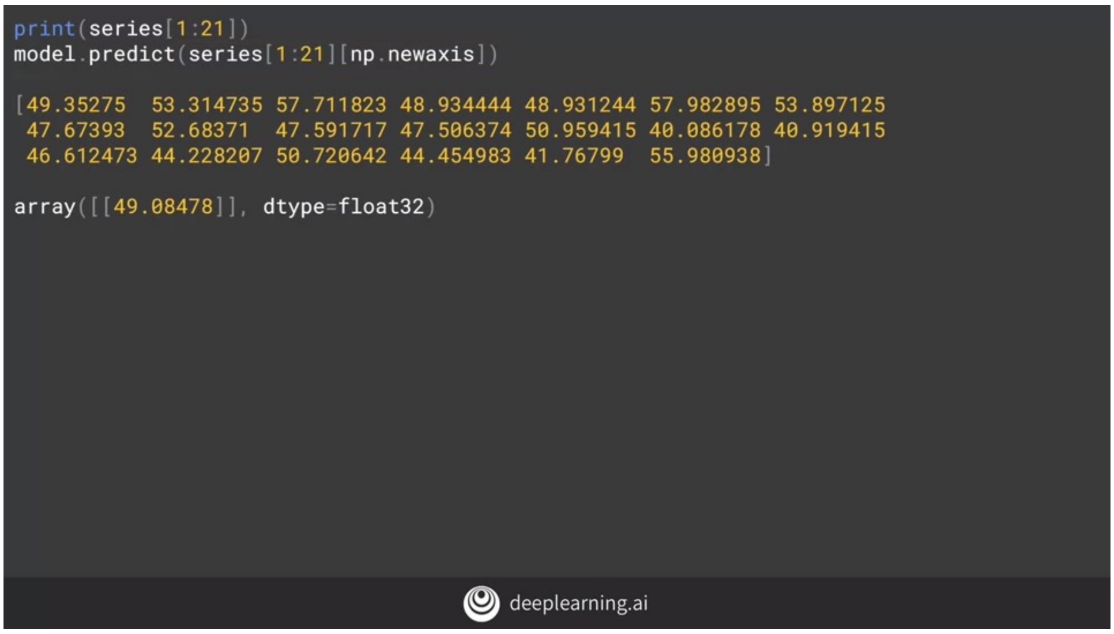
6. Prediction



```
print("Layer weights {}".format(l0.get_weights()))

Layer weights [array([[ 0.01633573],
        [-0.02911791],
        [ 0.00845617],
        [-0.02175158],
        [ 0.04962169],
        [-0.03212642],
        [-0.02596855],
        [-0.00689476],
        [ 0.0616533 ],
        [-0.00668752],
        [-0.02735964],
        [ 0.0377918 ],
        [-0.02855931],
        [ 0.05299238],
        [-0.0121608 ],
        [ 0.00138755],
        [ 0.0905595 ],
        [ 0.19994621],
        [ 0.2556632 ],
        [ 0.41660047]], dtype=float32), array([0.01430958], dtype=float32)]
```

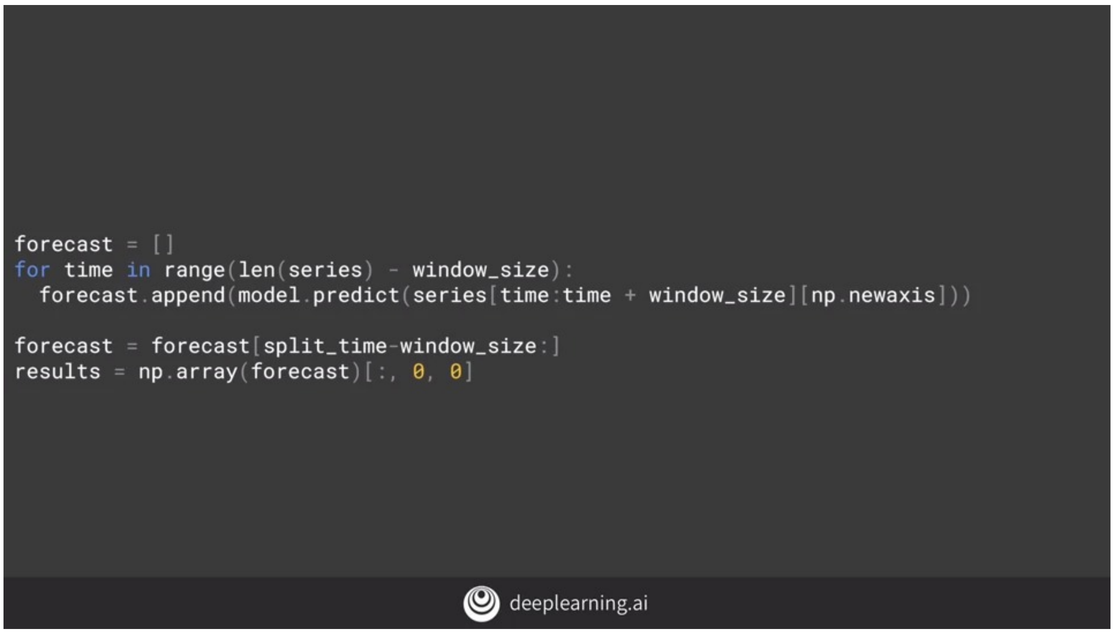
$$Y = W_{t0}X_0 + W_{t1}X_1 + W_{t2}X_2 + \ldots + W_{t19}X_{19} + b$$

