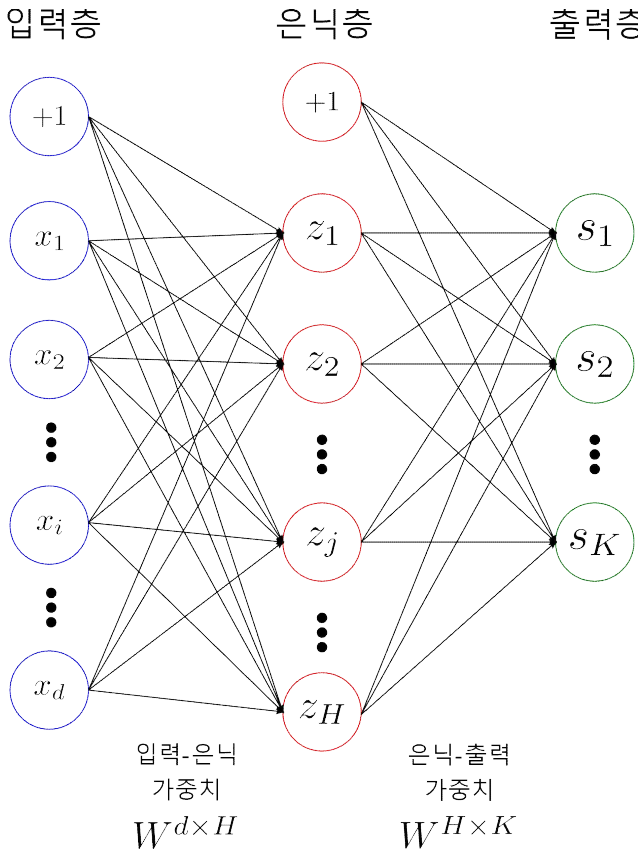
서론

냉전시대 이후 미국이 아폴로 계획 종료를 선언한 뒤 수많은 수학자와 물리학자들은 실업자가 되었다. 이때 많은 수학자와 물리학자는 월가로 진출하게 되며, 금융상품을 수치적, 계량적으로 해석하는 일을 맡았다. 이후 1973년 피셔 블랙과 마이런 숄즈가 아이슈타인의 브라운 운동 방정식으로부터 고안해낸, 옵션 가격을 산출하는 블랙 숄즈 방정식을 만들게 된다. 그 당시 월가 대부분의 사람들은 이 방정식을 사용하여 가격을 구하였고, 이를 통해서 큰 돈을 벌게되어 금융공학이라는 트렌드를 만들면서 수많은 이공계 사람들이 퀀트로 활약하게 되는 초석이 되었다. 블랙 숄즈 방정식의 발견으로 인하여 옵션시장은 급속히 성장하게 되는 발판이 되었다. 일반적으로 투자방법에는 크게 두 가지로 분류할 수 있다. 기술적 분석과 기본적 분석을 이용한 투자로 분류할 수 있다. 이 논문에서는 기본적 분석에 대해서는 다루지 않을 것이다. 기술적 분석에는 다양한 분석을 이용할 수 있고 일례로 랜덤 워크 이론을 이용해서 주가가 정규분포를 따른다고 가정한 뒤, 이를 바탕을 기댓값을 계산하는 알고리즘을 설계해서 시스템 트레이딩을 할 수 있다. 하지만 필자가 이전에 투자했던 방법과 같이 직접 손으로 제작된(hand-writting) 알고리즘보다 더욱 세밀하고 정확한 알고리즘을 만들 수 있는 기술이 현재 부상하고 있다. 그것이 바로 딥러닝이다. 현대 발달한 딥러닝 기술을 이용하여 금융 데이터의 시계열 데이터를 분석하고 패턴을 파악하도록 학습시킴으로써 주식 가격의 예측 가능성과 정확성에 대하여 논하고자 한다. 이 논문에서는 금융 데이터 예측의 대표적인 딥러닝 기술인 LSTM, RNN, DNN의 모델을 비교한 뒤 LSTM의 원리와 정확성을 극대화시키기 위한 방법, 그리고 나의 아이디어를 추가해서 LSTM을 이용한 코스피 주가 데이터의 주가 예측력을 최대한 높이는 방법과 결과를 소개할 것이다.

이론적배경

DNN, RNN, LSTM

DNN



입력 벡터가 자리잡는 층을 입력층(input layer), 최종 출력값이 자리잡는 층을 출력층(output layer), 입력층과 출력층 사이에 위치하는 모든 층을 은닉층(hidden layer)이라고 한다. 퍼셉트론을 기본 빌딩 블록으로 하여, 이런 패턴에 따라 2차원적으로 연결되어 구성되는 인공신경망의 일종을 특별히 다층 퍼셉트론(MLP: multi-layer perceptron)이라고 한다.

이런 입력층-은닉층-출력층의 경우, 다층 퍼셉트론뿐만 아니라, 다양한 인공신경망 구조에서 공통적으로 존재하는 층이다. 은닉층의 개수가 많아질수록 인공신경망이 깊어졌다(deep)고 부르며, 이렇게 충분히 깊어진 인공신경망을 러닝 모델로 사용하는 머신러닝 패러다임을 바로 딥러닝(Deep Learning)이라고 한다. 그리고, 딥러닝을 위해 사용하는 충분히 깊은 인공신경망을 심층 신경망(DNN: Deep neural network)이라고 통칭한다.

