

1. Convolutions

Last week, we looked at recurrent neural networks including a simple RNN and an LSTM. You saw how they could be useful in learning a sequence like the one that we've been looking at. And how the LSTMs removed some of the issues we had with an RNN.

This week, you'll go one step further, combining convolutions with LSTMs to get a very nicely fitting model. We'll then apply that to real world data instead of this synthetic data set that we've been working on since the beginning of this course.

```
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)
```



Here's the LSTM that you looked at last week, except that I've added something at the beginning of the sequential stack.

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.

Check back to the convolutions course to learn more about that.

2. Bi-directional LSTMs

```

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.

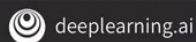
```

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

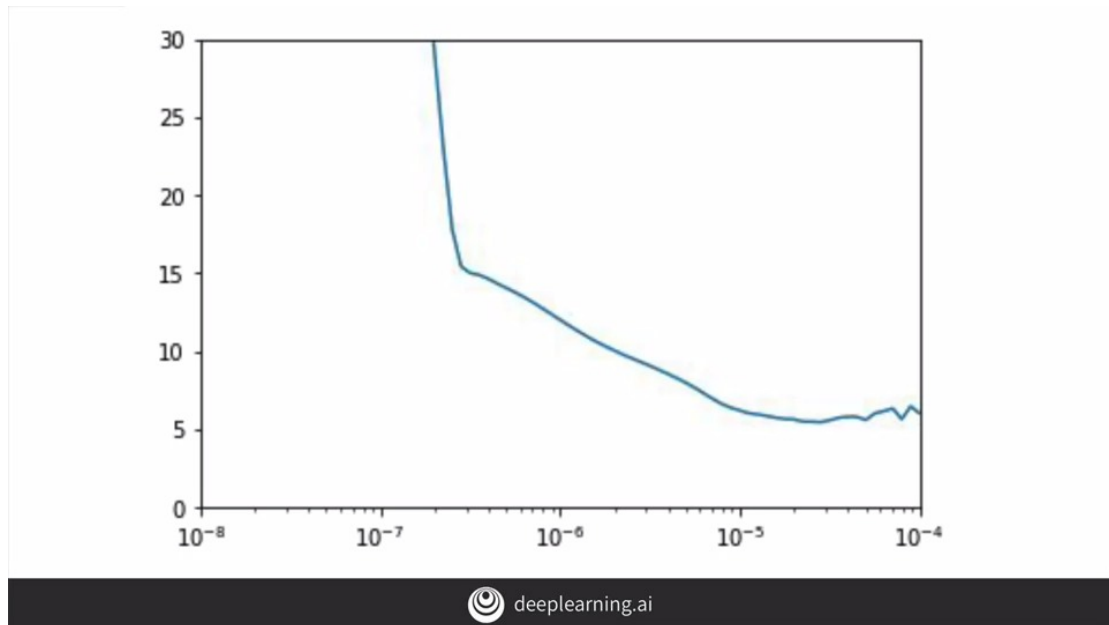
    ds = tf.data.Dataset.from_tensor_slices(series)
    ds = ds.window(window_size + 1, shift=1, drop_remainder=True)
    ds = ds.flat_map(lambda w: w.batch(window_size + 1))
    ds = ds.shuffle(shuffle_buffer)
    ds = ds.map(lambda w: (w[:-1], w[1:]))

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

```



This requires us to update the windowed_dataset 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.



Also similar to last week, the code will attempt lots of different learning rates changing them epoch by epoch and plotting the results. With this data and the convolutional and LSTM-based network, we'll get a plot like this. It clearly bottoms are around 10 to the minus five after which it looks a bit unstable, so we'll take that to be our desired learning rates.