So now, if we look at the values again and see that these are the weights for the values at that particular timestamp and b is the bias or the slope, we can do a standard linear regression like this to predict the value of y at any step by multiplying out the x values by the weights and then adding the bias.



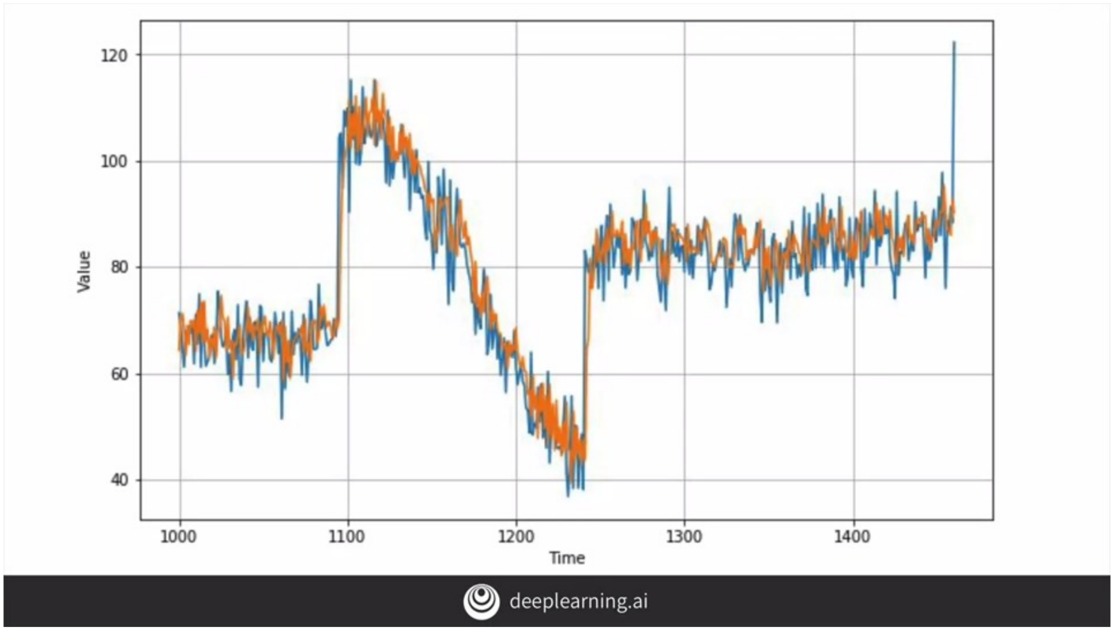
```
print(series[1:21])
model.predict(series[1:21][np.newaxis])

[49.35275  53.314735 57.711823 48.934444 48.931244 57.982895 53.897125
 47.67393  52.68371  47.591717 47.506374 50.959415 40.086178 40.919415
 46.612473 44.228207 50.720642 44.454983 41.76799  55.980938]

array([[49.08478]], dtype=float32)
```

So for example, if I take 20 items in my series and print them out, I can see the 20x values. If I want to predict them, I can pass that series into my model to get a prediction. The **NumPy new axis then just reshapes it to the input dimension that's used by the model.** The output will look like this. The top array is the 20 values that provide the input to our model and the bottom is the predicted value back from the model. So we've trained our model to say that when it sees 20 values like this, the predicted next value is 49.08478.



```
forecast = []
for time in range(len(series) - window_size):
  forecast.append(model.predict(series[time:time + window_size][np.newaxis]))

forecast = forecast[split_time-window_size:]
results = np.array(forecast)[:, 0, 0]
```

