[**https://www.thepythoncode.com/article/stock-price-prediction-in-python-using-tensorflow-2-and-keras**](https://www.thepythoncode.com/article/stock-price-prediction-in-python-using-tensorflow-2-and-keras)

**Preparing the Dataset**

As a first step, we need to write a function that downloads the dataset from the Internet and preprocess it:

def load\_data(ticker, n\_steps=50, scale=True, shuffle=True, lookup\_step=1, split\_by\_date=True,

test\_size=0.2, feature\_columns=['adjclose', 'volume', 'open', 'high', 'low']):

This function is long but handy, and it accepts several arguments to be as flexible as possible:

* The ticker argument is the ticker we want to load. For instance, you can use [TSLA](https://finance.yahoo.com/quote/TSLA/) for the Tesla stock market, [AAPL](https://finance.yahoo.com/quote/AAPL) for Apple, and so on. It can also be a pandas Dataframe with the condition it includes the columns in feature\_columns and date as an index.
* n\_steps integer indicates the historical sequence length we want to use; some people call it the window size, recall that we are going to use a recurrent neural network, we need to feed into the network a sequence data, choosing 50 means that we will use 50 days of stock prices to predict the next lookup time step.
* scale is a boolean variable that indicates whether to scale prices from 0 to 1; we will set this to True as scaling high values from 0 to 1 will help the neural network to learn much faster and more effectively.
* lookup\_step is the future lookup step to predict, the default is set to 1 (e.g., next day). 15 means the next 15 days, and so on.
* split\_by\_date is a boolean that indicates whether we split our training and testing sets by date. Setting it to False means we randomly split the data into training and testing using sklearn's train\_test\_split() function. If it's True (the default), we split the data in date order.

We will use all the features available in this dataset: open, high, low, volume, and adjusted close. Please check [this tutorial](https://www.thepythoncode.com/article/introduction-to-finance-and-technical-indicators-with-python) to learn more about what these indicators are.

The above function does the following:

* First, it loads the dataset using stock\_info.get\_data() function in yahoo\_fin module.
* It adds the "date" column from the index if it doesn't exist, this will help us later to get the features of the testing set.
* If the scale argument is passed as True, it will scale all the prices from 0 to 1 (including the volume) using sklearn's MinMaxScaler class. Note that each column has its own scaler.
* It then adds the future column, which indicates the target values (the labels to predict, or the y's) by shifting the adjusted close column by lookup\_step.
* After that, it shuffles and splits the data into training and testing sets and finally returns the result.

def create\_model(sequence\_length, n\_features, units=256, cell=LSTM, n\_layers=2, dropout=0.3,

loss="mean\_absolute\_error", optimizer="rmsprop", bidirectional=False):

Again, this function is flexible too, and you can change the number of layers, dropout rate, the RNN cell, loss, and [the optimizer](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers) used to compile the model.

The above function constructs an RNN with a dense layer as an output layer with one neuron. This model requires a sequence of features of sequence\_length (in this case, we will pass 50 or 100) consecutive time steps (which are days in this dataset) and outputs a single value which indicates the price of the next time step.

It also accepts n\_features as an argument, which is the number of features we will pass on each sequence, in our case, we'll pass adjclose, open, high, low and volume columns (i.e 5 features).

You can tweak the default parameters as you wish, n\_layers is the number of RNN layers you want to stack, dropout is the dropout rate after each RNN layer, units are the number of RNN cell units (whether it is [LSTM](https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM), [SimpleRNN](https://www.tensorflow.org/api_docs/python/tf/keras/layers/SimpleRNN), or [GRU](https://www.tensorflow.org/api_docs/python/tf/keras/layers/GRU)), bidirectional is a boolean that indicates whether to use [bidirectional RNNs](https://en.wikipedia.org/wiki/Bidirectional_recurrent_neural_networks), experiment with those!

So the above code is all about defining all the hyperparameters we gonna use; we explained some of them while we didn't explain the others:

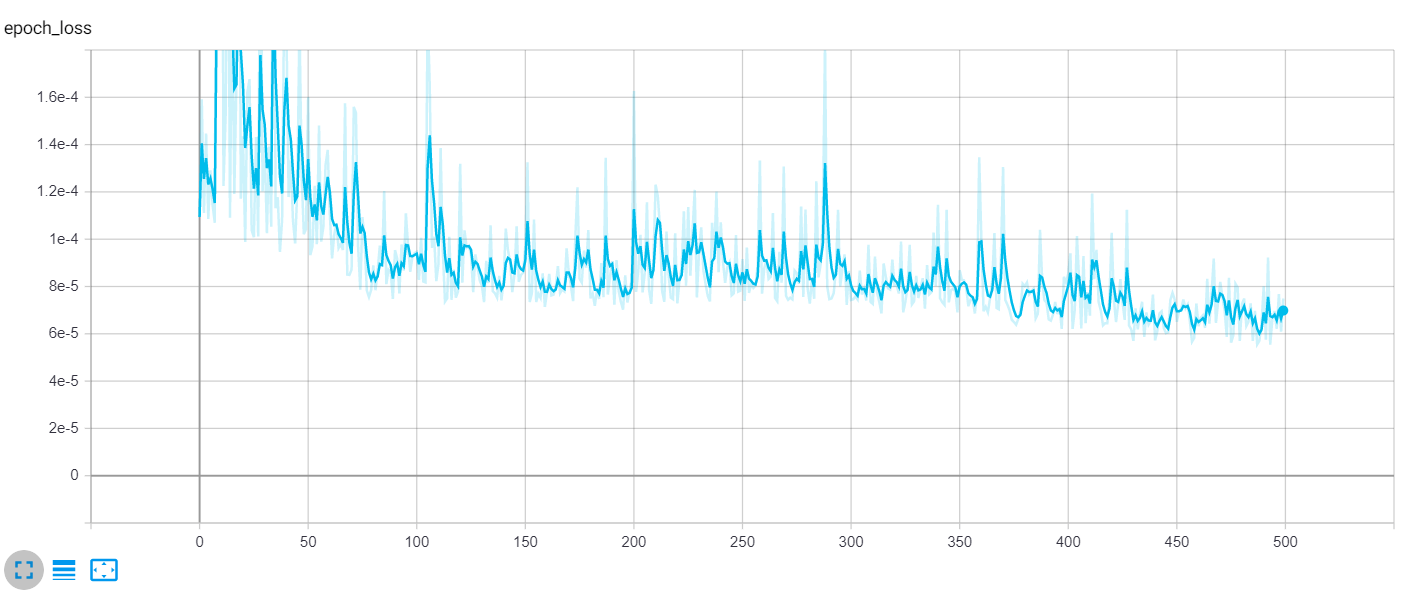
* TEST\_SIZE: The testing set rate. For instance, 0.2 means 20% of the total dataset.
* FEATURE\_COLUMNS: The features we gonna use to predict the next price value.
* N\_LAYERS: Number of RNN layers to use.
* CELL: RNN cell to use, default is LSTM.
* UNITS: Number of cell units.
* DROPOUT: The [dropout](https://www.thepythoncode.com/article/dropout-regularization-in-pytorch) rate is the probability of not training a given node in a layer, where 0.0 means no dropout at all. This regularization can help the model not overfit our training data. Check [this tutorial](https://www.thepythoncode.com/article/dropout-regularization-in-pytorch) for more information about dropout regularization.
* BIDIRECTIONAL: Whether to use [bidirectional recurrent neural networks](https://en.wikipedia.org/wiki/Bidirectional_recurrent_neural_networks).
* LOSS: Loss function to use for this regression problem, we're using [Huber loss](https://www.tensorflow.org/api_docs/python/tf/keras/losses/Huber), you can use mean absolute error (mae) or mean squared error (mse) as well.
* OPTIMIZER: Optimization algorithm to use, defaulting to [Adam](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam).
* BATCH\_SIZE: The number of data samples to use on each training iteration.
* EPOCHS: The number of times the learning algorithm will pass through the entire training dataset, we used 500 here, but try to increase it further.

We used [ModelCheckpoint](https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint?version=stable), which saves our model in each epoch during the training. We also used [TensorBoard](https://www.tensorflow.org/tensorboard) to visualize the model performance in the training process.

After the training ends (or during the training), try to run tensorboard using this command:

tensorboard --logdir="logs"

Now, this will start a local HTTP server at localhost:6006; after going to the browser, you'll see something similar to this:



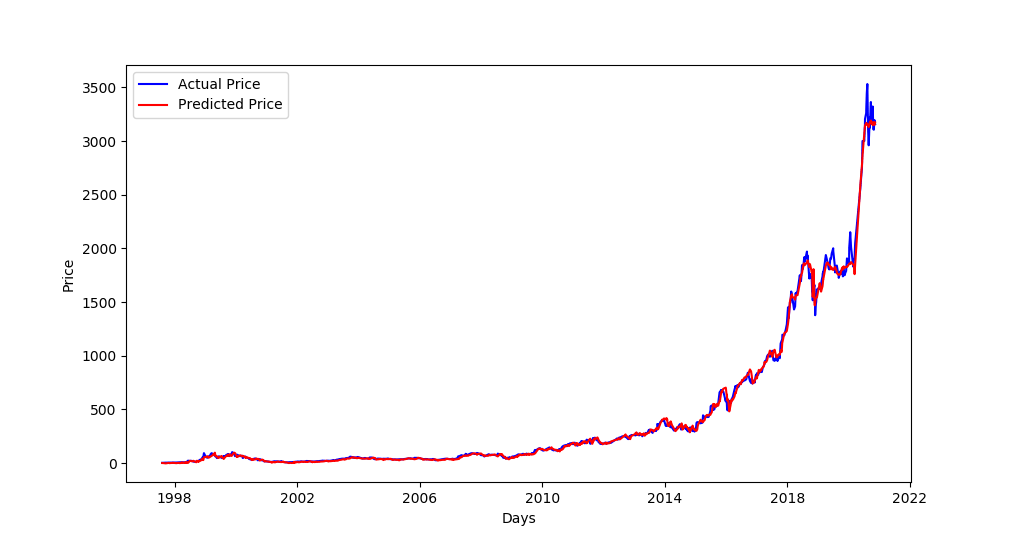
The loss is [Huber loss](https://www.tensorflow.org/api_docs/python/tf/keras/losses/Huber) as specified in the LOSS parameter (you can always change it to [mean absolute error](https://en.wikipedia.org/wiki/Mean_absolute_error) or [mean squared error](https://en.wikipedia.org/wiki/Mean_squared_error)), the curve is the validation loss. As you can see, it is significantly decreasing over time. You can also increase the number of epochs to get much better results.

Now that we've trained our model, let's evaluate it and see how it's doing on the testing set. The below function takes a pandas Dataframe and plots the true and predicted prices in the same plot using matplotlib. We'll use it later:

Great, the model says after 15 days that the price of AMZN will be 3232.24$, that's interesting!

Below is the meaning of the main metrics:

* **Mean absolute error**: we get about 20 as error, which means, on average, the model predictions are far by over 20$ to the true prices; this will vary from ticker to another, as prices get larger, the error will increase as well. As a result, you should only compare your models using this metric when the ticker is stable (e.g., [AMZN](https://finance.yahoo.com/quote/AMZN/)).
* **Buy/Sell profit**: This is the profit we get if we opened trades on all the testing samples, we calculated these on get\_final\_df() function.
* **Total profit**: This is simply the sum of buy and sell profits.
* **Profit per trade**: The total profit divided by the total number of testing samples.
* **Accuracy score**: This is the score of how accurate our predictions are. This calculation is based on the positive profits from all the trades from the testing samples.



* Excellent, as you can see, the blue curve is the actual test set, and the red curve is the predicted prices! Notice that the stock price has recently been increasing, as we predicted.
* Since we set SPLIT\_BY\_DATE to False, this plot shows the prices of the testing set spread on our whole dataset along with corresponding predicted prices (which explains the testing set starts before 1998).
* If we set SPLIT\_BY\_DATE to True, then the testing set will be the last TEST\_SIZE percentage of the total dataset (For instance, if we have data from 1997 to 2020, and TEST\_SIZE is 0.2, then testing samples will range from about 2016 to 2020).
* Finally, let's print the last ten rows of our final dataframe, so you can see what it looks like:

The dataframe has the following columns:

* Our testing set features (open, high, low, close, adjclose, and volume columns).
* adjclose\_15: is the predicted adjclose price after 15 days (since LOOKUP\_STEP is set to 15) using our trained model.
* true\_adjclose\_15: is the true adjclose price after 15 days; we get that by shifting our testing dataset.
* buy\_profit: This is the profit we get if we bought the stock at that date. A negative profit means we made a loss (it should be a sell trade, and we made a buy).
* sell\_profit: This is the profit we get if we sell the stock at that date.

**Conclusion**

Alright, that's it for this tutorial. You can tweak the parameters and see how you can improve the model performance, try to train on more epochs, say 700 or even more, increase or decrease the BATCH\_SIZE and see if it does change for the better, or play around with N\_STEPS and LOOKUP\_STEPS and see which combination works best.

You can also change the model parameters by increasing the number of layers or LSTM units or even trying the GRU cell instead of LSTM.

Note that there are other features and indicators to use, to improve the prediction, it is often known to use some other information like features, such as [technical indicators](https://www.thepythoncode.com/article/introduction-to-finance-and-technical-indicators-with-python), the company product innovation, interest rate, exchange rate, public policy, the web, and financial news and even the number of employees!

I encourage you to change the model architecture, try to use [CNNs](https://en.wikipedia.org/wiki/Convolutional_neural_network) or [Seq2Seq](https://towardsdatascience.com/understanding-encoder-decoder-sequence-to-sequence-model-679e04af4346) models, or even add bidirectional LSTMs to this existing model (setting BIDIRECTIONAL to True), see if you can improve it!