```
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)
])

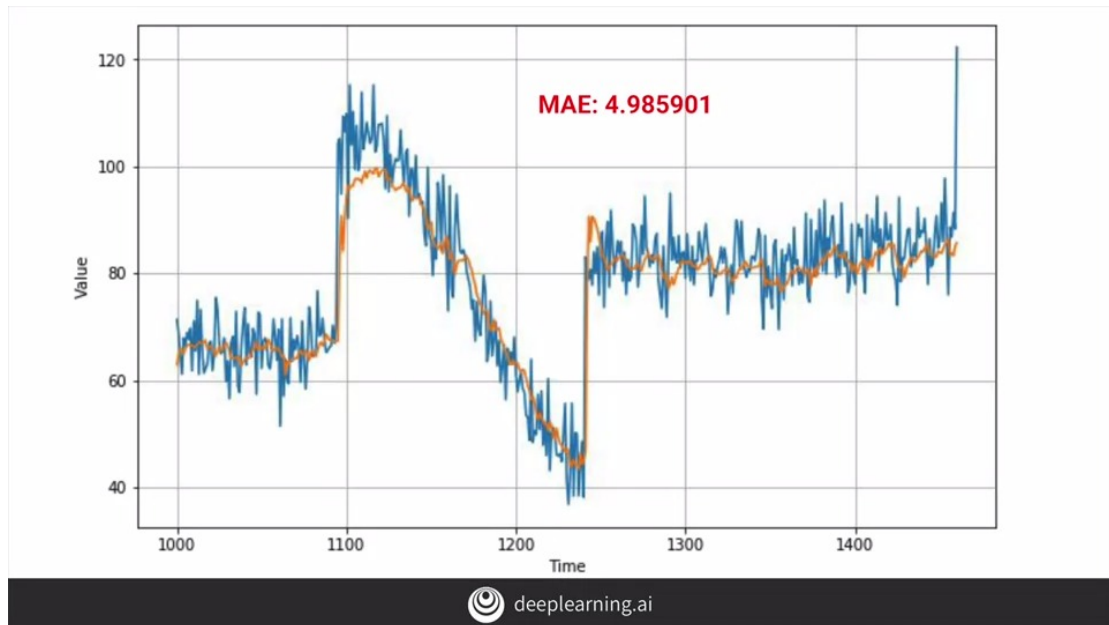
optimizer = tf.keras.optimizers.SGD(lr=1e-5, momentum=0.9)

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

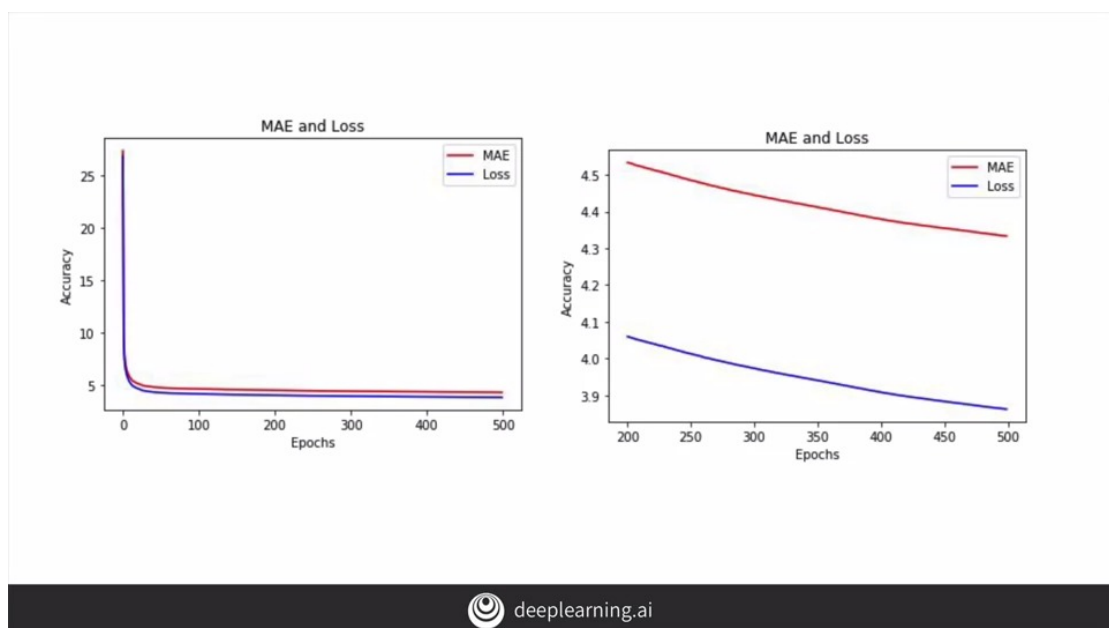
