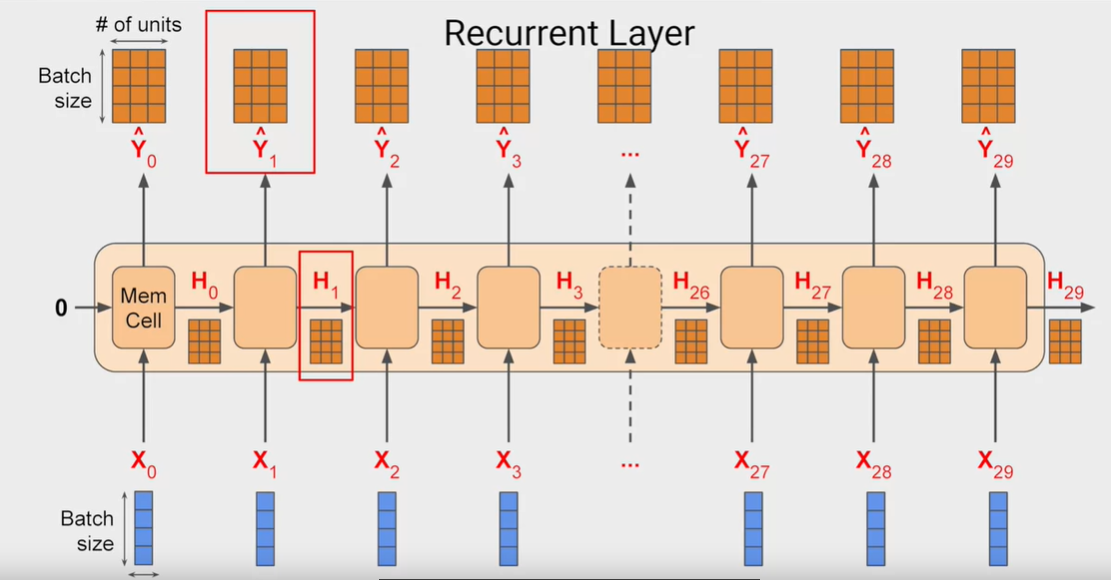
## Shape of the inputs in RNN :

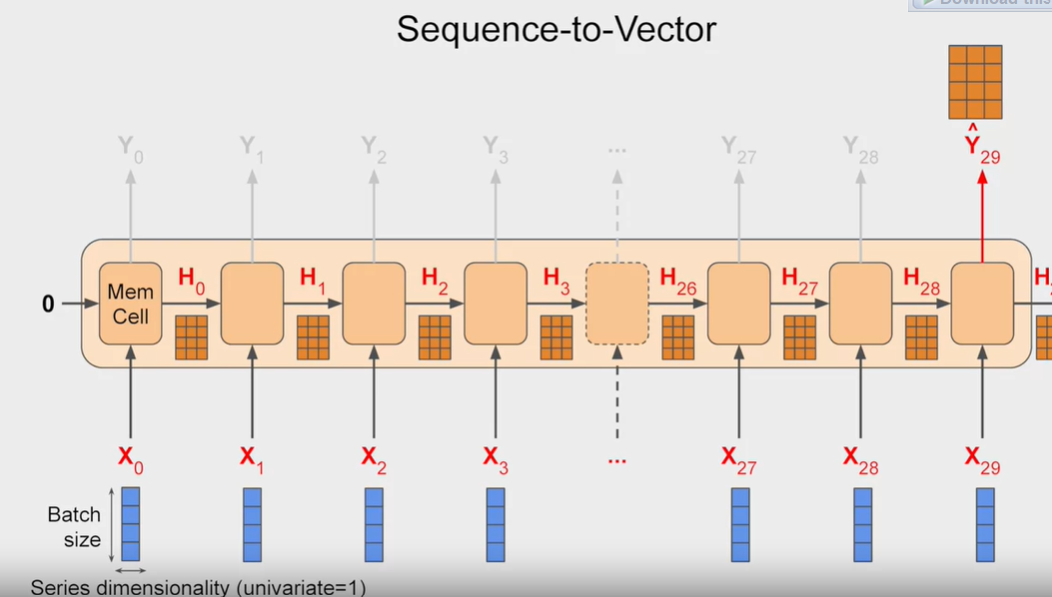


* The RNN layers ( and all the sequence layers ) expects a 3D input shape , the first dimension is the batch size , the second one is the window size and the third one is equal to the data length .
* For the univariant data the third dimension is equal to 1 ( since we are having one single factor to predict ) but for the multivariant data it’s equal to the number of factors to predict
* So to predict a temperature of a region with window\_size=20 and batch\_size=4 , the dimension will be 20\*4\*1
* The size of the memory cell H is equal to the Y dimension ( number of units in the out put layer \* batch\_size ) because in the default : Hi=Yi

