이 기사는 Deep LSTM Neural Network 아키텍처를 사용하여 Keras 및 Tensorflow를 사용하여 다차원 시계열 예측을 제공하는 데 중점을두고 있습니다. 특히 주식 시장 데이터 세트에서 주가의 모멘텀 지표를 제공합니다.

이 프레임 워크의 코드는 다음 GitHub 저장소에서 찾을 수 있습니다 (python 버전 3.5.x 및 requirements.txt 파일의 요구 사항 버전을 가정합니다.이 버전에서 벗어나면 오류가 발생할 수 있음). [**https://github.com/jaungiers / LSTM-Neural Network-for-Time-Series 예측**](https://github.com/jaungiers/LSTM-Neural-Network-for-Time-Series-Prediction)

다음 기사 섹션에서는 LSTM 뉴런 세포에 대해 간단히 살펴보고 사인파를 예측하는 장난감 예제를 제공 한 다음 응용 프로그램을 통해 확률 론적 시계열까지 살펴 봅니다. 이 기사는 간단한 심층 신경망에 대한 기본 실무 지식을 가정합니다.

LSTM 뉴런이란 무엇입니까?

전통적인 신경망 아키텍처를 오랫동안 괴롭힌 근본적인 문제 중 하나는 정보와 맥락을 위해 서로 의존하는 입력 시퀀스를 해석하는 능력이었습니다. 이 정보는 문맥에서 다음 단어가 무엇인지 예측할 수 있도록 문장의 이전 단어이거나 해당 시퀀스의 시간 기반 요소에 대한 컨텍스트를 허용하는 시퀀스의 시간 정보 일 수 있습니다.

간단히 말해 전통적인 신경망은 매번 독립형 데이터 벡터를 사용하며 메모리가 필요한 작업에 도움이되는 메모리 개념이 없습니다.

이를 해결하기위한 초기 시도는 네트워크에서 뉴런에 간단한 피드백 유형 접근 방식을 사용하여 출력을 입력으로 피드백하여 마지막으로 본 입력에 대한 컨텍스트를 제공하는 것입니다.

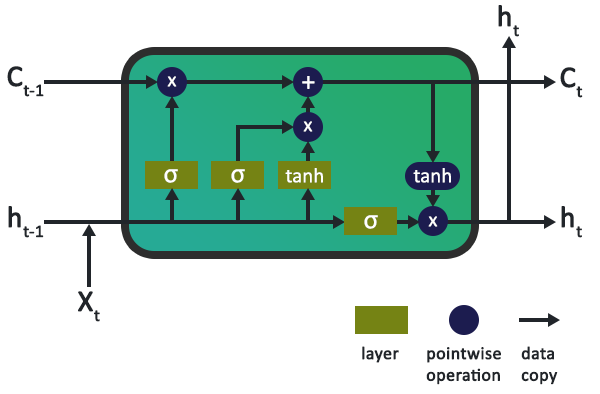
이를 재발 성 신경망 (RNN)이라고합니다. 이러한 RNN은 어느 정도 효과가 있었지만, 상당 부분 사용하면 소실 그라디언트 문제라는 문제로 이어질 것이라는 다소 큰 몰락이있었습니다.

RNN이이 문제로 인해 대부분의 실제 문제에 적합하지 않다는 것 외에는 사라지는 그래디언트 문제를 더 이상 확장하지 않을 것이므로 컨텍스트 메모리를 처리하는 다른 방법을 찾아야합니다.

이것이 바로 LTM (Long Short Term Memory) 신경망이 구조를 시작한 곳입니다. RNN 뉴런과 마찬가지로 LSTM 뉴런은 파이프 라인 내에서 메모리 컨텍스트를 유지하여 성능에 영향을 미치는 소실 구배 문제없이 순차적 및 시간적 문제를 해결할 수 있습니다.

LSTM 세포의 작용을 수학적으로 자세히 설명하는 많은 연구 논문과 기사를 온라인에서 찾을 수 있습니다. 그러나이 기사에서는 LSTM의 문제점에 대한 사용에 대해 더 우려하기 때문에 LSTM의 복잡한 작동에 대해서는 다루지 않습니다.

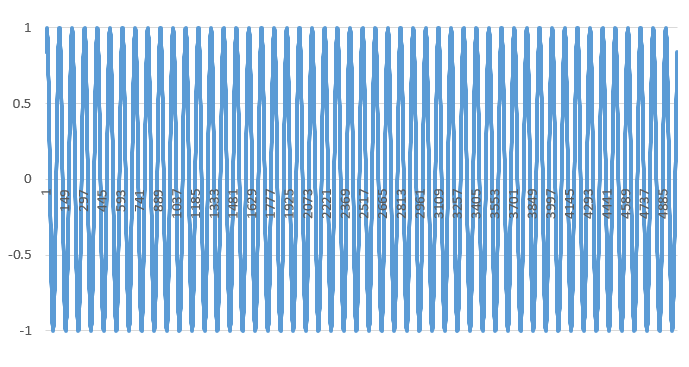
문맥 상, 아래는 LSTM 뉴런의 전형적인 내부 작동을 나타내는 다이어그램입니다. LSTM 셀 상태를 공급하는 데이터 입력, 출력 및 잊어 버림을위한 게이트 역할을하는 포인트 단위 연산과 여러 계층으로 구성됩니다. 이 셀 상태는 네트워크와 입력에서 장기 메모리와 컨텍스트를 유지합니다.



간단한 사인파 예제

시계열을 예측할 때 LSTM 신경망을 사용하는 방법을 설명하기 위해 가장 기본적인 것으로 생각할 수있는 시계열 인 신뢰할 수있는 사인파입니다. LSTM 네트워크가 훈련하기 위해이 기능의 많은 진동을 모델링하는 데 필요한 데이터를 생성 해 보겠습니다.