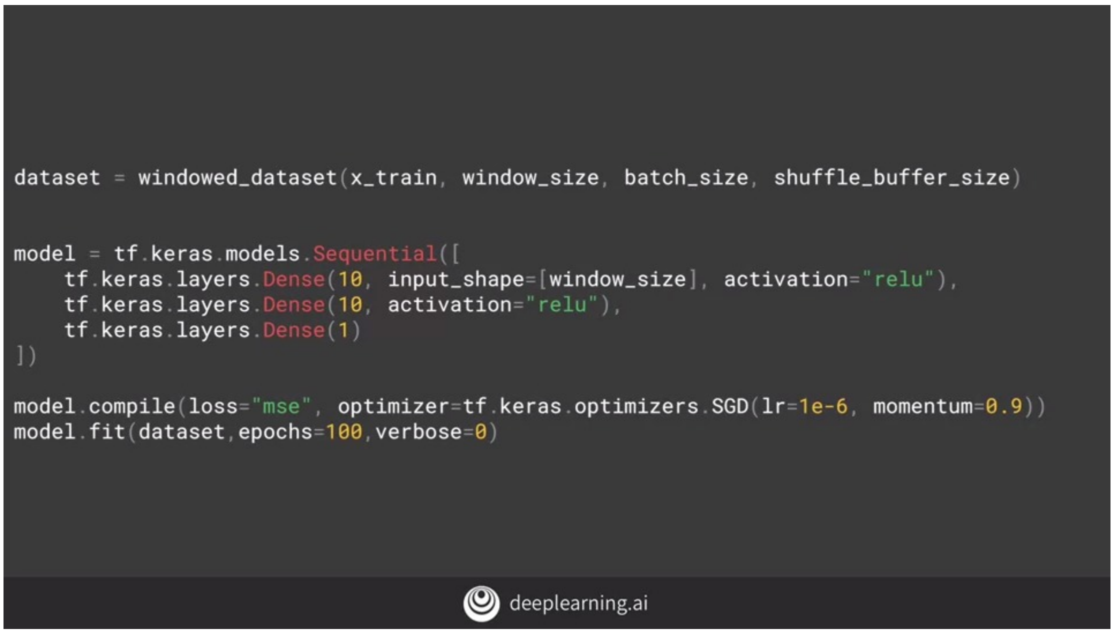
So if we want to plot our forecasts for every point on the time-series relative to the 20 points before it where our window size was 20, we can write code like this. We create an empty list of forecasts and then iterate over the series taking slices and window size, predicting them, and adding the results to the forecast list. We had split our time series into training and testing sense taking everything before a certain time is training and the rest is validation. So we'll just take the forecasts after the split time and load them into a NuimPy array for charting.



That chart looks like this with the actual values in blue and the predicted ones in orange. You can see that our predictions look pretty good and getting them was relatively simple in comparison with all the statistical gymnastics that we had to do in the last videos. So let's measure the mean absolute error as we've done before, and we can see that we're in a similar ballpark to where we were with a complex analysis that we did previously. Now that's just using a single layer in a neural network to calculate a linear regression. Let's see if we could do

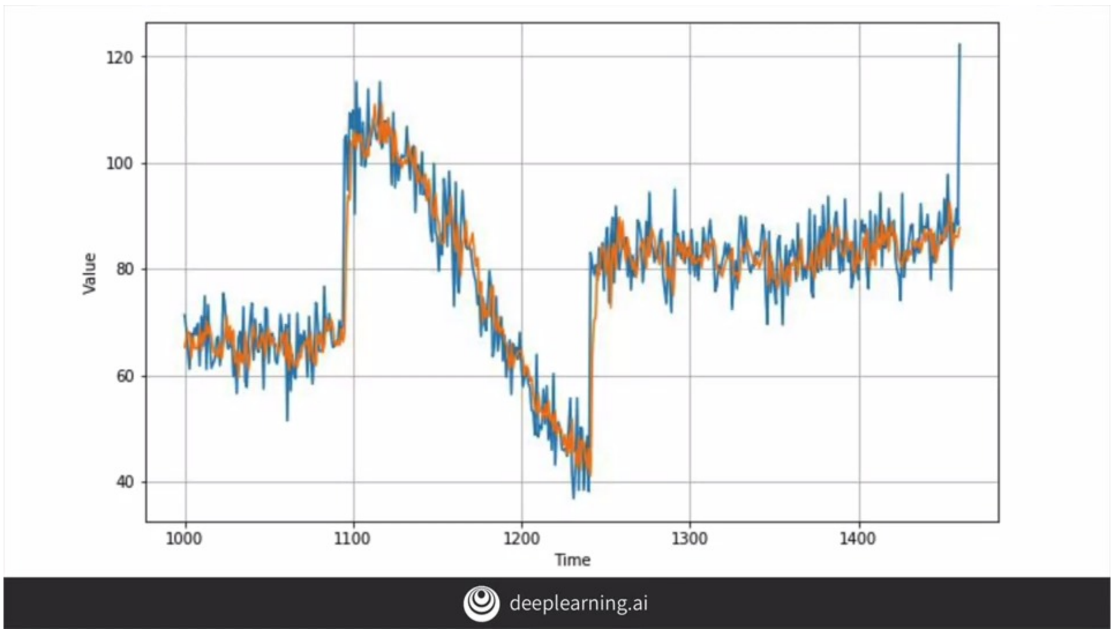
better with a fully-connected DNN next. Before you get to that, we'll go through the workbook for this lesson to ensure that you understand everything we've done up to now. The next video will be a screencast of going through that and then you'll work on the DNN after that.
S+P Week 2 Lesson 2

