Predicting the price of Bitcoin with multivariate Pytorch LSTMs

Using multivariate, multi-output forecasting models for financial data

Charlie O'Neill · Follow
14 min read · Jan 26, 2022

In a previous post, I went into detail about constructing an LSTM for univariate time-series data. This itself is not a trivial task; you need to understand the form of the data, the shape of the inputs that we feed to the LSTM, and how to recurse over training inputs to produce an appropriate output. This knowledge is fantastic for analysing curves where the only information we have is the past values of the curve itself. For instance, we can use univariate LSTMs to learn trigonometric curves, and plot them well outside the domain of training values.

Whilst it is all well and good to take one array of data, representing a sequence of values of a function over time, and attempt to predict future values, this often doesn't match the quantity of data available in the real world. More likely, the data we receive will be in tabular form, with each row representing a single time step. Each row probably consists of several different columns; these are the features, or *predictors*, that we have information about at each time step. One of these columns is not a feature, however. It is the **target** we want to predict. Consequently, we need a way to feed in these multiple values at each time step to our LSTM, and to produce a singular output representing the prediction at the next time step in return. In this way, we construct a **multivariate** LSTM.

Introduction: predicting the price of Bitcoin

For this problem, we're going to focus on financial data. Before we begin, I would like to point out that LSTMs will not make you rich, even if they are excellent forecasters for time-series data. **No** model will make you rich; there's a whole field of thinking on epistemic humility and how impossible it is for anything you do to detect any edge in the market; I won't go into it here. But know this: if you've found an inconsistency in the price of a stock (it’s too low, or too high and you want to capitalise on that), and you believe that no-one else has spotted this inconsistency, then you might want to rethink your surety. Recall that “no-one else” in this scenario includes the team of PhDs sitting in Goldman Sachs with a supercomputing cluster, the lowest latency in information and order processing in the market, and multi-billion dollar financial incentives to find exactly the same inconsistencies as you. If you really believe you’ve beaten them, you haven’t. For more detail on this line of thinking, see Eliezer Yudkowsky’s *Inadequate Equilibria*.

We’ll frame our problem as follows. We have historical price data for Bitcoin, which includes the following predictors for each day (where we have daily time steps):

- Opening price
- High price
- Low price
- Volume traded

Our goal is to take some sequence of the above four values (say, for 100 previous days), and predict the target variable (Bitcoin’s price) for the next 50 days into the future.

Preprocessing the data

We begin by importing the data and quickly cleaning it. Fortunately, financial data is readily available online. We will use Yahoo historical prices for Bitcoin, available back to September 17, 2014. This data is available here. Import the data using Pandas and have a look.

> *Note: you can download historical data from Yahoo for almost any financial instrument. Interested in Tesla shares? Recreate this LSTM to see if you should buy or sell in the coming month.*

```python
df = pd.read_csv('BTC-USD.csv', index_col = 'Date', parse_dates=True)
df.drop(columns=['Adj Close'], inplace=True)
df.head(5)
```

import.py hosted with ❤ by GitHub view raw

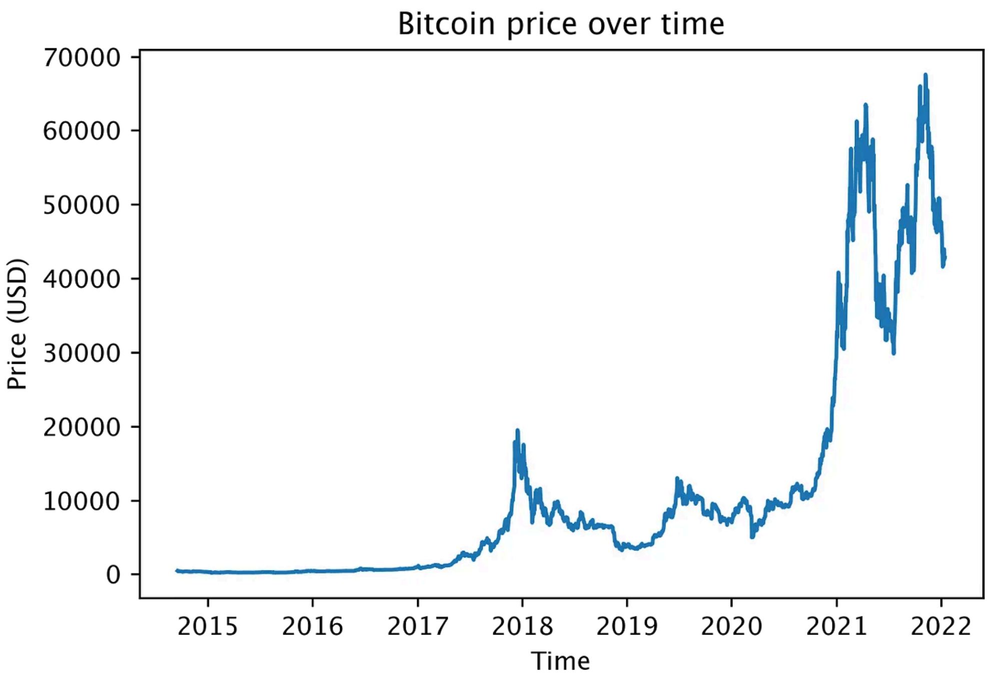
Date	Open	High	Low	Close	Volume
2014-09-17	465.864014	468.174011	452.421997	457.334015	21056800
2014-09-18	456.859985	456.859985	413.104004	424.440002	34483200
2014-09-19	424.102997	427.834991	384.532013	394.795990	37919700
2014-09-20	394.673004	423.295990	389.882996	408.903992	36863600
2014-09-21	408.084991	412.425995	393.181000	398.821014	26580100