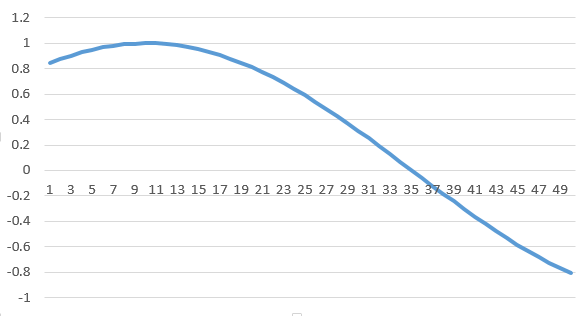
코드의 데이터 폴더에 제공된 데이터에는 우리가 만든 sinewave.csv 파일이 포함되어 있는데, 여기에는 진폭과 주파수 1 (각도 주파수 6.28 제공)과 시간 델타 0.01의 사인파의 5001 시간주기가 포함되어 있습니다. 플롯했을 때의 결과는 다음과 같습니다.

  
***사인파에 대한 데이터 세트***

이제 데이터를 얻었으므로 실제로 달성하려는 것은 무엇입니까? 글쎄, 우리는 LSTM이 우리가 공급할 데이터의 설정된 창 크기로부터 사인파를 배우기를 원하며 LSTM에게 시리즈의 다음 N- 단계를 예측하도록 요청하고 사인파를 계속 출력 할 수 있기를 바랍니다.

CSV 파일에서 팬더 데이터 프레임으로 데이터를 변환하고로드 한 다음 LSTM에 피드 할 numpy 배열을 출력하는 데 사용됩니다. Keras LSTM 레이어의 작동 방식은 3 차원 (N, W, F)의 numpy 배열을 취하는 것입니다. 여기서 N은 트레이닝 시퀀스의 수, W는 시퀀스 길이, F는 각 시퀀스의 특징 수입니다. 우리는 네트워크가 각 시퀀스에서 사인파의 모양을 엿볼 수 있도록 시퀀스 길이 (판독 창 크기)가 50 인 것을 선택했기 때문에 희망에 따라 시퀀스의 패턴을 작성하도록 스스로 희망 할 것입니다. 이전 창을 받았습니다.

시퀀스 자체는 슬라이딩 윈도우이므로 매번 1 씩 이동하여 이전 윈도우와 계속 겹칩니다. 시퀀스 길이가 50 인 일반적인 교육 창은 다음과 같습니다.

  
***사인파 데이터 셋 교육 창***

이 데이터를로드하기 위해 데이터로드 계층에 대한 추상화를 제공하기 위해 코드에 DataLoader 클래스를 만들었습니다. DataLoader 객체를 초기화하면 파일 이름과 함께 학습 대 테스트에 사용할 데이터의 백분율을 결정하는 분할 변수 및 하나 이상의 데이터 열을 선택할 수있는 열 변수가 전달됩니다. 1 차원 또는 다차원 분석 용.

class DataLoader():

def \_\_init\_\_(self, filename, split, cols):

dataframe = pd.read\_csv(filename)

i\_split = int(len(dataframe) \* split)

self.data\_train = dataframe.get(cols).values[:i\_split]

self.data\_test = dataframe.get(cols).values[i\_split:]

self.len\_train = len(self.data\_train)

self.len\_test = len(self.data\_test)

self.len\_train\_windows = None

def get\_train\_data(self, seq\_len, normalise):

data\_x = []

data\_y = []

for i in range(self.len\_train - seq\_len):

x, y = self.\_next\_window(i, seq\_len, normalise)

data\_x.append(x)

data\_y.append(y)

return np.array(data\_x), np.array(data\_y)

데이터를로드 할 수있는 데이터 객체를 얻은 후에는 딥 뉴럴 네트워크 모델을 구축해야합니다. 다시 추상화하기 위해 코드 프레임 워크는 config.json 파일과 함께 Model 클래스를 사용하여 구성 파일에 저장된 필수 아키텍처 및 하이퍼 파라미터가있는 경우 모델 인스턴스를 쉽게 빌드합니다. 네트워크를 구축하는 주요 기능은 구문 분석 된 구성 파일을 취하는 build\_model () 함수입니다.

이 함수 코드는 아래에서 볼 수 있으며 더 복잡한 아키텍처에서 나중에 사용하기 위해 쉽게 확장 할 수 있습니다.

class Model():

def \_\_init\_\_(self):

self.model = Sequential()

def build\_model(self, configs):

timer = Timer()

timer.start()

for layer in configs['model']['layers']:

neurons = layer['neurons'] if 'neurons' in layer else None

dropout\_rate = layer['rate'] if 'rate' in layer else None

activation = layer['activation'] if 'activation' in layer else None

return\_seq = layer['return\_seq'] if 'return\_seq' in layer else None

input\_timesteps = layer['input\_timesteps'] if 'input\_timesteps' in layer else None

input\_dim = layer['input\_dim'] if 'input\_dim' in layer else None

if layer['type'] == 'dense':

self.model.add(Dense(neurons, activation=activation))

if layer['type'] == 'lstm':

self.model.add(LSTM(neurons, input\_shape=(input\_timesteps, input\_dim), return\_sequences=return\_seq))

if layer['type'] == 'dropout':

self.model.add(Dropout(dropout\_rate))

self.model.compile(loss=configs['model']['loss'], optimizer=configs['model']['optimizer'])

print('[Model] Model Compiled')

timer.stop()

데이터가로드되고 모델이 구축되면 이제 훈련 데이터를 사용하여 모델 훈련을 진행할 수 있습니다. 이를 위해 Model과 DataLoader 추상화를 활용하여 훈련, 출력 및 시각화를 위해 이들을 결합하는 별도의 실행 모듈을 만듭니다.