7. Deep neural network training, tuning and prediction

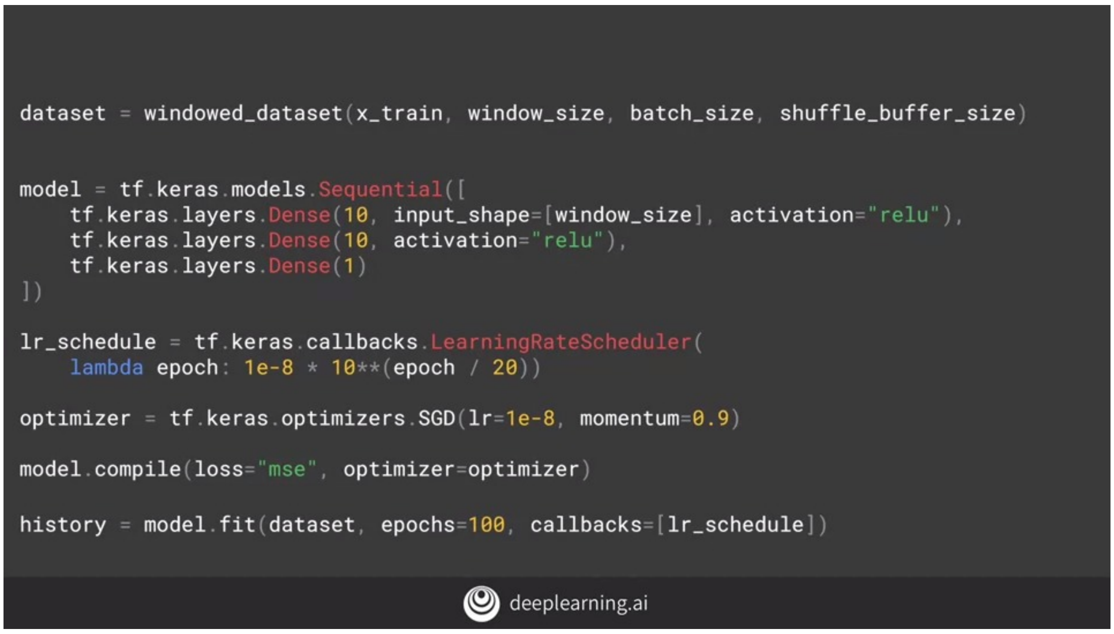


```
dataset = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)


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)
```

It's not that much different from the linear regression model we saw earlier. And this is a relatively simple deep neural network that has three layers. So let's unpack it line by line. First we'll have to get a data set which will generate by passing in x_train data, along with the desired window size, batch size, and shuffle buffer size.
We'll then define the model. Let's keep it simple with three layers of 10, 10, and 1 neurons. The input shape is the size of the window and we'll activate each layer using a relu.
We'll then compile the model as before with a mean squared error loss function and stochastic gradient descent optimizer.



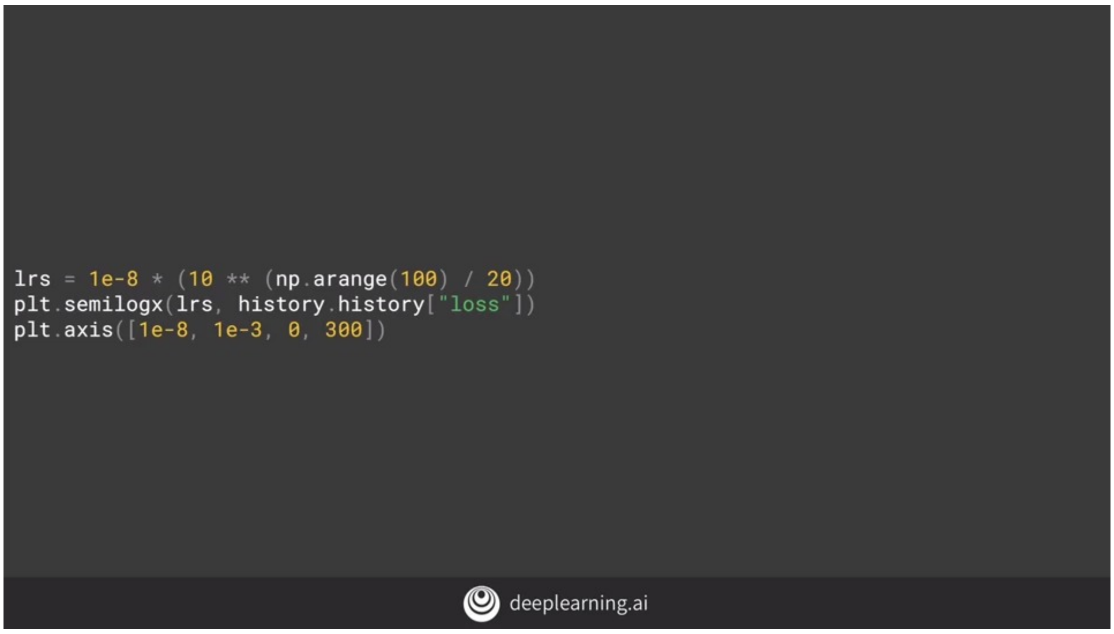
Finally, we'll fit the model over 100 epochs, and after a few seconds of training, we'll see results that look like this. It's pretty good still.
And when we calculate the mean absolute error, we're lower than we were earlier, so it's a step in the right direction. But it's also a somewhat a stab in the dark, particularly with the optimizer function. Wouldn't it be nice if we could pick the optimal learning rate instead of the one that we chose? We might learn more efficiently and build a better model.



