Notebook here:

<https://colab.research.google.com/drive/1El1ifPj9JaOIqsuI2druqsFDIBFEHCQJ?usp=sharing>

## RNN Layers

* 3D input shape: (Batch size, time steps, dimensionality of inputs)
* Time steps will be the window size

## Lambda Layers

model = Sequential([

Lambda(lambda x: tf.expand\_dims(x,axis=-1), input\_shape=[None]),

SimpleRNN(40, return\_sequences=True),

SimpleRNN(40),

Dense(1),

Lambda(lambda x: x \* 100.0)

])

* The dataset returned a 2D batch (batch size, no.of time steps)
* The RNN expects 3D (Need to add an extra dimension: dimensionality of inputs)
* Expand the array by one dimension (the last dimension, denoted by axis= -1). This will transform from 2D to a 3D tensor (With the last dimension having value 1, since it is a univariate time series)
* By setting input\_shape=[None], this means that the model can take sequences of any length (Note that this refers to the 2nd dimension, not the first)
* When you define the input\_shape, you are essentially defining the 2nd and the 3rd dimension (Since the first dimension will be automatically be batch\_size)
* Since the lambda layer is the first layer, you must define its input shape. As the data is initially 2D, you only have to specify the shape of the 2nd dimension
* By scaling outputs by 100, you can help training (Outputs of the RNN layers are between -1 and 1. Since the time series values are usually in the order of 10s like 40s, 50s…. You scale the outputs to be similar to these values which help with learning)

## Using LSTM

model = Sequential([

Lambda(lambda x: tf.expand\_dims(x,axis=-1), input\_shape=[None]),

Bidirectional(LSTM(32, return\_sequences=True)),

Bidirectional(LSTM(32)),

Dense(1),

Lambda(lambda x: x \* 100.0)

])

## 

## Using Conv1D

Sunspot notebook here:

<https://colab.research.google.com/drive/1onrtnWFda51E3wbHBnmXco_hls-t7rbb?usp=sharing>

* You will remove the lambda layer and instead carry out reshaping when creating the dataset

def windowed\_dataset(series, window\_size, batch\_size, shuffle\_buffer\_size):

series = tf.expand\_dims(series, axis=-1)

ds = Dataset.from\_tensor\_slices(series)

ds = ds.window(window\_size+1, shift=1, drop\_remainder=True)

ds = ds.flat\_map(lambda window: window.batch(batch\_size + 1))

ds = ds.shuffle(shuffle\_buffer\_size).map(lambda x: (x[:-1], x[1:]))

ds = ds.batch(batch\_size).prefetch(1)

return ds

* Here, you must expand\_dims when creating the dataset (And not in the lambda layers) since the first layer is Conv1D, not a RNN layer
* Also note that instead of your y being a single value, your y is now a range of values (from the 2nd value to the predicted value)
* Example:
* x : t=0 to t=39
* y: t=1 to t=40 (t=40 is the value to be predicted)

model = Sequential([

Conv1D(32,kernel\_size=5,strides=1, padding='causal', activation='relu', input\_shape=[None,1]),

Bidirectional(LSTM(32, return\_sequences=True)),

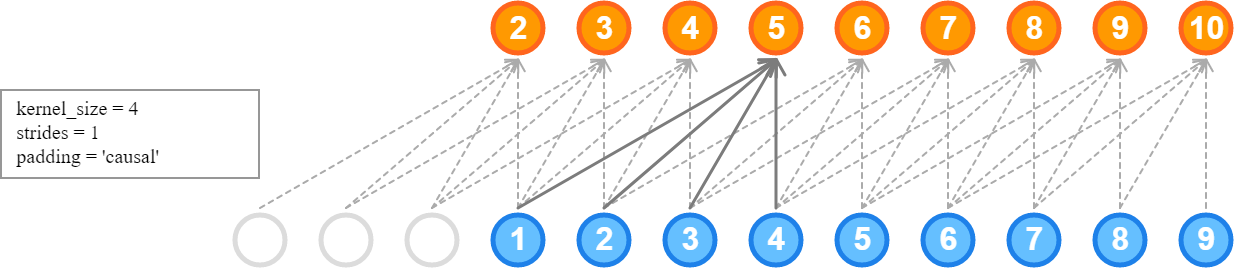
Bidirectional(LSTM(32)),

Dense(1),

Lambda(lambda x: x \* 100.0)