아래는 모델을 훈련시키기위한 일반적인 실행 스레드 코드입니다.

configs = json.load(open('config.json', 'r'))

data = DataLoader(

os.path.join('data', configs['data']['filename']),

configs['data']['train\_test\_split'],

configs['data']['columns']

)

model = Model()

model.build\_model(configs)

x, y = data.get\_train\_data(

seq\_len = configs['data']['sequence\_length'],

normalise = configs['data']['normalise']

)

model.train(

x,

y,

epochs = configs['training']['epochs'],

batch\_size = configs['training']['batch\_size']

)

x\_test, y\_test = data.get\_test\_data(

seq\_len = configs['data']['sequence\_length'],

normalise = configs['data']['normalise']

)

출력을 위해 두 가지 유형의 예측을 실행합니다. 첫 번째는 점 단위 방식으로 예측합니다. 즉, 매번 단일 지점을 미리 예측하고이 점을 예측으로 플로팅 한 다음 다음 창을 사용합니다. 전체 테스트 데이터와 함께 다음 지점을 다시 한 번 예측합니다.

두 번째 예측은 전체 시퀀스를 예측하는 것입니다.이를 통해 훈련 데이터의 첫 번째 부분으로 훈련 창을 한 번만 초기화합니다. 그런 다음 모델은 다음 점을 예측하고 점별 방법으로 창을 이동합니다. 차이점은 이전 예측에서 예측 한 데이터를 사용하여 예측한다는 것입니다. 두 번째 단계에서 이것은 단지 하나의 데이터 포인트 (마지막 포인트)만이 이전 예측으로부터 나온다는 것을 의미 할 것이다. 세 번째 예측에서 마지막 두 데이터 포인트는 이전 예측 등으로부터 올 것입니다. 50 번의 예측 후에 우리 모델은 그 자체의 이전 예측을 예측할 것입니다. 이를 통해 모델을 사용하여 많은 시간 단계를 예측할 수 있습니다.

아래에서 점별 예측과 전체 시퀀스 예측에 대한 코드와 각 출력을 볼 수 있습니다.

def predict\_point\_by\_point(self, data):

*#Predict each timestep given the last sequence of true data, in effect only predicting 1 step ahead each time*

predicted = self.model.predict(data)

predicted = np.reshape(predicted, (predicted.size,))

return predicted

def predict\_sequence\_full(self, data, window\_size):

*#Shift the window by 1 new prediction each time, re-run predictions on new window*

curr\_frame = data[0]

predicted = []

for i in range(len(data)):

predicted.append(self.model.predict(curr\_frame[newaxis,:,:])[0,0])

curr\_frame = curr\_frame[1:]

curr\_frame = np.insert(curr\_frame, [window\_size-2], predicted[-1], axis=0)

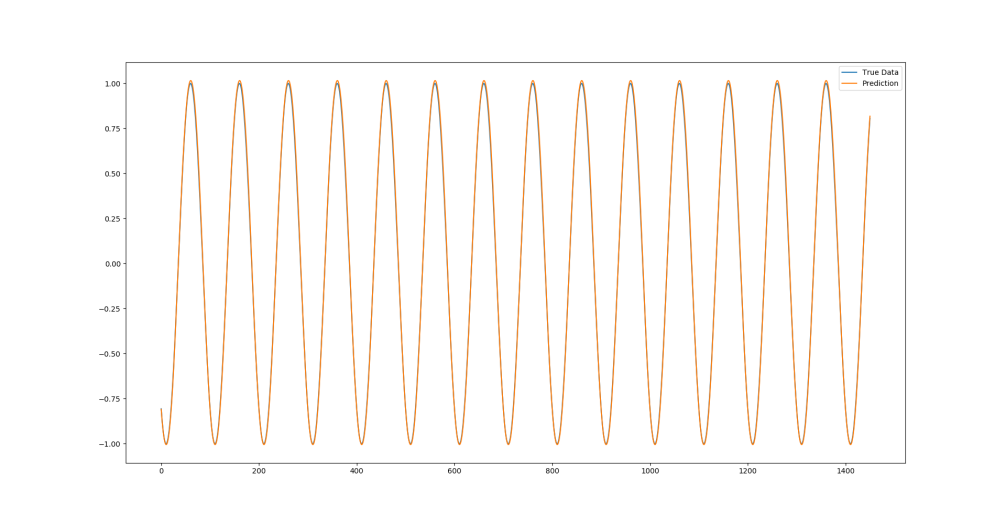
return predicted

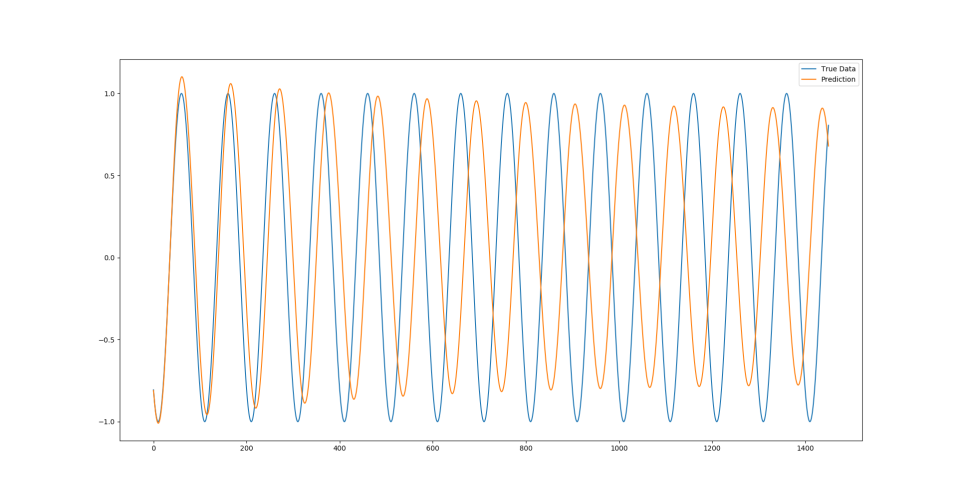
predictions\_pointbypoint = model.predict\_point\_by\_point(x\_test)

plot\_results(predictions\_pointbypoint, y\_test)

predictions\_fullseq = model.predict\_sequence\_full(x\_test, configs['data']['sequence\_length'])

plot\_results(predictions\_fullseq, y\_test)