우리의 뇌에 셀 수 없이 많은 뉴런들이 서로 연결되어 하나의 망(network)를 이루고 있고, 우리 또한 이러한 인공 신경들을 묶어 인공 신경망을 구성해야 사람처럼 학습을 하게 할 수 있을 것이다. 저렇게 이루어진 망으로 어떻게 학습을 하는 것일까? 개념 단계에서 모든 것을 다 말하는 것도 이상하지만 간단하게 말하면 내가 딸기 사진을 입력에 넣었다면 출력에는 딸기라는 결과가 나와야 맞는 것이다. 딸기가 맞다면 그냥 넘어가면 되지만, 딸기가 아니라고 결과가 나왔다면 이것은 문제가 있는 것이다. 따라서 다시 뒤로 돌아가면서 각 인공 신경의 가중치(weight)와 임계값(bias) 값을 수정하게 된다. 이러한 손실함수를 줄이는 과정을 반복하며 weight와 bias 값을 조정하고 딸기라는 결과가 나오도록 '학습' 하는 것이다. 기존의 if-else문의 알고리즘에서 네트워크가 데이터를 통해 학습을 하여 알고리즘을 스스로 작성하는 것이다.

RNN

LSTM을 설명하기에 앞서 RNN에 대한 기본적인 선행 이해가 필요하다. RNN은 글, 유전자, 음성 신호, 주가 등 시계열 데이터의 형태를 갖는 데이터에서 패턴을 인식하는 인공 신경망이다. RNN은 패턴을 '기억'할 수 있는 능력이 있다. 이 부분은 마치 사람의 기억과 기억력에 비유할 수 있다. 일반적인 인공 신경망은 Feed-Forward neural netwroks라고 부르는 MLP는 데이터를 입력하면 연산이 입력층에서 은닉층을 거쳐 출력층까지 차근차근 진행된다. 이 과정에서 입력데이터는 모든 노드를 딱 한번씩 지나가게 된다. 그러나 RNN은 은닉층의 결과가 다시 같은 은닉층의 입력으로 들어가도록 연결되어 있다. MLP는 시간 순서를 무시하고 현재 주어진 데이터만 가지고 판단을 한다. 즉, 이 데이터 전에 봤었던 데이터가 무엇인지 기억하려 들지 않는다. 이와 달리 RNN은 지금 들어온 입력 데이터와 과거에 입력 받았던 데이터를 동시에 고려한다. t-1시점의 RNN의 출력값은 t시점의 RNN의 출력값에도 영향을 준다. 결과적으로 RNN은 두 개의 입력을 가지고 있는 셈인데 하나는 현재 들어온 입력이고, 또 하나는 과거의 출력이다. 과거의 출력을 다시 입력으로 집어 넣는 구조 덕분에 RNN은 기억 능력이 있다고 할 수 있다.

$$\mathbf{h}_t = \phi\left(W\mathbf{x}_t + U\mathbf{h}_{t-1}\right),$$

시간 t에서 은닉층의 상태, 즉 은닉층이 갖고 있는 값을 h_t라고 하겠다. 이 값은 같은 시점 t에 들어온 입력 x_t와 계수 행렬 W, 시간 t-1에서 은닉층의 값 h_t-1, 그리고 h_t와 h_t-1의 관계를 나타내는 행렬 U의 함수다. (이 행렬 U는 Markov chain의 상태 전이 행렬(transition matrix)과 비슷하다.) 계수 W는 지금 들어온 입력과 보유하고 있던 기억(은닉층의 값)이 얼마나 중요한지 판단하는 값이다. 예를 들어 W가 아주 큰 값으로 이루어져 있다면 기억하고 있는 h는 별로 중요하지 않고, 현재 들어온 입력값 x_t를 위주로 판단을 내린다. FFNets과 마찬가지로 출력단에서 오차를 계산하고 이 오차는 다시 이 은닉층으로 내려오는데, 그 값을 기준으로 W를 업데이트한다.

입력 x와 기억 h의 합은 함수 φ를 통과하면서 압축된다. 보통 tanh나 로지스틱 시그모이드(logistic sigmoid) 활성화 함수를 사용한다. 이 함수는 출력값의 범위를 제한해주면서 전 구간에서 미분 가능하기 때문에 역전파가 잘 적용된다.

여기에서 h_t와 h_t-1의 피드백은 매순간마다, 즉 모든 t마다 이루어진다. 그런데 h_t의 값을 구하기 위해선 h_t-1이 필요하고, h_t-1의 값을 구하는데는 다시 h_t-2가 필요하다. 결과적으로 은닉층은 과거의 h를 전부 기억하고 있어야 하는데, 실제로 값을 무한히 저장할 수는 없으므로 사용 가능한 메모리 등 여러 상황에 맞추어 적당한 범위까지만 저장을 한다.