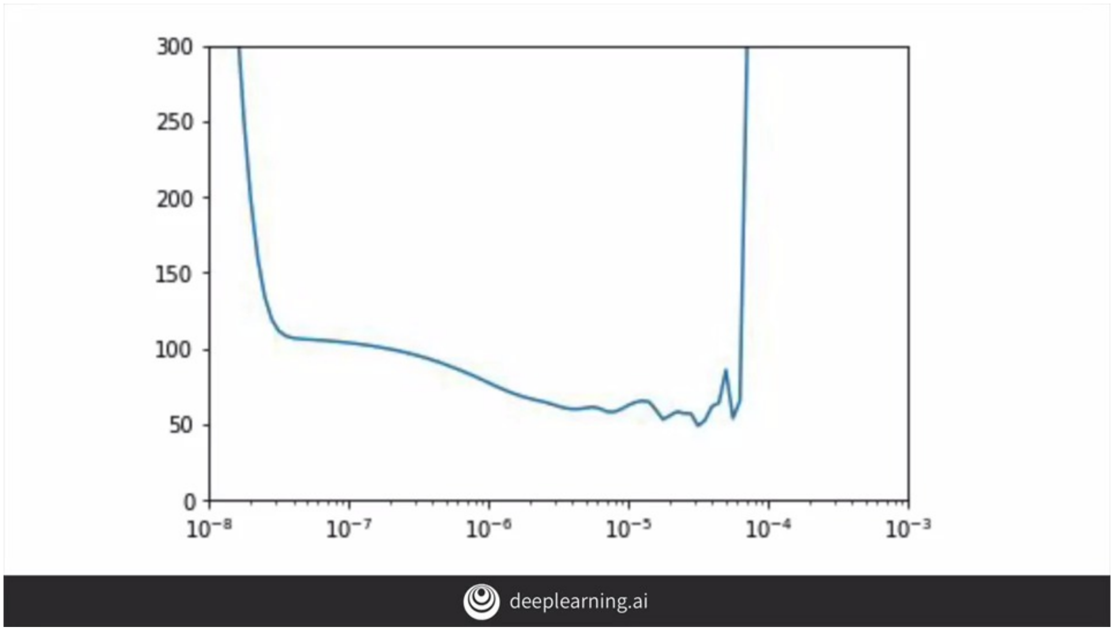
```
dataset = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)


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])
```

Now let's look at a technique for that that uses **callbacks** that you used way back in the first course. So here's a code for the previous neural network. But I've **added a callback to tweak the learning rate using a learning rate scheduler**. You can see that code here. This will be called at the callback at the end of each epoch. What it will do is change the learning rates to a value based on the epoch number. So in epoch 1, it is 1 times 10 to the -8 times 10 to the power of 1 over 20. And by the time we reach the 100 epoch, it'll be 1 times 10 to the -8 times