  
***사인파 포인트 별 예측***

  
***사인파 전체 시퀀스 예측***

참고로 사인파 예제에 사용 된 네트워크 아키텍처 및 하이퍼 파라미터는 아래 구성 파일에서 확인할 수 있습니다.

{

"data": {

"filename": "sinewave.csv",

"columns": [

"sinewave"

],

"sequence\_length": 50,

"train\_test\_split": 0.8,

"normalise": false

},

"training": {

"epochs": 2,

"batch\_size": 32

},

"model": {

"loss": "mse",

"optimizer": "adam",

"layers": [

{

"type": "lstm",

"neurons": 50,

"input\_timesteps": 49,

"input\_dim": 1,

"return\_seq": true

},

{

"type": "dropout",

"rate": 0.05

},

{

"type": "lstm",

"neurons": 100,

"return\_seq": false

},

{

"type": "dropout",

"rate": 0.05

},

{

"type": "dense",

"neurons": 1,

"activation": "linear"

}

]

}

}

실제 데이터와 중첩 된 LSTM 심층 신경망은 단 하나의 에포크 (epoch)와 합리적으로 작은 데이터 세트만으로도 사인 함수를 예측하는 데 상당히 훌륭한 작업을 수행했음을 알 수 있습니다.

미래에 점점 더 많이 예측할수록 이전 예측의 오류가 미래 예측에 사용될 때 점점 더 증폭됨에 따라 오류 마진이 증가한다는 것을 알 수 있습니다. 따라서 전체 시퀀스 예에서 예측의 빈도와 진폭이 실제 데이터와 비교 될 때 정확도가 떨어질 것으로 예측합니다. 그러나 sin 함수는 노이즈가없는 매우 쉬운 진동 함수이므로 과적 합 없이도 여전히이를 잘 예측할 수 있습니다. 이것은 에포크를 증가시키고 드롭 아웃 레이어를 만들어 모델을 쉽게 과적 합할 수 있기 때문에 중요합니다. 테스트 데이터와 동일한 패턴 인이 교육 데이터에서 거의 완벽하게 정확합니다.

다음 단계에서는 실제 데이터에 모델을 사용하여 효과를 확인합니다.

단순하지 않은 주식 시장

정확한 포인트 단위로 사인파의 수백 시간 간격을 예측했습니다. 이제 주식 시장 시계열에서 동일한 작업을 수행하고 즉시 이익을 얻을 수 있습니까? 불행히도 실제 세계에서는 그렇게 간단하지 않습니다.

사인파와 달리 주식 시장 시계열은 매핑 할 수있는 특정 정적 함수가 아닙니다. 주식 시장 시계열의 움직임을 설명하는 가장 좋은 속성은 무작위로 걷는 것입니다. 확률 론적 과정으로서, 실제 랜덤 보행은 예측 가능한 패턴이 없으므로 모델링 시도는 무의미합니다. 다행스럽게도 주식 시장은 순수한 확률 론적 과정이 아니라고 주장하는 많은 주장이 있습니다. 이는 시계열이 일종의 숨겨진 패턴을 가질 수 있다는 이론을 제시 할 수있게합니다. LSTM 딥 네트워크가 예측할 수있는 주요 후보는 바로 이러한 숨겨진 패턴입니다.

이 예제에서 사용할 데이터는 데이터 폴더의 sp500.csv 파일입니다. 이 파일에는 공개, 고가, 저가, 종가 및 2000 년 1 월부터 2018 년 9 월까지 S & P 500 주식 지수의 일일 거래량이 포함되어 있습니다.

첫 번째 사례에서는 종가 만 사용하여 단일 차원 모델 만 작성합니다. 새 데이터를 반영하도록 config.json 파일을 조정하면 대부분의 매개 변수를 동일하게 유지합니다. 그러나 필요한 한 가지 변화는 -1에서 +1 사이의 숫자 범위 만있는 사인파와 달리 종가는 주식 시장의 지속적으로 움직이는 절대 가격이라는 것입니다. 즉, 모델을 정규화하지 않고 모델을 학습하려고하면 수렴되지 않습니다.

이 문제를 해결하기 위해 각 n 크기의 훈련 / 테스트 데이터 창을 가져 와서 각 창을 정규화하여 해당 창 시작시의 백분율 변화를 반영합니다 (i = 0 지점의 데이터는 항상 0 임). 다음 방정식을 사용하여 예측 프로세스의 끝에서 정규화 한 후 비정규 화하여 실제 세계 수를 예측에서 얻습니다.