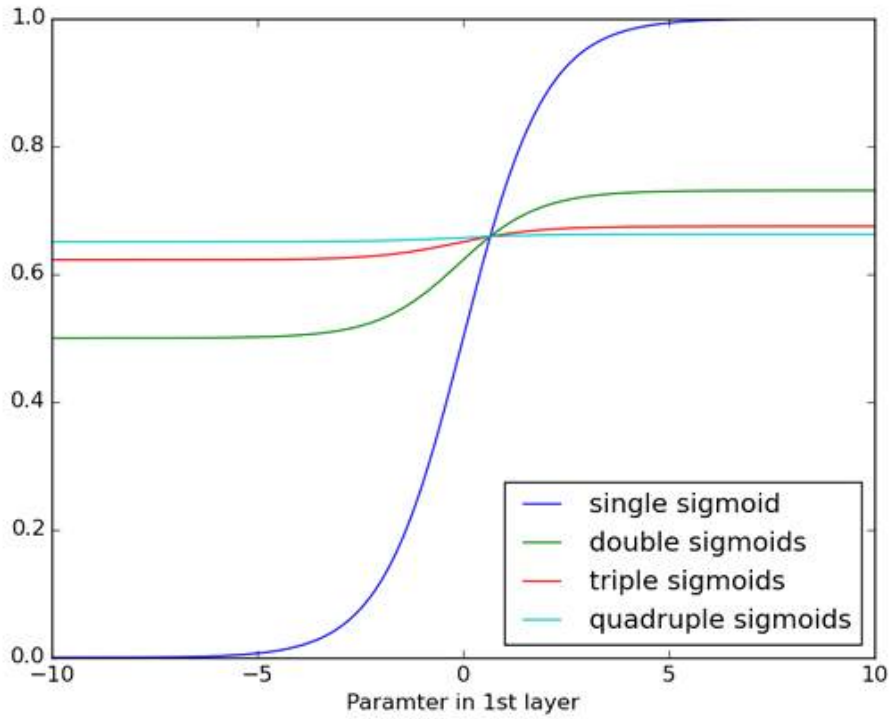
RNNs의 역사는 제법 길다. 1980년대에 이미 RNNs에 대한 논문이 여럿 나왔다. 그리고 1990년대 초반에 그라디언트 소실(vanishing gradient)이라는 문제가 나타났다. gradient의 개념은 아주 단순하다. x-y평면에 직선을 그으면, 직선의 미분값은 정의에 따라 x의 작은 변화량에 따른 y의 변화량의 비율을 나타낸ㄷ. 이를 인공 신경망에 적용하면 신경망의 모든 가중치와 오차의 관계를 구할 수 있다. 즉, 신경망의 값을 얼만큼 변화시키면 그 결과로 오차가 어떻게 변하는지를 알아낼 수 있다. 따라서 gradient의 계산은 아주 중요하다. 만일 gradient를 잘 구할 수 없다면 계수와 오차의 관계를 알 수가 없고, 결과적으로 학습이 잘 되지 않게된다.

RNNs은 시간을 거슬러 올라가며 과거 은닉값을 추적한다. 그런데 이 추적이 이어질수록 즉, 과거로 많이 거슬러 올라가면 gradient의 계산이 잘 되지 않는 경우가 있다. 이것은 신경망이 곱하기 연산을 기반으로 이루어져있기 때문이다.

은행의 적금 상품을 보면 1보다 아주 조금만 큰 값이라도 여러 번 곱하면 나중엔 제법 큰 값이 된다. 복리의 마법이라고도 하는데, 적은 이율로도 아주 오랜 기간을 보관하면 나중엔 엄청난 금액이 된다. 마찬가지로 1보다 아주 살짝 작은 값이라도 계속 곱하게 되면 나중엔 0에 가까운 값이 된다.

인공 신경망의 연산도 많은 곱하기로 이루어져 있고, 계속 곱해나가다보면 그라디언트가 완전 소실되거나(vanishing) 발산하는(exploding) 경우가 있다.
그라디언트가 발산하는 경우엔 최종적으로 컴퓨터가 다를 수 있는 가장 큰 숫자를 넘어서버린다. 그러나 발산은 비교적 제어하기가 쉽다. 그라디언트의 최대 범위를 지정해주면 된다. 문제는 바로 그라디언트가 소실되는 경우이다.

아래 그래프를 보면 시그모이드를 여러 번 곱하면 어떻게 되는지 알 수 있다. 딱 네 번 곱했을 분인데 굉장히 함수가 굉장히 평평해진다. RNN의 역전파도 마찬가지이다. 이렇게 평평해지면 기울기가 거의 모든 구간에서 0에 가까워진다. 즉, 그라디언트가 제대로 전파 되지 않는다.



Long Short-Term Memory Units

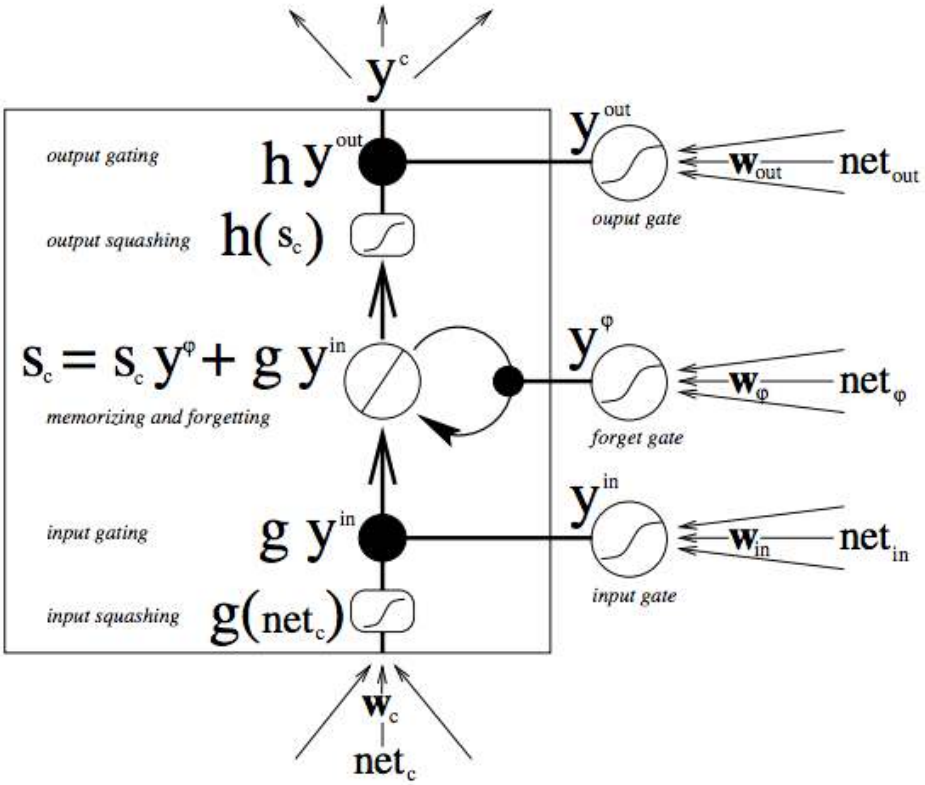
RNN의 변형인 LSTM(Long Short-Term Memory) 유닛은 90년대 중반에 처음으로 등장했다.