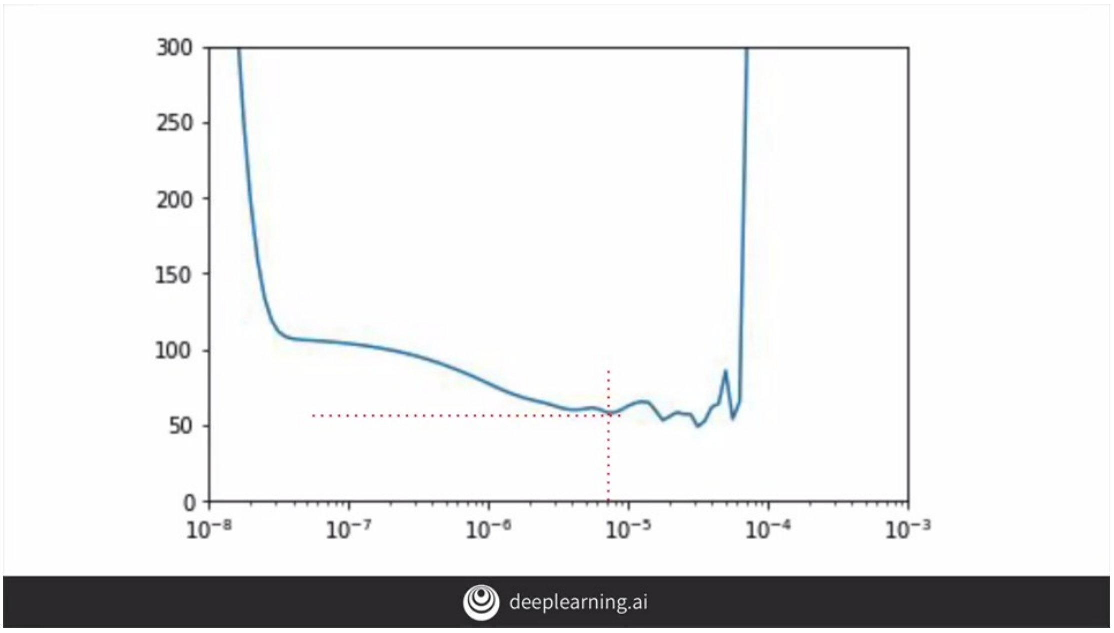
10 to the power of 5, and that's 100 over 20. This will happen on each callback because we set it in the callbacks parameter of modeled outfit.



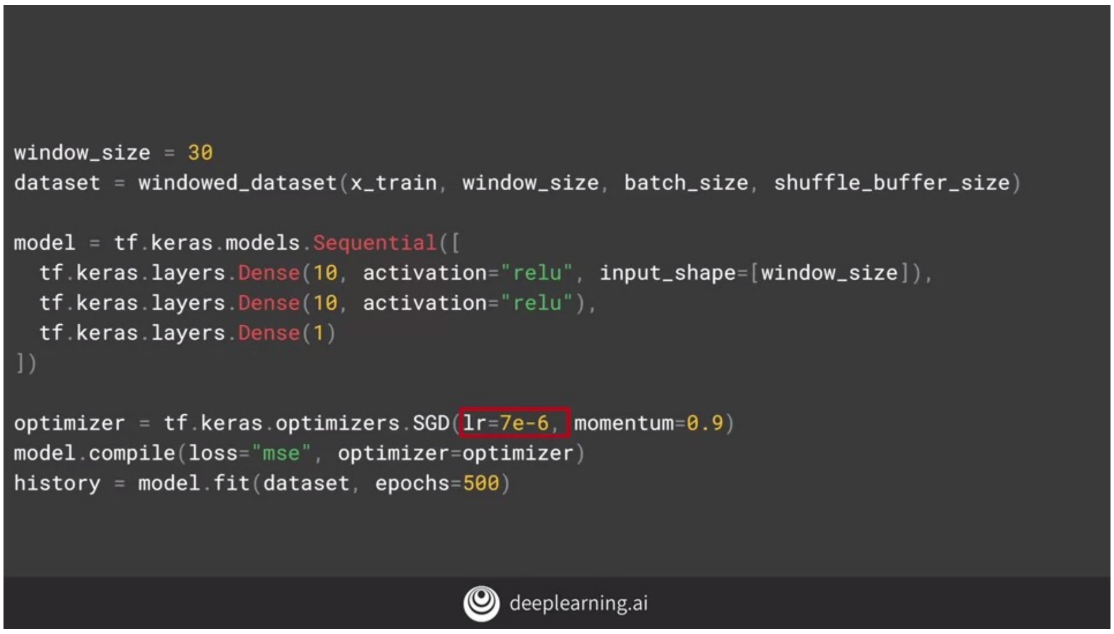
```
lrs = 1e-8 * (10 ** (np.arange(100) / 20))
plt.semilogx(lrs, history.history["loss"])
plt.axis([1e-8, 1e-3, 0, 300])
```



After training with this, we can then plot the last per epoch against the learning rate per epoch by using this code, and we'll see a chart like this. The y-axis shows us the loss for that epoch and the x-axis shows us the learning rate. We can then try to pick the lowest point of the curve where it's still relatively stable like this, and that's right around 7 times 10 to the -6.