The data structure for our Bitcoin prices dataframe. (Image by author)

At the bare minimum, your exploratory data analysis should consist of plotting the target variable of interest. (Some people will argue that you should do a lot more than this, such as regressing the target variable on the predictors, and looking for linear relationships between the variables.) Let's plot the Bitcoin price over time to see what we're actually trying to predict.

```
plt.plot(df.Close)
plt.xlabel("Time")
plt.ylabel("Price (USD)")
plt.title("Bitcoin price over time")
plt.savefig("initial_plot.png", dpi=250)
plt.show();
```

plt.py hosted with ❤️ by GitHub view raw



(Image by author)

Interesting. We want a realistic emulation of what would happen in the real-world. That is, we want a few years of historical data, and train an LSTM to predict what will happen to the price of Bitcoin in the next few months.

Setting inputs and outputs

Recall that our predictors will consist of all the columns except our target, closing price. Note that we want to use an `sklearn` preprocessor below, which requires reshaping the array if it consists of a single feature, as our target does. Hence, for the target `y`, we have to call `values`, which removes the axes labels and will allow us to reshape the array.

```python
X, y = df.drop(columns=['Close']), df.Close.values
X.shape, y.shape

>>> ((2677, 4), (2677,))
```

shape.py hosted with ❤ by GitHub view raw

We now have the task of standardising our features. We'll use standardisation for our training features `X` by removing the mean and scaling to unit variance. Standardisation helps the deep learning model to learn by ensuring that parameters can exist in the same multi-dimensional space; it wouldn't make much sense to have the weights have to change their size simply because all the variables have different scales. For our target `y`, we will scale and translate each feature individually to between 0 and 1. This transformation is often used as an alternative to zero mean, unit variance scaling.

```python
from sklearn.preprocessing import StandardScaler, MinMaxScaler
mm = MinMaxScaler()
ss = StandardScaler()

X_trans = ss.fit_transform(X)
y_trans = mm.fit_transform(y.reshape(-1, 1))
```

scaling.py hosted with ❤ by GitHub view raw

Now comes the slightly fiddly part. We want to feed in 100 samples, up to the current day, and predict the next 50 time step values. To do this, we need a special function to ensure that the corresponding indices of `X` and `y` represent this structure. Examine this function carefully, but essentially it just boils down to getting 100 samples from `X`, then looking at the 50 next indices in `y`, and patching these together. Note that because of this we'll throw out the first 50 values of `y`.

```python
# split a multivariate sequence past, future samples (X and y)
def split_sequences(input_sequences, output_sequence, n_steps_in, n_steps_out):
    X, y = list(), list() # instantiate X and y
    for i in range(len(input_sequences)):
        # find the end of the input, output sequence
        end_ix = i + n_steps_in
        out_end_ix = end_ix + n_steps_out - 1
        # check if we are beyond the dataset
        if out_end_ix > len(input_sequences): break
        # gather input and output of the pattern
        seq_x, seq_y = input_sequences[i:end_ix], output_sequence[end_ix-1:out_end_ix, -
        X.append(seq_x), y.append(seq_y)
    return np.array(X), np.array(y)

X_ss, y_mm = split_sequences(X_trans, y_trans, 100, 50)
print(X_ss.shape, y_mm.shape)

>>> (2529, 100, 4) (2529, 50)
```

