Investments: Selected Quantitative Tools

Yunxiang Guo; Yuan Li; Lundrim Azemi

Abstract— **Advancements in machine learning techniques have opened up new possibilities for enhancing active portfolio management. Effective management of corporate investments and optimization of a company's portfolio are critical for maintaining a competitive edge. This study focuses on managing corporate investments and optimizing the company's portfolio. We explore the strengths and limitations of autoencoder (AE), LSTM, and CNN models in capturing patterns and making predictions for active portfolio management. The AE model demonstrates effective pattern capture but may struggle with out-of-sample performance. The LSTM model exhibits good overall performance but has a longer training time. The CNN model stands out with its fast training speed and decent in-sample performance, although it may exhibit delayed predictions in some out-of-sample cases. Overall, machine learning has the potential to revolutionize active portfolio management when properly implemented.**

I. Introduction

Machine learning (ML), a subset of artificial intelligence, is a computational methodology that enables systems to learn, evolve, and improve with experience. Machine learning techniques have found a broad range of applications across various industries, with finance being a prominent sector where its impact is profound. One particular application where machine learning exhibits immense potential is in active portfolio management. By providing precise and comprehensive predictions, for instance, on corporate earnings based on corporate margins and revenues, ML can significantly enhance active portfolio management. In the realm of active portfolio management, machine learning techniques offer the potential to enhance decision-making processes, optimize investment strategies, and improve overall performance. Here, we will discuss some potential roles of machine learning in finance, focusing on active portfolio management.

Predictive Modeling and Risk Assessment. Machine learning algorithms can be employed to develop predictive models that estimate asset returns, volatility, and other risk factors. These models can capture complex patterns in historical data, detect non-linear relationships, and incorporate a wide range of information sources, such as financial statements, market data, and news sentiment. By accurately predicting future price movements, portfolio managers can make informed investment decisions and manage risk effectively [1].

Pattern Recognition and Market Analysis. Machine learning techniques can identify subtle patterns and relationships in financial data, enabling portfolio managers to gain insights into market behavior and make data-driven decisions. For example, clustering algorithms can group similar assets based on their characteristics, helping managers diversify portfolios and identify potential investment opportunities. Additionally, anomaly detection algorithms can flag unusual market behavior or outliers, providing early warnings and mitigating risks [2].

Portfolio Optimization and Asset Allocation. Machine learning algorithms can optimize portfolio allocation by considering multiple objectives and constraints simultaneously. Reinforcement learning techniques, for instance, can learn optimal investment policies by iteratively adjusting portfolio weights based on observed market conditions and rewards. By dynamically adapting to changing market environments, these algorithms can improve portfolio performance and enhance risk-adjusted returns [3].

Sentiment Analysis and News Filtering. Machine learning can aid in sentiment analysis by processing large volumes of news articles, social media feeds, and other textual data. Natural language processing (NLP) techniques can extract sentiment, identify relevant financial events, and assess their potential impact on markets. By incorporating sentiment analysis into portfolio management, managers can react quickly to market sentiment shifts and adjust their positions accordingly [4].

Overall, ML enables accurate modeling of asset returns and volatility, aiding in risk assessment. It excels in pattern recognition and market analysis, helping managers identify market behavior, diversify portfolios, and detect anomalies. ML algorithms optimize portfolio and asset allocation, improving performance and risk-adjusted returns by considering multiple objectives and constraints. Additionally, ML aids in sentiment analysis and news filtering, enabling swift reactions to market sentiment shifts. Overall, ML has immense potential in enhancing decision-making, optimizing investment strategies, and improving portfolio management in finance.

II. Objective

The objective is to develop a ML-based trading algorithm for a corporate finance. [1] Our comparison involves portfolios captured from 1988 till 2023, portfolios optimized via the stochastic gradient descent (often abbreviated SGD) technique. We illustrate that the ML approaches benefit more compared

◇ Institut für Banking und Finance, UZH

[1]Resources in: https://github.com/xiaoxiaoshikui/Investment-Selected-Quantitative-Tools.

to traditional historical returns. Our comparative study contributes to the body of research regarding the integration of machine learning into empirical asset pricing and portfolio theory. Predictions play a crucial role in trading algorithms for

stock portfolios. While financial markets are inherently uncertain, predictions provide valuable insights into potential price movements, allowing traders to make informed decisions. By leveraging machine learning techniques, traders can analyze historical data, identify patterns, and generate predictions about future stock behavior. These predictions serve as a guide for traders to formulate their trading strategies, manage risk, and seize profitable opportunities. However, it's important to acknowledge that predictions should be complemented with other factors, such as fundamental analysis and market indicators, to make well-rounded trading decisions.

III. Network Architectures

Deep feed-forward networks, also known as multilayer perceptrons (MLPs) or dense layers, are the quintessential deep learning models. The 'feed-forward' means that information flows through the function being evaluated from input to output, and the 'deep' refers to the model's use of multiple layers of neurons. Each layer in a feed-forward network transforms its input via a set of weights (parameters), and typically a non-linear activation function is applied. A key strength of feed-forward networks is their ability to model and solve complex non-linear relationships between input and output data.