model.fit(dataset, epochs=500)
```

deeplearning.ai

Thus when we define the optimizer will set the learning rate to be 1e-5 as shown here.



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.

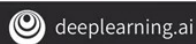
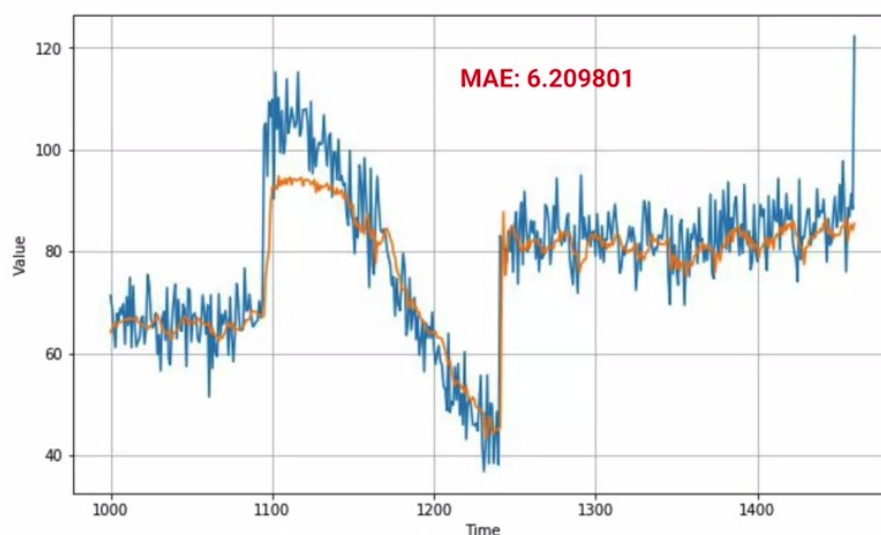
```