split_sequence.py hosted with ❤️ by GitHub view raw

Let's check that the first sample in `y_mm` indeed starts at the 100th sample in the original target `y` vector.

```python
assert y_mm[0].all() == y_trans[99:149].squeeze(1).all()

y_mm[0]

>>> array([0.00209093, 0.00222324, 0.00204426, 0.00206468, 0.00199688,
       0.00196819, 0.00210851, 0.00202031, 0.00203193, 0.00152813,
       0.00127754, 0.00143008, 0.00160392, 0.00172483, 0.00156177,
       0.00166652, 0.00143485, 0.00129928, 0.00133098, 0.00070869,
       0.        , 0.00047101, 0.00044509, 0.00031395, 0.00047836,
       0.00054546, 0.00049284, 0.00072407, 0.00082066, 0.00081284,
       0.00103495, 0.00112207, 0.00141522, 0.00126686, 0.00082821,
       0.00082224, 0.00071706, 0.00058409, 0.00072518, 0.00089223,
       0.00072957, 0.00072341, 0.00057885, 0.00065535, 0.00073678,
       0.00067235, 0.00062335, 0.00061933, 0.00060963, 0.0006479 ])

y_trans[99:149].squeeze(1)

>>> array([0.00209093, 0.00222324, 0.00204426, 0.00206468, 0.00199688,
       0.00196819, 0.00210851, 0.00202031, 0.00203193, 0.00152813,
       0.00127754, 0.00143008, 0.00160392, 0.00172483, 0.00156177,
       0.00166652, 0.00143485, 0.00129928, 0.00133098, 0.00070869,
       0.        , 0.00047101, 0.00044509, 0.00031395, 0.00047836,
       0.00054546, 0.00049284, 0.00072407, 0.00082066, 0.00081284,
       0.00103495, 0.00112207, 0.00141522, 0.00126686, 0.00082821,
       0.00082224, 0.00071706, 0.00058409, 0.00072518, 0.00089223,
       0.00072957, 0.00072341, 0.00057885, 0.00065535, 0.00073678,
       0.00067235, 0.00062335, 0.00061933, 0.00060963, 0.0006479 ])
```

check_y.py hosted with ❤️ by GitHub view raw

Above, we mentioned that we wanted to predict the data a several months into the future. Thus, we'll use a training data size of 95%, with 5% left for the remaining data that we're going to predict. This gives us a training set size of 2763 days, or about seven and a half years. We will predict 145 days into the future, which is almost 5 months.

```python
total_samples = len(X)
train_test_cutoff = round(0.90 * total_samples)

X_train = X_ss[:-150]
X_test = X_ss[-150:]

y_train = y_mm[:-150]
y_test = y_mm[-150:]

print("Training Shape:", X_train.shape, y_train.shape)
print("Testing Shape:", X_test.shape, y_test.shape)

>>> Training Shape: (2379, 100, 4) (2379, 50)
Testing Shape: (150, 100, 4) (150, 50)
```

We need to now, as usual, convert our data into tensors. This is fairly easy — we do so by calling `torch.tensor()` on our object, and setting the property `requires_grad=True`. Some old Pytorch tutorials might have you believe that we need to apply the wrapper `Variable` here. However, this is deprecated, and now the input tensor to be forward propagated has to be can facilitate automatic back propagation (through `backward()`) without being wrapped in a variable.

```python
# convert to pytorch tensors
X_train_tensors = Variable(torch.Tensor(X_train))
X_test_tensors = Variable(torch.Tensor(X_test))

y_train_tensors = Variable(torch.Tensor(y_train))
y_test_tensors = Variable(torch.Tensor(y_test))
```

If we look at the documentation for the multi-layer `torch.nn.LSTM`, we see that the input shape depends on whether the parameter `batch_first` is true. Since we are accustomed to having the first dimension of our data be the batch, we will set `batch_first` to true. The size of the input is then (*N*,*L*,*H_in*), where *N* is the batch size, *L* is the sequence length, and *H_in* is the input size (i.e. the number of features). In other words, we want the dimensions to be the rows of the dataframe in the first dimension, followed by the length of the dataframe in the next dimension (representing the length of the input sequence), and finally the features (of which we have four) in the final dimension.

To reshape the tensors into our required shape, we use `torch.reshape`. This takes as arguments the tensors we are reshaping, and then a tuple of the shape we need to reshape to. For the rows of the dataframe, we can simply look at the shape of the first dimension of `X_train_tensors.shape[0]`. Since we are applying an LSTM, we recall that the sequence length we feed in is simply 1. (Remember that the whole point of the LSTM is not feeding in tonnes of data at each point, because the memory is handled by the inner workings of the LSTM cell. For more on this, see my previous article.) Finally, we want the last dimension to be the number of features, which is stored in `X_train_tensors.shape[1]`.

```python
# reshaping to rows, timestamps, features
X_train_tensors_final = torch.reshape(X_train_tensors,
                                      (X_train_tensors.shape[0], 100,
                                       X_train_tensors.shape[2]))
X_test_tensors_final = torch.reshape(X_test_tensors,
                                     (X_test_tensors.shape[0], 100,
                                      X_test_tensors.shape[2]))

print("Training Shape:", X_train_tensors_final.shape, y_train_tensors.shape)
print("Testing Shape:", X_test_tensors_final.shape, y_test_tensors.shape)

>>> Training Shape: torch.Size([2379, 100, 4]) torch.Size([2379, 50])
Testing Shape: torch.Size([150, 100, 4]) torch.Size([150, 50])
```

