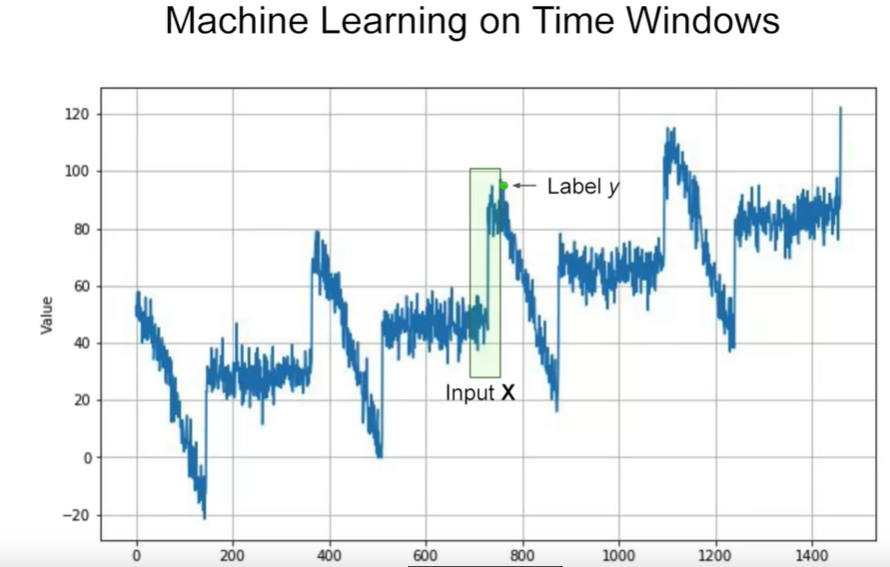
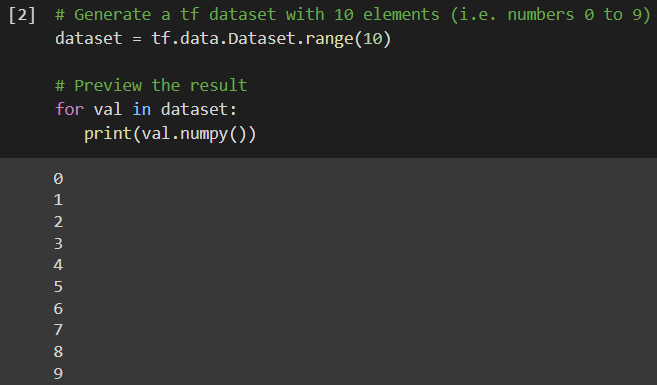
## Doing Time Series predictions in Machine Learning:

### The principle to follow :



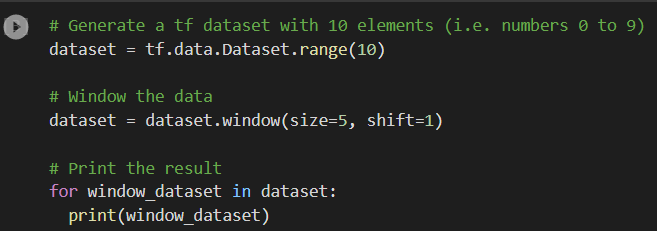
* Our main goal is to transform our timeseries data in a set of features and labels
* Using Windowing technique: the features of the label y corresponding to the time t will be the last M values right before the y ( the values in t-1 , t-2 , … , t-m ) (M is the window size ) : they are the input X

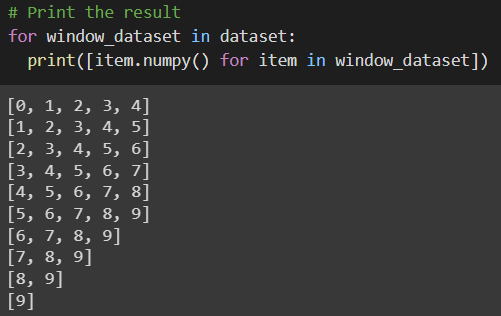
### Generating a random Dataset with tf.data.Dataset or using directly a ready-to-use data :



* Using the function range , I could create a data in form of sequential numbers from 0 to 9