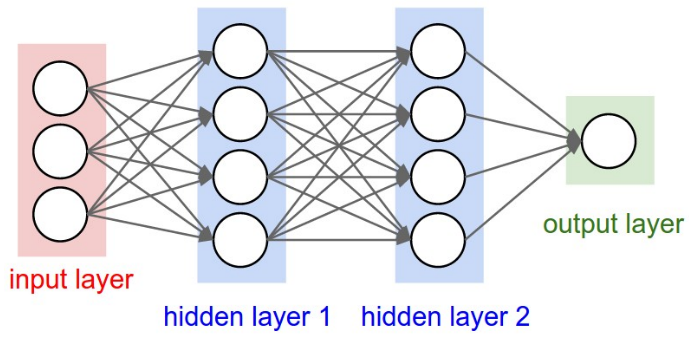

Fig. 1. A three-layer artificial neural network. (Image source: http://cs231n.github.io/convolutional-networks/conv)

A. Autoencoders

An autoencoder is a type of neural network designed to replicate its input to its output. An Autoencoder is primarily composed of three components: Encoder, Code and Decoder. The internal hidden layer known as a 'code' that serves to represent the input. Essentially, the autoencoder comprises two parts: an 'encoder' function h=f(x) that translates the input, and a 'decoder' that generates a reconstruction r=g(h), as shown in Fig.2.

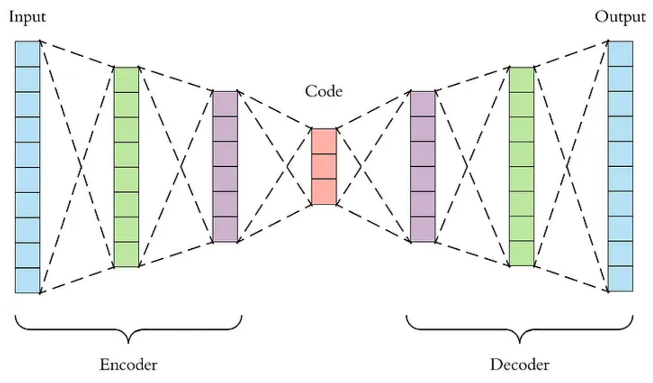

Fig. 2. The general structure of an autoencoder, mapping an input x to an output(called reconstruction) through an internal representation or code h. The autoencoder has two components: the encoder f(mapping x to h) and the decoder g(mapping h to r). (Image source: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798)

If an autoencoder only learns to set g(f(x)) =x everywhere, it wouldn't be particularly helpful. Therefore, autoencoders are constructed to prevent perfect copying. They are usually restricted in a way that they can only reproduce the input approximately and only input that is similar to the training data. This constraint compels the model to prioritize which input aspects to copy, often leading it to learn significant data properties.

Modern autoencoders have extended the concept of encoder and decoder from deterministic functions to stochastic mappings, referred to as pencoder(h — x) and pdecoder(x — h). The autoencoder concept has been part of the neural network's historical landscape for several decades (LeCun, 1987; Bourlard and Kamp, 1988; Hinton and Zemel, 1994). Initially, autoencoders were employed for dimensionality reduction or feature learning.

However, recent theoretical links between autoencoders and latent variable models have put autoencoders at the center of generative modeling. Autoencoders can be seen as a special case of feedforward networks and trained using similar techniques, typically minibatch gradient descent based on gradients computed by back-propagation.

Contrary to general feedforward networks, autoencoders can also be trained using recirculation (Hinton and McClelland, 1988), a learning algorithm that compares network activations on the original input to activations on the reconstructed input. Despite being seen as more biologically plausible than back-propagation, recirculation is seldom used in machine learning applications.

B. Recurrent Neural Networks

Recurrent neural network, short for "RNN", is born with the capability to process long sequential data and to tackle tasks with context spreading in time. The model processes one element in the sequence at one time step. After computation, the newly updated unit state is passed down to the next time step to facilitate the computation of the next element. Imagine the case when an RNN model reads all the Wikipedia articles, character by character, and then it can predict the following words given the context.

The RNN model we are about to build has LSTM cells as basic hidden units. LSTM (Long Short-Term Memory) cells are a type of recurrent neural network (RNN) unit that address a common challenge in traditional RNNs—the vanishing gradient problem. The vanishing gradient problem occurs when gradients diminish rapidly as they backpropagate through many time steps, making it difficult for the network to capture and retain long-term dependencies in sequential data.

LSTM cells contains the follwing components:
Memory Cells: LSTM cells have a memory cell that can store information over long periods, allowing them to capture and retain important dependencies in sequential data.
Input Gate: The input gate determines how much of the current input should be stored in the memory cell.
Forget Gate: The forget gate determines how much of the previous memory cell content should be discarded.
Output Gate: The output gate determines how much of the current memory cell content should be exposed as the output of the LSTM unit.

By using memory cells and the gating mechanisms described above, LSTM cells can selectively read, write, and forget information over multiple time steps. This enables them to capture and retain long-term dependencies in the sequential data, making them particularly effective for tasks that involve processing and understanding sequences, such as time series analysis.