tensor_shape.py hosted with ❤ by GitHub view raw

One more thing we want to check: the data logic of the test set. Sequential data is hard to get your head around, especially when it comes to generating a test-set for multi-step output models. Here, we want to take the 100 previous predictors up to the current time-step, and predict 50 time-steps into the future. In the test set, we have 150 batch feature samples, each consisting of 100 time-steps and four feature predictors. In the targets for the test set, we again have 150 batch samples, each consisting of an array of length 50 of scalar outputs.

Since we want a way to validate our results, we need to predict the Bitcoin price for 50 time steps in the test set for which we have the data (i.e. the test targets). Because of the way we wrote `split_sequence()` above, we simply need the last sample of 100 days in `X_test`, run the model on it, and compare these predictions with the last sample of 50 days of `y_test`. These correspond to a period of 100 days in `X_test`'s last sample, proceeded immediately by the next 50 days in the last sample of `y_test`.

We want to check that the 50 values we will be predicting match the last 50 values of `y` in the test set.

```python
X_check, y_check = split_sequences(X, y.reshape(-1, 1), 100, 50)
X_check[-1][0:4]

>>>  array([[4.47418828e+04, 4.69707617e+04, 4.39983164e+04, 3.72043123e+10],
       [4.67231211e+04, 4.93421523e+04, 4.66507070e+04, 3.47068675e+10],
       [4.93270742e+04, 4.97170195e+04, 4.83121992e+04, 4.05852053e+10],
       [4.88691055e+04, 4.94716094e+04, 4.81999414e+04, 2.53709754e+10]])

X.iloc[-149:-145]
```

check_2.py hosted with ❤ by GitHub view raw

Perfect, the first four rows of our data are as they should be. Note that `X_check[-1]` should be identical to `X.iloc[-149:-49]`, ending 50 days before the end of our dataset. So we are taking the 100 time-steps of information, up to the 26th of November 2021, and attempting to predict the 50 days after that, up to the 14th January 2022. For one final check, we make sure that the final batch sample in our test targets matches these dates for prediction.

```python
y_check[-1]

>>> array([53569.765625, 54815.078125, 57248.457031, 57806.566406,
       57005.425781, 57229.828125, 56477.816406, 53598.246094,
       49200.703125, 49368.847656, 50582.625   , 50700.085938,
       50504.796875, 47672.121094, 47243.304688, 49362.507813,
       50098.335938, 46737.480469, 46612.632813, 48896.722656,
       47665.425781, 46202.144531, 46848.777344, 46707.015625,
       46880.277344, 48936.613281, 48628.511719, 50784.539063,
       50822.195313, 50429.859375, 50809.515625, 50640.417969,
       47588.855469, 46444.710938, 47178.125   , 46306.445313,
       47686.8125  , 47345.21875 , 46458.117188, 45897.574219,
       43569.003906, 43160.929688, 41557.902344, 41733.941406,
       41911.601563, 41821.261719, 42735.855469, 43949.101563,
       42591.570313, 42801.679688])

df.Close.values[-50:]

>>> array([53569.765625, 54815.078125, 57248.457031, 57806.566406,
       57005.425781, 57229.828125, 56477.816406, 53598.246094,
       49200.703125, 49368.847656, 50582.625   , 50700.085938,
       50504.796875, 47672.121094, 47243.304688, 49362.507813,
       50098.335938, 46737.480469, 46612.632813, 48896.722656,
       47665.425781, 46202.144531, 46848.777344, 46707.015625,
       46880.277344, 48936.613281, 48628.511719, 50784.539063,
       50822.195313, 50429.859375, 50809.515625, 50640.417969,
       47588.855469, 46444.710938, 47178.125   , 46306.445313,
       47686.8125  , 47345.21875 , 46458.117188, 45897.574219,
       43569.003906, 43160.929688, 41557.902344, 41733.941406,
       41911.601563, 41821.261719, 42735.855469, 43949.101563,
       42591.570313, 42801.679688])
```

y_check.py hosted with ❤ by GitHub view raw

You can remove the `.values` from the code above to check for yourself that the dates match.

That was a lot of indexing and checking. To summarise, the main performance test for our model will be on the last batch sample in the test set. This will consist of predictors from the 100 time-steps up to the 26th November 2021, and this information will be used by our model to predict the next 50 days of Bitcoin prices, up to the 14th January 2022. In this way, we will validate model performance by comparing predictions to the actual prices in that 50 day window.