LSTM은 오차의 그라디언트가 시간을 거슬러서 잘 흘러갈 수 있도록 도와준다. 역전파하는 과정에서 오차의 값이 더 잘 유지되는데, 결과적으로 1000단계가 넘게 거슬러 올라갈 수 있다. 이렇게 그라디언트가 잘 흘러간다는 것은 다시 말해 더 오래 전 일도 잘 기억할 수 있도록 RNN을 진화시켰다는것을 의미한다.

LSTM 유닛은 여러 개의 게이트(gate)가 붙어있는 셀(cell)로 이루어져있으며 이 셀의 정보를 새로 저장/셀의 정보를 불러오기/셀의 정보를 유지하는 기능이 있다(컴퓨터의 메모리 셀과 비슷하다). 셀은 셀에 연결된 게이트의 값을 보고 무엇을 저장할지, 언제 정보를 내보낼지, 언제 쓰고 언제 지울지를 결정한다. 이 게이트가 열리거나(1) 닫히는(0) 디지털이 아니라 아날로그라는 점을 주의해야 한. 즉, 각 게이트는 0에서 1사이의 값을 가지며 게이트의 값에 비례해서 여러 가지 작동을 한다.

각 게이트가 갖는 값, 즉 게이트의 계수(또는 가중치, weight)는 은닉층의 값과 같은 원리로 학습된다. 즉 게이트는 언제 신호를 불러올지/내보낼지/유지할지를 학습하며 이 학습과정은 출력의 오차를 이용한 경사 하강법(gradient descent)을 사용한다.

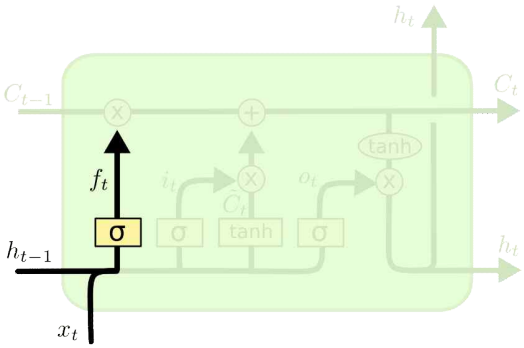
아래 그림은 LSTM 유닛과 게이트의 작동 방식을 시각화한 것이다.



우선 그림의 맨 아래 입력부터 보겠다. 3개의 화살표는 LSTM유닛에 입력되는 신호이다. 이 신호는 현재 입력신호와 과거의 셀에서 온 피드백을 합친 것인데, 바로 입력되는 것이 아니라 3개의 게이트로 들어가고 각 게이트에서는 입력값을 어떻게 다룰 것인지를 정한다.

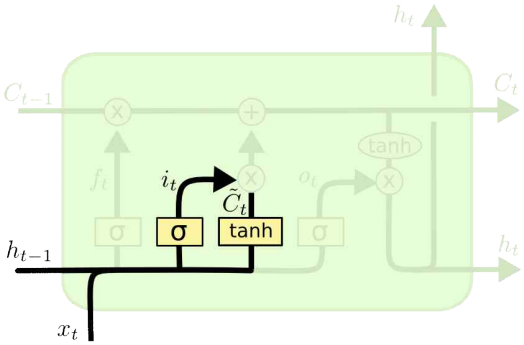
그림에서 검은색 점은 게이트를 나타낸다. 아래에 있는 게이트는 얼마나 입력값을 반영할지, 중간의 게이트는 현재 갖고있는 값 중 얼마를 잊을지, 위에 있는 게이트는 얼마나 출력할지를 결정한다. S_c는 현재 LSTM유닛의 셀의 상태이고 g_y_in은 셀에 입력되는 값이다. 모든 게이트는 샘플마다, 즉 매 시점마다 얼마나 열고 닫을지를 결정한다는걸 기억 해야한다. 그림에서 크게 써있는 문자가 각 게이트의 값을 반영한 결과이다.

LSTM의 게이트를 단계별로 시각화 해보겠다.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

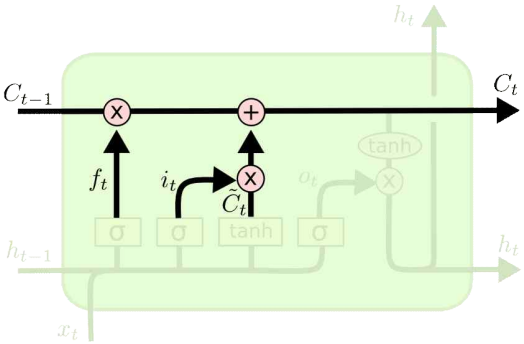
Forget gate : LSTM의 첫 단계로는 cell state로부터 어떤 정보를 버릴 것인지를 정하는 것으로, sigmoid layer에 의해 결정된다. 그래서 이 단계의 gate를 "forget gate layer"라고 부른다. 이 단계에서는 h_t−1과x_t를 받아서 0과 1 사이의 값을 C_t−1에 보내준다. 그 값이 1이면 "모든 정보를 보존해라"가 되고, 0이면 "모든 정보를 버려라"가 된다.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