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)
])

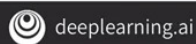
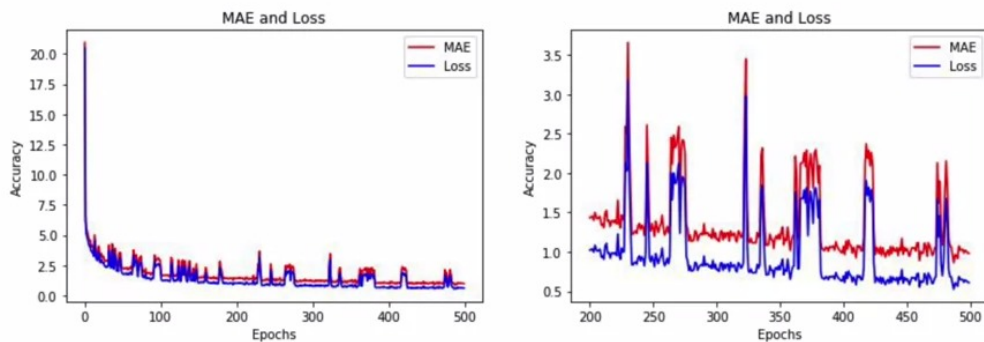
```



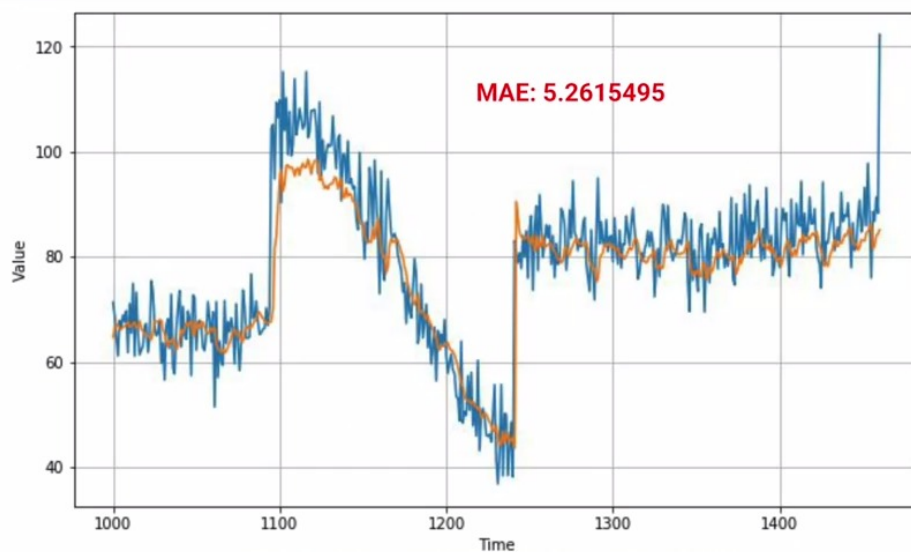
Now one method would be to make your LASTMs bidirectional like this. When training, this looks really good giving very low loss in MAE values sometimes even less than one.



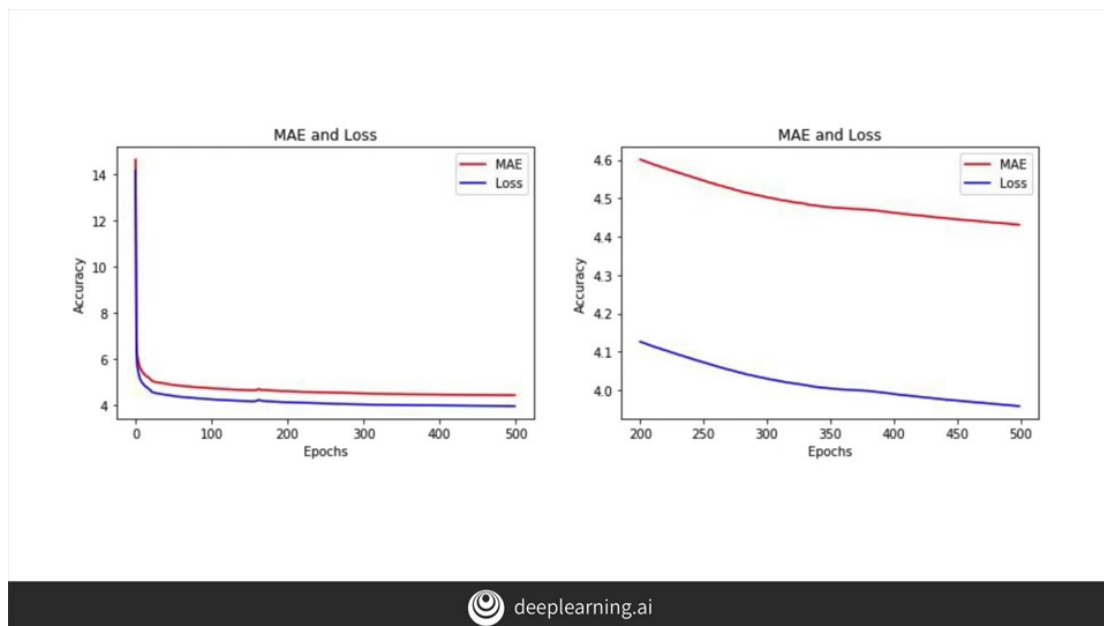
But unfortunately it's overfitting when we plot the predictions against the validation set, we don't see much improvement and in fact our MAE has gone down. So it's still a step in the right direction and consider an architecture like this one as you go forward, but perhaps you might need to tweak some of the parameters to avoid overfitting.



Some of the problems are clearly visualize when we plot the loss against the MAE, there's a lot of noise and instability in there. **One common cause for small spikes like that is a small batch size introducing further random noise.** I won't go into the details here, but if you check out Andrea's videos and his course on optimizing for gradient descent, there's some really great stuff in there. One hint was to **explore the batch size** and to make sure it's appropriate for my data. So in this case it's worth experimenting with different batch sizes.



So for example experimented with different batch sizes both larger and smaller than the original 32, and when I tried 16 you can see the impact here on the validation set, and here on the training loss and MAE data.



So by combining CNNs and LSTMs we've been able to build our best model yet, despite some rough edges that could be refined. In the next video, we'll step through a notebook that trains with this model so that you can see it for yourself, and also we'll maybe learn how to do some fine-tuning to improve the model even further. After that, we'll take the model architecture and apply it to some real-world data instead of the synthetic ones that you've been using all along.

3. LSTM


In the previous video, you saw how you could stack a convolutional layer with LSTMs and bidirectional LSTMs to do sequence prediction. In this video, we'll go through a workbook for this, which you can then try for yourself later. As always, let's check if we have tensorflow installed. If it isn't, install the latest either nightly or the latest release from tensorflow.org. Once you know you have version