*n = 가격 변동의 정규화 된 목록 [창]  
p = 조정 된 일일 반환 가격의 원시 목록 [창]*

표준화:

비정규 화 :

이 변환을 수행하기 위해 normalise\_windows () 함수를 DataLoader 클래스에 추가했으며 부울 정규화 플래그가 이러한 창의 정규화를 나타내는 구성 파일에 포함되어 있습니다.

def normalise\_windows(self, window\_data, single\_window=False):

'''Normalise window with a base value of zero'''

normalised\_data = []

window\_data = [window\_data] if single\_window else window\_data

for window in window\_data:

normalised\_window = []

for col\_i in range(window.shape[1]):

normalised\_col = [((float(p) / float(window[0, col\_i])) - 1) for p in window[:, col\_i]]

normalised\_window.append(normalised\_col)

*# reshape and transpose array back into original multidimensional format*

normalised\_window = np.array(normalised\_window).T

normalised\_data.append(normalised\_window)

return np.array(normalised\_data)

윈도우가 정규화되면 이제 사인파 데이터에 대해 실행 한 것과 동일한 방식으로 모델을 실행할 수 있습니다. 그러나 우리는이 데이터를 실행할 때 중요한 변경을했습니다. 프레임 워크의 model.train () 메소드를 사용하는 대신 생성 한 model.train\_generator () 메소드를 사용합니다. model.train () 함수가 전체 데이터 세트를 메모리에로드 한 다음 메모리의 각 창에 정규화를 적용하기 때문에 대용량 데이터 세트를 훈련하려고 할 때 메모리가 부족하기 쉽다는 것을 알았 기 때문에이 작업을 수행하고 있습니다. 메모리 오버플로를 쉽게 일으 킵니다. 대신 Keras의 fit\_generator () 함수를 사용하여 파이썬 생성기를 사용하여 데이터 세트를 동적으로 훈련하여 데이터를 가져옵니다. 즉, 메모리 사용률이 극적으로 최소화됩니다.

configs = json.load(open('config.json', 'r'))

data = DataLoader(

os.path.join('data', configs['data']['filename']),

configs['data']['train\_test\_split'],

configs['data']['columns']

)

model = Model()

model.build\_model(configs)

x, y = data.get\_train\_data(

seq\_len = configs['data']['sequence\_length'],

normalise = configs['data']['normalise']

)

*# out-of memory generative training*

steps\_per\_epoch = math.ceil((data.len\_train - configs['data']['sequence\_length']) / configs['training']['batch\_size'])

model.train\_generator(

data\_gen = data.generate\_train\_batch(

seq\_len = configs['data']['sequence\_length'],

batch\_size = configs['training']['batch\_size'],

normalise = configs['data']['normalise']

),

epochs = configs['training']['epochs'],

batch\_size = configs['training']['batch\_size'],

steps\_per\_epoch = steps\_per\_epoch

)

x\_test, y\_test = data.get\_test\_data(

seq\_len = configs['data']['sequence\_length'],

normalise = configs['data']['normalise']

)

predictions\_multiseq = model.predict\_sequences\_multiple(x\_test, configs['data']['sequence\_length'], configs['data']['sequence\_length'])

predictions\_fullseq = model.predict\_sequence\_full(x\_test, configs['data']['sequence\_length'])

predictions\_pointbypoint = model.predict\_point\_by\_point(x\_test)

plot\_results\_multiple(predictions\_multiseq, y\_test, configs['data']['sequence\_length'])

plot\_results(predictions\_fullseq, y\_test)

plot\_results(predictions\_pointbypoint, y\_test)

{

"data": {

"filename": "sp500.csv",

"columns": [

"Close"

],

"sequence\_length": 50,

"train\_test\_split": 0.85,

"normalise": true

},

"training": {

"epochs": 1,

"batch\_size": 32

},

"model": {

"loss": "mse",

"optimizer": "adam",

"layers": [

{

"type": "lstm",

"neurons": 100,

"input\_timesteps": 49,

"input\_dim": 1,

"return\_seq": true

},

{

"type": "dropout",

"rate": 0.2

},

{

"type": "lstm",

"neurons": 100,

"return\_seq": true

},

{

"type": "lstm",

"neurons": 100,

"return\_seq": false

},

{

"type": "dropout",

"rate": 0.2

},

{

"type": "dense",

"neurons": 1,

"activation": "linear"

}

]