LSTM model

Now we need to construct the LSTM class, inheriting from `nn.Module`. In contrast to our previous univariate LSTM, we're going to build the model with the `nn.LSTM` rather than `nn.LSTMCell`. This is for two reasons: firstly, it's nice to be exposed to both so that we have the option. Secondly, we don't need the flexibility that `nn.LSTMCell` provides. We know that `nn.LSTM` is essentially just a recurrent application of `nn.LSTMCell`. Thus, we would only use `nn.LSTMCell` if we wanted to apply other transformation in between different LSTM layers, such as batch-normalisation and dropout. Here however, we can implement dropout automatically using the `dropout` parameter in `nn.LSTM`. We've already standardised our data. Thus, there's not a whole lot of reasons to use the more fiddly `nn.LSTMCell`.

As per usual, we'll present the entire model class first, and then break it down line by line.

```python
class LSTM(nn.Module):

    def __init__(self, num_classes, input_size, hidden_size, num_layers):
        super().__init__()
        self.num_classes = num_classes # output size
        self.num_layers = num_layers # number of recurrent layers in the lstm
        self.input_size = input_size # input size
        self.hidden_size = hidden_size # neurons in each lstm layer
        # LSTM model
        self.lstm = nn.LSTM(input_size=input_size, hidden_size=hidden_size,
                            num_layers=num_layers, batch_first=True, dropout=0.2) # lstm
        self.fc_1 =  nn.Linear(hidden_size, 128) # fully connected
        self.fc_2 = nn.Linear(128, num_classes) # fully connected last layer
        self.relu = nn.ReLU()

    def forward(self,x):
        # hidden state
        h_0 = Variable(torch.zeros(self.num_layers, x.size(0), self.hidden_size))
        # cell state
        c_0 = Variable(torch.zeros(self.num_layers, x.size(0), self.hidden_size))
        # propagate input through LSTM
        output, (hn, cn) = self.lstm(x, (h_0, c_0)) # (input, hidden, and internal state
        hn = hn.view(-1, self.hidden_size) # reshaping the data for Dense layer next
        out = self.relu(hn)
        out = self.fc_1(out) # first dense
        out = self.relu(out) # relu
        out = self.fc_2(out) # final output
        return out
```

In our initialisation, as usual we initialisation our parent class, `nn.Module`. Most initialisations in a Pytorch model are separated into two distinct chunks:

1. Any variables that the class will need to reference, for things such as hidden layer size, input size, and number of layers.
2. Defining the layers of the model (without connecting them) using the variables instantiated above.

This is exactly what we do here. If you carefully read over the parameters for the `LSTM` layers, you know that we need to shape the LSTM with input size, hidden size, and number of recurrent layers. For instance, setting `num_layers=2` would mean stacking two LSTMs together to form a stacked LSTM, with the second LSTM taking in outputs of the first LSTM and computing the final results. Thus, we initialise these three variables in the first part of the `__init__`. We also need to force our model to output only one predicted value, so we also initialise a number of classes variable.

In the second part of `__init__`, we set out the layers of our network. Our first layer is obviously a recurrent application of LSTM cells, with all the parameters specified above.

```
self.lstm = nn.LSTM(input_size=input_size, hidden_size=hidden_size,
                    num_layers=num_layers, batch_first=True)
```

Next, we pass this to a fully connected layer, which has an input of `hidden_size` (the size of the output from the last LSTM layer) and outputs 128 activations. Then, we pass these 128 activations to another hidden layer, which evidently accepts 128 inputs, and which we want to output our `num_classes` (which in our case will be 1, see below). Finally, we pass this activation through a non-linear function, in our case ReLU.

```
self.fc_1 =  nn.Linear(hidden_size, 128) # fully connected
self.fc_2 = nn.Linear(128, num_classes) # fully connected last layer
self.relu = nn.ReLU()
```

Training

In our univariate example, we had a complicated loss function, because we said that the loss topology in sequential data is usually characterised by lots of long, flat valleys. Here, we're going to keep things simple, and see what performance we can get with a typical regression loss function like mean-squared error. This also has the added benefit that we don't have to restructure the training loop so as to include the callable `closure` in our parameter update. (If you don't know what I'm talking about, don't worry.)

```
def training_loop(n_epochs, lstm, optimiser, loss_fn, X_train, y_train,
                  X_test, y_test):
    for epoch in range(n_epochs):
        lstm.train()
        outputs = lstm.forward(X_train) # forward pass
        optimiser.zero_grad() # calculate the gradient, manually setting to 0
        # obtain the loss function
        loss = loss_fn(outputs, y_train)
        loss.backward() # calculates the loss of the loss function
        optimiser.step() # improve from loss, i.e backprop
        # test loss
        lstm.eval()
        test_preds = lstm(X_test)
        test_loss = loss_fn(test_preds, y_test)
        if epoch % 100 == 0:
            print("Epoch: %d, train loss: %1.5f, test loss: %1.5f" % (epoch,
                                                                      loss.item(),
                                                                      test_loss.item()))
```

training2.py hosted with ❤ by GitHub view raw

Let's very quickly recap what's going on here. We do the forward pass through our network by passing in our training tensors, which we shaped appropriately before. We then zero out the current gradients stored in the

Pytorch computational graph. We compare these outputs from the forward pass with the actual train targets using our loss function, and backpropagate using `loss.backward()` to calculate the gradients of the loss with respect to the parameters (our weights and biases). We then use this loss to update the parameters.