two, then this code will generate the synthetic time series for you, and this code will turn the array of data into a dataset for training. Note that we've expanded the dimensions on the first line. This helper function can perform the forecasting for us after training. This first copy of the neural network has run for a quick 100 Epoch run to try and pick the optimum learning rate for the optimizer. When it's done, we'll plot the results and we'll see the 10 to the minus 5 is the optimum value, so we'll set that on the SGD for the next training run. We'll train for 500 Epochs now and keep an eye on our loss and our mae. When it's done, we'll run our forecast and plot the results. Already the curve is looking much better and the plateau that we'd seen in previous training has vanished, and our mae is low, it's just above five at the validation set. Finally, if we plot our training loss and mae, we also see a healthy curve downwards.

S+P Week 4 Lesson 1

4. Real data – sunspots

Over the last few weeks, you looked at time-series data and examined a few techniques for forecasting that data including statistical analysis, linear regression, and machine-learning with both deep learning networks and recurrent neural networks. But now we're going to move beyond the synthetic data to some real-world data and apply what we've learned to creating forecasts for it. Let's start with this dataset from Kaggle, which tracks sunspots on a monthly basis from 1749 until 2018. Sunspots do have seasonal cycles approximately every 11 years. So let's try this out to see if we can predict from it.

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

 deeplearning.ai

It's a CSV dataset with the first column being an index, the second being a date in the format year, month, day, and the third being the date of that month that the measurement was taken. It's an average monthly amount that should be at the end of that month. You can download it from Kaggle or if you're using the notebook in this lesson, I've conveniently hosted it for you on my Cloud Storage. It's a pretty simple dataset, but it does help us understand a little bit more about how to optimize our code to predict the dataset based on the nature of its underlying data. Of course, one size does not fit all particularly when it comes to data that has seasonality. So let's take a look at the code.

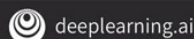