In our case we divide the sequential data into overlapping windows, where each window represents a subset of elements. We use values from the very beginning in the first sliding window W_0 to the window W_t at time t:

W_0 = (p_0, p_1, ..., p_{w-1})
W_1 = (p_w, p_{w+1}, ..., p_{2w-1})
...
W_t = (p_tw, $p_tw + 1$, ..., $p_(t + 1)w - 1$)

to predict the prices in the following window $W_t + 1$:
$W_t + 1$ = ($p_(t + 1)w$,$p_(t + 1)w + 1$,...,$p_(t + 2)w - 1$)

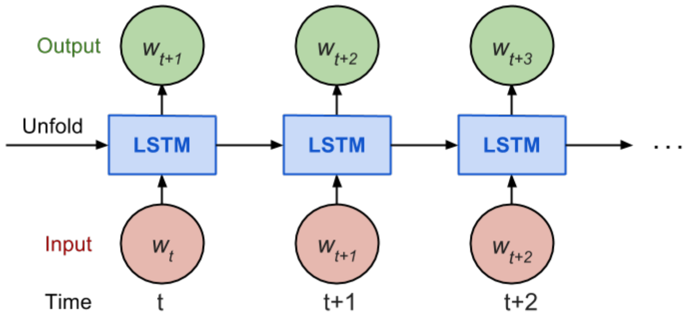

Fig. 3. The unrolled version of RNN.

C. Convolutional Neural Networks

Convolutional neural networks, short for "CNN", is a type of feed-forward artificial neural networks. Convolution is a mathematical term, referring to an operation between two matrices.

CNN's have three layers that process and extract features from data: Convolution Layer, Rectified Linear Unit (ReLU) and Pooling Layer.

The convolutional layer has a fixed small matrix defined, also called kernel or filter. As the kernel is sliding, or convolving, across the matrix representation of the input resource, it is computing the element-wise multiplication of the values in the kernel matrix and the original resource values. Specially designed kernels can process resource information for common purposes like blurring, sharpening, edge detection and many others, fast and efficiently. The convolution layer has several filters to perform the convolution operation.

The ReLU layer to perform operations on elements, and the output is a rectified feature map. Then the rectified feature map next feeds into a pooling layer. Pooling is a down-sampling operation that reduces the dimensions of the feature map.

The pooling layer converts the resulting two-dimensional arrays from the pooled feature map into a single, long, continuous, linear vector by flattening it. Finally, a fully connected layer forms when the flattened matrix from the pooling layer is fed as an input, which classifies and identifies the input resources. An sample of the image process using CNN model as shown in Fig.5.

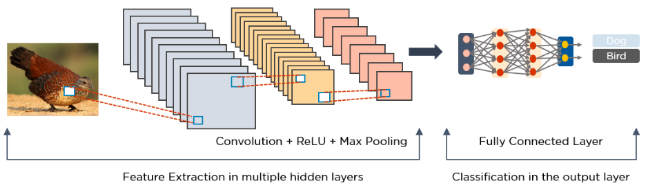
Dog
Bird
Convolution + ReLU + Max Pooling
Fully Connected Layer
Feature Extraction in multiple hidden layers
Classification in the output layer

Fig. 4. A CNN processes data through convolution, ReLU, pooling, and fully connected layers to extract features. (Image source: https://www.simplilearn.com/tutorials/deep-learning-tutorial/deep-learning-algorithm)

IV. Experiments

We build three ML models to predict the corporate earnings.

A. *Choice of signals*

TABLE I
Model Signal Selection

Column	CNN	RNN	AutoEncoder
EMP		Yes	Yes
PE			Yes
CAPE			Yes
DY			Yes
Rho			Yes
MOV	Yes		
MG	Yes	Yes	Yes
RV	Yes	Yes	Yes
ED	Yes	Yes	Yes
UN		Yes	
GDP		Yes	

- **CNN:** Consider selecting the CNN architecture for columns that might exhibit spatial or hierarchical relationships. CNNs are effective in capturing patterns in spatially organized data. For example, variables related to stock market trends (such as MOV, RV, and MG) might benefit from CNN-based analysis.
- **RNN:** Choose the RNN architecture for columns that have a temporal aspect or where past values influence future earnings predictions. RNNs are suitable for modeling dependencies over time. Variables like EMP, UN, GDP that have a time-dependent nature may be relevant for RNN-based analysis.
- **AutoEncoder:** Utilize the AutoEncoder architecture for dimensionality reduction or feature extraction purposes. If you have a large number of features or want to identify latent representations in the data, an AutoEncoder can help compress the information. Variables such as EMP, PE, CAPE, DY, Rho where feature extraction or dimensionality reduction is desired can be considered.

This experiment aims to explore the feasibility and efficacy of employing machine learning to predict corporate earnings. The selected predictive signals for this investigation are corporate margins and revenues, given their historical significance and influence on corporate earnings.

1) Data preparation: To fits the model, we processing the data by removing NA values and set the date as index. And denoise the data by applying normalization via equation 1.

$$
\begin{aligned}
df1m[i,j] &= \frac{df1[i,j] - \min(df1[:,j])}{\max(df1[:,j]) - \min(df1[:,j])} \\
df2m[i,j] &= \frac{df2[i,j] - \min(df1[:,j])}{\max(df1[:,j]) - \min(df1[:,j])}
\end{aligned} \tag{1}
$$

2) Train / Test Data: Since we always want to predict the future, we take the latest 20% of data as the test data.