Setting the appropriate variables below, we can instantiate an instance of our LSTM model.

```python
import warnings
warnings.filterwarnings('ignore')

n_epochs = 1000 # 1000 epochs
learning_rate = 0.001 # 0.001 lr

input_size = 4 # number of features
hidden_size = 2 # number of features in hidden state
num_layers = 1 # number of stacked lstm layers

num_classes = 50 # number of output classes

lstm = LSTM(num_classes,
              input_size,
              hidden_size,
              num_layers)
```

init.py hosted with ❤ by GitHub view raw

We use MSE as our loss function, and the well-known Adam optimiser.

```python
loss_fn = torch.nn.MSELoss()    # mean-squared error for regression
optimiser = torch.optim.Adam(lstm.parameters(), lr=learning_rate)
```

Let's train for 1000 epochs and see what happens. Recall in the previous article that a key part of LSTM debugging is visual cues. Here, our training is fast enough that we can just plot the result at the end, and if it's off, we can change our parameters and run it again.

```
training_loop(n_epochs=n_epochs,
              lstm=lstm,
              optimiser=optimiser,
              loss_fn=loss_fn,
              X_train=X_train_tensors_final,
              y_train=y_train_tensors,
              X_test=X_test_tensors_final,
              y_test=y_test_tensors)

Epoch: 0, train loss: 0.08615, test loss: 0.61831
Epoch: 100, train loss: 0.01013, test loss: 0.08719
Epoch: 200, train loss: 0.00363, test loss: 0.01195
Epoch: 300, train loss: 0.00240, test loss: 0.01414
Epoch: 400, train loss: 0.00203, test loss: 0.01665
Epoch: 500, train loss: 0.00188, test loss: 0.01706
Epoch: 600, train loss: 0.00182, test loss: 0.01743
Epoch: 700, train loss: 0.00179, test loss: 0.01763
Epoch: 800, train loss: 0.00177, test loss: 0.01762
Epoch: 900, train loss: 0.00175, test loss: 0.01749
```

Prediction

A key part of prediction, if the variables have been standardised, is knowing what we need to pass to the model. That is: what do we need to standardise before passing it to the model? Well, we can answer this question easily: we just apply the same transforms to the predictors and targets so that whatever we feed to the model is exactly what the model is used to seeing. It wouldn’t make sense to not standardise our inputs; the model would be confused as to why they no longer have zero mean and unit variance.

Let’s plot our results for the whole dataset.

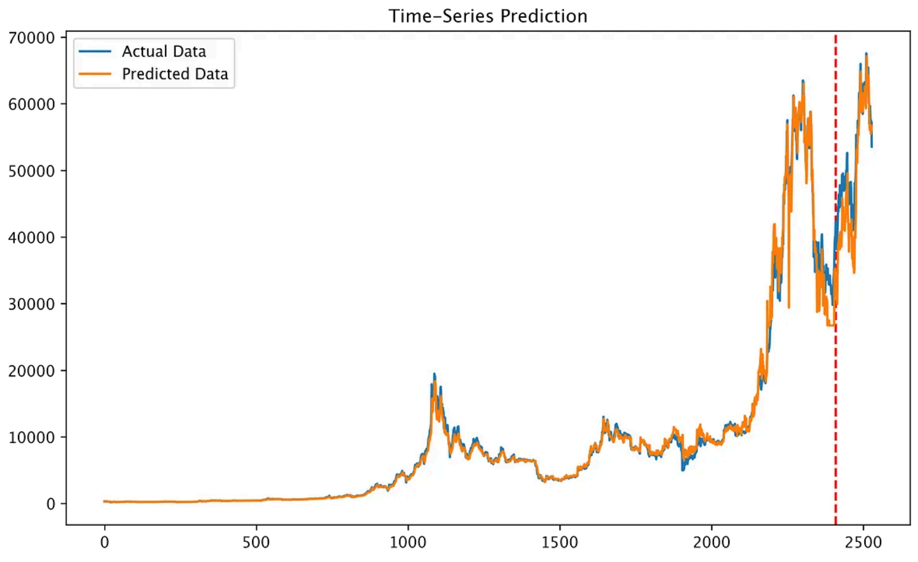
```python
df_X_ss = ss.transform(df.drop(columns=['Close'])) # old transformers
df_y_mm = mm.transform(df.Close.values.reshape(-1, 1)) # old transformers
# split the sequence
df_X_ss, df_y_mm = split_sequences(df_X_ss, df_y_mm, 100, 50)
# converting to tensors
df_X_ss = Variable(torch.Tensor(df_X_ss))
df_y_mm = Variable(torch.Tensor(df_y_mm))
# reshaping the dataset
df_X_ss = torch.reshape(df_X_ss, (df_X_ss.shape[0], 100, df_X_ss.shape[2]))

train_predict = lstm(df_X_ss) # forward pass
data_predict = train_predict.data.numpy() # numpy conversion
dataY_plot = df_y_mm.data.numpy()

data_predict = mm.inverse_transform(data_predict) # reverse transformation
dataY_plot = mm.inverse_transform(dataY_plot)
true, preds = [], []
for i in range(len(dataY_plot)):
    true.append(dataY_plot[i][0])
for i in range(len(data_predict)):
    preds.append(data_predict[i][0])
plt.figure(figsize=(10,6)) #plotting
plt.axvline(x=train_test_cutoff, c='r', linestyle='--') # size of the training set

plt.plot(true, label='Actual Data') # actual plot
plt.plot(preds, label='Predicted Data') # predicted plot
plt.title('Time-Series Prediction')
plt.legend()
plt.savefig("whole_plot.png", dpi=300)
plt.show()
```