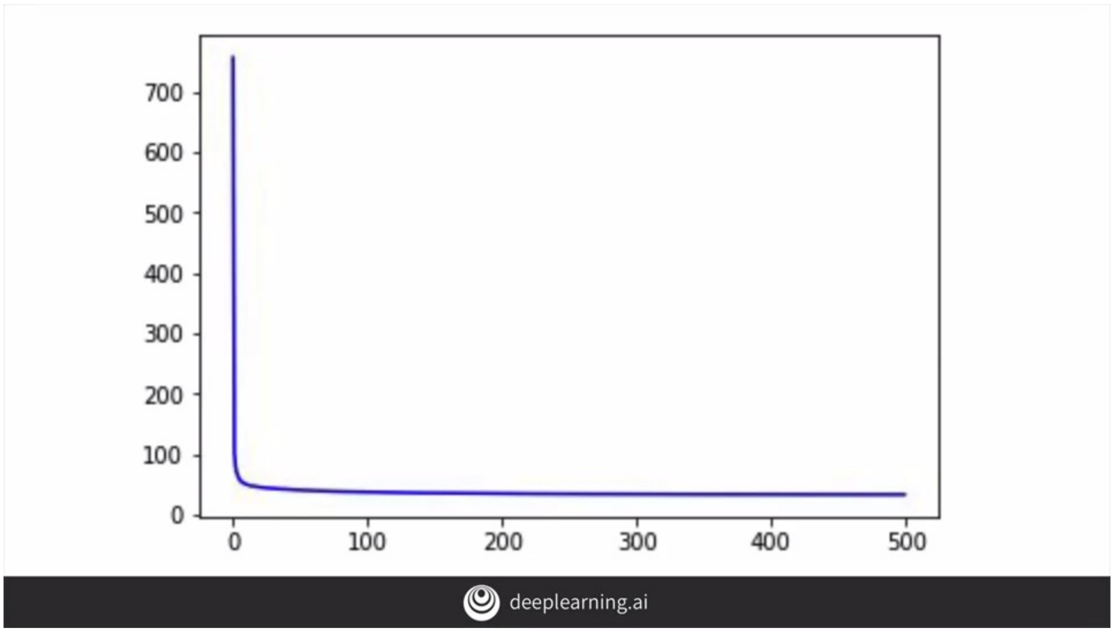


So let's set that to be our learning rate and then we'll retrain. So here's the same neural network code, and we've updated the learning rate, so we'll also train it for a bit longer.

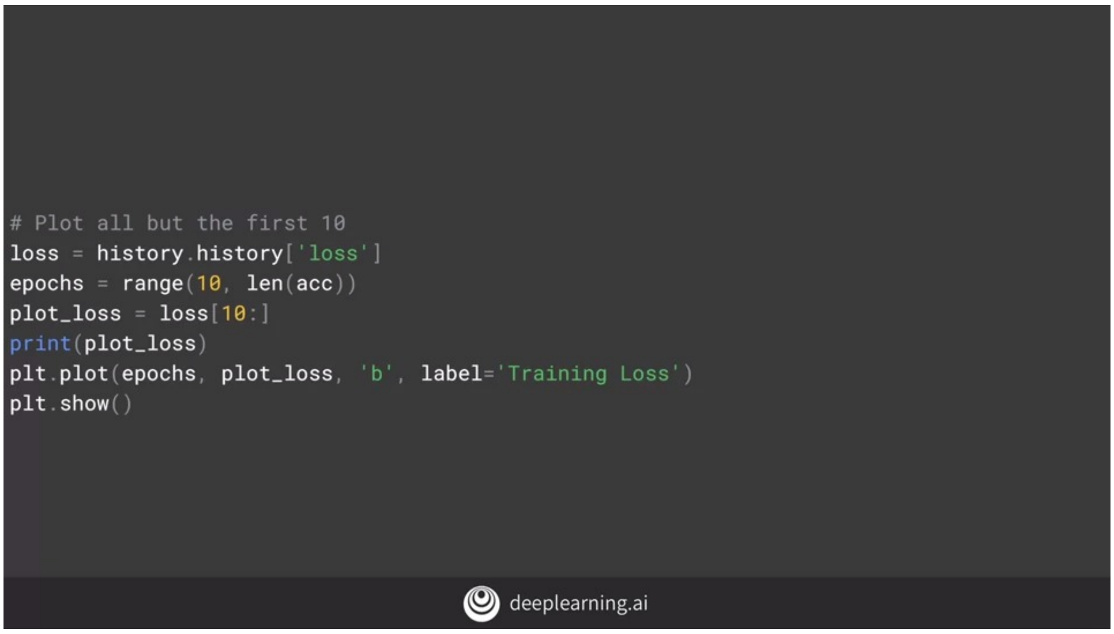


```
window_size = 30
dataset = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)

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)
```