### Windowing the data :

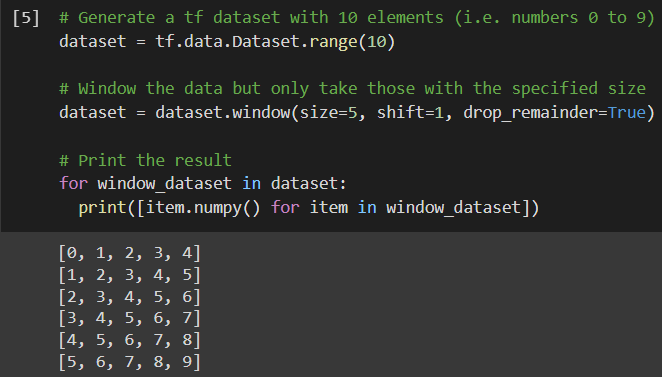




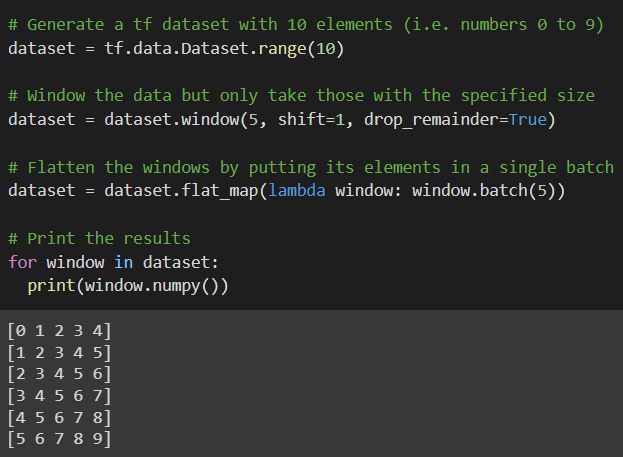
* Our plan is to for each consecutive 5 items , we will take the first four ones as a features and the last one as its label , so if our data is : [0,1,2,3,4,5,6,7,8,9]

Our dataset will contain as first row : [0,1,2,3] as the features and [4] is their label

And the next row will contain [ 1 , 2 , 3 , 4 ] as features with [5] as label

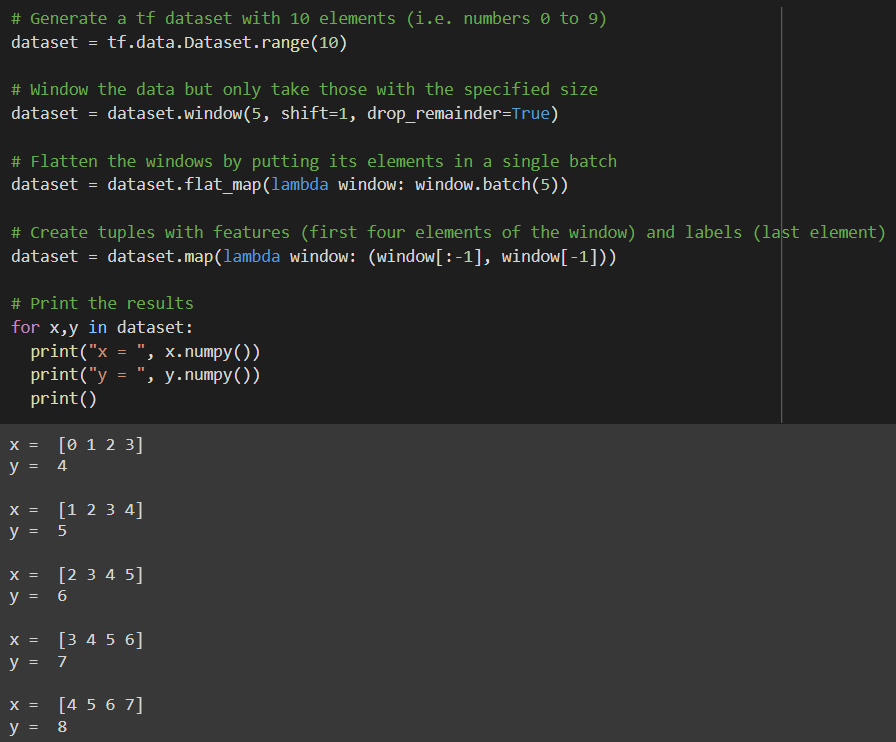
* Thanks to window function , by providing size=5 as argument and shift=1 , Our data will be splitted into arrays of array of 5 consecutive elements .
* By reaching to the end of our data , the size of the generated array cannot reach the size argument , so well use the argument drop\_reminder=True like it shown in the image below to get rid of the generated arrays with length inferior than the specified size