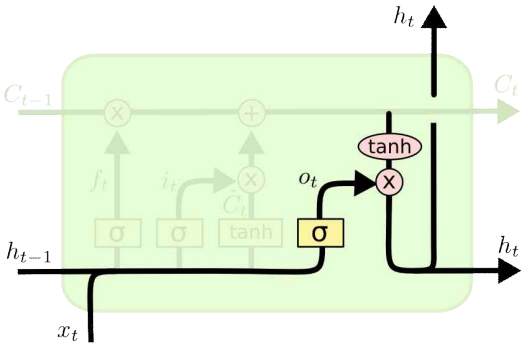
Input gate : 다음 단계는 앞으로 들어오는 새로운 정보 중 어떤 것을 cell state에 저장할 것인지를 정한다. 먼저, "input gate layer"라고 불리는 sigmoid layer가 어떤 값을 업데이트할 지 정한다. 그 다음에 tanh layer가 새로운 후보 값들인 C_t 라는 vector를 만들고, cell state에 더할 준비를 한다. 이렇게 두 단계에서 나온 정보를 합쳐서 state를 업데이트할 재료를 만들게 된다.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Cell state update : 이제 과거 state인 C_t−1를 업데이트해서 새로운 cell state인 C_t를 만들 것이다. 이미 이전 단계에서 어떤 값을 얼마나 업데이트해야 할 지 다 정해놨으므로 여기서는 그 일을 실천만 하면 된다.

우선 이전 state에 f_t를 곱해서 가장 첫 단계에서 잊어버리기로 정했던 것들을 진짜로 잊어버린다. 그리고나서 I_t∗C_t를 더한다. 이 더하는 값은 두 번째 단계에서 업데이트하기로 한 값을 얼마나 업데이트할 지 정한 만큼 scale한 값이 된다.



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

output gate : 마지막으로 무엇을 output으로 내보낼 지 정하는 일이 남았다. 이 output은 cell state를 바탕으로 필터된 값이 될 것이다. 가장 먼저, sigmoid layer에 input 데이터를 태워서 cell state의 어느 부분을 output으로 내보낼 지를 정한다. 그리고나서 cell state를 tanh layer에 태워서 -1과 1 사이의 값을 받은 뒤에 방금 전에 계산한 sigmoid gate의 output과 곱해준다. 그렇게 하면 우리가 output으로 보내고자 하는 부분만 내보낼 수 있게 된다.

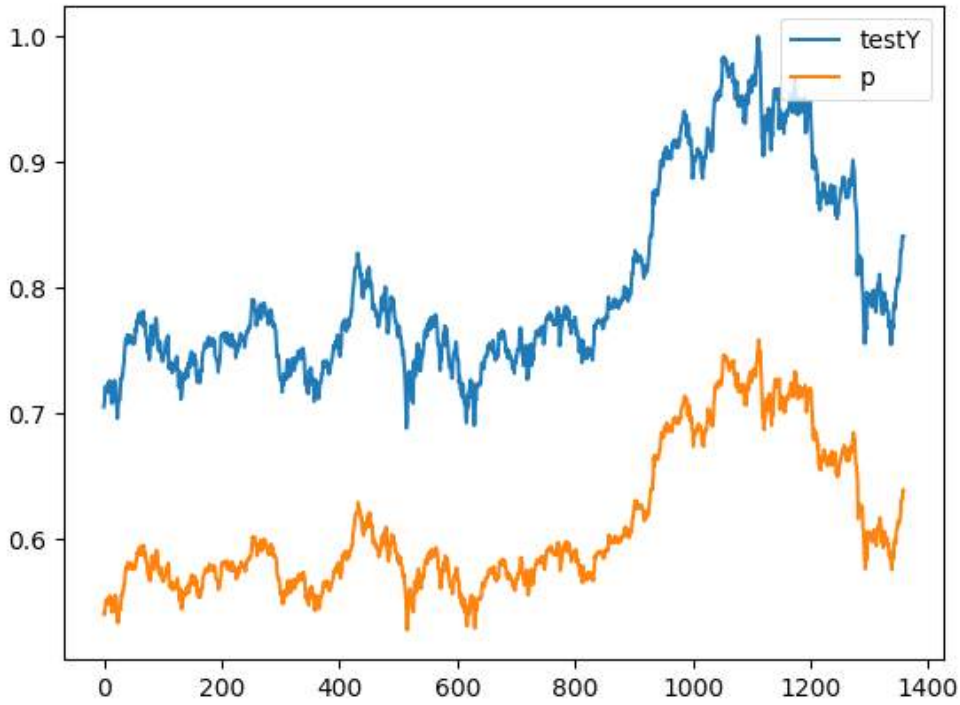
일반적인 RNNs의 유닛은 곱하기로만 이루어져있는데, LSTM은 피드백을 더하기로 잇고 있다. 따라서 위에서 이야기한, sigmoid를 곱하다 보니 생기는 그라디언트 소실 문제가 없어진다. 기억력을 오래 보존하는 것이 목적인데 왜 LSTM은 망각 게이트를 가지고 있는 걸까? 쉽게 설명하면 때론 잊는 것이 좋을 때가 있기 때문이다. 예를 들어 읽던 문서가 다 끝나고 다음 문서를 다루기 시작하는 경우엔 기억을 싹 지우고 새로 시작하는 것이 더 정확할 수 있다.

RNNs과 FFNets의 아주 큰 차이를 간략히 언급하면, FFNets은 입력 하나에 출력 하나, 즉 입력:출력이 1:1이다. 그런데 RNNs은 설정하기에 따라 1:1, 1:N(이미지 설명하기), N:N(기계 번역), N:1(음성신호 인식) 등 다양하게 적용할 수 있다.

연구방법 및 연구결과
MinMaxScalar를 사용하여 0~1 사이의 값으로 정규화 시킨다.
데이터 세트를 train 80% : test 20%로 나눈다.
optimizer는 rmsprop을 사용하고 loss는 mean_squared_error를 사용한다.
epochs는 기본으로 10으로 설정하고 batch_size는 16으로 설정했다.
위 환경을 고정 시킨 뒤, 서로 다른 모델을 적용시켜 실험을 진행한다.