#### The size of the output :

* The size of the output is equal to the batch\_size \* Number of units in the output layer of our model \* window\_size ( size of the sequence )
* The third dimension represents the number of outputs Y ( which is 30 in the image )
* To get the whole Y sequence ( one to one ) , We have to specify the argument return\_sequences=True in the RNN layer

#### For the Sequence-to-vector RNN case :



* For sequence to vector models , We are ignoring the output of all the outputs Y except the last one Y29 , so the size of the output is in 2D ( batch\_size \* output layer dimension ) instead of 3D ( there isn’t window\_size )

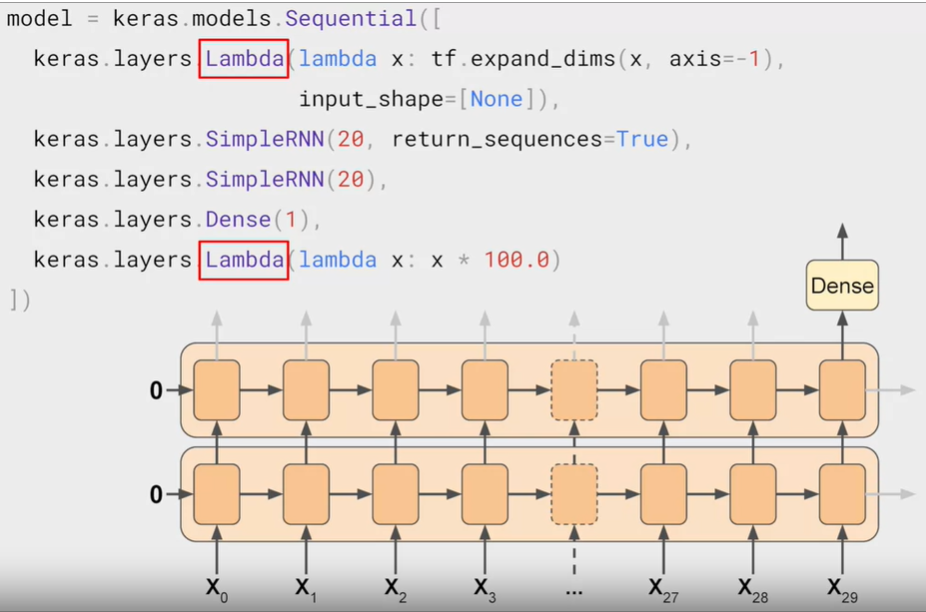
## Deep RNN for sequence-to-vector:

* It’s pretty intuitive what we should do to construct a Deep architecture: we should return the whole sequence in the first sequence (by specifying the argument return\_sequence=True in the first layer) and in the final RNN layer we only expect the output of the final Y
* In the Input shape we specified it as [ None , 1 ] ( 2D )
  + TensorFlow actually expects an input shape of 3D ( because that’s what RNN layer expects ) , the first dimension is for the batch\_size and since it’s can be any size : we don’t need to put a restriction it, we just ignore it in the input\_shape
  + None for the batch\_size , we set it None to say that there isn’t a restriction in the batch size to execute the model
  + The 1 in the third dimension is for the data size , and since we are handling univariant data : the dimension have to be set to 1

## Deep RNN for sequence-to-sequence:

* If we want to return the whole sequence and not only the final output , we just need to specify return\_sequence