### Transfer the Dataset structure into Tensors :



* Thanks to the flat\_map() function with batch function inside : we could convert our dataset to Tensors that Tensorflow usually works with
* We called the function .batch(5) to get exactly the same arrays we got when we did the windowing where 5 represents the window size

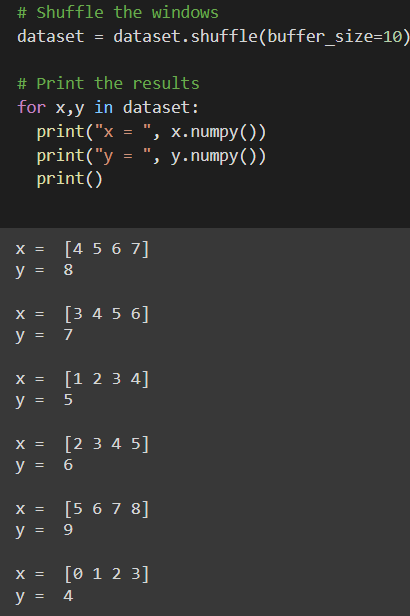
### Split each array to features and their label :



* In this step will just use the map python function to transform each array into array of features ( first 4 elements ) and their label ( the last element )

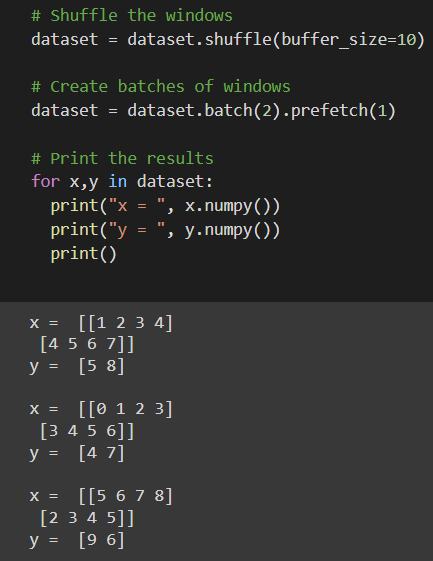
### Shuffling the data is important to avoid the overfitting:

* It is good practice to shuffle your dataset to reduce sequence bias while training your model. This refers to the neural network overfitting to the order of inputs and consequently, it will not perform well when it does not see that particular order when testing. You don't want the sequence of training inputs to impact the network this way so it's good to shuffle them up

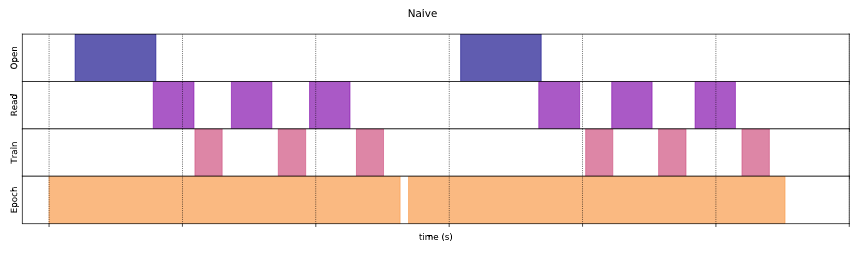


* We will just call shuffle function with buffer\_size as argument
* The buffer\_size value should be equal or greater than the number of rows in our dataset( which is 6) to get best shuffle possible , so we specified 10