MLP (epoch 100)

____
Layer (type) Output Shape Param #
===
dense_1 (Dense) (None, 32) 64
____
dropout_1 (Dropout) (None, 32) 0
____
dense_2 (Dense) (None, 32) 1056
____
dropout_2 (Dropout) (None, 32) 0
____
dense_3 (Dense) (None, 32) 1056
____
dropout_3 (Dropout) (None, 32) 0
____
dense_4 (Dense) (None, 1) 33
===
Total params: 2,209
Trainable params: 2,209
Non-trainable params: 0



Train Score: [0.004482719383945659, 0.00013798813302056022]
Test Score: [0.03693357944992906, 0.0007358351729212656]
MSE: 0.03693358
총 파라미터는 2,209개이고, 학습이 충분히 이루어 지도록 epoch를 100으로 설정하였다. 단순히 은닉층을 여러개 연결한 MLP의 경우 학습이 이루어져도 이전의 값들을 기억하지 못하므로 앞으로의 제대로 예측이 불가능 하는 것을 알 수 있다. MSE 또한 0.03693358로 매우 큰 것을 알 수 있다.
RNN

_____

Layer (type) Output Shape Param #

==========

simple_rnn_1 (SimpleRNN) (None, 32) 1088

_____

dropout_1 (Dropout) (None, 32) 0

_____

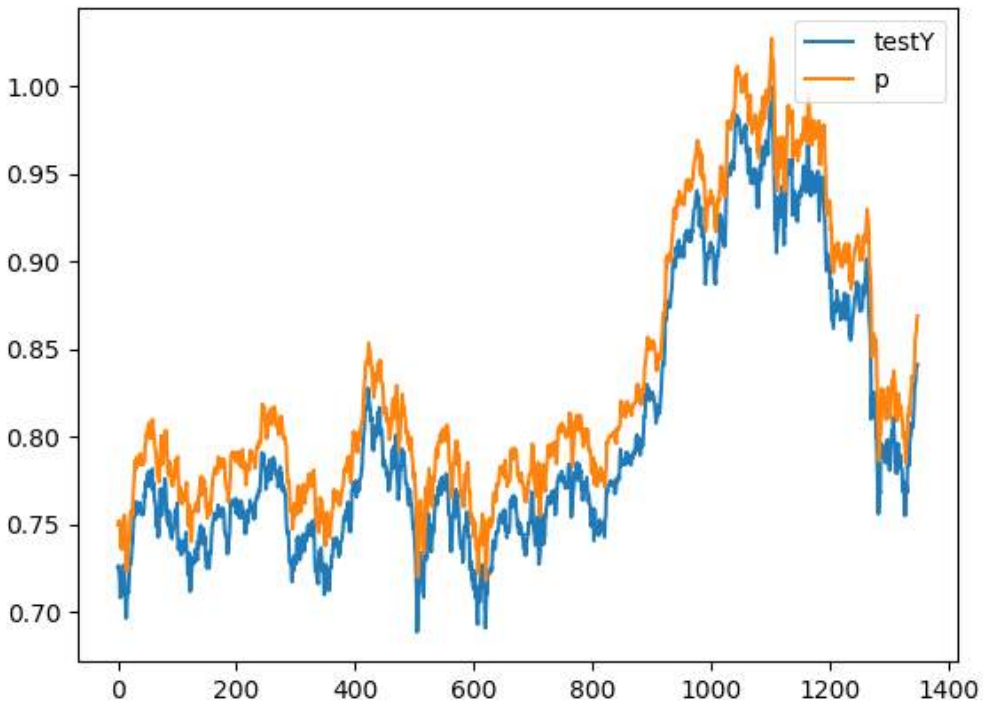
dense_1 (Dense) (None, 1) 33

==========

Total params: 1,121
Trainable params: 1,121
Non-trainable params: 0

_____

Train on 7238 samples, validate on 443 samples



Train Score: [0.00022912525406860077, 0.00013815971262779773]
Test Score: [0.0008584694710939571, 0.00074074074074074074]
MSE : 0.00085846946
RNN의 경우 1,126개의 파라미터로 구성 되어있고, MLP보다는 나은 성능을 보인다. 이전의 20개의 데이터를 참고하여 다음 가격을 예측하는 모델로 구성되어 있다. MSE의 경우 0.0008로 꽤 우수한 성능을 보임을 알 수 있다.

LSTM(epochs 10)