## Lambda layers to preprocess the data for the intermediate layers:



### Role of the first lambda layer :

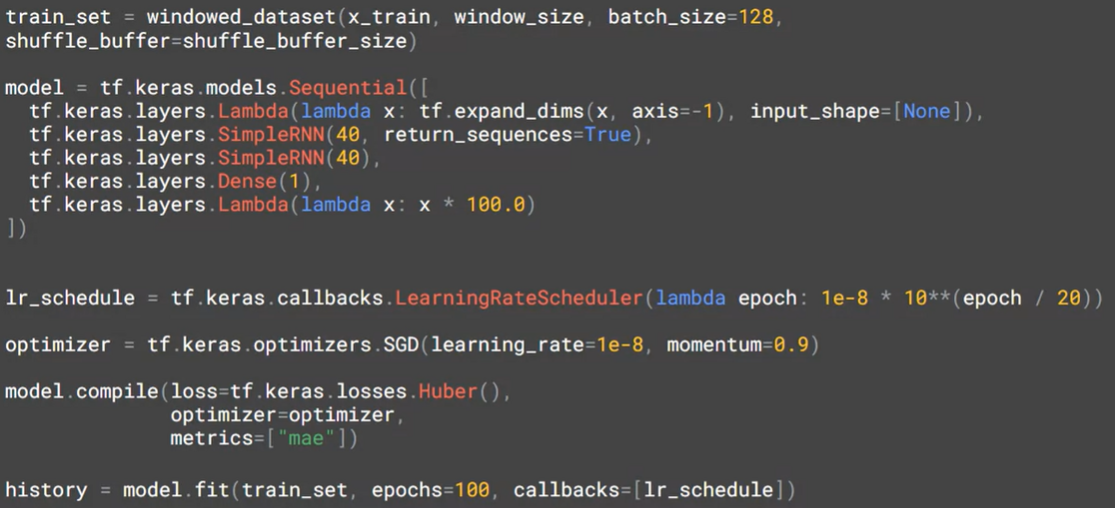
* We are planning to pass to our model the output of windowed\_dataset function which returns a 2D array ( batch\_size\*window\_size ) , but the RNN expects a 3D dimension data as input: the third one is for the Series demonsiality ( which is 1 for the univariant data ) and that’s why we added a new axis using the lambda layer : it will add a empty array which means there is no restriction in the series dimensionality

### Role of the second lambda layer :

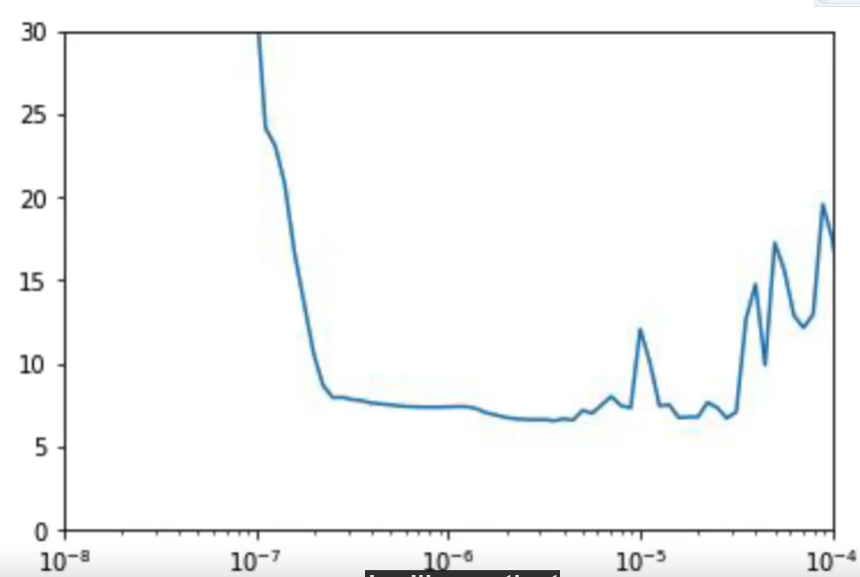
* Briefly, the role of that lambda layer is to scale the output of the Model to match the input data , this is will increase the quality of the learning ( btw , the default activation function in the RNN layers is “tanh” which has values between [ -1 , 1 ] ) and by multiplying the output by 100 we will get values like 40s , 50s , 60s , 70s which is In the same scale of our input data

## Adjusting the learning rate and choosing the optimal one :

* As we used to do, We are going to use the Learning rate Scheduler:

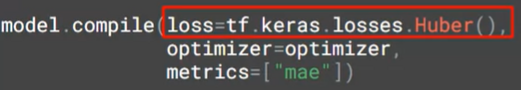


* And that’s we will get for this example:

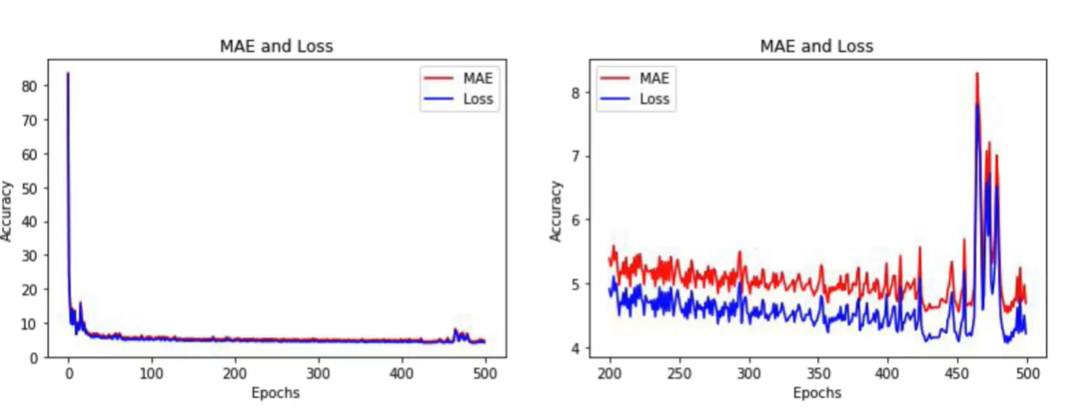


* We can see that the optimal learning rate is between 10^-6 and 10^-5 so we define it to be : 5\*10^-6

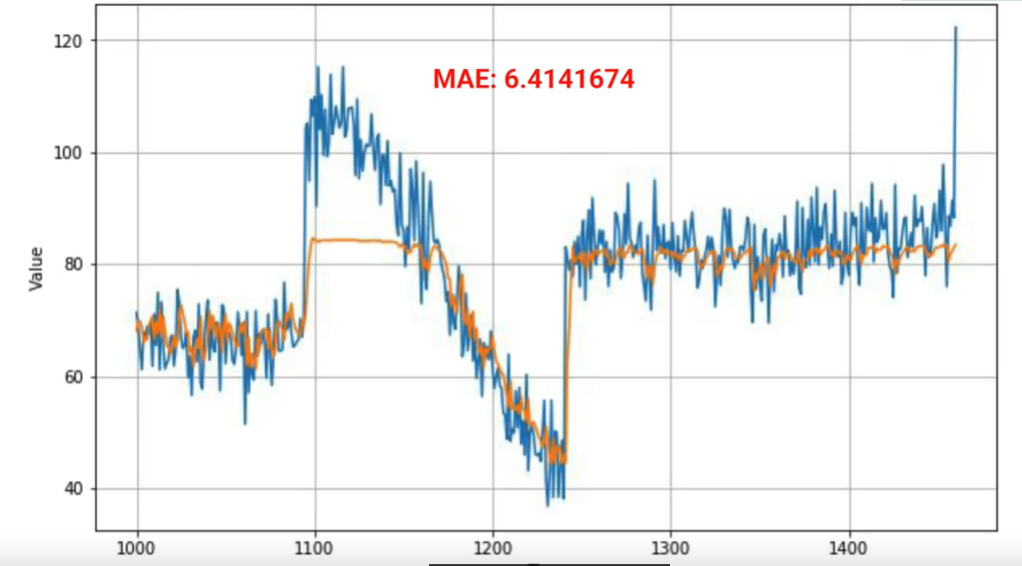
### Huber is an interesting Loss function for Sequential Data :



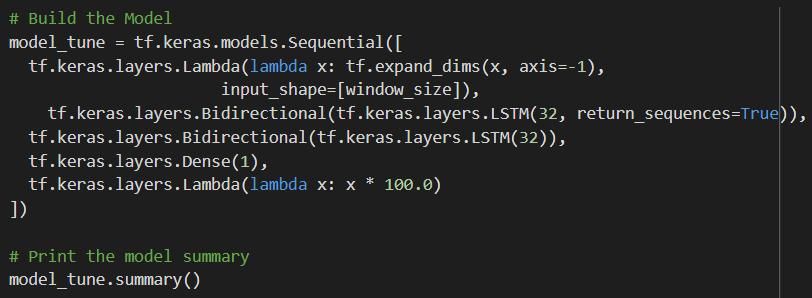
* In our Model we used a completely new Function called Huber , this function is special and unique because it’s less sensitive the outliners ( which is our noisy data in this case which has spikes )



* By visualizing the MAE and loss ( Huber ) we can see that Huber has a litter values
* We notice also that the losses got unstable after the 400th epochs , So We will just train our model for 400 epochs to have less time execution + to avoid the non-stabilization of the loss values
* And here is the final result:



## Using LSTM instead of SimpleRNN :



* It’s the pretty same code as SimpleRNN , we just replace the two RNN layers by two Bidirectional LSTM layers