对于正弦曲线我试着运行两种类型的预测:第一个将以逐点的方式来预测,即只预测每一次的一个点,将这一点绘制成预测,然后将下一个窗口和完整的测试数据,并再次预测下一个点。

第二个预测是预测一个完整的序列,通过这个,只在培训数据的第一部分中初始化一个培训窗口。模型之后预测下一个点,同时把窗口移开,就像点逐点法一样。不同的是,之后预测中将使用我在先前预测中预测的数据。在第二步中,这只意味着一个数据点(最后一点)将来自先前的预测。在第三个预测中,最后两个数据点将来自先前的预测,等等。在50次预测之后,模型随后将预测自己的先前预测。这让我可以使用模型来预测未来的许多时间,但正如它预测的预测,然后反过来是基于预测,这将增加预测的误差速度。

下面我可以看到代码和各自的输出,点对点的预测结果和完整的序列预测。

1. **def** predict\_point\_by\_point(self, data):
2. #Predict each timestep given the last sequence of true data, in effect only predicting 1 step ahead each time
3. predicted = self.model.predict(data)
4. predicted = np.reshape(predicted, (predicted.size,))
5. **return** predicted
7. **def** predict\_sequence\_full(self, data, window\_size):
8. #Shift the window by 1 new prediction each time, re-run predictions on new window
9. curr\_frame = data[0]
10. predicted = []
11. **for** i **in** range(len(data)):
12. predicted.append(self.model.predict(curr\_frame[newaxis,:,:])[0,0])
13. curr\_frame = curr\_frame[1:]
14. curr\_frame = np.insert(curr\_frame, [window\_size-2], predicted[-1], axis=0)
15. **return** predicted
17. predictions\_pointbypoint = model.predict\_point\_by\_point(x\_test)
18. plot\_results(predictions\_pointbypoint, y\_test)
20. predictions\_fullseq = model.predict\_sequence\_full(x\_test, configs['data']['sequence\_length'])
21. plot\_results(predictions\_fullseq, y\_test)

***Sinewave point-by-point prediction***

***Sinewave full sequence prediction***

通过正确的数据,我可以看到,只有1个epoch和一组相当小的数据,LSTM深神经网络已经做了一个很好的预测正弦函数的工作。

你可以看到,随着我预测的越来越多,随着未来的预测被用于未来的预测,错误的误差增加了。因此,我看到,在整个序列的例子中,我预测的未来,我预测预测的频率和振幅与真实数据相比。sin函数可以在不太合适的情况下预测它,这很重要,因为我可以通过增加epochs来适应模型,并把dropout层移出来,使其在这个训练数据上几乎完全准确,这与测试数据是相同的,但对于其他真实的例子来说,在训练数据上过于适应模型会导致测试精度的下降,因为模型不会一般化。

由此可以证明LSTM适合被用来预测股票走势，并从中提取潜在的周期信息。

与sinewave不同,股票市场时间序列并不是任何可以映射的特定静态函数。描述股票市场时间序列的最佳属性是随机漫步。作为一个随机过程,一个真正的随机行走没有可预测的模式,因此试图建模它将是毫无意义的。幸运的是,有很多人认为股票市场不是纯粹的随机过程,这使得我可以从理论上讲,时间系列很可能有某种隐藏的模式。正是这些隐藏的模式,LSTM深层网络是预测的主要候选模型。

这个示例将使用的数据是之前经过清洗和正交化的数据，已经保存为CSV格式。数据文件夹中的csv文件。该文件包含了从2000年1月到2018年9月的标准普尔500股票指数的开放、高、低、收盘价以及标准普尔500指数的每日成交量。

在第一个实例中,我只创建一个单一的维度模型。调整配置。json文件来反映新数据,我将保留大多数参数。然而,需要的一个变化是,不像sinewave,它只在1到+ 1之间的数值范围是一个不断移动的股票市场的绝对价格。这意味着,如果我试着在没有正常化的情况下对模型进行训练,它就永远不会收敛。

为了解决这个问题,我将采用每个大小的窗口进行培训/测试数据,并将每个人的标准定义为从窗口开始的百分比变化(所以点i = 0的数据将永远是0),我将使用以下方程在预测过程结束时进行规范化,然后在预测过程的结束时进行标准化,以获得一个真实的世界数字:

其中：

n=价格变化的正交化列表（窗口）

P=调整后的每日成交价列表（窗口）

标准化：svg

去标准化：svg (1)

我将normalise\_window()函数添加到DataLoader类中来做这个转换,而且一个布尔值的布尔值标志包含在配置文件中,该文件表示这些窗口的规范化。

1. **def** normalise\_windows(self, window\_data, single\_window=False):
2. '''''Normalise window with a base value of zero'''
3. normalised\_data = []
4. window\_data = [window\_data] **if** single\_window **else** window\_data
5. **for** window **in** window\_data:
6. normalised\_window = []
7. **for** col\_i **in** range(window.shape[1]):
8. normalised\_col = [((float(p) / float(window[0, col\_i])) - 1) **for** p **in** window[:, col\_i]]
9. normalised\_window.append(normalised\_col)
10. # reshape and transpose array back into original multidimensional format
11. normalised\_window = np.array(normalised\_window).T
12. normalised\_data.append(normalised\_window)
13. **return** np.array(normalised\_data)