whole_plot.py hosted with ❤️ by GitHub view raw



(Image by author)

Although this may seem fantastic, it’s not quite as good as it seems. This plot above, where the test data is any time step beyond the red-dashed line, seems to suggest that our model is amazingly accurate at predicting the price of Bitcoin, quite a few months into the future. However, what’s happening here is a form of **data leakage:** information about the test targets has leaked into our test features. That is, whenever we run the model, it has access to the `Open` price for that day, which is obviously going to be extremely close to the price Bitcoin ends up at for that particular time-step. When we loop over our predictions on the model, we append the *first* predicted value for that time-step. This means that every prediction we are plotting here has the benefit of being the *next predicted value*, rather than a sequence of 50 values into the future, like we set up in our problem definition.

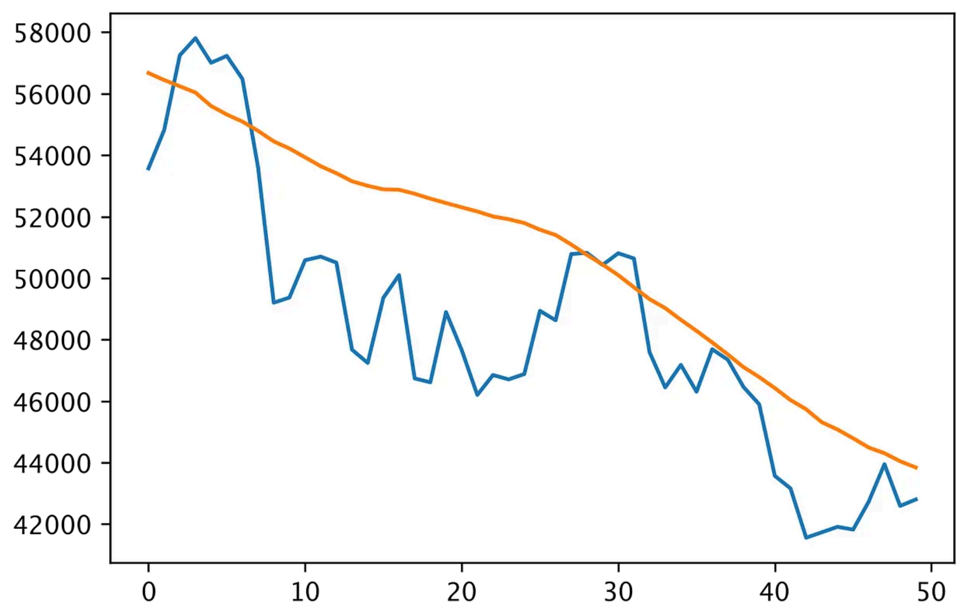
If we really want to see how our model is performing, we have to feed it the 100 time-steps of features before the final value in the test set, and then use **one model forward pass** to calculate the 50 time-step prices in the test set. Recall from above that we double-checked our test features in the final batch sample in the test set were the 100 time steps up to the time-step we're attempting to predict. That is, we feed to model 100 days of information up to 26th November 2021, and get it to predict 50 days of Bitcoin prices, from this day to the 14th January 2022.

```python
test_predict = lstm(X_test_tensors_final[-1].unsqueeze(0)) # get the last sample
test_predict = test_predict.detach().numpy()
test_predict = mm.inverse_transform(test_predict)
test_predict = test_predict[0].tolist()

test_target = y_test_tensors[-1].detach().numpy() # last sample again
test_target = mm.inverse_transform(test_target.reshape(1, -1))
test_target = test_target[0].tolist()

plt.plot(test_target, label="Actual Data")
plt.plot(test_predict, label="LSTM Predictions")
plt.savefig("small_plot.png", dpi=300)
plt.show();
```