```
!wget --no-check-certificate \
https://storage.googleapis.com/laurencemoroney-blog.appspot.com/Sunspots.csv \
-O /tmp/sunspots.csv
```



Okay, first of all, if you're using a codelab, then you'll need to get the data into your codelab instance. This code will download the file that I've stored for you. You should really get it from Kaggle and store it on your own server or even manually upload it to codelab, but for convenience, I've stored it 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]))
```

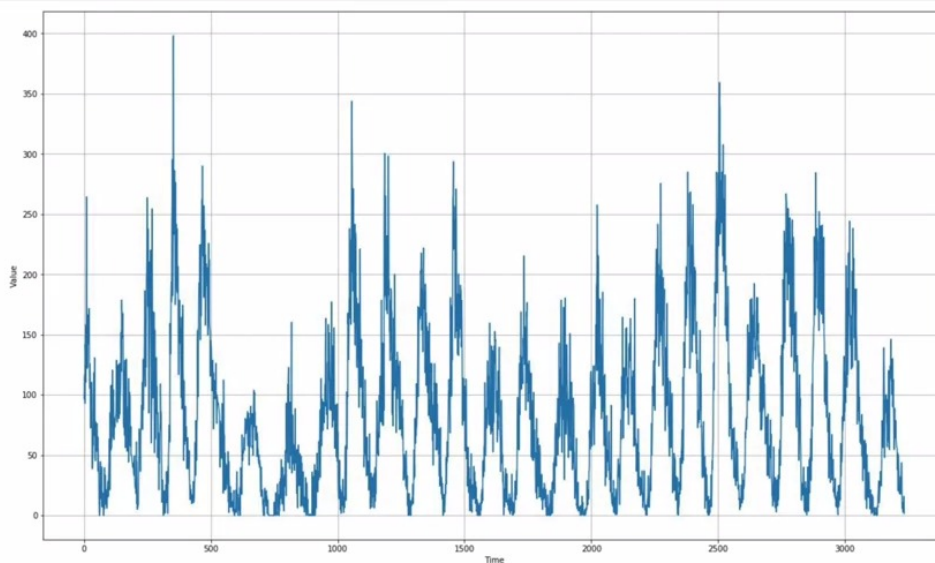


Here's the code to read the CSV file and get its data into a list of sunspots and timestamps. We'll start by importing the CSV library. Then we'll open the file. If you're using the codelab and the W get code that you saw earlier, downloads the CSV and puts it into slash temp. So this code just reads it out of there. 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. Then, we will look through the reader reading the file line by line. Our sunspots are actually in column 2 and we want them to be converted into a float. As the file is read, every item will be read as a string so we may as well convert them now instead of iterating through the list later and then converting all the datatypes. Similarly, we'll read the time steps as integers.

```
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.



If we plot our data it looks like this. 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.

```

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

```



As before, let's split our series into a training and validation datasets. We'll split at time 1,000. We'll have a window size of 20, batch size of 32, and a shuffled buffer of 1,000. We'll use the same window dataset code that we've been using all week to turn a series into a dataset which we can train on.

5. Train and tune the model

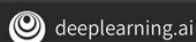
```

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

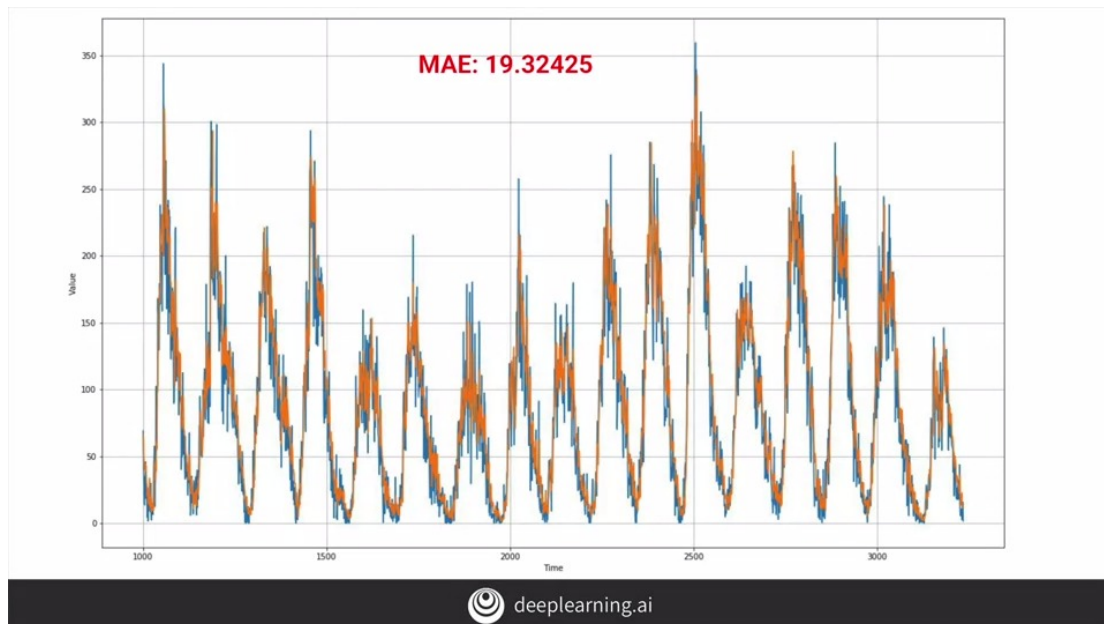
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(10, input_shape=[window_size], activation="relu"),
    tf.keras.layers.Dense(10, activation="relu"),
    tf.keras.layers.Dense(1)
])