])

* Padding = ‘causal’ means that you pad the input with zeros in the front so you can predict values of early timesteps



* In this case, since your kernel\_size = 4 you apply padding to the front 3 values (So you can use the first actual input for all 4 times)
* This is much like Conv2D padding=’same’ where you pad so the corner pixels will be used the same number of times with the middle pixels

def model\_forecast(model, series, window\_size):

series = tf.expand\_dims(series, axis=-1)

ds = tf.data.Dataset.from\_tensor\_slices(series)

ds = ds.window(window\_size, shift=1, drop\_remainder=True)

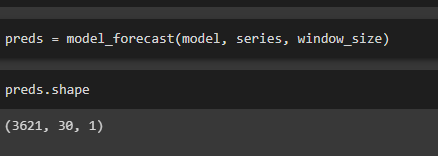
ds = ds.flat\_map(lambda x: x.batch(window\_size))

ds = ds.batch(32).prefetch(1)

forecast = model.predict(ds)

return forecast

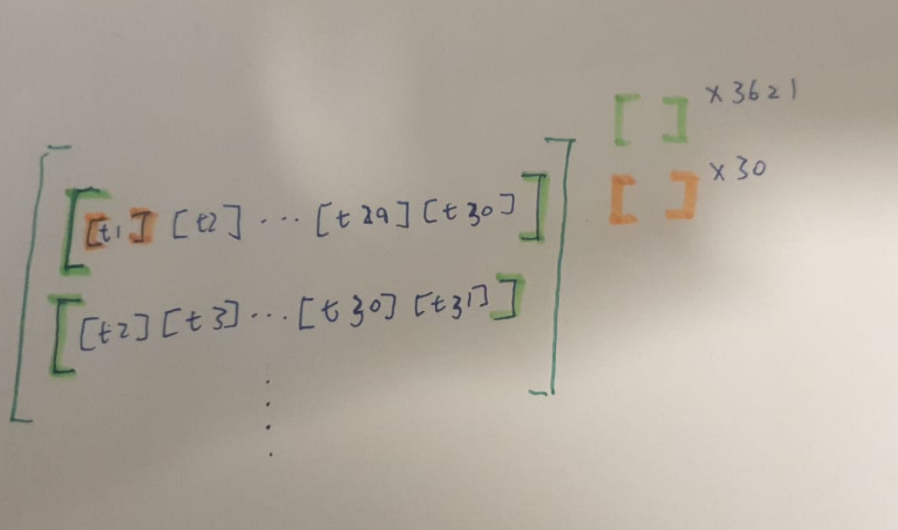
* Almost the same as when you defined the widowed\_dataset
* Only difference is that since you don’t have a y value to return, you don’t need to use window\_size + 1 and no need to .map()



* Since your window size is 30, you end up with 3261 X windows (where each window contains 30 elements)



* Since the y value is not a single value but a range of values (e.g t=1 to t=39), you want to index the last value out (i.e t=39)
* Split\_time = 2500
* You want to get the predictions from split\_time to the end (i.e from t=2500 to t=3650)
* Hence, you get the Xs from t=2470 (since that window will contain values from t=2471 to t=2500) and you index the last element (to get t=2500) using -1
* Then, since you want to access the raw element, use 0 (Because its like [value])
* So val\_preds will be a 1D array containing all the last values of all the windows (from t=2500)



* This is how the whole preds array look like. (Not the val\_preds array)
* The green bracket represents one window (there are 3621 windows (or Xs) in the entire preds array)
* Each window contains 30 time series values