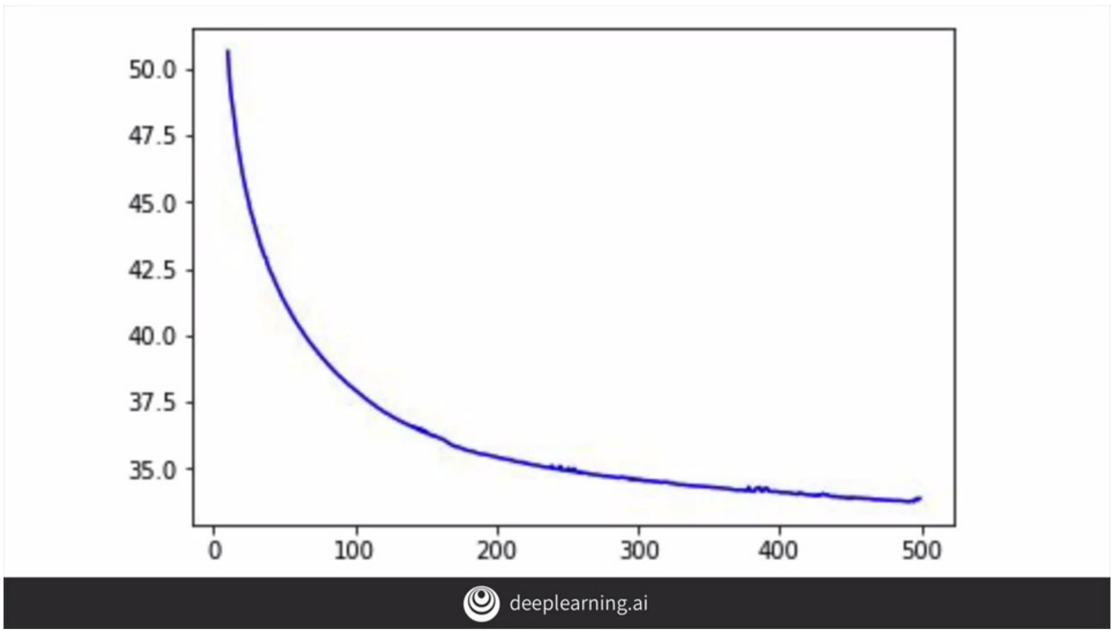
Let's check the results after training for 500 epochs. Here's the codes to plot out the loss that was calculated during the training, and it will give us a chart like this.



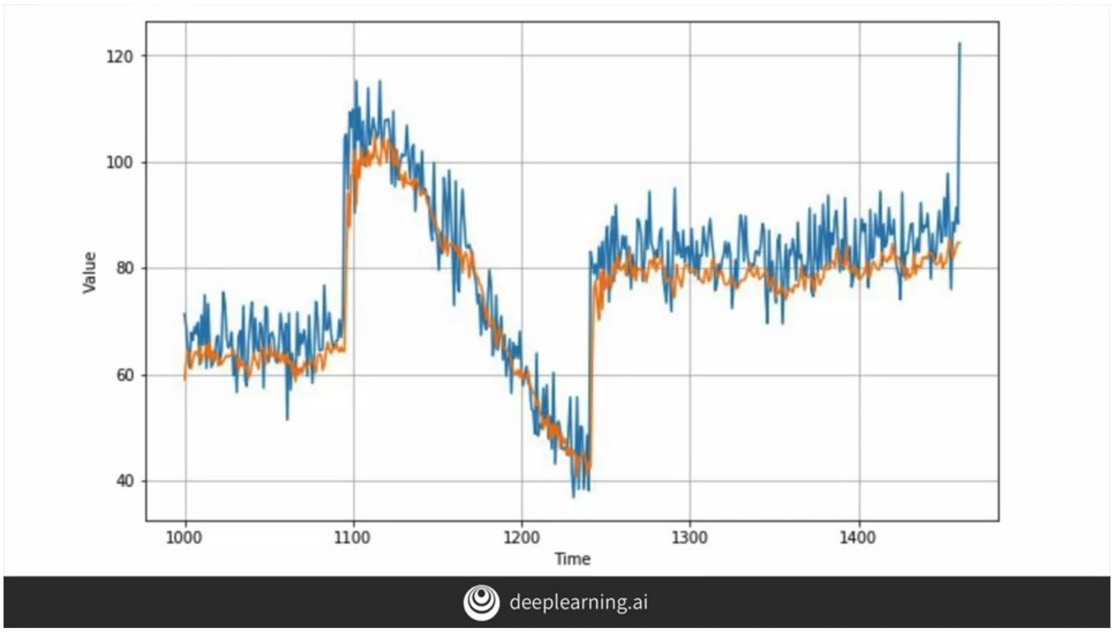
Which upon first inspection looks like we're probably wasting our time training beyond maybe only 10 epochs, but it's somewhat skewed by the fact that the earlier losses were so high. If we cropped them off and plot the loss for epochs after number 10 with code like this, then the chart will tell us a different story.



```
# 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()
```



We can see that the loss was continuing to decrease even after 500 epochs. And that shows that our network is learning very well indeed. And the results of the predictions overlaid against the originals looks like this.



And the mean absolute error across the results is significantly lower than earlier.
I'll take you through a screencast of this code in action in the next video. Using a very simple DNN, we've improved our results very nicely. But it's still just a DNN, there's no sequencing taken into account, and in a time series like this, the values that are immediately before a value are more likely to impact it than those further in the past. And that's the perfect set up to use RNS like we had in the natural language course. Now we'll look at that next week, but first, let's dig into this code.

S+P Week 2 Lesson 3