### Everything is Ready , I just have to batch the training set ! :

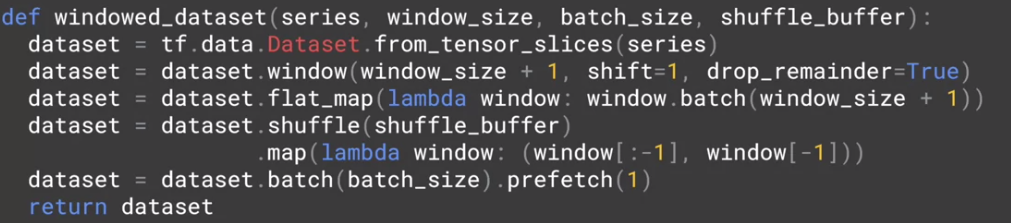


* After generating our dataset and splitting it into features and labels using the windowing technique : We will call the Batch method to split our dataset for the training , We specify the argument as 2 , that means each batch will contain two rows represented by X which contains the features and the Y which contains the labels
* The prefetch function is nice and important to reduce the execution time while learning, Tensorflow will prepare the next one batch in advance (i.e. putting it in a buffer) while the current batch is being consumed by the model:



## Doing real manipulations in TensorFlow:

### Windowing the Data :

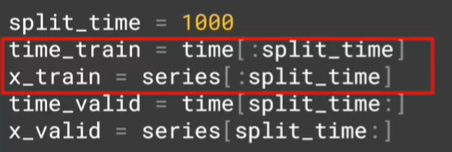


1. Convert our series Data to tf.data.Dataset instance using the method from\_tensor\_slices
2. Applying the windowing process with drop\_reminder=True to ensure all the data has the same length
3. We call after that flat\_map method to flat the data and make it easier to be manipulated , the argument is to specify the size of each chunk ; and it’s clearly equal to window\_size+1 because it’s the size of a row
4. Then , we call the shuffle method to avoid the sequence bias ( which helds to overfitting ) , The shuffle buffer argument is a numerical argument ( 1000 for example ) to tell tensorflow to grab randomly 1000 data from the series to RAM and then doing the random selelction from them , instead of loading the whole Data to Ram to do the RAM selection to have a shorter time execution and lower the RAM consumption
5. And finally we batch our data ( batch\_size means the number of rows to be passed in the same iteration inside our Neural network ) , we use the prefetch method to prepare the next batch in advance to accelerate the execution

### Training with Linear regression ( Single layer neural network )

* In TensorFlow, a Single Dense layer with a one unit is the equivalent for linear regression:

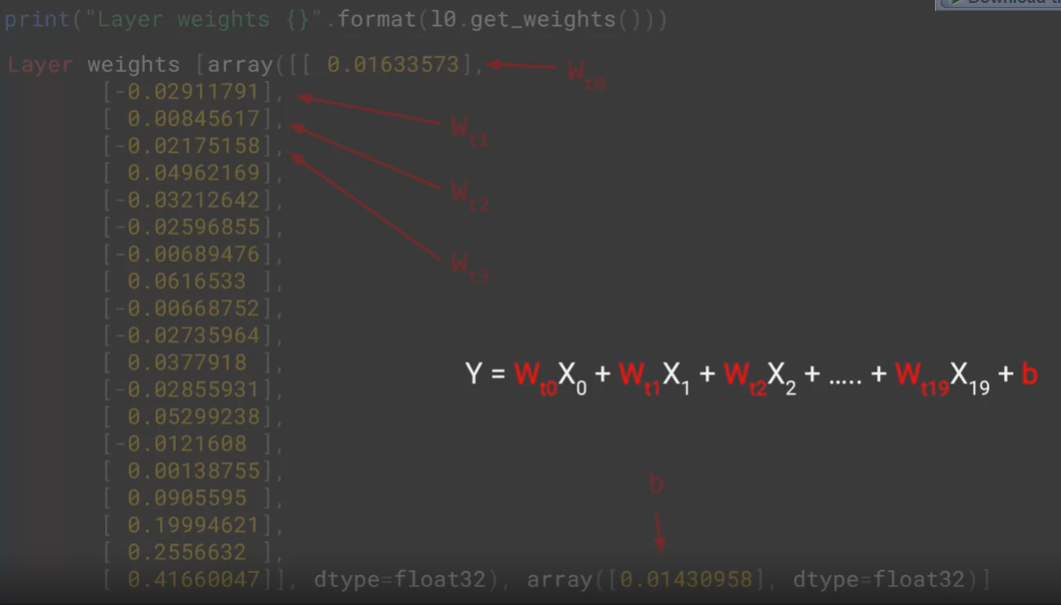
#### Splitting the data ( series and its time into training and testing ) :