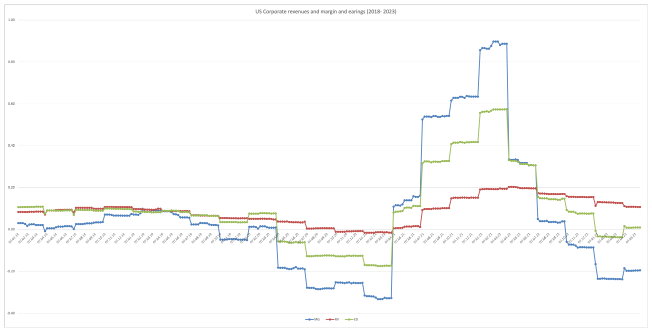

Fig. 5. Signals MG, RV, ED (2018 - 2023)

3) Sliding window: The corporate revenue is a time series of length N, defined as p_0, p_1, ..., $p_N - 1$ in which p_i is the close price on day i, $0 \leq i < N$. We have a sliding window of a fixed size (we refer to this as input_size) and every time we move the window to the right by size W, so that there is no overlap between data in all the sliding windows.

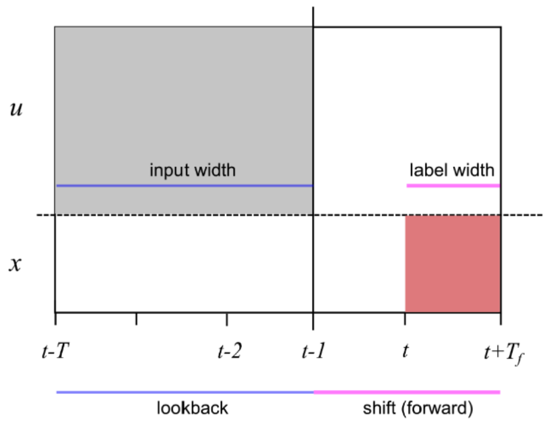

Fig. 6. Choice of sliding window

```
# define sliding window
lf =52    # look forward
ks =14    # kernel size
lw =1     # label width
lb =52

# look back
window =WindowGenerator(input_width=lb,
   label_width=lw, shift=lf,
   input_columns=['MOV','RV', 'MG'],
   label_columns=['ED'])
td =window.make_dataset(train_df,
   batchsize=150, shuffle=True)

# cross-validation
is_data =td.take(5)
os_data =td.skip(5)
```

B. *Choice of network type and parameters*

1) Parameters Definition: As shown in TABLE II.

TABLE II
PARAMETERS OF MODEL CONSTRUCTION

Parameter	Definition
lstm_size	number of units in one LSTM layer.
num_layers	number of stacked LSTM layers.
keep_prob	percentage of cell units to keep in the dropout operation.
learning_rate	the learning rate to start with.
learning_rate_decay	decay ratio in later training epochs.
epoch	number of epochs using the constant learning_rate.
max_epoch	total number of epochs in training.
input_size	size of the sliding window / one training data point.
batch_size	number of data points to use in one mini-batch.

2) Network Selection and Parameterization: We explore three different types of neural network models: Autoencoders (AE), Recurrent Neural Networks (RNN), and Convolutional Neural Networks (CNN). Each network type has its unique strengths and is more suited to particular types of data. The selection of a network type and its parameters will be governed by the nature of our task and the characteristics of our data.

Autoencoders (AE): Autoencoders are used for data compression, noise reduction, and feature extraction tasks. In the context of predicting corporate earnings, an AE could be utilized to extract valuable features from high-dimensional data or to reduce noise in the financial time series. Key parameters include the number of hidden layers and units per layer, the type of activation function, and the choice of optimizer.

Algorithm 1 Training an Autoencoder Neural Network

```
1: procedure TRAINAUTOENCODER(model, is_data, os_data)
2:     lb, num_inputs, num_hidden, kernel_size, pooling    ▷ Define parameters
3:     model ← InitializeAutoencoder(lb, num_inputs,
                   num_hidden, kernel_size, pooling)
4:     lr_schedule ← ExponentialDecay(0.01, 50, 0.97)
5:     model ← CompileModel(model, lr_schedule)
6:     EnableEagerExecution(model)
7:     early_stopping ← EarlyStopping('loss', 50)
8:     TrainModel(model, is_data, os_data, 500, early_stop)
9:     PrintModelSummary(model)
10: end procedure
```

Recurrent Neural Networks (RNN): RNNs are especially suited for sequential data, as they have 'memory' of previous inputs in their hidden layers. When dealing with time-series data like corporate margins and revenues over successive quarters, RNNs can be a good fit. Important parameters for RNNs include the number of hidden units, the number of layers, the type of RNN cell (like LSTM or GRU), the sequence length, and the learning rate.

Convolutional Neural Networks (CNN): CNNs are typically used for image or signal processing tasks, where spatial relationships in the data are important. In financial forecasting, a 1-D CNN can be employed to identify local patterns or trends in time series data. Key parameters for CNNs include the number and size of convolutional filters, the number of layers, the stride length, padding, the type of pooling, and the choice of optimizer and learning rate.