small_plot.py hosted with ❤ by GitHub view raw

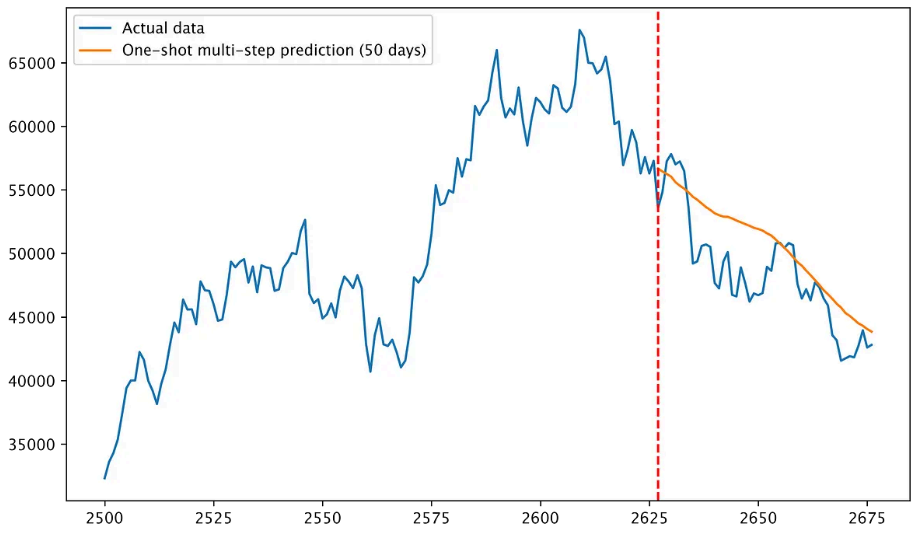


(Image by author)

This is good. If we feed in the last 100 days of information, our model successfully predicts a steady decline in the price of Bitcoin over the next 50 days. For one last plot, let's put this in perspective of the scale of the data.

```python
plt.figure(figsize=(10,6)) #plotting
a = [x for x in range(2500, len(y))]
plt.plot(a, y[2500:], label='Actual data');
c = [x for x in range(len(y)-50, len(y))]
plt.plot(c, test_predict, label='One-shot multi-step prediction (50 days)')
plt.axvline(x=len(y)-50, c='r', linestyle='--')
plt.legend()
plt.show()
```

final_plot.py hosted with ❤ by **GitHub** view raw



(Image by author)

So, if we'd run our model on 26th November 2021, we would have been correct in selling off our Bitcoin. The model correctly predicts a price drop, as well as the rate at which it drops.

Here's the amazing part, which took me a while to process. This orange whole curve is generated **without looking at any target data**. Yes, to be fair, the model is very familiar with all the actual targets before the dashed red line. However, once we move past the training set into the test set, the model has no idea what the test targets are. It has to generate its predictions based on the input features alone. It's quite remarkable, then, that our orange predictions curve so closely matches that of the actual price.

Conclusion

Interestingly, there's essentially no information on the internet on how to construct multi-step output LSTM models for multivariate time-series data. Hopefully, this article gave you both the intuition and technical understanding for building your own forecasting models. Just remember to think through your input and output shapes very carefully, and construct tensors that represent **past data predicting future data**. Pytorch's LSTM class will take care of the rest, so long as you know the shape of your data.

In terms of next steps, I would recommend running this model on the most recent Bitcoin data from today, extending back to 100 days previously. See what the model thinks will happen to the price of Bitcoin over the next 50 days. You could also play with the time being fed to the model and the time being forecast; try for longer periods and see if the model can pick up on longer-term dependencies. Finally, you should note that these types of LSTMs are not the only solution to these multivariate, multi-output forecasting problems. There are many other deep learning solutions, including encoder-decoder networks for variable-length sequences, that you should look into. Saying that, LSTMs are a great place to start, and can give incredible performance if you know how to utilise them.

Pytorch Bitcoin Cryptocurrency Deep Learning Lstm

Written by Charlie O'Neill

232 Followers · 25 Following

Follow

Researcher at Macuject, ANU. Interests include integration of deep learning, causal inference and meta-learning. Twitter: @charles0neill.

Responses (9)

What are your thoughts?

Respond