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

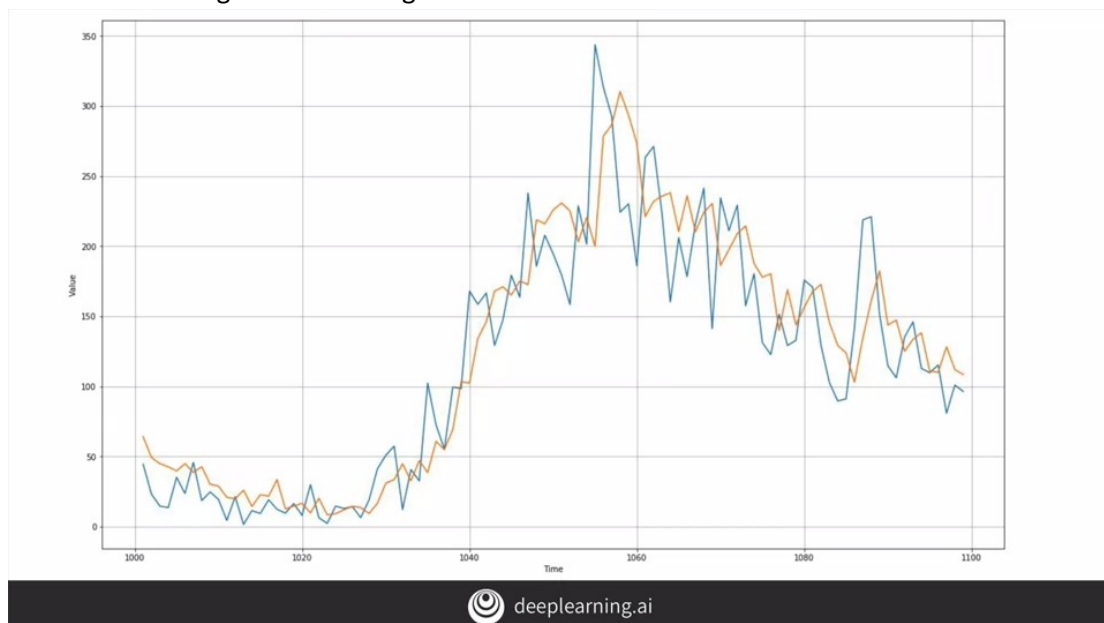
```



We'll go back to the simple DNN that we saw way back in week two for training on and we'll see what happens.



We get 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. 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.

But if you remember this chart, 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 interleaving with each other. So what would happen if we retrain with a window size of 132, which is 11 years worth of data as our window size.

Now while this chart looks similar, 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?

Well, by 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. Zooming in on the data again, we'll see something like

this where it's just the typical time series. Values later on are somewhat related to earlier ones, but that's a lot of noise. So maybe we don't need a huge window of time in order to train. Maybe we should go with something a little bit more like our initial 20, let's try 30.

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. And then when we retrain, we'll get this. Our MAE has improved to 15 but can we make it even better? Well, one thing we can try is to edit the neural network design and height of parameters. If you remember, we had three layers of 10, 10, and 1 neurons. Our input shape is now larger at 30. So maybe try different values here, like 30, 15, and 1, and retrain.

Surprisingly, this was a small step backwards, with our MAE increasing.

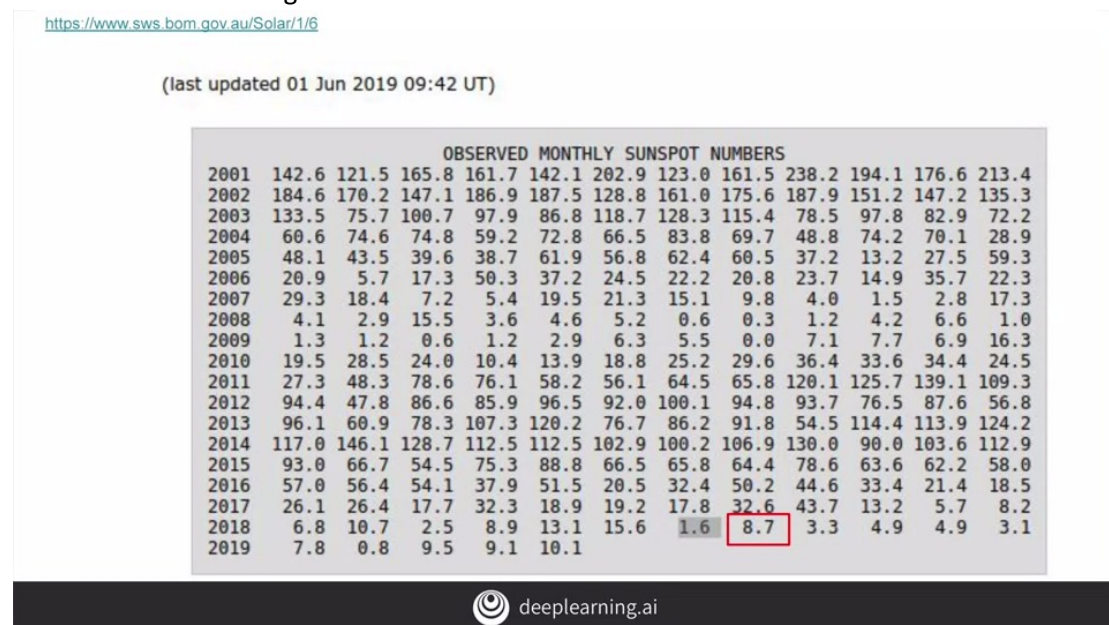
It also wasn't worth the extra compute time for the extra neuron layers. So let's switch back to 10, 10, 1 and instead look at the learning rate.

Let's tweak it a little.

Now after retraining, I can see my MAE has decreased a bit which is good.

6. Prediction

Out of interest, let's do a prediction. The window size I'm using is 30 steps, and the dataset is 3,235 steps long. So if I want to predict the next value after the end of my dataset, I would use this code. And I would get the result 7.0773993.



The dataset goes up to July 2018, so I'm actually predicting 7.077 sunspots for August 2018. And if I look at this chart of observations, which does have some slightly different data from my dataset, I can see that the actual recorded number of sunspots in August 2018 was 8.7. So the prediction isn't too bad, but let's see if we can improve on it.