#### Training with Single layer network : the linear regression

* We use the MSE loss , with SGD as optimizer with modified parameters for learning\_rate and momentum
  + And the momentum is: “Momentum [1] or SGD with momentum is method which helps accelerate gradients vectors in the right directions, thus leading to faster converging. It is one of the most popular optimization algorithms and many state-of-the-art models are trained using it”
* The input\_size must be equal to the window\_size so our model will expect an array of 20 values ( the series of the past ) to generate the 21th value ( the predicted value )

#### Analyzing the result of the linear regression after the fitting:



* Since our window\_size=20 , which represents the number of parameters that are gonna used to do the prediction of the coming value : the weights of our single layer represents the weights W1,W2,…..,W20 associated to X1,X2,….,X20 to calculate the prediction X21
* The second array with a single value represents the bias b
* And these parameters together represent the generated formula from the linear regression: **X21=W1X1+W2X2+…. +W20X20 + b**

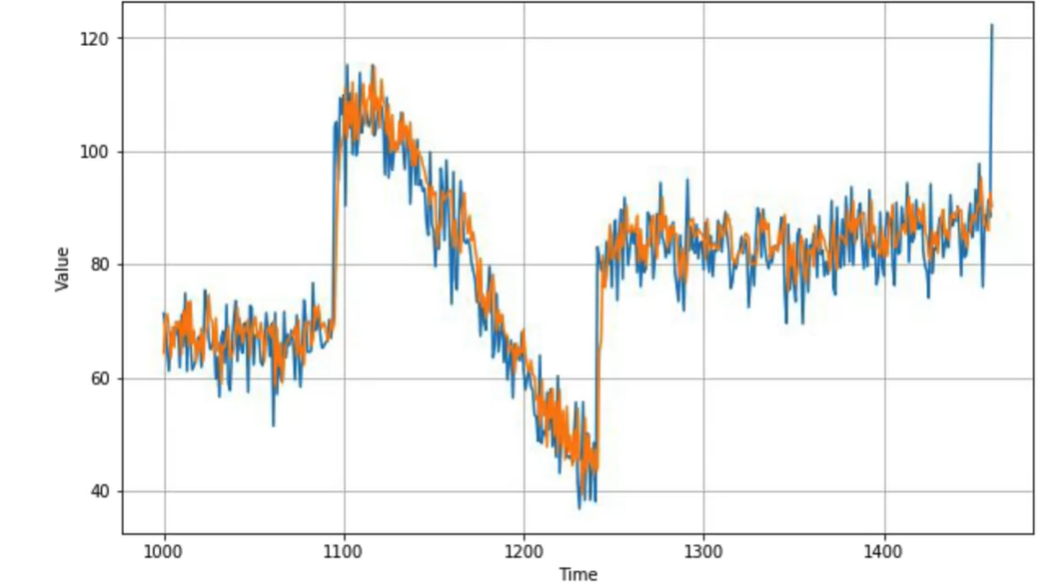
#### Calling the. predict () method:

* As expected the output of the predict method is an array with a single value which represents the predicted value

#### Storing the predicted values to plot them :

* We initialize an empty array
* We iterate through the data and and we passed the last 20 values ( time:time+window\_size ) to the poredicted method and we store the predicted value into the forecast array
* Finally, we slice the forecast array to get only the predicted value generated in the validation time only

#### Plotting the predicted values in the validation time beside the real data:



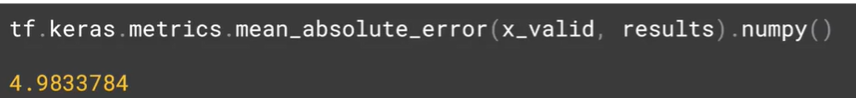
#### Calculating the error :