Algorithm 2 Training a LSTM Neural Network

```
1: procedure TRAINLSTM(model, is_data, os_data, rnn_units)
2:     model ← InitializeSequentialModel()
3:     model ← AddLSTMLayer(model, rnn_units, True, 'relu')
4:     model ← AddDropoutLayer(model, 0.2)
5:     model ← AddDenseLayer(model, 1, 'sigmoid')
6:     model ← AddDenseLayer(model, 1)
7:     lr_schedule ← ExponentialDecay(0.01, 150, 0.95)
8:     model ← CompileModel(model, lr_schedule)
9:     DisableEagerExecution(model)
10:    early_stopping ← EarlyStopping('loss', 100)
11:    TrainModel(model, is_data, os_data, 500, 150)
12:    PrintModelSummary(model)
13: end procedure
```

Algorithm 3 Training a Convolutional Neural Network (CNN)

```
1: procedure TRAINCNN(model, is_data, os_data, ks, lb, lw)
2:     model ← InitializeSequentialModel()
3:     model ← AddConv1DLayer(model, 4, ks, 'relu')
4:     model ← AddMaxPooling1DLayer(model, 4, 1)
5:     model ← AddConv1DLayer(model, 32, ks, 'relu')
6:     model ← AddMaxPooling1DLayer(model, 4, 1)
7:     fs ← $lb - 2 \times (ks - 1) - (lw - 1)$
8:     model ← AddConv1DLayer(model, 4, fs, None)
9:     model ← AddDenseLayer(model, 1)
10:    lr_schedule ← ExponentialDecay(0.01, 150, 0.95)
11:    model ← CompileModel(model, lr_schedule)
12:    DisableEagerExecution(model)
13:    early_stopping ← EarlyStopping('loss', 100)
14:    TrainModel(model, is_data, os_data, 500, 150)
15:    PrintModelSummary(model)
16: end procedure
```

The provided algorithm above outlines the procedure for constructing and training a Convolutional Neural Network (CNN) in a sequential manner, employing 1D convolutional layers suitable for time-series data or certain Natural Language Processing tasks. It involves building the network architecture with Conv1D and MaxPooling1D layers, using Rectified Linear Unit (ReLU) activation functions, and specifying a custom kernel size for one of the convolutional layers. The model also incorporates a Dense (fully connected) layer before the output.

The training of the model leverages the Stochastic Gradient Descent (SGD) optimizer with an Exponential Decay learning rate schedule and is compiled with a Mean Squared Error loss function, thus aligning the architecture for regression tasks. The model is trained on an input data set with a predefined number of epochs and batch size, and validated on an output data set. The model's summary provides an overview of the network's architecture and its parameters.

3) Parameter Tuning: Once the initial models are set up with base parameters, we will use techniques like Grid Search, Random Search, or Bayesian Optimization to tune the parameters. This step aims to find the optimal parameters that yield the best performance on the validation data.

Each of the three model types – AE, RNN, and CNN – will be evaluated based on their predictive performance on a hold-out validation set. Performance metrics could include Mean Squared Error (MSE), Mean Absolute Error (MAE), and R-Squared (R²) for regression tasks.

C. Results

Given training pairs (u, y), we fit three models: in the feed-forward set-up the neural net approximates a function that transforms inputs to targets in the "usual" way: y = f (u). An alternative is to combine x = [u y]T and build a model that learns to produce a copy of x, the so-called auto-encoder. The architecture of the network remains that same but the input and hidden layers assume the role of an encoding function h = f (x) while the output layer implements the decoding function = b(h), where is an (imperfect) reproduction of x. The auto-encoder does not make a difference between u and y and can therefore also be used in the context of un-supervised learning (i.e., where the target variable y is missing). The key is to constrain the capacity of the auto-encoder such as to avoid learning the identity map x = b(f (x)).

1) Hypothesis: The model describes corporate margin, corporate revenues and earnings in a market where the future earnings should be estimated before market margins are observed i.e., based on profit expectations xˆ of x = p p¯ , where p is price and p¯ is the (known) production cost. We expect the corporate earnings will keep 5 % rate in next two years.

2) Regression and Prediction: In this section, we would like to discuss the results about the training loss and the prediction.

AE

The result in Figure 7 shows that during the training phase, the training loss steadily decreases until around 0.35 over iterations, indicating that the auto-encoder is effectively capturing the underlying patterns in the selected signals. This smooth and gradual decline suggests that the model is learning the important features and minimizing the reconstruction errors, resulting in accurate representations of the input data.

Furthermore, the validation loss line closely aligns with the training loss line, indicating the autoencoder's ability to generalize well to unseen data. The low and stable validation loss signifies that the model is not overfitting to the training data but instead has learned a compact representation of the input data that can accurately reconstruct new, unseen inputs. This alignment between the training and validation loss lines demonstrates the robustness and reliability of the autoencoder, ensuring that it can effectively capture and reconstruct the essential features of the data. Figure 8 shows the prediction

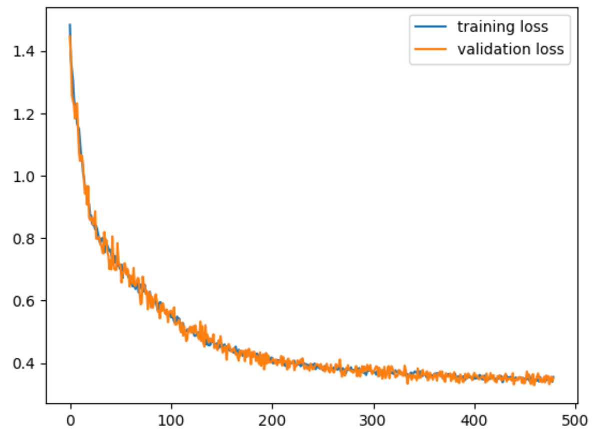

Fig. 7. AE training loss

results of AE. The mse in sample is 0.35 that is smaller than 0.44 from out sample. This indicates that AE performs better in the in sample situation. While the IR (-0.65) of out of sample is greater than -0.72 from in sample. A higher Information Ratio indicates a better risk-adjusted performance, suggesting that the model is effective in generating positive returns while managing risk effectively. This means that AE could capture more useful information in the out of sample situation. From Figure 8, the predicted lines fit better to the underlying one.