```

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))

```



With these settings, I got the MAE down to 13.75, and the prediction was 8.13, which is much closer to the actual real reading of 8.7. There is a random elements in creating models, however, so your results may vary. 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.

So here, we looked at using a DNN to predict sunspot values. With a little bit of tuning, we reduced our MAE a bit. And when we tried to predict the next month's value using this model, we got quite close to the actual value. In the next video, you'll go through this workbook to see it in action. And then you should try it out for yourself, to see if you can improve on what I got. After that, you'll look into the RNNs again and see if you can get better predictions using those.

7. Combining our tools for analysis

So we've looked at the sunspot data using a standard DNN like way back at the beginning of this course. We've also been working a lot with RNNs and a little with convolutional neural networks. So what would happen if we put them all together to see if we can predict the sunspot activity? This is a difficult dataset because like we've seen already, while it's seasonal the period is really long, around 11 years, and it's not perfectly seasonable during that period. So let's take a look at using all the tools we have to see if we can build a decent prediction using machine learning. So here's the first piece of code we can try. I've gone a little crazy here, so let's break it down piece by piece.

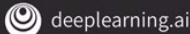

```

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])

```



First of all I'm setting the batch size to 64 and the window size to 60. Then we'll start with a 1D convolution that we'll learn 32 filters. This will output to a couple of LSTMs with 32 cells each before feeding into a DNN similar to what we saw earlier, 30 neurons, then 10, and one. Finally, as our numbers are in the 1-400 range, there is a Lambda layer that multiplies out our X by 400. With the first test run to establish the best learning, rate we get this chart. This suggests the best learning rate for this network will be around 10 to the minus 5. So when I trained for 500 epochs with this setup, here's my results. It's pretty good with a nice low MAE. But 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, and as we saw from earlier videos, one of the best things to look at in these circumstances is the batch size. So I'll increase it to 256 and retrain. 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, which then brings me to another thing to try. 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? My DNN already look good, so I won't change them. 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 and I got these results. 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. Experimenting with hyperparameters like this is a great way to learn the ins and outs of machine learning, not just with sequences but with anything. I thoroughly recommend spending time on it and seeing if you can improve on this model. In addition, you should accompany that work with looking

deeper into how all of these things in machine learning work and Andrews courses are terrific for that. I strongly recommend them if you haven't done them already.