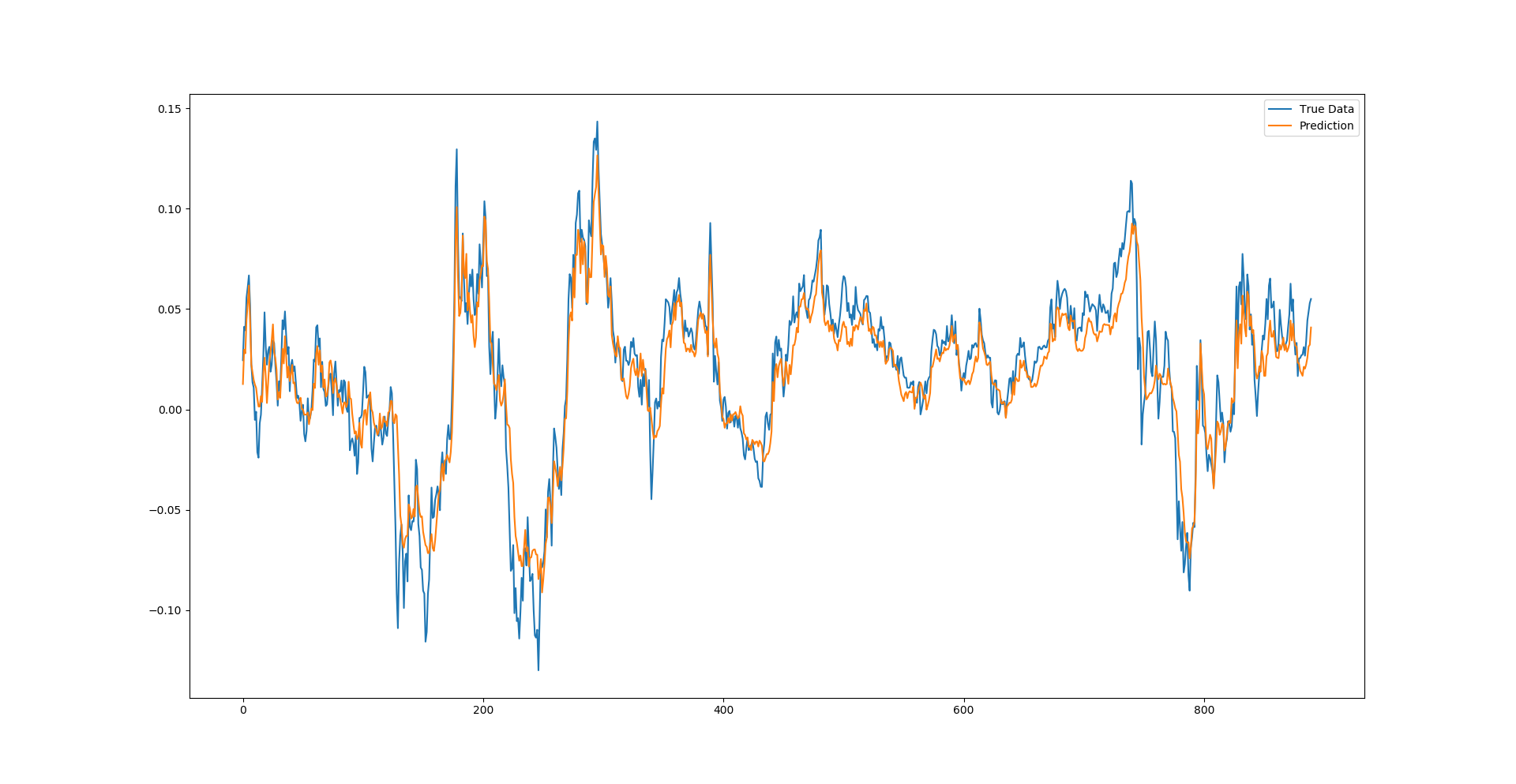
}

}

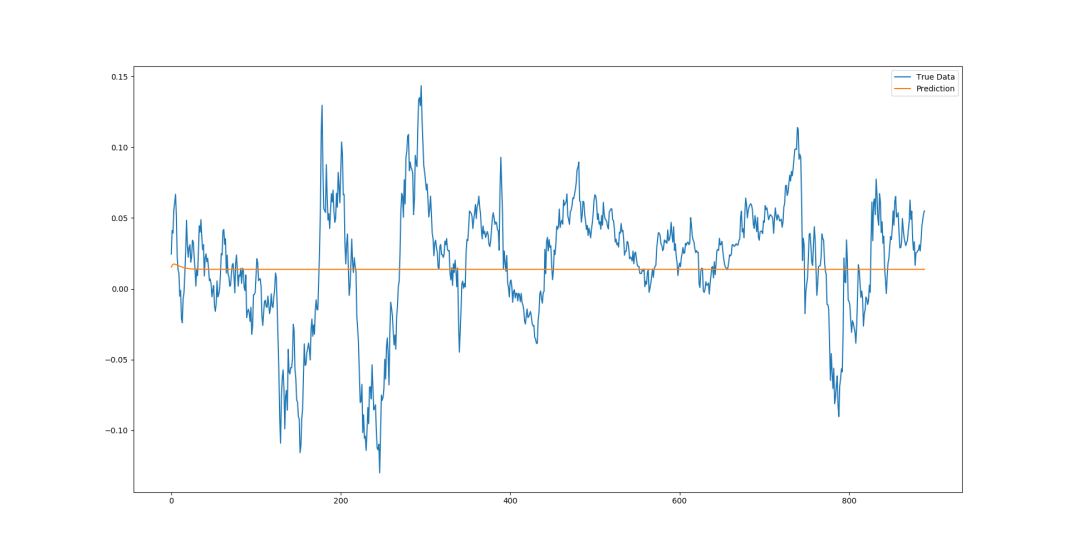
위에서 언급 한대로 단일 포인트 단위 예측에서 데이터를 실행하면 리턴과 매우 일치하는 것을 얻을 수 있습니다. 그러나 이것은 약간 기만적입니다. 더 자세히 살펴보면, 예측 라인은 이전의 전체 실제 히스토리 창을 가진 단일 예측 포인트로 구성됩니다. 이 때문에 네트워크는 다음 각 지점이 마지막 지점에서 너무 멀지 않을 것 이외의 시계열 자체에 대해 많이 알 필요가 없습니다. 따라서 포인트에 대한 예측이 잘못 되더라도 다음 예측은 실제 히스토리를 고려하여 잘못된 예측을 무시하지만 오류가 다시 발생할 수 있습니다.

이것이 다음 가격대의 정확한 예측에 대해 유망한 것으로 보이지는 않지만 중요한 용도가 있습니다. 정확한 다음 가격이 무엇인지 알 수 없지만 다음 가격의 범위를 매우 정확하게 나타냅니다.