* We got an interesting (little) error in our machine learning with only single layer ! which is almost like the error we got from the statistical methods ( what we used to do in the first week ) : 4.92 means that in the average the difference between the prediction and the real value is 4.92

### Training with Deep layer network :

* There isn’t a big difference between training with single layer and multiple layers , we just added two layers of dense layer with “relu” as activation function

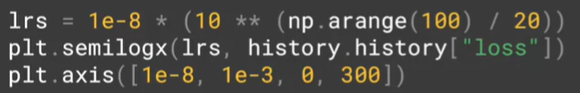
#### The resulted error :

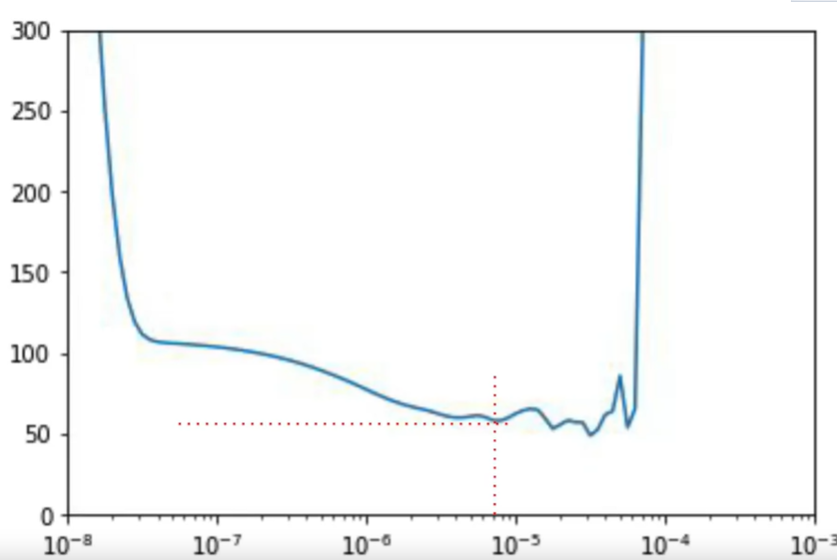


#### Change the learning rate of the optimizer based on the number of epochs:

* We can do that using LearningRateScheduler from the callbacks of keras :
  + In this example we specified the learning rate by :
    - Lr(epoch)= 10^-8 \* 10^(epoch/20)
  + So for the first epoch learning rate = 10^-8 \* 10^1/20 = 10^(-8+(1/20))
  + For the lase epoch : 10^-8 – 10^5 = 10^-3
    - This will help us to determine the best learning rate which minimizes the loss
* The role of this fitting is not training our model but to test the training with different learning rates in order to choose the optimal one later on

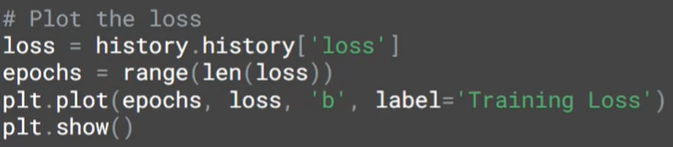
#### (Important ) plotting the loss per epoch to choose the best learning rate :

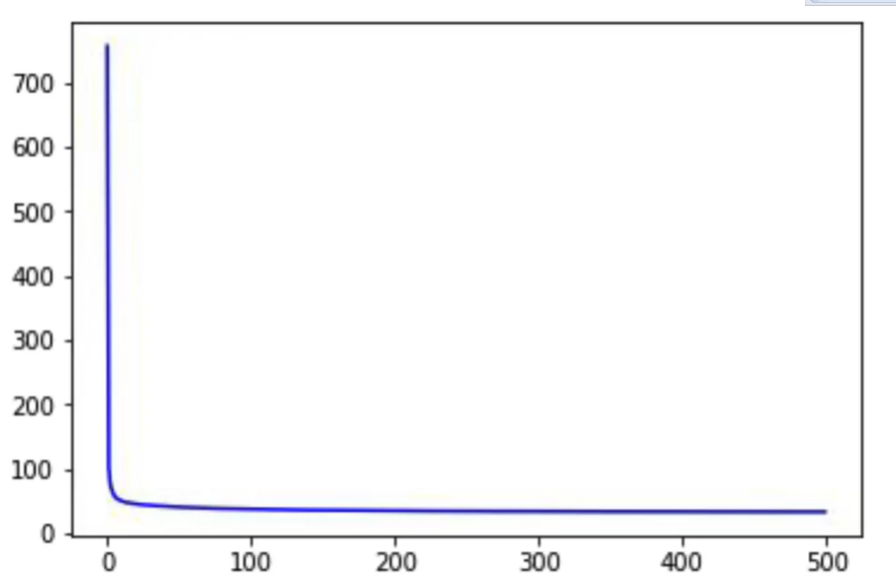
* Since each epoch has its own learning rate , it would be interesting to plot the loss for each learning rate in order to choose the best learning rate