_____

Layer (type) Output Shape Param #

==========

lstm_1 (LSTM) (None, 32) 4352

_____

dropout_1 (Dropout) (None, 32) 0

_____

dense_1 (Dense) (None, 1) 33

==========

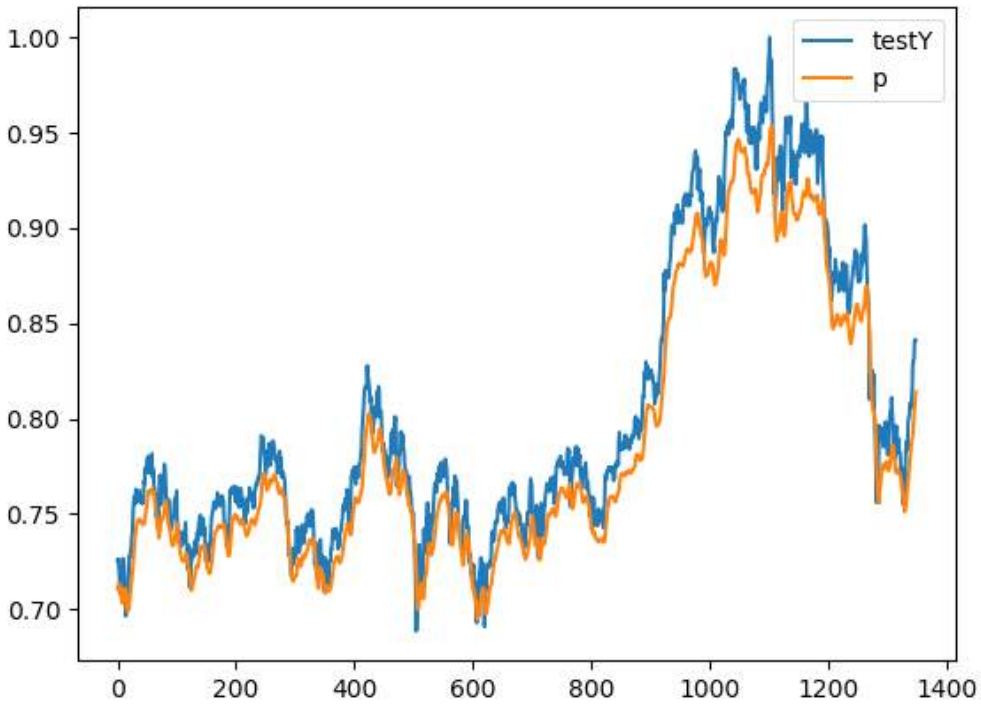
Total params: 4,385

Trainable params: 4,385

Non-trainable params: 0

_____

Train on 7238 samples, validate on 443 samples



Train Score: [0.00021257683106305135, 0.00013815971262779773]

Test Score: [0.00041856445975739645, 0.00074074074074074074]

MSE : 0.00041856448

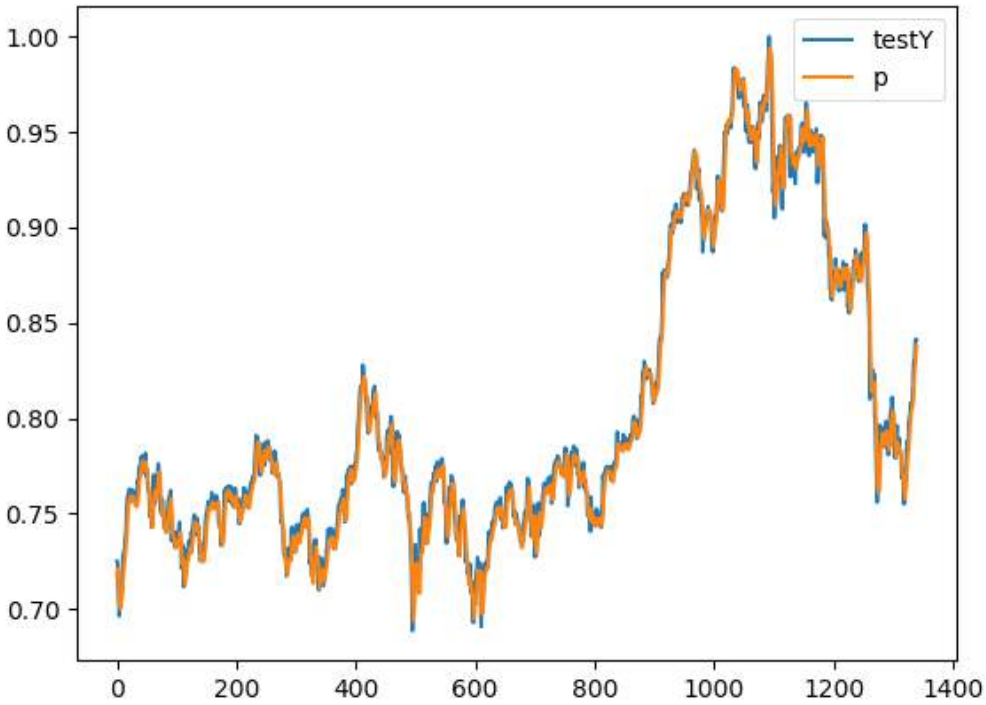
LSTM의 경우 RNN보다 수치적으로 봤을 때 약 2배의 성능 향상을 보인다. RNN

의 장기 의존성 문제와 vanishing gradiant 문제를 해결함으로써 더욱 향상된 성능을 보임을 알 수 있다. 시각적으로 보이게도 test셋과 큰 격차를 보이지 않는 것을 알 수 있다.

LSTM + MLP (epochs 10)

Layer (type)	Output Shape	Param #
lstm_1 (LSTM)	(None, 32)	4352
dropout_1 (Dropout)	(None, 32)	0
dense_1 (Dense)	(None, 64)	2112
dropout_2 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 1)	65

Total params: 6,529
Trainable params: 6,529
Non-trainable params: 0



Train Score: [0.00019781659078935298, 0.00013835085777531 82]
Validataion Score: [9.702389305548885e-05, 0.0]
Test Score: [6.658997308070173e-05, 0.00074626865671641 79]
MSE : 0.00006658998
실험환경은 이전과 같고, 단순히 Dense(64)를 추가해보았다.
LSTM에 Hidden Layer를 추가하여 FC로 연결하였더니 성능이 향상되었다.
LSTM의 세밀하지 못한 예측을 FC layter에서 특징을 한번 더 필터링 함으로써 추세에 대한 특징이 예측에 대한 성능을 보완해주는 역할을 한다고 볼 수 있다.