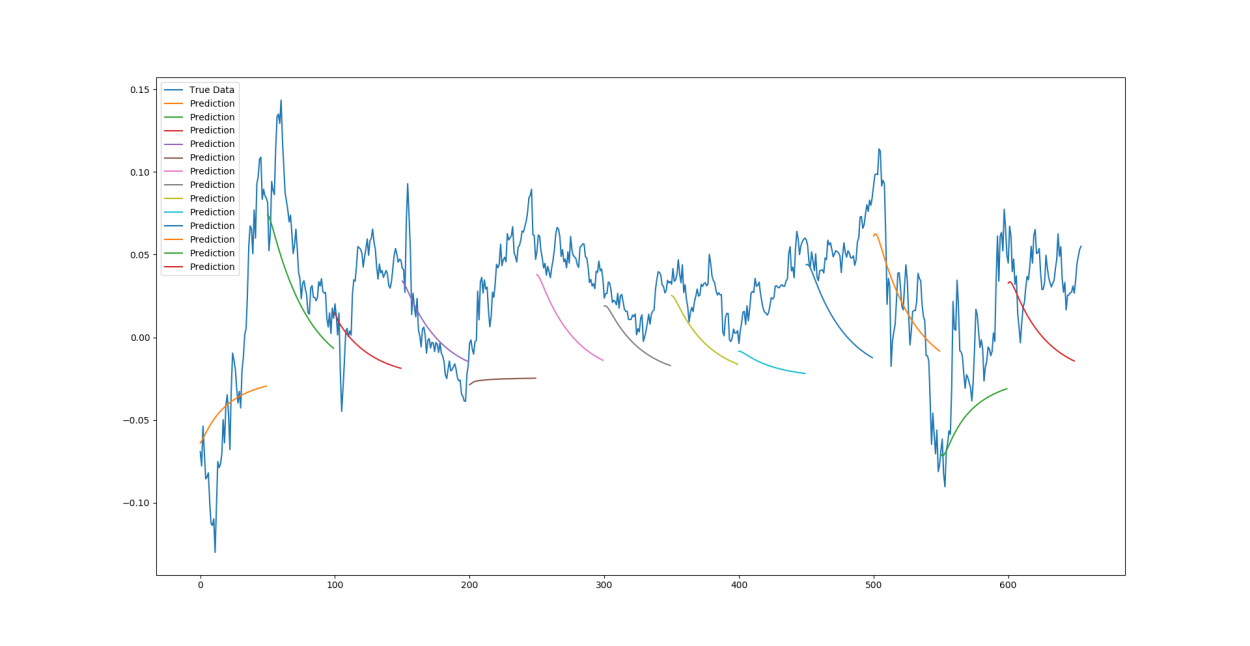
이 정보는 변동성 예측 (시장에서 변동성이 높거나 낮은 기간을 예측할 수있는 특정 거래 전략에 매우 유리할 수 있음)과 같은 응용 프로그램에서 사용될 수 있거나 거래에서 멀어지면 좋은 지표로 사용될 수 있습니다. 이상 감지 용. 다음 지점을 예측 한 다음 들어올 때이를 실제 데이터와 비교하여 이상 감지를 수행 할 수 있으며, 실제 데이터 값이 예측 된 포인트와 크게 다른 경우 해당 데이터 포인트에 대해 이상 플래그가 발생할 수 있습니다.

  
***S & P500 지점 별 예측***

전체 시퀀스 예측으로 넘어 가면이 유형의 시계열에 대해 가장 유용한 예측 인 것 같습니다 (적어도이 하이퍼 파라미터를 사용하여이 모델에 대해 훈련 된 것임). 모델이 어떤 종류의 운동량을 따랐던 예측의 시작에서 약간의 충돌을 볼 수 있지만, 모델이 가장 최적의 패턴이 시계열의 평형에 수렴하는 것으로 결정한 것을 매우 빨리 볼 수 있습니다. 이 단계에서 이것은 많은 가치를 제공하지 않는 것처럼 보일지 모르지만, 반품 거래자는 모델이 변동성을 제거 할 때 가격 시리즈가 되돌릴 평균을 단순히 찾는다는 것을 의미하기 위해 거기에 개입 할 수 있습니다.

  
***S & P500 전체 시퀀스 예측***

마지막으로이 모델에 대한 세 번째 유형의 예측을 만들었습니다.이를 다중 시퀀스 예측이라고합니다. 이것은 여전히 ​​테스트 데이터로 테스트 창을 초기화하고 그 위에 다음 점을 예측하고 다음 점으로 새 창을 만드는 점에서 전체 시퀀스 예측의 혼합입니다. 그러나 입력 창이 과거 예측으로 완전히 구성된 지점에 도달하면 중지되고 전체 창 길이로 앞으로 이동하고 실제 테스트 데이터로 창을 재설정 한 후 프로세스를 다시 시작합니다. 본질적으로 이것은 테스트 데이터에 대한 여러 추세선 예측을 제공하여 모델이 미래 모멘텀 추세를 얼마나 잘 포착 할 수 있는지 분석 할 수 있습니다.

  
***S & P500 다중 시퀀스 예측***

다중 시퀀스 예측에서 네트워크가 대부분의 시계열에 대한 추세 (및 추세의 진폭)를 정확하게 예측하는 것으로 보입니다. 완벽하지는 않지만 순차적 및 시계열 문제에서 LSTM 심층 신경망의 유용성을 나타냅니다. 신중한 하이퍼 파라미터 튜닝으로 더 큰 정확도를 달성 할 수 있습니다.

다차원 LSTM 예측

지금까지 모델은 1 차원 입력 (S & P500 데이터 세트의 경우 "종가"가격) 만 사용했습니다. 그러나 더 복잡한 데이터 세트의 경우 데이터 세트를 향상시키고 따라서 모델의 정확도를 향상시키는 데 사용할 수있는 시퀀스에 대해 다양한 차원이 자연스럽게 존재합니다.

S & P500 데이터 세트의 경우 5 개의 가능한 차원을 구성하는 Open, High, Low, Close 및 Volume이 있습니다. 우리가 개발 한 프레임 워크는 다차원 입력 데이터 셋을 사용할 수있게 해주므로이 모델을 활용하기 위해 열과 lstm 첫 번째 레이어 input\_dim 값을 적절히 편집하는 것입니다. 이 경우 2 차원을 사용하여 모델을 실행합니다. "닫기"및 "볼륨".