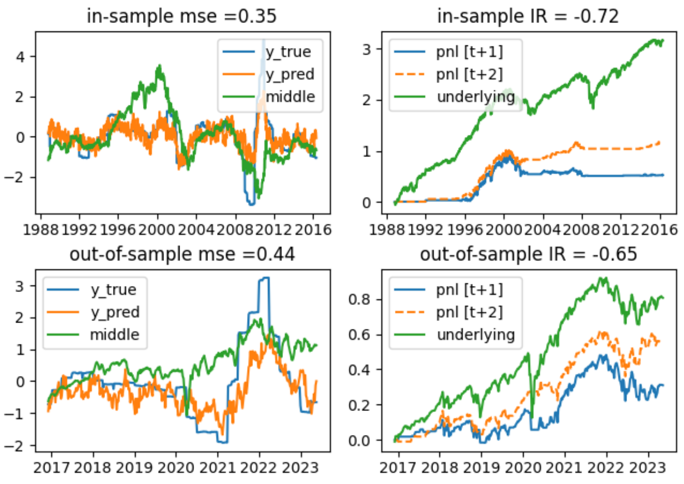

Fig. 8. AE training with prediction

RNN-LSTM

The result in Figure 9 shows that during the training phase, the training loss dynamically decreases until around 0.6 over iterations. Compared with the one from AE, it is more dynamically, so as the validation loss line. Training the LSTM model typically takes at least twice as much time compared to the AE model.

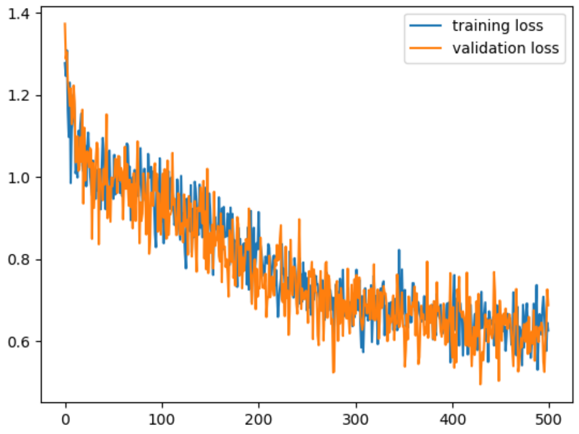

Fig. 9. RNN training loss

In term of the prediction (see Figure 10), it is observious that the LSTM model performs better in the in smaple situation with the mse loss only 0.62, comparing 1.30 from out of sample case. The in sample IR is greater than the one from out of sample (-0.41 VS -0.43), the difference between them is minor. LSTM gets similar effective information from both cases.

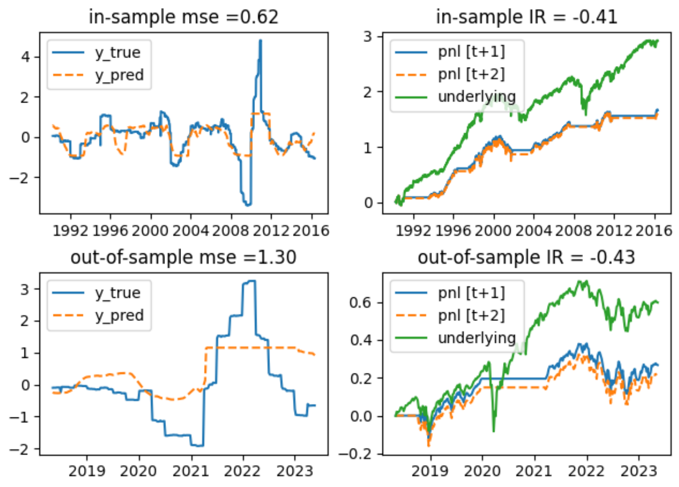

Fig. 10. RNN training with prediction

CNN

The training and validation loss curves of the CNN model steadily decrease and fall between the LSTM and AE models. However, they eventually reach a smaller loss value, around 0.1, and the training speed is the fastest among the three models (please check Figure 11). This shows that CNN might suit the best for given the task.

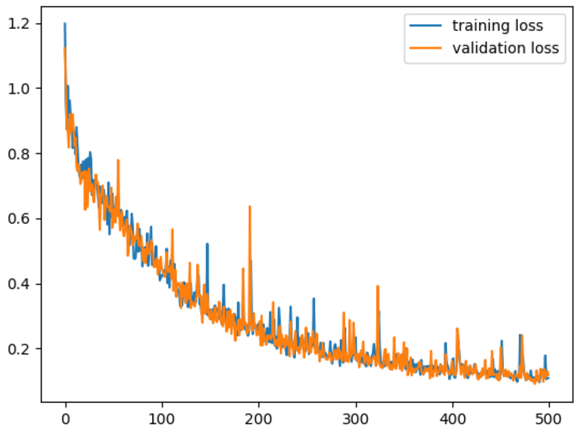

Fig. 11. CNN training loss

CNN has the highest IR (-0.29) from the in-sample case among models, and the lowest IR (-0.87) in the out of sample case.