随着windows的规范化,我现在可以以同样的方式运行模型,以对抗sinewave数据。然而,我在运行此数据时做出了一个重要的改变;而不是使用我的框架的模型. train()方法,我将使用我创建的模型. train\_generator()方法。我这样做是因为我发现,当试图训练大型数据集时,它很容易耗尽内存,因为模型. train()函数将完整的数据集载入内存,然后将正常化应用到内存中的每个窗口中,很容易引起内存溢出。因此,我使用了Keras的fit\_generator()函数,允许使用python生成器来绘制数据来动态地对数据集进行动态培训,这意味着内存利用率将大幅减少。下面的代码详细介绍了运行三种类型的预测(逐点、完整序列和多个序列)的新运行线程。

1. configs = json.load(open('config.json', 'r'))
3. data = DataLoader(
4. os.path.join('data', configs['data']['filename']),
5. configs['data']['train\_test\_split'],
6. configs['data']['columns']
7. )
9. model = Model()
10. model.build\_model(configs)
11. x, y = data.get\_train\_data(
12. seq\_len = configs['data']['sequence\_length'],
13. normalise = configs['data']['normalise']
14. )
16. # out-of memory generative training
17. steps\_per\_epoch = math.ceil((data.len\_train - configs['data']['sequence\_length']) / configs['training']['batch\_size'])
18. model.train\_generator(
19. data\_gen = data.generate\_train\_batch(
20. seq\_len = configs['data']['sequence\_length'],
21. batch\_size = configs['training']['batch\_size'],
22. normalise = configs['data']['normalise']
23. ),
24. epochs = configs['training']['epochs'],
25. batch\_size = configs['training']['batch\_size'],
26. steps\_per\_epoch = steps\_per\_epoch
27. )
29. x\_test, y\_test = data.get\_test\_data(
30. seq\_len = configs['data']['sequence\_length'],
31. normalise = configs['data']['normalise']
32. )
34. predictions\_multiseq = model.predict\_sequences\_multiple(x\_test, configs['data']['sequence\_length'], configs['data']['sequence\_length'])
35. predictions\_fullseq = model.predict\_sequence\_full(x\_test, configs['data']['sequence\_length'])
36. predictions\_pointbypoint = model.predict\_point\_by\_point(x\_test)
38. plot\_results\_multiple(predictions\_multiseq, y\_test, configs['data']['sequence\_length'])
39. plot\_results(predictions\_fullseq, y\_test)
40. plot\_results(predictions\_pointbypoint, y\_test)

以下为网络结构：（Netrons）

1. {
2. "data": {
3. "filename": "sp500.csv",
4. "columns": [
5. "Close"
6. ],
7. "sequence\_length": 50,
8. "train\_test\_split": 0.85,
9. "normalise": true
10. },
11. "training": {
12. "epochs": 1,
13. "batch\_size": 32
14. },
15. "model": {
16. "loss": "mse",
17. "optimizer": "adam",
18. "layers": [
19. {
20. "type": "lstm",
21. "neurons": 100,
22. "input\_timesteps": 49,
23. "input\_dim": 1,
24. "return\_seq": true
25. },
26. {
27. "type": "dropout",
28. "rate": 0.2
29. },
30. {
31. "type": "lstm",
32. "neurons": 100,
33. "return\_seq": true
34. },
35. {
36. "type": "lstm",
37. "neurons": 100,
38. "return\_seq": false
39. },
40. {
41. "type": "dropout",
42. "rate": 0.2
43. },
44. {
45. "type": "dense",
46. "neurons": 1,
47. "activation": "linear"
48. }
49. ]
50. }
51. }

在上面提到的单个逐点预测上运行数据,可以很好地匹配返回的结果。但这有点欺骗性。在更仔细的检查中,预测线是由奇异的预测点组成的,这些预测点已经有了他们身后的全部历史窗口。因为这样,网络不需要知道很多关于时间序列本身的了解,而不是每下一点,很可能不会离最后一点太远。因此,即使它得到了错误的预测,下一个预测也会影响到真实的历史,忽视错误的预测,但再次允许做出错误。

虽然这可能最初听起来不太可能对下一个价格点的准确预测,但它确实有一些重要的用途。虽然它不知道下一步的价格是什么,但它确实给出了一个非常准确的范围,即下一个价格应该在哪里。

这种信息可以在诸如波动性预测等应用程序中使用(能够预测市场上的高或低波动性的时期,对特定的交易策略是非常有利的),或者偏离交易,这也可能被用作异常检测的一个很好的指标。通过预测下一点,可以实现异常检测,然后将其与真实数据进行比较,如果真实的数据值与预测点有明显不同,则可以提高数据点的异常标志。

***S&P500 full sequence prediction***

多序列预测。这是整个序列预测的混合,它仍然以测试数据初始化测试窗口,预测下一个点,并在下一个点创建一个新窗口。然而,一旦它到达了一个点,输入窗口完全由过去的预测完成,它就会停止,切换到一个完整的窗口长度,以真实的测试数据重新设置窗口,并再次启动这个过程。在本质上,这给了多个趋势线,比如对测试数据的预测,以分析模型能如何选择未来的动量趋势。

***S&P500 multi-sequence prediction***

我可以从多序列预测中看到,网络似乎正确地预测了大多数时间序列的趋势(和趋势的振幅)。虽然不完美,但它确实表明了LSTM深神经网络在顺序和时间序列问题上的有用性。更大的准确性最好是用谨慎的超参数调优来实现。