{

"data": {

"filename": "sp500.csv",

"columns": [

"Close",

"Volume"

],

"sequence\_length": 50,

"train\_test\_split": 0.85,

"normalise": true

},

"training": {

"epochs": 1,

"batch\_size": 32

},

"model": {

"loss": "mse",

"optimizer": "adam",

"layers": [

{

"type": "lstm",

"neurons": 100,

"input\_timesteps": 49,

"input\_dim": 2,

"return\_seq": true

},

{

"type": "dropout",

"rate": 0.2

},

{

"type": "lstm",

"neurons": 100,

"return\_seq": true

},

{

"type": "lstm",

"neurons": 100,

"return\_seq": false

},

{

"type": "dropout",

"rate": 0.2

},

{

"type": "dense",

"neurons": 1,

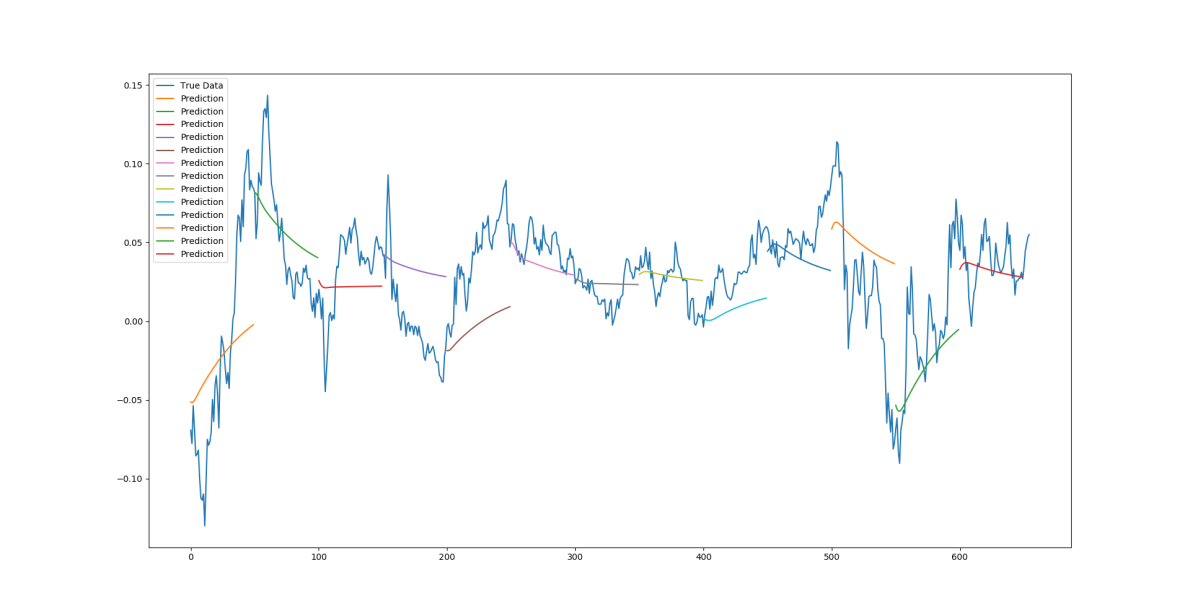
"activation": "linear"

}

]

}

}

  
***"닫기"및 "볼륨"을 사용하는 S & P500 다차원 다중 시퀀스 예측***

"Close"와 함께 두 번째 "Volume"차원이 추가되어 출력 예측이 더 세분화됨을 알 수 있습니다. 예측 트렌드 라인은 시작부터 일반적인 트렌드뿐만 아니라 트렌드 라인의 정확도도 향상되는 것처럼 약간의 다가오는 딥을 예측하기 위해 더 정확한 것으로 보입니다.

결론

이 기사는 실제로 LSTM 심층 신경망의 실제 사례를 제공하는 것을 목표로하지만, 순차 및 일시적 문제에서 잠재력과 적용의 표면을 긁어 냈습니다.

글을 쓰는 시점에서 LSTM은 여기에 설명 된 고전적인 시계열 문제부터 텍스트 자동 수정, 이상 감지 및 사기 감지, 자체 운전 자동차 기술의 핵심 개발에 이르기까지 다양한 실제 문제에 성공적으로 사용되었습니다. .

현재 위에 설명 된 바닐라 LSTM을 사용하는 경우, 특히 재무 시계열을 사용할 때 몇 가지 제한 사항이 있습니다.이 시리즈 자체에는 고정되지 않은 속성이있어 모델링하기가 매우 어렵습니다. 시계열의 비정상 성을 다루기 위해). 또한 일부 응용 분야에서는 신경망에 대한주의 기반 메커니즘의 새로운 발전이 LSTM보다 성능이 뛰어나고 (이러한주의 기반 메커니즘과 결합 된 LSTM은 자체적으로 성능이 우수함) 확인되었습니다.

그러나 현재 LSTM은 관계를 비선형으로 모델링 할 수 있고 비선형 방식으로 여러 차원의 데이터를 처리 할 수 ​​있도록보다 고전적인 통계 시계열 방식을 크게 개선했습니다.

우리가 개발 한 프레임 워크의 전체 소스 코드는 다음 GitHub 페이지의 GNU GPLv3 (GNU General Public License)에서 확인할 수 있습니다 (우리는이 코드가 어디에 있든 크레딧이 "Jakob Aungiers, Altum Intelligence ltd"라고 명확하게 요구합니다 사용) : [**https://github.com/jaungiers/LSTM-Neural-Network-for-Time-Series-Prediction**](https://github.com/jaungiers/LSTM-Neural-Network-for-Time-Series-Prediction)