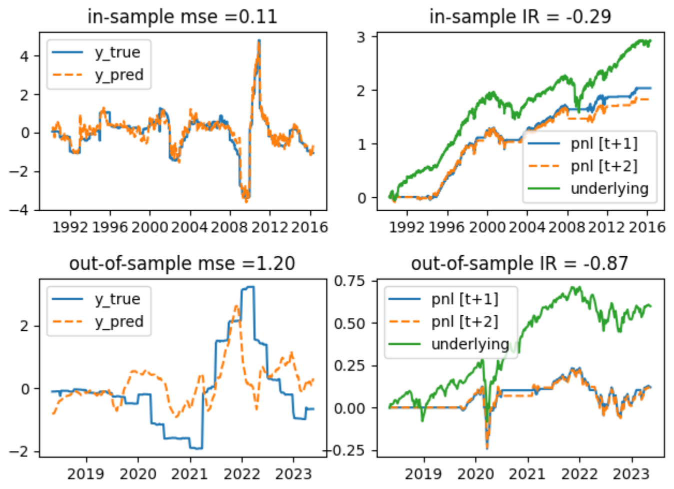

Fig. 12. CNN training with prediction

LSTM VS CNN

The same signals (”RV”, ”MG”, ”ED”) were taken to conduct the experiments in this part. As shown in Figure 13, ”sequential” is LSTM model and ”sequential 1” is the CNN model. There is no obvious differences in the training and validation stage.

However, it seems that the CNN model exhibits delayed predictions in the out of sample case (see Figure 14), which may be related to the size of the window. In term of IR, ”P/L

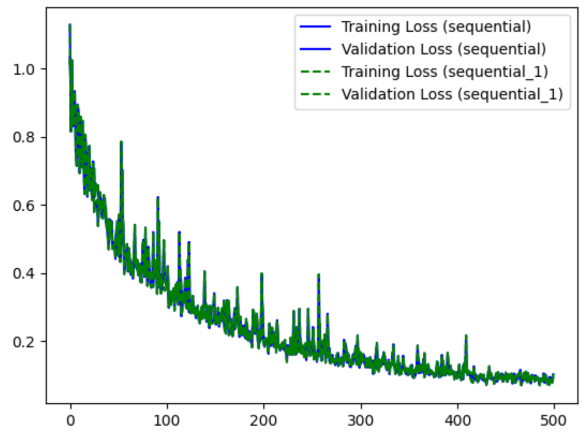

Fig. 13. lstm vs cnn training loss

(t+2)" line fits the underlying one nicely, whereas the P/L lines from LSTM seem far away from the underlying one. As for the out of sample IR, all the lines fit well except for the one "P/L (t+1)" from CNN, which is considerable higher than the underlying.

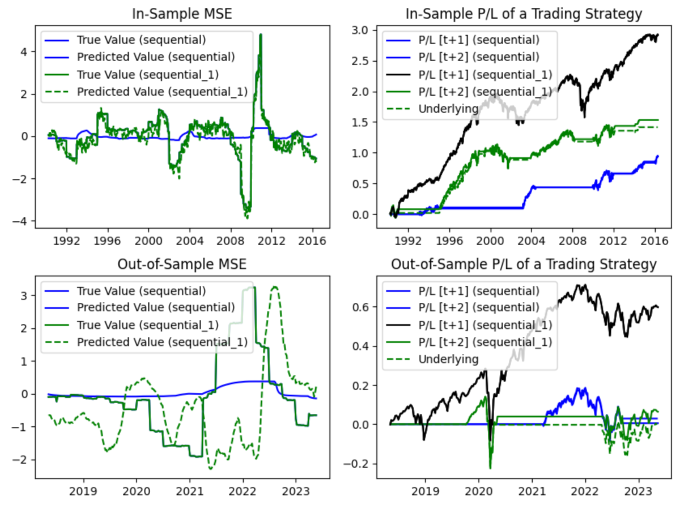

Fig. 14. lstm vs cnn training with prediction

3) Discussion: Based on the results obtained, the autoencoder (AE), LSTM, and CNN models each exhibit unique strengths and weaknesses. The AE model shows effective pattern capture but struggles with out-of-sample performance. The LSTM model performs well overall but has a longer training time. The CNN model has fast training speed and good in-sample performance but shows delayed predictions in the out-of-sample case, potentially related to the window size. Each model has its advantages and considerations, and further improvements can be explored to enhance their performance.

V. Conclusion

In conclusion, the analysis of the autoencoder (AE), LSTM, and CNN models reveals their respective strengths and limitations in capturing patterns and making predictions. The AE model demonstrates its ability to effectively capture underlying patterns in the input data, leading to accurate representations and reconstruction. However, its performance in out-of-sample scenarios may be limited. The LSTM model showcases its capability to capture long-term dependencies and exhibit good performance both in-sample and out-of-sample. Despite its longer training time, it proves to be a reliable model for sequential data analysis. The CNN model stands out for its fast training speed and decent in-sample performance, although it exhibits delayed predictions in some out-of-sample cases. Overall, each model has its unique characteristics and considerations, providing a range of options for various applications and datasets. Further research and optimization can help maximize their strengths and mitigate their limitations in order to achieve more accurate predictions and insights.