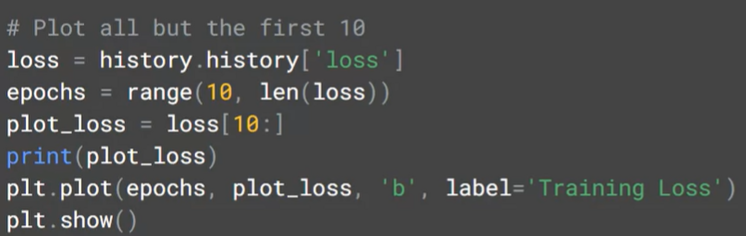


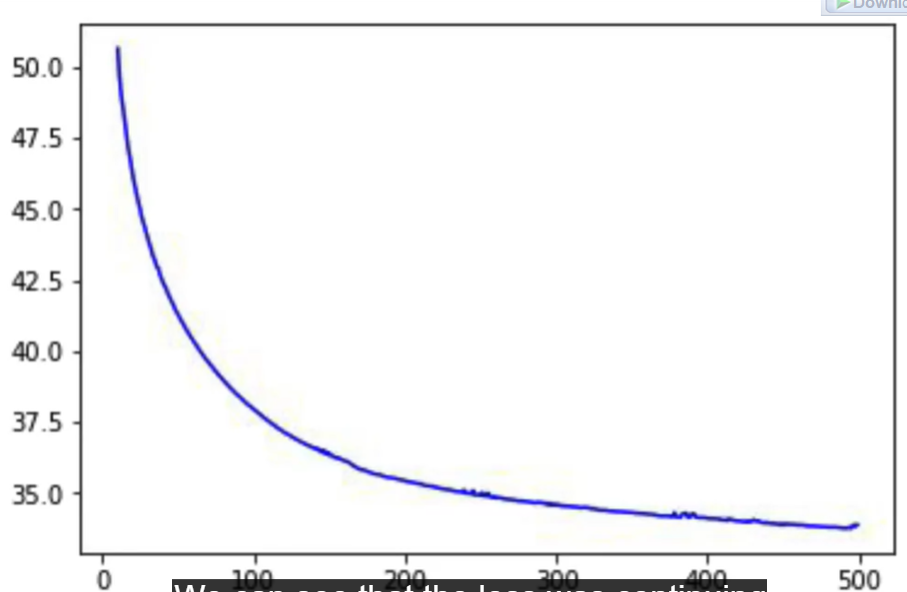
* From the chart we can see that the minimal loss come when learning rate was 7\*10^-6 (7e-6)

#### Plotting the loss after specifying the optimal learning rate :



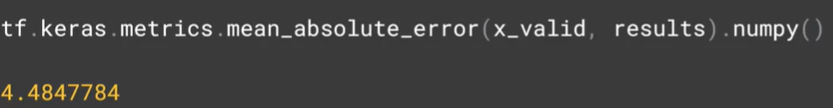


* It seems that the loss decreased very well in the first epochs and then the next epochs were useless , but it’s not true , because if we slice the first 10 epochs :
* That’s we’ve got:



* We see that the loss keeps decreasing that means that increasing the number of epochs helps on the converge of the model

And that’s we got as a MEA after choosing the best leaning rate:



* This is so far the best MEA we’ve got !