Deep LSTM + MLP(epochs 10)

_____

Layer (type) Output Shape Param #

==========

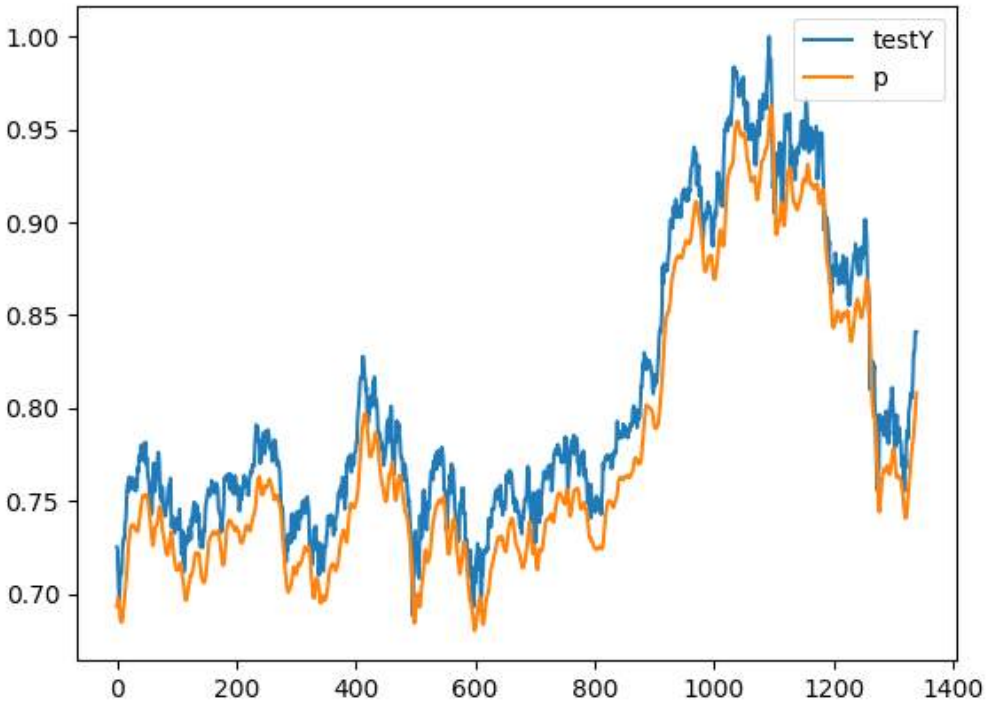
======

lstm_1 (LSTM) (None, None, 32) 4352

_____

dropout_1 (Dropout)	(None, None, 32)	0
lstm_2 (LSTM)	(None, 32)	8320
dropout_2 (Dropout)	(None, 32)	0
dense_1 (Dense)	(None, 64)	2112
dropout_3 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 64)	4160
dropout_4 (Dropout)	(None, 64)	0
dense_3 (Dense)	(None, 1)	65

Total params: 19,009
Trainable params: 19,009
Non-trainable params: 0



Train Score: [0.0003354621024078323, 0.0001383508577753182]
Validataion Score: [0.0006830131211081481, 0.0]
Test Score: [0.0007010780469472728, 0.00074626865671641791]
MSE : 0.00070107804

은닉층의 LSTM과 Dense를 추가하였는데 오히려 학습이 제대로 이루어지지 않았다. 주어진 데이터에 비에 param의 개수가 너무 많아 vanishing gradient가 발생한 것으로 보인다. 이처럼 데이터에 비해 너무 복잡한 모델을 사용하면 오히려 정확도가 떨어진다는 것을 알 수 있다.

결론

MLP	0.03693358
RNN	0.00085846946
LSTM	0.00041856448
LSTM + MLP	0.00006658998
(LSTM + MLP) * 2	0.00070107804

MLP의 경우 시간에 대한 감각이 없다고 할 수 있다. 계속해서 학습할수록 성능은 좋아 지겠지만 시계열 데이터를 대상으로 학습할때는 그 한계가 있다. 왜냐하면 이전에 어떤 데이터가 존재 했는지에 대한 정보가 없기 때문에 그때 그때 들어오는 입력에 대해서만 바로 결과를 내놓게 된다. 하지만 실험 결과에서 나왔듯이 테스트 세트 데이터에 대한 상승 및 하락에 대한 추세는 예측하는 경향이 있다.

RNN의 경우 시계열 데이터의 형태를 갖는 데이터에 대하여 이전의 데이터를 현재의 데이터의 입력값으로 받게 되어 이전의 값을 참고 할 수 있다. 따라서 이전의 데이터를 기반으로 다음 시계열 데이터를 예측하는데 도움이 되긴 하지만 vanishing/exploding gradient 문제가 존재한다는 단점이 있다.

LSTM의 경우 NN에서 발생하는 vanishing/exploding gradient 문제와 장기 의존성 문제를 해결하기 위해 고안되었다. 오차의 gradient가 시간을 거슬러서도 0에 가까워지지 않고 끝에서도 학습이 될 수 있도록 장치를 만든 것이다. 어떤 결과에서는 1000, 2000 단계 넘어서까지 오차가 전달이 될 수 있다고 하니까, 기존의 RNN보다 더 장기적인 기억을 가지고 있다고도 이야기 할 수 있다.

주식시장이 내포하고 있는 랜덤워크라는 변동성에 대해서 LSTM 만으로는 정확한 예측이 힘들었고, Dense 계층을 추가하여 정확성을 높였다. 입력 데이터의 크기에 알맞은 모델의 복잡도가 중요하였다. 너무 복잡한 모델은 과도한 파라미터 개수를 가지므로 오히려 정확성이 떨어지게 되는 모습을 볼 수 있었다.

지금까지 코스피 데이터의 종가 라는 데이터 하나만 가지고 모델을 학습시켜 최대한 정확도가 높은 모델을 만들어 보았다. 종가 데이터 외에 OHCLV의 주식의 기본 데이터 모두 학습시키거나 보조지표중 중요한 특성을 갖는 데이터를 추가하는 등의 연구가 필요하고, LSTM과 Dense의 결합 외에도 다른 머신러닝 기법을 사용하여 정확성을 높이는 방향으로 추가적인 연구를 진행하면 더 좋은 결과를 얻을수 있다고 생각한다.