While machine learning holds great promise in enhancing active portfolio management, it is not without challenges. Ensuring data quality, choosing the right algorithms, and interpreting results accurately requires skill and experience. However, with proper implementation, machine learning has the potential to revolutionize active portfolio management by providing more accurate predictions, efficient risk management, and enhanced decision-making. By leveraging machine learning for tasks such as predicting corporate earnings based on corporate margin and revenues, active portfolio management can be significantly improved.

Acknowledgements

We would like to express our gratitude to Prof. Dr.Matthias Feiler for his outstanding teaching and expertise, which greatly contributed to our understanding and knowledge in the field. Thanking him for the valuable comments and assistance throughout the entire project. His guidance and insights significantly improved the quality and depth of our work.

References

[1] X. Guo, Q. Liang, and X. Liu, "Portfolio optimization based on machine learning: A survey," *Complexity*, pp. 1–24, 2020.

[2] A. Lim, "Application of machine learning techniques in portfolio management: A review," *The Journal of Finance and Data Science*, vol. 5, no. 3, pp. 199–207, 2019.

[3] A. Agrawal, J. Gans, and A. Goldfarb, *Prediction Machines: The Simple Economics of Artificial Intelligence*. Boston, MA: Harvard Business Review Press, 2019.

[4] T. Loughran and B. McDonald, "When is a liability not a liability? textual analysis, dictionaries, and 10-ks," *The Journal of Finance*, vol. 71, no. 1, pp. 399–436, 2016.

Appendix

```
# data preparation
self.data =self.data.set_index(self.data['Date'])
self.data =self.data.drop(columns='Date')
self.data =self.data.dropna()
```

```
# set up train/ test data.
data_processor =DataProcessor('market_data.xlsx')
train_data, test_data =
   data_processor.split_train_test(test_size=0.2)
```

```
# denoising set-up
df_n =df +1.0 *np.random.normal(0, 1, df.shape)
cols_n =[str(i)+'_n' for i in cols]
df_n.columns =cols_n
df =df.join(df_n)
```

```
# autoencoder model
autoencoder_units =128
dropout_rate =0.2
autoencoder_latent_dim =1

autoencoder_model =create_model('autoencoder',
   lb =lb, num_inputs_auto =3, num_hidden_auto =1,
   kernel_size_auto =25, pooling_auto =pooling)

lr_schedule =
   tf.keras.optimizers.schedules.ExponentialDecay(
   0.01, decay_steps=50, decay_rate=0.97,
   staircase=True)

autoencoder_model.compile(
   loss=tf.losses.MeanSquaredError(),optimizer=
   tf.optimizers.SGD(learning_rate=lr_schedule),
   metrics=[tf.metrics.MeanSquaredError(), ])

autoencoder_model.run_eagerly =True
early_stopping =tf.keras.callbacks.EarlyStopping(
   monitor='loss', patience=50, mode='min')
history =autoencoder_model.fit(
   is_data, validation_data=os_data, epochs=500,
   callbacks=[early_stopping])
autoencoder_model.summary()
```

```
# RNN model
rnn_units =128
dropout_rate =0.2

rnn_model =create_model('rnn', rnn_units =256)
lr_schedule =
   tf.keras.optimizers.schedules.ExponentialDecay(
   0.01, decay_steps=150,
   decay_rate=0.95, staircase=True)

rnn_model.compile(loss=tf.losses.MeanSquaredError(),
   optimizer=tf.optimizers.SGD(
   learning_rate=lr_schedule),
   metrics=[tf.metrics.MeanSquaredError()])
rnn_model.run_eagerly =False

early_stopping =tf.keras.callbacks.EarlyStopping(
   monitor='loss', patience=100, mode='min')
history =rnn_model.fit(is_data,
   validation_data=os_data, epochs=500,
   batch_size=150, callbacks=[early_stopping])
rnn_model.summary()
```

```
# CNN model
model =tf.keras.Sequential()
# Version 1: Convolutional Network
model.add(tf.keras.layers.Conv1D(
   filters=4, kernel_size=ks,
   activation='relu', use_bias=False))
model.add(tf.keras.layers.MaxPooling1D(
   pool_size=4, strides=1, padding='same'))
model.add(tf.keras.layers.Conv1D(
   filters=32, kernel_size=ks,
   activation='relu', use_bias=False))
model.add(tf.keras.layers.MaxPooling1D(
   pool_size=4, strides=1, padding='same'))
model.add(tf.keras.layers.Conv1D(
   filters=4, kernel_size=lb-2*(ks-1)-(lw-1),
   activation=None, use_bias=False))
model.add(tf.keras.layers.Dense(units=1))

# Version 2: Recurrent Network
lr_schedule =
   tf.keras.optimizers.schedules.ExponentialDecay(
   0.01, decay_steps=150,
   decay_rate=0.95, staircase=True)
model.compile(loss=tf.losses.MeanSquaredError(),
   optimizer=tf.optimizers.SGD(
   learning_rate=lr_schedule),
   metrics=[tf.metrics.MeanSquaredError()])
model.run_eagerly =False
early_stopping =tf.keras.callbacks.EarlyStopping(
   monitor='loss', patience=100, mode='min')
history =model.fit(is_data,
   validation_data=os_data, epochs=500,
   batch_size=150)#, callbacks=[early_stopping])
model.summary()
```